École Nationale Supérieure d'Informatique et d'Analyse des systèmes
Centre d’Études Doctorales en Sciences des Technologies de l'Information et de l'Ingénieur

THÈSE DE DOCTORAT

Formal Design of Anomaly-Free Security Policies

Présentée par

Meryeme AYACHE

Le 25/09/2018

Formation doctorale : Informatique
Structure de recherche : ITM
Information Technology and Management

JURY

Professeur Mahmoud NASSAR
PES, ENSIAS, Université Mohammed V de Rabat

Professeur Mohammed ERRADI
PES, ENSIAS, Université Mohammed V de Rabat

Professeur Habiba CHAOUI
PES, ENSA, Université Ibn Tofail, Kénitra

Professeur Anas ABOU EL KALAM
PH, ENSA, Université Cadi Ayyad, Marrakech

Professeur Hanan EL BAKKALI
PES, ENSIAS, Université Mohammed V de Rabat

Professeur Driss BOUZIDI
PH, ENSIAS, Université Mohammed V de Rabat

Président
Directeur de thèse
Rapporteur
Rapporteur
Examineur
Acknowledgements

I would like to offer my sincerest gratitude to my supervisor, Prof. Mohammed Erradi. He has been supportive throughout my typical path at ENSIAS. His rigorous monitoring and valuable directions, during my engineering’s degree, have encouraged me to pursue my studies for a PhD degree. He is an interesting person that impacted my personal life due to his serious and hard working character. For this, I cannot thank him enough. I am forever grateful.

Special thanks go to Prof. Ahmed Khoumsi, my co-supervisor, who lightened my path during my PhD research. His support, guidance, and advice, have helped me improve the thesis. Prof. Khoumsi taught me a lot about research and showed me the art of scientific writing. This dissertation would look completely different without him. So thank you.

I express my sincere gratitude to my examination committee: Prof. Mahmoud Nassar, Prof. Habiba Chaoui, Prof. Anas Abou El Kalam, Prof. Hanane El Bakkali and Prof. Driss Bouzidi, for their insightful comments and encouragement. Their guidance through this process, their discussions, ideas, and feedback have been absolutely invaluable.

I would like to extend my thanks to Prof. Bernd Freisleben. He invited me twice to his Mathematik und Informatik laboratory in Marburg University, Germany. Thank you for the fruitful discussions, guidance, and encouragement.

I am so grateful to Prof. Mira Mezini and Dr. Tom Dinkelaker. I have been invited to work with their team at TU Darmstadt in Germany. They gave me the opportunity to explore interesting research areas such as Aspect Oriented Programming. These experiences had a great impact on the result of this thesis.

I would like to thank the staff of the Emergency Medical Service (EMS) of Rabat, especially Dr. Marouane Hamdani for all the information they provided for us. Their feedback helped us to understand the process of collaborative remote diagnoses that we integrated in this thesis as case study scenario.

I acknowledge the people who mean a lot to me, my parents, for showing faith in me and giving me freedom to choose what I desired. I salute you all for the selfless love, care, pain and sacrifice you did to shape my life. Although you hardly understood what I researched on, you were willing to support any decision I made. I would never be able to pay back the love and affection showered upon by my parents.

Finally, I owe thanks to a very special person, my husband, Najib for his continued and unfailing support and understanding during my pursuit of Ph.D. You were always around at times I thought that it is impossible to continue, you helped me to keep things in perspective. I appreciate my little baby Loqmane for abiding my ignorance and the patience he showed during my thesis writing. Words would never say how grateful I am to both of you.
Dedication

I am dedicating this thesis to two beloved people who have meant and continue to mean so much to me. Although they are no longer of this world, their memories continue to regulate my life. First and foremost, to my maternal grandmother whose love for me knew no bounds and, who taught me the value of hard work. I love you and miss you beyond words. May Allah (SWT) grant you Jannah Firdaws.

I also want to remember my uncle, Lhssen Jtit, who taught me the meaning of patience and perseverance. He impacted my life with his wisdom and his kindness. Thank you so much dear uncle, I will never forget you.
Abstract

Healthcare is increasingly a cooperative business that involves many individuals and organizations. Data sharing across such organizations is just one of the major pillar in improving the health care business. In fact, it enhances the collaboration between different stakeholders by exchanging medical records. Such Collaborations guarantee the efficiency and cost-effectiveness of medical diagnoses. However, due to the current Big data exponential growth, solutions that store, exchange and process medical data are of a great interest. In this direction, cloud computing represents a great solution for such needs.

Cloud Computing is the most suitable environment for the interoperability of multiple medical organizations in e-health systems. The interoperability is warranted by multi-tenancy cloud architectures. In such architectures, multiple customers, called tenants, transparently share the cloudâ€™s resources and services. Multi-tenancy permits cloud providers to establish collaborative relationships among cloud users, especially at the storage service level. However, the usage of cloud computing for medical environments raises several issues such as reliability and security. Due to the sensitivity of data in the health-care domain, ensuring data sharing makes matters even more problematic. Hence, each medical organization provides a certain security level to its own data by enforcing a set of security policies.

In this thesis, our main contributions consist in developing approaches to manage security policies imposed by medical organizations in the cloud. Most of the cloud environments use XACML as a policy specification language. Yet, XACML has many limitations. For instance, XACML lacks a mechanism to detect policy conflicts and redundancies. However, e-Health domain does not tolerate such errors due to the sensitivity of patient’s health data. In this direction, we propose automata-based approaches to detect and resolve anomalies in XACML policies such as: conflicting rules, redundancies, inconsistencies, policy similarities, and ambiguities; and to verify the completeness property that guarantees that each access request is either accepted or denied by the medical policy. The approaches are integrated in a middleware as cloud services. The main idea behind the proposed approaches is to represent the policies using automata. Then, it uses the final states of the resulting automata to detect and resolve anomalies. The resolution approach uses Aspect Oriented Finite State Machine (AO-FSM) to dynamically choose the adequate combining algorithm. The automata-based policy evaluation approach shows a significant improvement in term of policy/request evaluation comparing to existing XACML implementations.

keywords - Cloud Computing; OpenStack; e-Health; XACML; Anomalies; Redundancies; Ambiguities; Conflicts; Automata; Policy Evaluation; Aspect Oriented Finite State Machine.
Résumé

La santé est de plus en plus une entreprise coopérative qui implique de nombreuses personnes et organisations. Le partage de données entre ces organisations est l’un des principaux piliers de l’amélioration de l’activité de soins de santé. En fait, les échanges des dossiers médicaux garantissent l’efficacité et la rentabilité des diagnostiques médicaux. Cependant, en raison de la croissance exponentielle actuelle des grandes données, on a toujours besoin d’un environnement qui nous permet de stocker et échanger les données médicales.

Cloud Computing est l’environnement le plus approprié pour l’interopérabilité de plusieurs organisations médicales dans les systèmes de e-santé. En fait, le cloud offre des architectures multi-tenancy, dans lesquelles plusieurs clients, appelés locataires, partagent de manière transparente les ressources et les services du cloud. Le multi-tenancy permet aux fournisseurs du cloud d’établir des relations de collaboration entre les utilisateurs du cloud, en particulier au niveau du service de stockage. Cependant, l’utilisation du cloud computing pour les environnements médicaux soulève plusieurs problèmes tels que la fiabilité et la sécurité. Par conséquent, chaque organisation médicale assure la sécurité de ses propres données en imposant une politique de sécurité.

Dans cette thèse, nos principales contributions consistent à développer des approches pour gérer les politiques de sécurité imposées par les organisations médicales dans le cloud. La plupart des environnements cloud utilisent XACML comme langage de spécification de politique. Pourtant, XACML a de nombreuses limitations. Par exemple, XACML n’a pas de mécanisme pour détecter la non-complétude et les redondances dans les politiques. Cependant, le domaine e-santé ne tolère pas ces erreurs en raison de la sensibilité des données de santé du patient. Dans ce sens, nous proposons des approches basées sur les automates pour détecter et résoudre les anomalies dans les politiques XACML telles que : les conflits, les redondances, les incohérences, les similarités de politiques, et les ambiguïtés ; et pour vérifier la propriété de complétude qui garantit que chaque demande d’accès est acceptée ou refusée par une politique de sécurité. Les approches sont intégrées dans un middleware en tant que services cloud. L’idée principale derrière les approches proposées est de représenter les politiques en utilisant des automates. Ensuite, on utilise les états finaux des automates pour déterminer et résoudre les anomalies. L’approche fondée sur les automates montre une amélioration significative en terme de l’évaluation des requêtes vis-a-vis les politiques par rapport aux implémentations XACML existantes.

Mots-clés - Cloud Computing ; OpenStack ; e-santé ; XACML ; Anomalies ; Redondances ; Ambiguïtés ; Conflits ; Automates ; Evaluation des politiques ; Automates orientés aspect.
Contents

List of Figures viii
List of Tables x
Listings xi
List of Abbreviation xii
List of Publications xiv

Chapter 1:  General Introduction 1
1.1 Motivation ......................................................... 1
1.2 Problem Statement ............................................. 3
1.3 Contributions .................................................... 3
1.4 Dissertation’s organization ..................................... 5

Chapter 2:  Background 6
2.1 Introduction ..................................................... 6
2.2 Scenario and Notation .......................................... 6
   2.2.1 Subjects: Human Resources ............................... 7
   2.2.2 Objects: Stored Information ............................. 9
   2.2.3 Medical Organizations .................................. 9
2.3 Access Control Policies: XACML ............................. 11
   2.3.1 XACML Policies specification ......................... 11
   2.3.2 Abstract Syntax for XACML requests ................. 15
   2.3.3 Example of XACML-based e-Health policies ......... 16
2.4 Automata ....................................................... 18
   2.4.1 Basics of Finite State Automata ....................... 18
   2.4.2 Synchronous Composition of Automata ................ 19
2.5 Cloud Computing: Openstack ............................... 19
2.6 Object Storage System: Swift OpenStack .................. 22
2.7 Summary ....................................................... 25

Chapter 3:  State of the Art 27
3.1 Introduction .................................................... 27
Chapter 4: Automata-based modeling of XACML policies

4.1 Introduction

4.2 Automata Model of XACML policies
   4.2.1 Step 1: Attribute extraction from XACML policies
   4.2.2 Step 2: Automaton for each rule
   4.2.3 Step 3: Standardize the intervals of the automaton
   4.2.4 Step 4: Composition of automata

4.3 Other examples of Synthesis of automata

4.4 Chapter Summary

Chapter 5: Automata-based evaluation of XACML policies

5.1 Introduction

5.2 curlX: a Middleware to Enforce Policies in OpenStack
   5.2.1 Architecture
   5.2.2 CurlX Translator
   5.2.3 X2Automata: An efficient policy evaluation engine

5.3 Implementation and Evaluation

5.4 Chapter Summary

Chapter 6: Anomalies detection in XACML policies

6.1 Introduction

6.2 Multi-Tenancy Collaboration

6.3 Anomalies Categorization

6.4 CPVS: Policy Verification Service
   6.4.1 Anomalies Detection Process
   6.4.2 Verification of the Completeness Property

6.5 Evaluation of Space and Time Complexities

6.6 Implementation and Evaluation

6.7 Chapter Summary

Chapter 7: Anomalies resolution in XACML policies

7.1 Introduction

7.2 Aspect Oriented Finite State Machine: AO-FSM
List of Figures

2.1 Organisation involved in the collaborative diagnosis ........................................ 10
2.2 Example of an automaton’s graph ................................................................. 19
2.3 Deployment models of Cloud computing ......................................................... 20
2.4 Architecture of Openstack ............................................................................. 22

4.1 Automaton $A_2$ obtained in Step 2 which models rule $h_2$ of Table 2.1 ............ 40
4.2 Automaton $A_2^*$ obtained from $A_2$ of Fig. 4.1 ........................................... 41
4.3 Automata $A_{H_1}$ modeling the security policy of $H_1$ ....................................... 42
4.4 Automata $A_{H_2}$ modeling the security policy of $H_2$ ....................................... 43
4.5 Automata $A_{H}$ Modelling the Security Policy of Table 2.3 ................................. 43

5.1 Interaction between different components of the curlX middleware ...................... 46
5.2 Automata $A_{H_1} \times A$ modeling $Acc = (generalist, PR, create)$ and $SecPol_x$ .... 51
5.3 Pre-processing (Automata construction) time on synthetic XACML policies .......... 52
5.4 Processing time difference between Sun PDP and X2Automata .......................... 52

6.1 Multi-tenant outsourced data sharing ............................................................. 55
6.2 Anomaly categories ....................................................................................... 56
6.3 The three modules of CPVS (Cloud Policy Verification Service) ........................ 57
6.4 Automaton modeling the combination of $H_1$ and $H_2$ of Tables 2.1 and 2.2 ....... 60
6.5 Effect of number of rules for Intra-policy Anomalies Detection .......................... 65
6.6 Effect of rules’ number for Inter-policy Anomaly Detection ............................... 66
6.7 Effect of rules’ number for the Completeness Property Verification .................... 66

7.1 Workflow representing the anomaly resolution strategy ..................................... 71
7.2 Different pointcut patterns ............................................................................. 72
7.3 Different advice patterns .............................................................................. 73
7.4 Anomalies detection and resolution performance ............................................ 76
7.5 Detected and resolved anomalies ................................................................. 76

8.1 Global architecture of the middleware ............................................................ 79
8.2 Curl Primitive Translation (CPT) and its interactions. ......................... 80
8.3 Policy Automata Generation (PAG) and its interactions. ....................... 81
8.4 Policy Evaluation (PE) and its interactions with AAG. ......................... 81
8.5 Completeness Verification (CV) and its interactions. .......................... 82
8.6 Intra-Policy Anomalies Detection (Intra-PAD) and its interactions with PAG and AR. .................................................. 82
8.7 Inter-Policies Anomalies Detection (Intra-PAD) and its interactions with PAG and AR. .................................................. 83
List of Tables

2.1 XACML policy of hospital $H_1$ .................................................. 16
2.2 XACML policy of hospital $H_2$ .................................................. 17
2.3 XACML policy of hospital $H$ containing anomalies .................... 17

3.1 The Capabilities of the Proposed Access Control Verification Approaches ................................................. 35

4.1 How the constituents of a XACML policy are represented in the corresponding automaton ................................................................. 39

7.1 Prioritization of decisions. .............................................................. 71
Listings

2.1 A simple ALFA specification of rule with a condition ........................................ 12
2.2 Example of a XACML Policy .............................................................................. 14
2.3 Example of a XACML Request .......................................................................... 15
2.4 Example of a XACML response ......................................................................... 16
5.1 Example of curl query translated to XACML request ........................................ 48
1  the groovy FSM representing the hospital’s policy of Table 2.1. ......................... 101
2  Groovy code of the FinalStateJoinPoint .............................................................. 102
3  Groovy code of the ExitStateJoinPoint ............................................................... 102
4  Excerpt of RedundancyPC to define final states with redundancies ..................... 102
5  Implantation of removeRedundancyAdvice ....................................................... 103
# List of Abbreviation

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAC</td>
<td>Attribute Based Access</td>
</tr>
<tr>
<td>AO-FSM</td>
<td>Aspect Oriented Finite State Machines</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>CAT</td>
<td>Computed Axial Tomography</td>
</tr>
<tr>
<td>DAC</td>
<td>Discretionary access control</td>
</tr>
<tr>
<td>EEG</td>
<td>Electroencephalography</td>
</tr>
<tr>
<td>EMS</td>
<td>Emergency Medical Service</td>
</tr>
<tr>
<td>EPR</td>
<td>Electronic Patient Records</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machines</td>
</tr>
<tr>
<td>HH</td>
<td>Host Hospital</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>MAC</td>
<td>Mandatory access control</td>
</tr>
<tr>
<td>MRA</td>
<td>Magnetic Resonance Angiogram</td>
</tr>
<tr>
<td>MRI</td>
<td>Magnetic resonance imaging</td>
</tr>
<tr>
<td>OrBAC</td>
<td>Organization-Based Access Control</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>PR</td>
<td>Patient’s Personal</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role Based Access Control</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>TBAC</td>
<td>Task Based Access Control</td>
</tr>
<tr>
<td>XACML</td>
<td>eXtensible Access Control Markup Language</td>
</tr>
</tbody>
</table>
List of Publications

This thesis presents the research work done during my PhD under the supervision of Prof. Mohammed Erradi and the co-supervision of Prof. Ahmed Khoumsi. I was invited twice by Prof. Bernd Freisleben to the Mathematik und Informatik laboratory in Marburg University, Germany, where I worked with his Distributed Systems Group on security policies in Openstack. I had also the opportunity to visit Prof. Mira Mezini’s lab in Darmstadt University, Germany, where I worked on Aspect Oriented Programming (AOP) under the supervision of Dr. Tom Dinkelaker:

• M. Ayache, M. Erradi, B. Freisleben, and A. Khoumsi, "Using Aspect-Oriented State Machines for Resolving Conflicts in XACML Policies" in Proceedings of International Conference on NETworked sYStems (NETYS), 17-19 May 2017. (Chapters 6 and 7).


• M. Ayache, M. Erradi, and B. Freisleben, “curlx: a MiddleWare to enforce access control policies within a cloud environment”, in Proceedings of IEEE CNS 2015 Poster Session, Florence, Italy, Sep. 2015, pp. 771–772. (Chapter 5).

Chapter 1

General Introduction

1.1 Motivation

Until the 1950s, hospitals stored manually medical information of their patients, using essentially paper and ink. Any simple operation, such as searching a patient’s medical folder, requires much time. Less simple operations, such as sharing patients’ information between medical organizations, were very limited, even inexistent in most cases. Since 1960s, several medical organizations have begun progressively to explore the introduction of computer and communications technologies for storing and processing electronically their patients’ medical records [49]. Records using paper and ink have been replaced by digital records, called electronic patient records (EPR).

The EPR consists of a set of information, for example: medical history, diagnoses, medications, treatment plans, immunization dates, allergies, radiology images, laboratory and test results. A first advantage of an EPR is that it provides accurate, up-to-date, and complete information about patients. Another advantage of an EPR is that it can be quickly accessed by doctors, and hence allows more effective diagnoses and reduces risks of medical errors. Yet another advantage of EPR is that it favors to share information with other medical organizations. A typical situation is that of a patient (e.g., victim of an accident) that has to be transferred to the closest hospital $H_1$, while the patient’s EPR is stored in the database of another hospital $H_2$. In such a case, it is clearly indispensable that $H_2$ shares the EPR’s content with doctors of $H_1$. What is also strongly more desirable is that the EPR’s content can also be shared with doctors of other hospitals than $H_1$ and $H_2$, in order to regroup several relevant competencies to study the situation of the patient. Such a sharing can be realized through collaborative sessions involving doctors from various medical organizations.

In a collaborative session, various doctors have access to information in the patient’s EPR. Simply speaking, a collaborative session can be seen as a sequence of accesses, where each access specifies a type of action of a doctor to particular data in the EPR.
Example of access: a neurologist reads a magnetic resonance imaging (MRI) of the patient. Collaborative sessions are motivated by the desire to regroup several relevant competencies that contribute to obtain a diagnosis as precise and correct as possible for the considered patient. The collaborative approach consists in sharing different medical data such as: patient’s medical records, prescription information, X rays, test results, physician’s references, physicians availability, etc. This information is primordial for making decisions, obtaining better diagnosis and treatments to yield better results. However, sharing a huge amount of data is just one of the major impediments in the health care business that hinders progress towards efficiency and cost-effectiveness.

In parallel to the rapid development of information technologies, it is increasingly convenient and efficient to provide good healthcare information services. However, the production, collection, sharing, processing and storage of data is expanding at an astonishing pace. Therefore, there is a need for more innovative and cost-effective solutions to address this growing problem. Cloud computing has the potential to develop some of the solutions needed to address these problems. In fact, Cloud technology is used to create a network between patients, doctors, and healthcare institutions by providing applications, services and also by keeping all the data in the cloud. The current technology adopting Cloud computing in the medical field can improve and solve many collaborative information issues within healthcare organizations. For instance, In November 2010, IBM and ActiveHealth Management, a subsidiary of Aetna (a leading American provider of healthcare services), worked together to create the Collaborative Care Solution [25] that gives physicians and patients access to the information they need to improve the overall quality of care, without the need to invest in new infrastructures. The solution aims to enable easy access by medical and health carers to a wide range of information from disparate sources such as electronic medical records (EMRs), claims, medication and laboratory data.

Cloud Computing offers a new business model which enables several advantages to the healthcare community. By adopting the Cloud in medical services both patients and healthcare organizations would obtain a huge benefit in patient’s quality of service, collaboration between healthcare organizations as well as reductions in IT cost in healthcare companies. In fact, due to the multi-tenancy characteristic of the cloud, several medical organizations may share resources which permits the interchangeability of data and hence the collaboration between different medical organizations. Cloud technology provides also a storage service that can be accessed anywhere and everywhere by authorized entities which facilitates the remote diagnoses of patients’ ERP in different locations.

When moving to the cloud, there is also a very important beneficial factor for healthcare organizations, which is the IT costs [104]. By adopting the cloud model, all the IT processes are migrated to a remote cloud computing infrastructure. Therefore, medical organizations do not
need anymore to purchase expensive hardware infrastructure, or to keep/train in-site staff for maintenance, security, replications because the cloud computing providers takes care of them.

1.2 Problem Statement

Human life is priceless, and heath stakeholders have very critical roles where any leakage or decisions’ error may cause very complicated situations that may lead to death. Therefore, healthcare data have strict confidentiality, privacy and security concerns. Healthcare organizations need to take into consideration some security concerns before moving to Cloud Computing infrastructures. While the cloud appears to present several benefits, it also appears to present special risks to healthcare data with respect to interoperability, privacy and security.

Healthcare data, unlike other kinds of data, have strict confidentiality [84]. Data confidentiality refers to only authorized parties can access the data. Yet delegating data control to the Cloud providers may increase the number of data compromises. Therefore, each medical information needs to be associated to a set of access control rules imposed by its owner organization. The problem is that the current Cloud architectures do not offer facilities to its users to express their own security policies.

Interoperability is one of the biggest challenges when moving healthcare systems to the cloud [3]. Healthcare organizations are using diverse programming languages, operating systems, security protocols, etc. System interoperability can be resolved by sharing a common data model and design products that can interact with each other [39]. Interoperability can be also reached by using universal standards, which makes version control, updating and maintenance easier. However, security policy interoperability remains a challenging task for medical organizations especially during collaborations. Medical organizations may use different specification languages to represent their own security policies. Hence, Composing the policies during collaboration may generate both language interoperability and security interoperability (anomalies) such as redundant rules, ambiguous rules, conflicting rules, policy inconsistencies, and policy similarities. Unauthorized access can become possible through the exploitation of such interoperabilities, bringing up issues of data availability and integrity.

1.3 Contributions

Most of the cloud environments use XACML (eXtensible Access Control Markup Language) as a policy specification language. Yet XACML has many limitations in terms of policy anomaly detection. For instance, XACML lacks a mechanism to detect policies’ conflicts and redundancies. Moreover, XACML performs a brute force search to compare an access request to all existing rules in a given policy. This decreases the decision process (i.e., policy evaluation)
performance especially for policies with a large number of rules. On the other hand, health care requires quick interventions to save lives, and anomaly-free policies to guarantee the privacy of patients’ data. To overcome these limitations, we propose formal approaches based on automata in order to analyze and, if necessary, correct the XACML policies.

The contributions of this thesis are summarized below:

- **Translation of XACML policies into automata**: We propose a translation of XACML policies into automata. Each security policy is then modeled by an automaton consisting of an initial state, a set of states and transitions labeled by subjects and objects, and a set of final states associated to authorizations. As we will see in the following contributions, the model of automata is convenient for different policy studies and verification. (Chapter 4)

- **An automata-based approach to increase the policy evaluation performance of XACML**: We propose an automata-based approach to evaluate a XACML Policy against a given request. This necessitates to check whether a request satisfies a XACML policy. The automata-based approach consists in modeling both the policy and the request by two automata, the final states of their synchronous composition permit the policy evaluation. This approach replaced the evaluation by brute force search of XACML implementations which reduces the policy evaluation performance. (Chapter 5)

- **A generic automata-based approach to detect anomalies**: We propose an automata-based approach that uses the same automata-model to detect different anomalies: intra-policy anomalies (redundancies, ambiguities and conflicts), and inter-policies anomalies (consistencies and similarities). The same model permits also the verification of the completeness property which guarantees that each access request is either accepted or denied by the access control policy. (Chapter 6)

- **A dynamic automata-based approach to resolve anomalies**: The objective of resolution approach is removing anomalies by modifying appropriately the policy so that it becomes anomaly-free. We use aspect oriented finite state machines (AO-FSM) for the resolution strategy. The approach consists in describing a set of finite state machines’ patterns for each anomaly. The pattern then triggers a set of methods denoted advices to remove the anomalies. (Chapter 7)

- **A Cloud Middlware for security policies’ management**: We provide a cloud middleware to implement the proposed automata-based approaches as cloud services. (Chapter 8)

The proposed approaches were implemented as three services within a cloud middleware.
denoted $curlX$: (a) *curl Translator* translates the user’s requests into the XACML syntax, (b) *$X2Automata$*: represents a XACML policy evaluation engine (i.e. it determines whether a given request satisfies a policy), and (c) *CPVS* (Cloud Policy Verification Service): detects and resolves anomalies between XACML policies and verifies completeness of XACML policies.

The proposed cloud middleware has the following advantages:

1. The cloud users can specify their own security policies instead of delegating access control to the cloud providers. They can also analyze and correct their own security policies through anomaly detection and resolution, respectively.

2. The adoption of XACML as a unified language eliminates the policy language’s interoperability.

3. The automata-based approach proposed to detect and resolve anomalies is generic; it can detect and resolve several anomalies using the same formal model.

### 1.4 Dissertation’s organization

This thesis consists of seven chapters. These chapters will now be introduced to give an overview of each of them.

- Chapter 2 gives a brief overview of the medical scenario presenting the case study. It also elaborates on definitions of XACML, automata and Cloud Computing.

- Chapter 3 presents a state of art related to access control models and policy specification languages. It also discusses different existing approaches to detect and resolve anomalies and existing policy evaluation systems.

- Chapter 4 shows how to represent a XACML policy by an automaton. This model will be useful for the elements of chapters 5, 6 and 7. In fact, those three chapters justify the use of the model of automata.

- Chapter 5 uses the automata-based model for the policy evaluation process.

- Chapter 6 uses the automata-based model of the policy to verify its completeness and detect its anomalies.

- Chapter 7 uses the automata-based model of the policy to resolve the detected anomalies, if any.

- Chapter 8 presents the software architecture of the cloud middleware that we have developed. The architecture details different middleware’s features and their interactions.

- Chapter 9 concludes the dissertation and sets the perspectives of this thesis.
Chapter 2

Background

2.1 Introduction

This chapter introduces the basic concepts related to this thesis. It describes a healthcare scenario used as a reference case study for the rest of this document. Moreover, it covers an overview of Cloud computing, and the languages and techniques used in the context of this research work, such as XACML (policy specification language) and automata. Section 2.2 presents a healthcare scenario provided by a Moroccan emergency medical service based on a stroke accident, the section presents also formal notations of different medical components. The notations are going to be used in the rest of this document. The following sections present the languages and technologies used to define and develop the proposed approaches. Section 2.3 introduces XACML policy specification language. An overview of Finite state automata (briefly automata) is described in Section 2.4. Section 2.5 gives an overview of Cloud’s definitions, deployment models and solutions including OpenStack. In Section 2.6, we present the storage service of OpenStack denoted Swift. Section 2.7 summarizes this chapter.

2.2 Medical Environment: Scenario and Notation

Healthcare organizations provide several services to their patients: emergency services, day procedures, diagnostic services, therapy services, etc. To realize a service, an organization may have to produce documents (e.g. personal records, X-ray, brain scan, electroencephalography (EEG), etc). Those documents are typically stored in the organization data center. Data of an organization should be accessible by stakeholders of other organizations, so that they can collaborate in elaborating diagnoses. However, the information sharing must be regulated in order to guarantee the integrity and confidentiality of the shared information. This leads to the necessity of having policies regulating the medical data sharing.

Hereafter, we consider a reference scenario of stroke accident presented by the Moroccan
emergency medical service of Rabat [89]. In this scenario, three kinds of medical organizations are involved: two hospitals ($H_1$ and $H_2$), one emergency medical service ($EMS$) and two university hospitals ($UH_1$ and $UH_2$). These organizations are involved in a collaborative session (presented by a sequence of accesses) in order to perform an effective diagnosis to the transferred patient. In fact, doctors located in the host hospital (the hospital where the patient has been transferred) can make use of the experiences of specialists located in other medical organizations.

Our proposed scenario is that of a patient that has an accident and is transferred to the nearest hospital, which we call host hospital and denote $H_1$. We assume that this patient had his electronic medical records (EMR) in another hospital $H_2$. Once the patient is in $H_1$, the generalist calls $EMS$ and a regulator receives the call and writes the patient’s information into the system. The regulator looks for the available specialist doctors: radiologists, cardiologists, neurologists, etc, in $H_2$, $UH_1$ and $UH_2$. Once he finds a list of available doctors, the regulator creates a collaborative session. In the meantime (before the collaboration starts), the doctors in $EMS$ try to understand the medical state of the patient by holding a phone call with the generalist in $H_1$.

During the collaborative session (whose aim is to elaborate a diagnosis), the specialist doctors may ask $H_1$ to provide them with some scans prepared by a local radiologist. The scans and a part of the electronic medical record of the patient may be shared with other doctors participating to the collaborative session. According to this scenario, employees of various medical organizations collaborate remotely in order to provide a diagnosis for a patient.

In the rest of the paper, we describe formally three essential concepts that are used in a collaborative session: subjects, objects, and organizations. Simply speaking: an organization represents a medical institution, such as a hospital or a medical emergency service; a subject represents an employee or a customer of an organization, such as a doctor, a nurse or a patient; an object represents recorded data owned by an organization, such as a scan or an electroencephalogram. A collaborative session is specified as a sequence of accesses, where each access is a triplet $(S, O, a)$ where $S$ (resp. $O$) is a set of subjects (resp. objects) and $a$ is an action. $(S, O, a)$ means that a subject of $S$ applies the action ‘$a$’ on an object of $O$.

Let $Org$ denote the set of organizations involved in the collaborative session.

### 2.2.1 Subjects: Human Resources

In this section, we propose how to specify formally subjects, which correspond to employees or customers of medical organizations. For the sake of simplicity, we consider only three categories of subjects: doctors, nurses and patients, which are sufficient for the purpose of our study.

- **Doctors:** For the sake of simplicity, we consider only the following categories of doctors: generalists, radiologists, regulators, and neurologists. We use the following notations:
Doctors is the set of doctors of all organizations, and \( \text{doctors}_x \) is the set of doctors of an organization \( x \in \text{Org} \). In the same way, we define the following sets of subjects: generalists, \( \text{generalists}_x \), radiologists, \( \text{radiologists}_x \), regulators, \( \text{regulators}_x \), neurologists, \( \text{neurologists}_x \), cardiologists, \( \text{cardiologists}_x \). We have therefore a partition by organization which is indicated as an index, and a partition by specialty. We have obviously the following formulas:

- Doctors = \( \bigcup_{x \in \text{Org}} \text{doctors}_x = \text{generalists} \cup \ldots \cup \text{cardiologists} \)
- generalists = \( \bigcup_{x \in \text{Org}} \text{generalists}_x \)
- radiologists = \( \bigcup_{x \in \text{Org}} \text{radiologists}_x \)
- regulators = \( \bigcup_{x \in \text{Org}} \text{regulators}_x \)
- neurologists = \( \bigcup_{x \in \text{Org}} \text{neurologists}_x \)
- cardiologists = \( \bigcup_{x \in \text{Org}} \text{cardiologists}_x \)

**Nurses:** Like doctors, nurses are partitioned by organizations and by specialties. The difference with doctors is that specialties are indicated by \( \text{nurses}^1, \text{nurses}^2, \ldots \), instead of being named specifically. \( \text{nurses}_x \) denotes the set of nurses of an organization \( x \in \text{Org} \), and \( \text{nurses}^i_x \) denotes the part of \( \text{nurses}_x \) that are of specialty \( i \). We have therefore the following formulas:

- Nurses = \( \bigcup_{x \in \text{Org}} \text{nurses}_x = \bigcup_{i=1}^\ldots \text{nurses}^i \)
- \( \text{nurses}_x = \bigcup_{i=1}^\ldots \text{nurses}^i_x \)
- \( \text{nurses}^i = \bigcup_{x \in \text{Org}} \text{nurses}^i_x \)

**Patients:** Like doctors and nurses, we use two partitions: by organization and by illness (instead of specialty). We use the same type of notations and we obtain the following formulas:

- Patients = \( \bigcup_{x \in \text{Org}} \text{patients}_x = \bigcup_{i=1}^\ldots \text{patients}^i \)
- \( \text{patients}_x = \bigcup_{i=1}^\ldots \text{patients}^i_x \)
- \( \text{patients}^i = \bigcup_{x \in \text{Org}} \text{patients}^i_x \)

If we consider only the three categories of doctors, nurses and patients, the set of all subjects can be formally defined as follows:

\[
\text{Subjects} = \text{Doctors} \cup \text{Nurses} \cup \text{Patients}.
\]
2.2.2 Objects: Stored Information

An object represents data recorded and owned by an organization. For the sake of simplicity, we consider here only the following categories: personal records (\(PR\)), scans, audios, list of available doctors, and history of the collaborative session’s discussion. As we have done for subjects, for each category of objects, we consider two partitions: by organization and by type (instead of specialty). We obtain therefore formulas like:

\[
\text{Objects} = \bigcup_{x \in \text{Org}} \text{objects}_x
\]

\[
\text{objects}_x = \text{PR}_x \cup \text{scans}_x \cup \text{audios}_x \cup \text{listDoctors}_x \cup \text{collSessDisc}_x
\]

Here are formulas for the partitions of scans, these formulas can be obviously adapted for the other categories: audios, etc.

- \(\text{scans} = \bigcup_{x \in \text{Org}} \text{scans}_x = \bigcup_{i=1}^{\ldots} \text{scans}^i\)
- \(\text{scans}_x = \bigcup_{i=1}^{\ldots} \text{scans}^i_x\)
- \(\text{scans}^i = \bigcup_{x \in \text{Org}} \text{scans}^i_x\)

Example of types of scans:

- \(\text{scans}^1 = \text{MRI} \) (Magnetic resonance imaging)
- \(\text{scans}^2 = \text{CAT} \) (Computed Axial Tomography)
- \(\text{scans}^3 = \text{EEG} \) (Electroencephalography)
- \(\text{scans}^4 = \text{MRA} \) (Magnetic Resonance Angiogram) etc.

Each object is identified by several attributes, such as:

- its type: e.g. scan
- its category: e.g. brain scan
- its owner (organization): e.g. university hospital \(\text{UH}_1\)
- the name of the scanned person

If we consider only the categories above-mentioned, the set of all objects can be formally defined as follows:

\[
\text{Objects} = \text{PR} \cup \text{scans} \cup \text{audios} \cup \text{listDoctors} \cup \text{collSessDisc}.\]

2.2.3 Medical Organizations

Each organization has:

i. Subjects: they are human resources and patients.
ii. Objects: they are physical and computer resources (hardware, software).

For the purpose of illustrating our study, we will consider only the following three categories of medical organizations that are present in most scenarios: hospitals, university hospitals, and emergency medical services. For the sake of simplicity, we will restrict their sets of subjects and objects as follows (Fig 2.1):

(a) Hospitals $H_1, H_2, H_3, \ldots$

The subjects of a hospital are: generalists, radiologists, and nurses. Formally

$$\text{subjects}_{H_i} = \text{doctors}_{H_i} \cup \text{nurses}_{H_i}$$

$$\text{doctors}_{H_i} = \text{generalists}_{H_i} \cup \text{radiologists}_{H_i}.$$ 

The objects of a hospital are: personal records of patients regularly followed by the hospital, or of patients transferred to the hospital in emergency cases; and scans consisting of MRI, MRA, CAT, and EEG. Formally:

$$\text{objects}_{H_i} = \text{PR}_{H_i} \cup \text{scans}_{H_i}$$

$$\text{scans}_{H_i} = \text{MRI}_{H_i} \cup \text{MRA}_{H_i} \cup \text{CAT}_{H_i} \cup \text{EEG}_{H_i}.$$ 

(b) University hospitals $UH_1, UH_2, UH_3, \ldots$

The subjects of a university hospital are specialist doctors (for our case, neurologists, cardiologists and radiologists), and nurses. Formally:

$$\text{subjects}_{UH_i} = \text{doctors}_{UH_i} \cup \text{nurses}_{UH_i},$$

$$\text{doctors}_{UH_i} = \text{neurologists}_{UH_i} \cup \text{radiologists}_{UH_i} \cup \text{cardiologists}_{UH_i},$$

The objects of a university hospital are the personal records of its patients. Formally:

$$\text{objects}_{UH_i} = \text{PR}_{UH_i}.$$
Emergency medical services $EMS_1$, $EMS_2$, …

The subjects of an emergency medical service are regulators and generalists. Formally:

$$ subjects_{EMS_i} = doctors_{EMS_i} = generalists_{EMS_i} \cup regulators_{EMS_i}. $$

The objects of an emergency medical service are personal records of the hosted people and who passed through the emergency case, audio records, list of the doctors of hospitals and university hospitals, and history of the collaborative session’s discussions. Formally:

$$ objects_{EMS_i} = PR_{EMS_i} \cup audios_{EMS_i} \cup collSessDisc_{EMS_i} \cup listDoctors_{EMS_i}. $$

Each medical organization is responsible of securing data it owns. Therefore, each medical organization imposes a set of access control rules forming a security policy. Security policies are usually expressed using a specific language. XACML (eXtensible Access Control Markup Language) is one of the most known policy specification language. Section 2.3 gives an overview of XACML.

2.3 Access Control Policies: XACML

XACML (eXtensible Access Control Markup Language) has been widely used as a policy specification language in both academia and industry. Its first version was released by Anderson et al. [9] in 2003 and used in the context of distributed systems [75]. Two years later, OASIS extended the old version and proposed XACML 3.0 [100]. XACML, actually, describes both a policy specification language and a request/response language. The specification language is used to express access control policies (who can access what and when), while the request/response language expresses the subjects queries to access a resource (request) and describes the answers to these queries.

2.3.1 XACML Policies specification

A security policy in general consists of a set of filtering rules [93], where each rule is specified as a triplet $(S, O, u_A)$, where $S$ is a set of subjects, $O$ is a set of objects, and $u_A$ is an access permission or prohibition. More precisely, in $u_A$ we have $u = p$ (for Permit) or $d$ (for Deny), and $A$ is a set of actions like read ($r$), write ($w$), create ($c$) and delete ($d$). $p_{r,w,c,d}$ is also noted $p_a$ and called admin permission, because it corresponds typically to the permission granted to an administrator. The meaning of $(S, O, u_A)$ is:

- if $u=p$, then any action in $A$ applied by a subject of $S$ to an object of $O$ is permitted;
- if $u=d$, then any action in $A$ applied by a subject of $S$ to an object of $O$ is forbidden.

It is worth noting the notational similarity between a filtering rule $(S, O, u_A)$ and an access $(s, o, a)$. The semantic difference is that an access indicates the effective application of an action
a, while a rule indicates the decision (permission/prohibition) of applying any action of $A$.

The general formal specification of security rules $(S, O, u_A)$ is used by developers to express XACML Policies. There are several XACML policy specification tools, but the most used tools are SUN XACML [90], HERAS [29], WSO2 [96] and ALFA [13]. SUN, HERAS and WSO2 permit the evaluation of XACML policies as well. The policy evaluation process consists in determining whether a given set of attributes, represented by a request, satisfies the policy. To evaluate a request, they parse the XML policies and then visit parts of the generated DOM (Document Object Model) trees for calculating the authorization decision. The DOM defines the logical structure of documents and the way a document is accessed and manipulated. ALFA (Axiomatics Language for Authorization) is an eclipse plugin developed by the company Axiomatics. It converts the eclipse programming IDE to a dedicated editor of authorization policies using ALFA syntax. ALFA policies can then easily be converted into real XACML 3.0 policies and loaded into XML files.

In this thesis, we used ALFA to describe policies from which we automatically build the corresponding XACML model. A policy modeled in XACML will be called XACML policy. The following ALFA example represents a XACML policy which contains a single rule. The rule has a condition which is used to compare 3 attributes (subject’s role, resource’s type and the action’s ID).

Listing 2.1: A simple ALFA specification of rule with a condition

```java
namespace hospital{
    policy Hospital_1{
        apply deny-overrides
        rule readPR{
            permit
            condition actionId == "read" and userRole == "generalist" and resourceType == "PR"
        }
    }
}
```

There are three main policy components in XACML, namely PolicySet, Policy and Rule. PolicySet is the sequence (combination) of two or more policies. Policy is the sequence (combination) of several rules. While a Rule is the most elementary unit of a policy. All XACML policies start with a root which can be either a PolicySet or a Policy. Rules include conditions on the values of attributes to determine which rule can be applied to each access request. Here are two simple examples of rules $h_1$ and $h_2$, but note that in general we can have more complicated rules with restrictions on time for example:

- $h_1$: Subjects having the role “generalist” can read the documents having category “Personal Records” (PR);
• $h_2$: Subjects having the role “radiologist” can read/write the documents having category “scans”.

Listing 2.2 describes the XACML representation of the first rule. In order to allow a succinct treatment of the important aspects of XACML, we omitted some of the rule’s components such as the Target and the Environment. We focus only on the core aspects of XACML corresponding to Subjects, Resources, Actions and Effects. The XACML specification of rule $h_1$ can be explained as follows:

• Line 1: represents a standard XML document tag indicating which XML version is being used.

• Lines 2-5: represent the Policy tag which delimits the start of the XACML Policy. It consists of XML namespace declaration (lines 2 to 3), policy identifier (line 4), rule combining algorithm identifier (line 4), and the XACML’s version (line 5).

• Line 6: Description of the policy.

• Line 7-40: The Rule element of this policy. It consists of Rule identifier and its effect (line 7). Rule’s description are presented in lines 8-10. Lines 11-39 describe the condition that defines the attributes of subjects, resources and actions which regulate the access. The condition contains several elements such as:

  – AttributeDesignator elements are used to identify one or more attribute values in the request context.

  – The Apply element represents the function that needs to be applied in case of multiple attributes. For instance, if we consider that the rules permits the read and the write actions to be performed. Using the function string-at-least-one-member-of in line 30 implies that the request’s action attribute needs to mach at least one of them (either read or write).
XACML 3.0 separates the Attribute category into sub-categories using AttributeId entity. To simplify the policy’s automata model described in chapter 4, we do not consider this separation. For instance, in XACML 3.0, a subject (e.g. doctor) designate a category and subject-role (e.g. radiologist) designate an AttributeId. We only model the attributeId, i.e., subject-roles (radiologists).
2.3.2 Abstract Syntax for XACML requests

A Request contains a set of attributes giving further information about the access query that should match the policy’s Target. A request can be formally specified by the triplet access\(= (s, o, a)\) where \(a\) represents the requested action to perform over the object \(o\) by the subject \(s\). XACML 3.0 supports multiple requests using the element MultiRequests. However, in this thesis, we omit MultiRequests entity since we only consider single requests (single accesses).

As an example, let us consider that “a generalist wants to create a personal record (PR)”. When the context handler receives a request, first of all it retrieves attributes related to the request: the subject’s role, resource’s type, etc. As an example, a Context Handler provides the following attributes: subject = generalist, resource = PR and action = create. Then, the context handler sends a request in XACML format to PDP (Policy Decision Point). The encoding of XACML 3.0 of this request is represented in Listing 2.3:

Listing 2.3: Example of a XACML Request

```xml
<? xml version ="1.0" encoding =" utf -8" ?>
<Request xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:
                     http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd"
ReturnPolicyIdList="false">
<Attributes Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
<Attribute IncludeInResult="false"
AttributeId="urn:oasis:names:tc:xacml:1.0:subject:role">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">generalist</AttributeValue>
</Attribute>
</Attributes>
<Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource">
<Attribute IncludeInResult="false"
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:type">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PR</AttributeValue>
</Attribute>
</Attributes>
<Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action">
<Attribute IncludeInResult="false"
AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">create</AttributeValue>
</Attribute>
</Attributes>
</Request>
```

The original syntax of XAML request uses some elements beside the attributes elements such as:

- The ReturnPolicyIdList entity which requests the PDP to return a list of all applicable Policy and PolicySet elements that were in the decision as a part of the decision response of multiple requests [92]. In the example above, the ReturnPolicyIdList equals false because
we consider only one request.

- The IncludeInResult entity is used to include the attribute in the result. The default value for this is false. This is useful to correlate requests with their responses in case of multiple requests [92].

The request’s attributes cannot be null (empty), because empty attributes mean there is no request. The PDP retrieves policies that match the request. If no policy matches, The PDP sends the NotApplicable response. Otherwise, it returns the Effect of the corresponding rule. For instance, the example presented in listing 2.3 does not match the policy example of listing 2.2. Therefore, the response is NotApplicable. The PDP sends a response in XML format as shown in Listing 2.4:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <Result>
    <Decision>NotApplicable</Decision>
  </Result>
</Response>
```

### 2.3.3 Example of XACML-based e-Health policies

In order to facilitate the representation of XACML policies, we represent them in the form of tables. Hereafter, we consider, as a XACML example, the three policies presented in Tables 2.1, 2.2 and 2.3. For instance, the policy of Table 2.1 consists of six rules regulating the access to different resources (objects). We consider the term “Objects” to specify all the objects of an organization. For the sake of simplicity, the term 'Others' means all the subjects of an organization, except those in the previous rules. For example, in Table 2.1, the term “Others” of $h_6$ can be represented otherwise as: $Subjects \ \{\text{generalist};\text{neurologist};\text{radiologist}\}$.

<table>
<thead>
<tr>
<th>RuleID</th>
<th>Effect</th>
<th>Subject</th>
<th>Resource</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>Permit</td>
<td>Generalist</td>
<td>PR</td>
<td>read</td>
</tr>
<tr>
<td>$h_2$</td>
<td>Permit</td>
<td>Generalist</td>
<td>Scans</td>
<td>read/write</td>
</tr>
<tr>
<td>$h_3$</td>
<td>Deny</td>
<td>Neurologist</td>
<td>Objects</td>
<td>read</td>
</tr>
<tr>
<td>$h_4$</td>
<td>Permit</td>
<td>Radiologist</td>
<td>PR</td>
<td>read/write/create/delete</td>
</tr>
<tr>
<td>$h_5$</td>
<td>Permit</td>
<td>Radiologist</td>
<td>Scans</td>
<td>read/write</td>
</tr>
<tr>
<td>$h_6$</td>
<td>Deny</td>
<td>Others</td>
<td>Objects</td>
<td>read/write/create/delete</td>
</tr>
</tbody>
</table>
The automata modeling these policies are presented in Chapter 4. These examples will be used later to illustrate the results obtained in chapters 5, 6 and 7.

Table 2.2: XACML policy of hospital \( H_2 \)

<table>
<thead>
<tr>
<th>RuleID</th>
<th>Effect</th>
<th>Subject</th>
<th>Resource</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>Permit</td>
<td>Generalist</td>
<td>PR</td>
<td>read/write</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>Deny</td>
<td>Generalist</td>
<td>Scans</td>
<td>read/write</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>Permit</td>
<td>Nurse</td>
<td>Objects</td>
<td>read</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>Deny</td>
<td>Neurologist</td>
<td>Scans</td>
<td>read/write</td>
</tr>
<tr>
<td>( r_5 )</td>
<td>Permit</td>
<td>Neurologist</td>
<td>PR</td>
<td>read/write/delete/create</td>
</tr>
<tr>
<td>( r_6 )</td>
<td>Deny</td>
<td>Others</td>
<td>Objects</td>
<td>read/write/delete/create</td>
</tr>
</tbody>
</table>

Table 2.3: XACML policy of hospital \( H \) containing anomalies

<table>
<thead>
<tr>
<th>RuleID</th>
<th>Effect</th>
<th>Subject</th>
<th>Resource</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>Permit</td>
<td>generalist</td>
<td>PR</td>
<td>read</td>
</tr>
<tr>
<td>( R_2 )</td>
<td>Permit</td>
<td>neurologist</td>
<td>EEG</td>
<td>read</td>
</tr>
<tr>
<td>( R_3 )</td>
<td>Permit</td>
<td>radiologist</td>
<td>Scans</td>
<td>write</td>
</tr>
<tr>
<td>( R_4 )</td>
<td>Deny</td>
<td>radiologist</td>
<td>Scans</td>
<td>write</td>
</tr>
<tr>
<td>( R_5 )</td>
<td>Deny</td>
<td>generalist</td>
<td>PR</td>
<td>read</td>
</tr>
<tr>
<td>( R_6 )</td>
<td>Permit</td>
<td>neurologist</td>
<td>EEG</td>
<td>read</td>
</tr>
<tr>
<td>( R_7 )</td>
<td>Permit</td>
<td>generalist</td>
<td>PR</td>
<td>read</td>
</tr>
</tbody>
</table>

Policies can be combined using \( PolicySet \) that specifies the combining algorithms in case where two security policies provoke contradictory actions, i.e. permit and deny, respectively.

XACML offers four combining algorithms:

- permit-override: If at least one policy is evaluated as "permit", the integrated output will also be "permit".

- deny-override: If at least one policy is evaluated as "deny", the integrated output will also be "deny".

- first applicable: The combined result is the same as the result of evaluating the first policy.

- only-one-applicable: The result is the one of the only applicable policy. If we have more than one policy, then the result is Not Applicable.

A XACML policy contains hundreds and thousands of rules, which makes it difficult to detect policy conflicts directly from the XML file. Yet, identifying conflicts in XACML policies is a primordial task for their designers. In fact, the choice of the combining algorithms relies essentially on the information from conflict diagnosis. The XACML policies may contain two kinds of conflicts: intra-policy (conflict between rules of the same policy) and inter-policies (conflict between rules of several policies defined under the \( PolicySet \)).

There are several formal approaches that represent XACML security policies such as Unified Modelling Language (UML), Answer Set Programming (ASP) and decision trees. In this work,
we formally represent XACML policies using Finite State automata. The formal definition and the mathematical expression of automata are detailed in Sect. 2.4.

2.4 Automata

Automata are abstract of machines that performs computations by describing a set of states or configurations. They can be used, for example, for pattern matching in text editors [5], for lexical analysis in compilers, for communication protocol specifications, for language recognition [24], and for firewall design analysis [60]. The word automaton itself, closely related to the word “automation”, denotes automatic processes carrying out the production of specific processes.

The main objective of automata is to develop methods that help human beings to describe the dynamic behavior of systems which are characterized by three main components: inputs, outputs and states. There are four types of automata namely: (a). finite state automata, (b). pushdown automata, (c). Linear-bounded automata, and (d). turning machine. In this thesis we use the first type of automata, i.e. finite state automata.

2.4.1 Basics of Finite State Automata

The first people to use finite state machines (or simply automata) consists of set of scientists, biologists, and computer scientists. In fact, the first definition of finite state machines (FSM) was presented by two neurologists Warren McCulloch and Walter Pitts [78] in 1943. They modeled the human neural network using FSM.

An automaton can be formally defined by \( A = (\Sigma, Q, q^0, Q_f, \delta) \) where \( \Sigma \) is a finite set of events (also called alphabet), \( Q \) is a finite set of states, \( q^0 \in Q \) is the initial state and \( Q_f \subseteq Q \) contains the final states of \( Q \). \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function, where \( \delta(q, \sigma) = r \) means that the execution of the event \( \sigma \) (or the reading of the term \( \sigma \) from state \( q \) leads to state \( r \). \( \delta(q, \sigma) = r \) can also be written as \( q \xrightarrow{\sigma} r \).

An automaton \( A \) is represented by a graph whose nodes and arcs are the states and the transitions of \( A \), respectively. There is one initial state (with a small incoming arrow) and one or more final states (double circled).

In this thesis, we use the notation \( S = \{\sigma_1, \sigma_2, \ldots, \sigma_p\} \) (it can be also denoted as \( S = \{\sigma_1\} \cup \{\sigma_2\} \cup \ldots \{\sigma_p\} \)) for a set of events. The notation \( q \xrightarrow{S} r \) means that if \( q \) is the current state, then every event \( \sigma_k \) from the set \( S \) leads to the state \( r \). The arc labeled \( S \), linking \( q \) and \( r \), is equivalent to many arcs labeled \( \sigma_1, \ldots, \sigma_p \), linking \( q \) and \( r \). Figure 2.2 presents an example of the automaton’s graph. the state \( \text{initial} \) represents the initial state, \( \text{final} \) depicts the final state, \( \text{error} \) represents a failure state. The labels \( a \) and \( b \) represent the automaton’s language.

A finite event sequence (more briefly, sequence) is accepted by \( A \) if it starts in the initial
state $q^0$ and terminates in one of the final states of $A$. The language of $A$, denoted $L_A$, is the set of sequences accepted by $A$.

2.4.2 Synchronous Composition of Automata

The rich theory of automata allows us to compose models of systems, behaviors, mechanisms due to the operations that can be performed over automata. For instance, the synchronous composition of two automata $A_1$ and $A_2$ over the alphabet $\Sigma$ is an automaton over the alphabet $\Sigma$ whose language is $L_{A_1} \cap L_{A_2}$. The resulting automata depicts the parallel execution of the two sub-systems represented by $A_1$ and $A_2$. Each of its states is the combination of states of $A_1$ and $A_2$. Its initial state is the combination of the initial states of the composed automata.

2.5 Cloud Computing: Openstack

The concept of Cloud computing is not totally new. In fact, back in 1961, computing pioneer John McCarthy predicted that “computation may someday be organized as a public utility”; that means that computing power and resources can be obtained as utility similar to electricity. Users can simply request information and computation and have them delivered to him without the necessity to know where are the data stored and processed [101]. Grid computing is defined as “a hardware and software infrastructure that provides dependable consistent, pervasive, and inexpensive access to high-end computational capabilities”.

Cloud Computing is the convergence of Grid Computing and Utility Computing [101]. It represents the external deployment of IT resources, such as computational power or storage, and obtaining them as services [23]. Cloud computing is a model for enabling convenient, on-demand network access, to a shared pool of configurable computing resources, (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [79]. The main idea of Cloud computing is to offer everything as a service (XaaS). The most known Cloud services are [79]:

- **Infrastructure as a Service (IaaS):** offers virtualized infrastructures (i.e., virtual machines) as a service. The most known IaaS are: Amazon Web Services Elastic Compute Cloud
Platform as a Service (PaaS): is an application development and deployment platform delivered as a service to developers over the Web. It provides all the facilities required to support the complete life cycle of building and delivering web applications and services. As one of the most known PaaS examples, we can cite Google Apps Engine [111].

Software as a Service (SaaS): offers the users a hosted set of software (running on a platform and infrastructure) that he does not own but pay for some element of utilization. A SaaS provider typically hosts and manages a given application in their own data center and makes it available to multiple tenants (users) over the Web. Salesforce [94], and oracle Netsuite [86] are some of the well known SaaS examples.

As shown in Fig. 2.3, there are four deployment models of Cloud computing: private Cloud, community Cloud, public Cloud and hybrid Cloud. They are described by the National Institute of Standards and Technology (NIST) as follows [79]:

- **Public Cloud**: The Cloud infrastructure is made available to the general public or a large industry group and is owned by one or more organizations selling Cloud services.

- **Private Cloud**: The Cloud infrastructure is operated for a private organization. It may be managed by the organization or a third party.

- **Community Cloud**: The Cloud infrastructure is shared by several organizations and supports a specific community that has communal concerns (e.g., collaborative diagnosis, conference). It may be managed by the organizations or a third party.
• Hybrid Cloud: consists of several (two or more) Cloud environments (private, community, or public) that share standardized technology to enable the portability of applications and data.

Due to the great advance of Cloud computing technology, there are several solutions in this area. These solutions are clustered into two main categories: commercial and open source platforms. For instance, Amazon EC2 is a commercial IaaS, while OpenStack is an open source IaaS. More than 500 enterprises have entered the business of Cloud providers: Amazon and their Elastic Compute Cloud (EC2), Google’s App Engine, Microsoft’s Azure Platform, and Salesforce among many others. Similarly, The Open Source Community has also been very active in the Cloud computing. They contributed also in related areas such as system and network virtualization.

The most prominent examples of open source Cloud computing platforms are OpenStack, Eucalyptus, OpenNebula, Xen Cloud Platform, Apache VCL, TPlatform and CloudStack. Several works provide comparisons and discussions related to the merits and issues of these open source Clouds. For instance, Endo et al. [45] present and discuss the state-of-the-art of open-source solutions for Cloud computing. Their comparison is based on the services and infrastructures provided by the Cloud solutions. Cordeiro et al. [30] also provide a comparison between XCP, Eucalyptus and OpenNebula. Their discussion proves that each solution may be more appropriate for its use.

Openstack [85] is an opensource Cloud IaaS. It was first introduced in June 2010, born with its initial code from “Rackspace Hosting” and “NASA”. OpenStack has released fifteen versions: from Austin (October 2010) to Ocata (February 2017). Pike is the last version of OpenStack but it is not released yet. OpenStack is an open-source software for building private and public Clouds. It is currently organized around five main projects i.e. compute (Nova), object storage (swift), image service (glance), block storage (cinder), network (Quantum). Along with these projects, dashboard (horizon) becomes an important component in providing interface to administrators and users for provisioning and release of resources. Keystone is also an identity management project that was integrated in OpenStack. The interaction between these components is presented in Fig. 2.4.

In this research work, we use OpenStack because of several reasons:

• It is open-source and growing dramatically.

• it is vendor neutral i.e. the user can add middlewares and features in the original kernel.

• It has several limitations in terms of security. For instance, in the existing authorization system, it is impossible to specify permissions at the object level, that is why customers of OpenStack Object Storage Swift are limited to using container based permissions [98]. For
example if a user wants to share an object with other users, he has two solutions: either to change the permissions of the whole container, or to create new container containing all the objects to share.

Since object storage Swift is used during the course of this thesis, it is detailed in the next section.

### 2.6 Object Storage System: Swift OpenStack

Swift was a thoughtful and creative response of the unprecedented and precipitous growth in data. In 2009 a group of developers of Rackspace including John Dickinson, started developing what would become Swift. Swift is an object storage system that differs from traditional storage systems which are either block or file systems. In fact Swift has the ability to scale to an extremely large number of concurrent connections and extremely large sets of data.

Nowadays, the demands on storage systems are increasing exponentially due to the era of connected devices and Internet of Things (IoT). To understand the scale of this growth in data, consider that in 2012 the International Data Corporation (IDC) declared that the volume of digital content worldwide exceeded 2,765 exabytes. This was an increase of almost 2,048 exabytes in 2 short years. And stored data is continuing to grow at ever faster rates, by 2020 the IDC estimates that the amount of data in the world will reach 3,5840 exabytes. Divided by the world’s population, that is 4 TB per person [10].

The majority of stored data are “unstructured”, this means that the data does not have a predefined data model. Much of this unstructured data is the ever-proliferating images, videos,
emails, documents, and files of all types. However, different types of data have different access patterns and therefore can be best stored on different types of storage systems. There are three broad categories of data storage: block storage, file storage, and object storage [11].

- **Block storage**: stores structured data, which is represented as equal-size blocks (say, $2^{12}$ bits per block) without putting any interpretation on the bits. A common use for block storage is databases, which can use a raw block device to efficiently read and write structured data.

- **File storage**: takes a hard drive and exposes a filesystem on it for storing unstructured data. Although file storage provides a useful abstraction on top of a storage device, there are challenges as the system scales. File storage needs strong consistency, which creates constraints as the system grows and is put under high demand.

- **Object storage** provides access to whole objects or blobs of data through an API specific to that system. Objects are accessible via URLs using HTTP protocols. Object storage abstracts these locations as URLs so that the storage system can grow and scale independently from the underlying storage mechanisms. This makes object storage ideal for systems that need to grow and scale for capacity, concurrency, or both.

Swift is a multi-tenant, highly scalable, and durable object storage system designed to store large amounts of unstructured data at low cost [11]. It is open-sourced as part of the OpenStack project. Swift is designed to scale linearly based on how much data needs to be stored. To scale up, the system grows by adding storage nodes to increase storage capacity, and adding proxy nodes as requests increase. Moreover, the distributed architecture of swift ensures the availability and durability of data. Indeed, swift creates copies of data and stores them in different clusters and regions to which ensure disaster recovery sites. Swift installations can run robustly on commodity hardware (low-cost server components), and even on regular desktop drives rather than more expensive enterprise drives.

OpenStack Swift allows users to store their data objects with a canonical name containing three parts: account, container, and object. The combination of one or more of these parts allows the system to form a unique storage location for data.

- **Accounts** are the root storage location for data. Sometimes compared to filesystem volume, accounts are uniquely named spaces in the system. Each account has a database that stores both the metadata (descriptive information) for the account and a listing of all containers within the account. Accounts are designed for both single-user access and multi-users access. The multiuser access can be granted in case of projects where many team members collaborate with each other and require access to the account.

- **Containers** are user-defined segments of the account namespace that provide the storage
location where objects are found. They are conceptually similar to directories or folders in a filesystem. Each container has a database to store metadata and a record for each object that it contains. Example of container’s metadata: X-Container-Read which provides the authorization list for who can read the container.

- **Objects** records objects it contains. They could be photos, videos, documents, backups, or any kind of data. Similarly to accounts and containers, objects has a database to store metadata as well. Object’s metadata provides important information about the object. For instance, a genome sequence could contain information about the process by which it was generated.

Every object must belong to a container which belongs as well to an account. When an object is stored in the cluster, users will always reference it by the object storage location (/account/container/object). There is no limit to the number of objects that a user may store within a container. Moreover, there is no limit of containers that a user may create within an account. Since the parts are joined together to make up the locations, the container and object names do not need to be unique within the cluster.

The Cloud client (user) communicates with swift and keystone using the curl (client url Request) library [102]. The user sends a request to keystone for authentication. The request consists of a user name, an account name and a password; e.g. `curl -v -H 'X-Auth-User:generalist account: hospital1' -H 'X-Auth-Key: password' http://localhost/auth/v1.0/`. If the credentials are correct, Swift service returns an http response with an authentication token and storage url.

```
HTTP/1.1 200 OK
X-Storage-Url: https://swift/v1/AUTH-account
X-Auth-Token: Auth-tk60217960bccf44b7d6b3a98c77ce8d
content-Length: 0
```

The authentication token and the storage url are included in subsequent requests to the Swift cluster. By default, Swift rejects unauthenticated requests with HTTP 401 unauthorized. Authentication tokens are valid for 24 hours, and can be used in each request until it expires.

After the authentication, the user can access the Swift storage service. Using the storage url returned by Swift, the user can construct his URLs. The authentication token has to be passed as a request header. In order to get status information from the cluster, the user may use the HTTP HEAD method. The information contain statistics for the entire cluster as follows:

```
HEAD /v1/Auth-account HTTP/1.1
X-Auth-Token: Auth-tk60217960bccf44b7d6b3a98c77ce8d
```

Swift responds:
HTTP/1.1 No Content
content-Length: 0
X-Account-Container-Count: 2
X-Account-Object-Count: 2
X-Account-Bytes-Used: 2048

This response provides information about how many containers and objects are in the cluster and how many bytes of storage have been used. To list the containers in the account, the user makes a GET request to the storage URL as follows:

GET /v1/Auth-account HTTP/1.1
X-Auth-Token: Auth-tk60217960bcecf4e44b7d6baa98c77ce8d

To create a container, the user uses the PUT HTTP method, and attaches the desired container name to the storage URL. The PUT request is used to upload objects as well. The only difference in uploading objects is that we append the desired object name to the container’s resource URL and the object content in the request body:

PUT /v1/Auth-account/animals/bears.txt HTTP/1.1

The Swift HTTP API allows the users to delete object using DELETE method. It supports the pagination and result filtering as well.

Once swift receives the request, it translates it into the storage system using two basic components. First, the proxy service which routes the incoming requests to read or store data to the appropriate node. next the storage service which stores the actual data. In order to map the partitions to physical locations on disk, swift uses a data structure called the ring. when other components need to perform any operation on a an object, container or account, they need to interact with the ring to determine its location in the cluster.

In addition to the basic CRUD (create, read, update and delete) operations, swift offers the users the possibility to add access control lists (ACLs). In some cases (such as collaborative eHealth) the Cloud clients need to share some data with some unauthorized users. Therefore, swift adds a meta-data represented by X-container-Read or X-Container-Write request headers to the curl syntax to specify the read acl and the write acl respectively.

2.7 Summary

In this chapter, we have described a healthcare scenario used as a reference case study for the rest of this document. Furthermore, we have introduced basic concepts related to this thesis, such as XACML (the policy specification language), and finite state machines (automata). The chapter gives an overview of the Cloud Computing and its open-source solutions such as
openstack. In addition, we have presented the storage service of openstack denoted Swift. Swift is the basic component used in the course of this thesis. The next chapter presents and discusses related work.
Chapter 3

State of the Art

3.1 Introduction

Access control ensures that only authorized subjects can access the requested resources. Hence, access control protects systems against unauthorized disclosure. Access control policy defines the high-level rules specifying the conditions under which subjects are permitted to access targets [36]. Policy enforcement is the mechanism that ensures that a policy is applied correctly. Enforcement can be as simple as denying an unauthorized person to access a file, or as complex as the execution of the appropriate combining algorithm in collaborative applications. Security policy languages support the expression and the enforcement of policies, and some of them support combination of multiple security policies. Such combination may lead to conflicting and redundant rules. Therefore, the implementation of access control in terms of low-level mechanism such as ACLs is not sufficient. There is a need to an access control model to bridge the existing abstraction gap between the policy implementation and the policy specification [26]. Moreover, access control models provide a formal representation of security policies which allows the proof of properties about an access control system.

In this chapter, we give an overview of related works on access control policy specification and verification. Section 3.2 presents some policy specification languages. In section 3.3, we present the existing access control models in the literature. In this thesis, we use XACML (eXtensible Access Control Markup) as policy specification Language. Hence, an overview of the XACML evaluation tools is provided in Section 3.4. Sections 3.5 and 3.6 give an overview of existing solutions to detect and resolve security policy anomalies. Finally, section 3.7 summarizes this chapter.
3.2 Policy Specification Languages

Policy specification languages are domain-specific programming languages intended to facilitate the expression and the enforcement of security policies on untrusted software. Many of these languages are used for expressing generic assertions about subjects \[52\] (i.e. principles or users) such as the association of a user with a role, the membership of a user in a group, or the right of a user to perform a certain operation at a specified time \[36\]. Some languages aim to support also trust management. For instance, Ponder \[33\] is a declarative, object-oriented language that can be used to specify both security and management policies. Ponder authorisation policies can be implemented using various access control mechanisms for firewalls, operating systems, databases and Java \[31\].

Several of the most recent language designs rely on concepts from logic programming. For instance, Binder \[35\] is an open logic-based security language that encodes security statements as components of communicating distributed logic programs. In the same direction, Li et al. developed different logic-based languages denoted D1LP \[70\], D2LP \[71\] and RT \[72\]. These languages provide a logical framework for studying delegation, negation of authority, conflicts between authorities, and their interplay. Even though logic-based specifications have many benefits, they may be complicated or even intimidating to some users. Yet, they can be difficult to use and are not always directly translatable into efficient implementations. Indeed, security administrators and end users need simple languages that allow them easily to understand the system behavior and maintain control over security specifications.

Recently, many applications either use XML as their data model, or export relational data as XML data. Hence, it becomes primordial to investigate the problem of access control for XML. In this regard, many XML-based access control languages have been proposed \[19, 32, 65, 22\]. The most relevant of these languages is the eXtensible Access Control Markup Language (XACML) which contains both an access control policy language and a request/response language. Another XML-based language that supports authorization decision request is Enterprise Privacy Authorization Language (EPAL). While EPAL and XACML are very similar, and share a common authorization model, they differ in important ways \[8\]. For instance, XACML can combine results of multiple policies developed by potentially independent policy issuers via policySet, whereas EPAL does not support nested policies. Moreover, XACML supports conditions requiring principals to be in multiple, independent hierarchical roles or groups. Therefore, Anderson concluded, from the comparison of both languages \[8\], that XACML is the most promising language that can answer the needs of scalable and distributed systems such as cloud computing \[54\].

Policy specification languages are essential but they are just a way to represent the policies and rules. Therefore, it is important to have an access control model to express all security
requirements. Next section gives a brief overview of existing access control models and emphasis
on the most used model in XACML language.

3.3 Access Control Models

The most ultimate goal is to express access control models using existing policy specification
languages. Access control models can be implemented in many places and at different levels.
for instance, operating systems implement access control models to regulate accesses to files
and folders, while cloud computing uses access control models to ensure that some services are
only available to certain users. An access control model is an abstract mathematical description
of an authorization syntax that defines relationships among permissions, actions, objects, and
subjects. We distinguish the difference between users, the people who use the resources, and
subjects, processes that act on behalf of the users. Most of the proposed access control models
in the literature organize this relationship. In this section, we present the most known access
control models: discretionary access control (DAC) [57], mandatory access control (MAC) [106],
Task based access control (TBAC) [105], organization based access control model (OrBAC)
[42], attribute based access control (ABAC) [109], and role-based access control model (RBAC)
[95].

- **Discretionary Access Control (DAC):** usually requires each resource to have an owner,
and only the owner has the discretion over who can access the object. In other words, an
individual user is able to decree who is allowed to have access to the resources he owns. In
DAC the requests are evaluated based on the identity of the requesting user or the group to
which he belongs. Therefore, dictionary access control is also called *Identity-based access
control.*

DAC is based on the concept of an access matrix proposed by Lampson [68]. An access
matrix $M$, also known as the protection matrix, has rows indexed by users and columns
indexed by objects. Each combination of a user and an object defines the set of allowed
actions. For instance, user Bob (more precisely processes invoked by user Bob) is allowed
to write and read both scans and medical records objects and user Alice is denied to read
the prescriptions objects. The access matrix can be read either: (a) by rows to represent
capabilities lists that defines what is allowed for each user, or (b) by columns to represent
the access control list (ACLs) that defines the permissions associated to each object.

- **Mandatory Access Control (MAC):** In most organizations, users do not usually have
the ownership of the data they are granted access to. In fact, the organization is the only
owner of data, and the users are not allowed to define their own permissions. Hence, DAC
is not useful for these kind of situations. To overcome this limitation, MAC has been
developed. MAC was initially designed to handle the classified documents in computer science (such as military ones). MAC affects subjects and objects to levels of security called clearance: unclassified, confidential, secret, and top-secret. The higher the level is, the more confidential data are. The security levels represent the hierarchical order in an organization. Therefore, users are allowed to write into higher level in order to transmit the data within their hierarchy. Whereas, the users can only read data (documents) of their security level and lower.

The increasing number of users and the exponential growth of data (big data) can not be modeled by the subject object view of access control (DAC or MAC). Therefore, new access control models try to overcome this limitation by regrouping the subjects and objects. For instance, TBAC regroups the subjects into a set of tasks, OrBAC regroups the subjects into roles, regroups the objects into views and regroups the actions into activities.

- **Task Based access control** (TBAC): a task can be defined as a unit in a work-flow that correlates a set of subjects into tasks. TBAC assigns permissions to tasks and users can use these permissions only during the execution of the task.

- **Organization Based Access Control** (OrBAC): in OrBAC a security policy is defined as a collection of permissions, prohibitions, obligations and recommendations. This access model is centered on the concept of *organization*, and all the concepts used to define a security policy in this model depend on a given organization. OrBAC proposed to regroup a set of subjects under the concept of *roles* they play in their organization. The objects of an organization are regrouped into *views*, and the actions are regrouped into the concept of *activities*.

- **Attribute Based Access Control** (ABAC): uses attributes as building blocks to define a security policy and the access request. Usually, attributes are labels or properties that describe an entry in the rules’ components: subjects, resources, actions and environments. Each attribute consists of a key-value pair, for instance, “role=doctor”.

- **Role Based Access Control** (RBAC): the permissions are assigned to roles and each user can be assigned to the role via sessions in order to get the permission.

The most frequently used and implemented model in XACML is RBAC via a *profile*. A profile in general specifies two things:

1. **Attributes that should be used for specific information**: this is useful in case of cross-organizational structures, where several organizations need to communicate and share their data. For instance, consider an organization *A* which identifies the customers
with their user-name, while organization B identifies its customers with their login-ID. Profiles can unify identifiers for each attribute.

2. **The structure that the policy should take:** it can be useful for higher-level management of security policy. It’s often necessary to extract information from the policy and present it to the user, and limiting the structure in a profile can be used as document that specifies how the policy can be represented.

The RBAC profile of XACML presents a way to use the XACML access language within the RBAC model [7]. XACML does not support directly RBAC model. However, the proposed profile maps the XACML attributes to the main components of RBAC (roles, users, objects, permissions, sessions, and operations) as follows:

- **Users** are implemented using XACML Subjects.
- **Objects** are expressed by XACML Resources.
- **Operations** are defined by XACML Actions
- **Roles** are expressed using one or more XACML Subject Attributes.
- **Permissions** are expressed using XACML Role (PolicySet) and Permission (PolicySet). Where the latter contains the actual permissions associated with a given role. It contains (Policy) elements and (Rules) that describe the resources and actions that subjects are permitted to access. Whereas Role (PolicySet) limits the applicability of (PolicySet) to only subjects holding a given role attribute.

Later, Haidar et al. [2] extended the RBAC profile of XACML by adding more functionalities in order to respond to more advanced access control requirements such as user-user delegation, access elements abstractions and contextual applicability of the policies.

XACML implements the ABAC which makes it flexible and used to describe general access control requirements in the information system. However, XACML is a large and complex language and error prone. Hence, the formal verification and analysis of XACML policies remain difficult. The next sections present and discuss existing policy evaluation tools and different approaches for detecting and resolving anomalies within XACML policies.

### 3.4 Security Policy Evaluation Systems

The adoption of XACML as the standard for specifying access control policies for various applications, especially cloud computing services, is vastly increasing. This highlights the need for high performance XACML policy evaluation engines. Several research works proposed
different approaches and frameworks to improve the policy evaluation time in XACML. Prior work in optimizing the policy evaluation procedure in XACML can be split into three groups: using an adaptive reordering approach [76], using a decision diagrams approach [74, 88, 87] and using mathematical expressions [82].

Liu et al. [74] proposed XEngine which is a scheme to improve XACML policies evaluation using two techniques: numericalization and normalization. The first technique converts a textual XACML policy to a numerical policy. From the implementation perspectives, numericalization is a hash function that converts every attribute type into integers stored in a hash table with the objective of an efficient comparison. The second technique converts the output from the first technique to a hierarchical structure. Finally, the approach represents the normalized numerical policy using tree structures for efficient processing of requests. Their approach improves the performance of the XACML Sun PDP (Policy Decision Point) by three to four order of magnitude. The preprocessing time of XEngine includes the time for normalizing the policy and the time for building the internal data structure. This time can neither be reduced nor eliminated, because the preprocessing time is required for each access request.

In the same direction as XEngine, Ngo et al. [87] presented mechanisms to transform XACML policies into decision tree diagrams. They reduced the time by aggregating data interval partitions and by adopting a new decision diagram combinations. The authors proved the completeness and correctness of their proposed approach. However it is only experimented on small scale policies up to 360 rules. Therefore, their approach remains limited and cannot be applied in flexible and scalable systems such as Cloud environments.

Mourad and Jebbaoui [82] proposed SBA-XACML, a scheme to evaluate XACML policies. SBA-XACML is a set-based language composed of all the elements and constructs needed for the specification of XACML based policies. It is a mathematical intermediate representation of policies based on set theory. SBA-XACML ameliorates the techniques of XEngine, but it is scalable only for evaluating requests in a single independent organization. Therefore, it cannot be applied for collaborative frameworks. The experimental results explore that SBA-XACML evaluation of large and small sizes policies offers a better performance than XEngine and Sun PDP by a factor ranging between 2.4 and 15 times faster depending on policy size.

Marouf et al. [76] proposed a clustering technique which regroups policies and rules based on subjects. They also provide a re-ordering technique based on statistical analysis of previous requests. These processes reduce the policy evaluation time, because the most requested rules and policies are given priority to be evaluated first. Although this approach seems interesting, it eliminates some rules and policies that can be useful in case of emergencies. The evaluation in this case takes more time and may slow down the quick intervention.

The proposed approaches reduced the policy evaluation time of the XACML policies
compared to Sun PDP. However, most of them share the same limitations in terms of evaluation of policies related to collaborative and scalable environments such as Cloud computing. Our approach differs from the aforementioned ones in different aspects. First, the pre-processing step of our approach (composition of automata presented in Chapter 4) is required only if the policy changes. Therefore, this step is optional and can be ignored during the request/policy evaluation process which reduces the evaluation time. Second, it supports the representation of multiple XACML policies imposed by different collaborating organizations. In fact, the automata-based approach allows us to represent policies of different organizations. Their collaboration is modeled by the composition of automata (Chapter 4). Third, our experiments in Sect. 5.3 show that our approach outperforms the current approaches.

Next section gives an overview of existing approaches that detect anomalies within the XACML policies.

3.5 Security Policy Anomalies Detection

Access control protects the system's resources against unauthorized access via a set of policies. Jansen [54] proposed XACML as a policy specification language for cloud applications. Yet, XACML policies may contain conflicting and redundant rules, since XACML policies are sometimes managed by more than one administrator [51]. Moreover, in the collaborative applications, the XACML policies are aggregated from collaborative parties which may raise conflicts between rules in different policies.

Several works make use of verification techniques such as model checking in order to detect XACML policy anomalies. For instance, to detect conflicts between rules in a given policy, Martin et al. [99] encode the rules in Coq [21], a tool built specifically for formal theorem proving. A rule is a Coq record with two fields: the first field has the effect type, and the second field contains srac type which combines the four elements of XACML: subject-resource-action-condition. In order to compare the elements of type srac independently, the authors split them into a defined normal form \( DNF \). If two rules have overlap (srac types are identical) with different effects, the rules are then in conflict. Otherwise, if the effects of two overlapping rules are similar, then the rules are redundant. However, using Coq does not allow the dynamic anomaly detection. On the other hands, Mourad et al. [83] use the Unified Modelling Language (UML) to detect conflicting and redundant rules prior to their enforcement in the system. However, this technique does not allow the completeness verification; i.e. each access is either accepted or denied.

Regarding inter-policies conflict detection, Ramli [91] uses Answer Set Programming (ASP) in order to detect incompleteness, conflicting and unreachable XACML policies. As a limitation
of this approach, it is difficult to model XACML expressions dealing with types of attributes that
do not belong to AnsProlog [103], such as strings. Huonder [53] proposes another approach to
detect and resolve conflicts in XACML policies based on mapping each target to n-dimensional
space and overlapping the policies with different effects. The intersection of all dimensions
defines an inter-policies conflict. Yet, this technique cannot verify the policy’s completeness
property.

Besides verification-based techniques, many research efforts consider representing XACML
policies as decision trees to detect and resolve conflicts. In this direction, Hu et al.[51] make
use of Binary Decision Diagram (BDD). In this work, each XACML attribute is encoded into
an atomic boolean expression. The rules are then functions of these expressions. Fisler et
al. [46] suggested an extended version of BDD called Multi-Terminal Binary Decision Diagram
(MTBDD). Also, Gouglidis et al. [48] transform the XACML policies into Computation Tree
Logic (CTL). These tree-based approaches have a main drawback, the state explosion in the
decision trees.

A comparison of the approach proposed in this paper with the seven existing methods is
presented in Table 3.1. We have adopted the metrics proposed by Li et al. [69]:

1. **Completeness detection**: i.e. the capacity to verify if that any access request has a
response by the access control policy: permit or deny.

2. **Detection of policy anomalies**: the anomalies can be divided into two categories,
namely intra-policy anomalies (redundancy, ambiguity and conflicting rules) and
inter-policies anomalies (inconsistency and similarity):
   - **Redundancy**: the existence of two rules that have the same effect (permit or deny)
   and such that one of the two rules can be removed without changing the result of the
   policy.
   - **Ambiguity**: the existence of two rules that have the same effect (permit or deny) but
   with different set of actions (read, write, create, and delete).
   - **Conflicting rules**: two rules are in conflict in the same security policy.
   - **Inconsistency**: the existence of two or more rules in different policies that are in
   conflict.
   - **Policy similarity**: two policies can be equivalent and represented differently.

Table 3.1 underlines our contributions compared to existing works. The proposed approach
uses automata to describe formally the XACML policies. By using only the formalism of
automata, we have been able to detect several XACML anomalies (intra and inter policies
conflicts and redundancies) and verify the completeness of XACML policies. In addition, the
Table 3.1: The Capabilities of the Proposed Access Control Verification Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Technique</th>
<th>Completeness detection</th>
<th>Redundancy</th>
<th>Ambiguity</th>
<th>Conflicting rules</th>
<th>Inconsistency</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>UML</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ASP</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>n-dimension</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BDD</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MTBDD</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CTL</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Our Proposed method</td>
<td>Automata</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
approach has the ability to detect conflicting rules at runtime. In fact, the proposed approach
tmodels each security policy with an automaton. To verify if a new rule raises conflicts with
the existing ones, the proposed approach consists in composing the rule’s automaton and the
policy’ automaton by applying a so-called synchronous composition operator (Chapter 6). The
anomalies detection process is then applied to the resulted automaton. Hence, there is no need
to integrate the new rule into the policy to do the verification.

The obtained result of anomaly detection is an input of the resolution approach proposed in
Chapter 7. Next section presents a state-of-the-art of existing approaches to resolve anomalies
within security policies.

3.6 Security Policy Anomalies Resolution

Since the conflicts in XACML policies always exist and are hard to eliminate [50],
XACML defines four different policy combining algorithms (PCAs) to resolve conflicts [9]:
Deny-Overrides, Permit-Overrides, First Applicable and Only-One-Applicable. The first two
algorithms are trivial. For instance, permit-overrides returns permit if one of the conflicting
policies evaluates to permit. Otherwise, the result is deny. First Applicable returns the result
of the first policy that matches the request. Only-One Applicable obligates the system to have
one and only one policy that matches the request, otherwise the system returns Not-applicable.

Some XACML policy evaluation engines, such as Sun PDP [90] and XEngine [74], have been
developed to handle the request/policy evaluation. These implementations can detect conflicts
by checking if a request matches multiple rules with different effects. The conflicts are then
resolved by one predefined XACML’s PCA which allows the policy administrator to apply one
strategy for all the identified conflicts. In contrast, XAnalyzer [50] provides policy analysis at
policy design time and gives the designer the ability to choose the adequate PCA for each specific
policy. However, these systems are based on the intervention of the designer or the administrator
to set up the adequate combining algorithm. Yet, these techniques are not suitable for dynamic
environments such as e-health, for example, that necessitate the policy to change depending on
a set of parameters: context, patient’s state, etc.

Policy conflict resolution strategies are based on categories [73]: priority-based, context-based,
taking a simple majority vote, and taking a weighted majority vote. Each category may
present itself either statically or dynamically. For instance, Wang et al. [108] define three
static conflict resolution strategies depending on their types. They proposed rule priorities
to resolve modality conflicts, the principle of “specific-take-precedence” for redundancies, and
policy priority assignment for inconsistencies. According to Wang et al., a rule is redundant if
a permission or a prohibition having the same access authorisation can be derived from a set of
rules with higher priorities. Hence, redundant rules are never applied. The principle of specific take precedence gives a higher priority to the detected redundant rules.

On the other hand, other authors [56, 77, 81] deal with the dynamic detection and resolution of conflicts. For instance, Kagal et al. [56] have considered the low priority technique to resolve the conflict, i.e., negative authorizations are allowed. Matteucci et al. [77] have proposed a strategy for policy conflict resolution based on multi-criteria decisions. The decision is taken based on some calculations of multiple criteria retrieved from the policies’ attributes and represented in a matrix. However, in case of contradictory policies that exploit exactly the same set of attributes, the multi-criteria strategy cannot be applied. In fact, the policies do not give meaningful information on which policy should be enforced. Another research to resolve conflicts was provided by Mohan et al. [81] where they select the adequate PCA based on a set of environmental attributes. Further, in case of emergencies, they choose to switch from "deny-overrides" to "permit-overrides".

With respect to the solutions proposed in the previously mentioned papers, our approach aims at defining a generic strategy for conflict resolution. Our approach consider three types of anomalies: conflicting rules, redundant rules and ambiguous rules. Based on these categorization, we adopt a specific resolution strategy (Chapter 7). For instance, if a conflicting rules are detected, then this anomaly is resolved by using context-based approach to choose the adequate PCA. The redundancies are resolved by removing one of the rules. As for the ambiguity, we adopt the rule prioritizing technique. The proposed resolution approach uses the Aspect Oriented Finite State Machine (AO-FSM) model to intercept, prevent, and dynamically manipulate rules that cause conflicts (Chapter 7).

3.7 Chapter Summary

In this chapter, we have presented the existing policy specification languages that can be divided into three categories: object-oriented: Ponder; logic-based: Rei, Binder; D1LP, D2LP, and RT; and XML-based: XACML, and EPAL. XACML has many advantages compared to existing specification languages. In fact, it supports policy combination, and it contains policy evaluation for multiple request. Therefore, XACML is the most appropriate solution for distributed collaborative environments such as Cloud computing. This chapter covered different XACML policy evaluation tools. Furthermore, the chapter discussed existing approaches to detect security policies anomalies. Finally the chapter presented existing solutions to resolve these anomalies. The next chapter describes in detail the XACML-based policies and requests, and provides some e-health policy examples.
Chapter 4

Automata-based modeling of XACML policies

4.1 Introduction

XACML has become a standard for specifying security policies for distributed systems. However, due to the exponential growth of data, XACML policies are growing rapidly in size and complexity. Hence, processing and analyzing such policies is becoming difficult. Therefore, representing XACML policies differently may facilitate these tasks. Several research works have proposed different XACML policies representations (Chapter 3). In this thesis, we present a new representation of XACML policies using Finite State Machines.

This chapter describes a synthesis procedure that receives an XACML policy as input and generates as output an automaton that models the XACML policy. That synthesis procedure is an adaptation of the synthesis procedure of [59, 60, 64] to our context (Cloud, e-health, XACML). The obtained automaton is then used for different policy studies: (a) policy evaluation process, i.e. a process to find the permission (if any) generated by a policy for a given request (chapter 5), (b) policy anomalies detection and completeness verification (chapter 6), and (c) to resolve the detected anomalies (chapter 7). The rest of this chapter is organized as follows: Section 4.2 describes the fours steps of the synthesis procedure to model a XACML policy by an automaton. Section 4.3 presents examples of some security policies’ automata representation. Finally, Section 4.4 concludes the chapter.

Parts of this chapter have already been published in [18].

4.2 Automata Model of XACML policies

Our proposed automata-based model [18] is realized as follows: From the XACML representation of a policy $F$, we construct an automaton $A$ that models $F$, and then our analyses
of $F$ are done on $\mathcal{A}$. The automaton $\mathcal{A}$ generated from $F$ has the following characteristics: from the initial state of $\mathcal{A}$, we have several possible paths where each path consists of a pair of transitions that leads to a final state associated to an authorization (permission or prohibition) $u_A$. Each path is described as follows: the first and second transitions are labeled $S$ and $O$ respectively, and the reached state is associated to $u_A$. The set of paths of $\mathcal{A}$ represents therefore a set of rules that constitute $F$. Table 4.1 indicates how the constituents of a XACML policy are represented in the corresponding automaton.

Table 4.1: How the constituents of a XACML policy are represented in the corresponding automaton

<table>
<thead>
<tr>
<th>XACML Policies</th>
<th>Finite State Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>Word</td>
</tr>
<tr>
<td>Set of subjects and objects</td>
<td>Alphabet</td>
</tr>
<tr>
<td>Action attributes</td>
<td>Actions associated to the final states</td>
</tr>
<tr>
<td>Subject attributes</td>
<td>Labels of first transitions</td>
</tr>
<tr>
<td>Resource attributes</td>
<td>Labels of second transitions</td>
</tr>
</tbody>
</table>

Consider a medical organization $x$ and its security policy consisting of rules $x_1, x_2, \ldots, x_n$, where $n$ is the number of rules. The construction of the automaton from the policy is done in four steps [59, 60]. The rest of this section explains in details these steps and illustrate each step by an example.

### 4.2.1 Step 1: Attribute extraction from XACML policies

This step consists in extracting all pertinent elements from the XACML policies. In XACML, a rule is described by: an **Effect** and a **Target**. The **Effect** can have two values: "Permit" and "Deny". The **Target** is a combination of **attributeID** elements. Each **attributeID** element describes an attribute that a **Request** should match in order to activate the policy. There are four attribute categories in XACML 3.0, namely: (a) subject attribute is the entity requesting the access, e.g., generalist, radiologist, etc; (b) resource attribute is the object or the required data, e.g., EEG, MRA, etc; (c) action attribute defines the type of access requested, e.g, read, write, delete, etc; and (d) environment attribute defines additional information such as time. Environment attributes describe the operational, technical, and even organizational environment or context in which the information access occurs. In this thesis we do not consider the environmental attributes. We only consider the most relevant attributes in the policy: subject, resource and action (described in section 2.3.1). However, in some examples we may use the environment attribute to present the owning organization if necessary.

Algorithm 1 parses the XACML policy using the function $getDetailPolicy$. The latter extracts rules from the XML file and expresses each of them formally by a triplet $(S, O, u_A)$. Each rule has: **Effect** ($u$), **Subject** ($S$), **Resource** ($O$), and **Action** ($A$). For instance, if we consider the
second rule of Table 2.1, the algorithm will return the following attributes: subject=generalist, resource=scans, effect=permit and action=\{\textit{read, write}\}. Hence, we can formally express the rule as follows: $h_2 = \text{(generalist, scans, } P_{r,w} \text{)}$.

**Algorithm 1** Algorithm of Step 1

**Input:** XACML Policy  
**Output:** $S, O, u, A$

1: 
   \textbf{procedure} getDetailPolicy(Policy.xml) 
2:    \textbf{document} ← parse(Policy.xml)  
3:    \textbf{root} ← \textbf{document}.getDocumentElement()  
4:    \textbf{while} \textbf{root} $\neq \text{EndOfDocument}$ \textbf{do}  
5:       \textbf{for} $i < \text{nbRootNodes}$ \textbf{do}  
6:          \textbf{if} node.getName = "Rule" and node.getAttributes $\neq \text{null}$ \textbf{then}  
7:             $u$ ← node.getAttributes.getNamedItem("Effect")  
8:          \textbf{if} node.getNodeName = "Subject" \textbf{then}  
9:             $S$ ← Attribute.getTextContent  
10:          \textbf{else if} node.getNodeName = "Resource" \textbf{then}  
11:             $O$ ← Attribute.getTextContent  
12:          \textbf{else if} node.getNodeName() = "Action" \textbf{then}  
13:             $A$ ← Attribute.getTextContent  
14:      \textbf{end if}  
15:  \textbf{end for}  
16: \textbf{end while}  
17: \textbf{return} $S, O, u, A$  
18: \textbf{end procedure}

4.2.2 Step 2: Automaton for each rule

Each rule $x_i = \text{(S, O, uA)}$ obtained in Step 1 is described by an automaton with four states $x_0^i, x_1^i, x_2^i$ and $x_3^i$, where $x_0^i$ represents the initial state, and $x_2^i$ and $x_3^i$ are final states. The pair of states $x_0^i$ and $x_1^i$ are linked by a transition labeled S, and the pair of transition $x_1^i$ and $x_2^i$ are linked by a transition labeled O. The permission $u_A$ is associated to the final state $x_2^i$. Transitions labeled $\neq$S and $\neq$O link $x_0^i$ and $x_1^i$ to $x_3^i$ respectively. The final state $x_2^i$ is called match state, because it is reached for any request matching the attribute values of the rule $x_i$, i.e. for any subject $s \in S$ and object $o \in O$. The final state $x_3^i$ is called no-match state, because it is reached if the request does not match the attribute values of $x_i$. As an example, Fig. 4.1 represents the automaton obtained from the rule $h_2$ of Table 2.1. other rules are represented in the same way.

![Figure 4.1: Automaton $A_2$ obtained in Step 2 which models rule $h_2$ of Table 2.1.](image-url)
4.2.3 Step 3: Standardize the intervals of the automaton

The automata obtained in Step 2 do not have the same alphabet, the objective here is therefore to rewrite the transitions of the automata so that they have the same alphabet (this rewriting is useful for Step 4). This is realized by partitioning each of the domains of subjects and objects into a set of disjoint sets. Such partitioning permits to express a transition of an automaton as a union of sets of the partition.

For example, in the automaton of Fig. 4.1 the transition labeled $\text{scans}_{H_1}$ can be partitioned into multiple transitions. The partition consist in splitting this transition into multiple transitions labeled by sub-categories of $\text{scans}_{H_1}$ as it was described in section 2.2.2: $\text{scans} = \bigcup_{i=1}^{\ldots} \text{scans}^i = MRI \cup MRA \cup CAT \cup EEG$. Therefore, the automaton of of Fig. 4.1 can be transformed into the automaton of Fig. 4.2. The set $\text{scans}_{H_1}$ has been partitioned into 4 sets $MRI_{H_1}, MRA_{H_1}, CAT_{H_1}$ and $EEG_{H_1}$, which implies that the transition labeled $\text{scans}_{H_1}$ is replaced by four transitions labeled $MRI_{H_1}, MRA_{H_1}, CAT_{H_1}$ and $EEG_{H_1}$, respectively.

![Figure 4.2: Automaton $A^*_2$ obtained from $A_2$ of Fig. 4.1](image)

It is worth noting that the same partition can be applied to the six automata representing rules of Table 2.1. For instance, using the categorization of section 2.2.2, the transition labeled $\text{Objects}$ in rule $h_3$ can be split into 5 transitions labeled: $PR$, $\text{scans}$, $\text{audios}$, $\text{listDoctors}$ and $\text{collSessDisc}$. Each category of $\text{Objects}$ can be split into sub-categories presenting types.

4.2.4 Step 4: Composition of automata

In order to model the security policy defined in a XACML policy file, we combine the automata resulting from Step 3 by an operator called synchronous composition. Hereafter, we consider the policy presented in Table 2.1 as an example. It contains six rules regulating access to different resources (objects).

The resulting automaton representing the policy of an organization $x$ is denoted $A_x$. Each of its states is a combination of states $(u_1, u_2, \ldots, u_n)$ of the various combined automata, hence each $u_i = x_i^j$ or $x_i^\#$, for $j = 1, 2$. A final state $x^2_{i_1,i_2,\ldots}$ may be associated to one or more permissions. For the sake of clarity, a state of $A_x$ is noted $x^j_{i_1,i_2,\ldots}$, where we indicate only the indices $i_k$ such that $u_{i_k} = x^j_{i_k}$ (i.e. $u_{i_k} \neq x^\#_{i_k}$), for $j = 1, 2$. For example, the initial state is noted $x^0_{i_1,i_2,\ldots,n}$. A state is noted $x^\#$ if all its components are $x_i^\#$. For example, if we apply the four
steps to the policy of Table 2.1, we obtain the automaton of Fig. 4.3.

Let us show in an example how this automaton is interpreted. Consider a radiologist $r$ of $H_1$ and a personal record (PR) $pr$ of $H_1$. Notice that $r$ and $pr$ are elements of Radiologist and PR respectively. Let us assume that $r$ tries to have access to $pr$. Using the automaton $A_{H_1}$ of Figure 4.3, we execute the 2 transitions labeled Radiologist and PR, respectively, which leads to state $h_4^2$. The radiologist $r$ is permitted any access to $pr$, because the permission associated to the reached state $h_4^2$ is $p_a$.

Figure 4.3: Automata $A_{H_1}$ modeling the security policy of $H_1$

4.3 Other examples of Synthesis of automata

Using the same procedure of section 4.2, we obtain the automaton $A_{H_2}$ (Fig. 4.4) which models the XACML policy of hospital $H_2$ represented in Table 2.2. Furthermore, we obtain automaton $A_{H}$ (Fig. 4.5) which Models the Security Policy of Table 2.3. The Automata will be used for the policy studies presented in chapters 5, 6 and 7.
4.4 Chapter Summary

This chapter presented the synthesis procedure to model XACML security policies using automata. The procedure is composed of four steps. The first step extracts relevant attributes from the policy structured in the XML file. Then the second step constructs an automaton for each rule using the extracted attributes. However, the resulted automata do not have the same alphabet. Therefore, step 3 consists in unifying and standardizing the intervals of each automaton by partitioning each of the domains of subjects and objects into a set of disjoint sets. Finally, step 4 combines all the rules’ automata to model the whole security policy. We provided some examples of automata that are going to be useful for the rest of this document. Next chapter shows how the automata model is used to evaluate a policy, i.e. to determine the action (if any) generated by a policy for a given request.
Chapter 5

Automata-based evaluation of XACML policies

5.1 Introduction

Due to the current Big data exponential growth, several medical organizations outsourced their data to the Cloud. On the other hand, most of Cloud environments such as openStack offer an infrastructure as service (IaaS) to manage policy authorizations. Such authorization services guarantee a low level control implemented as a fine-grained access control. Low-level control does not give cloud users the possibility to define their own security policies. However, in collaborative environments, such as e-health, organizations need to impose their own security policies to regulate their data sharing. Therefore, there is a need to introduce facilities in the cloud layers to support access control policies specification at a high level, and hence to permit cloud users to have control over their outsourced data.

XACML specification [75] suite for expressing access control policies for complex distributed systems [34]. It can combine results of multiple policies developed by potentially independent policy issuers via its PolicySet component. Therefore, XACML remains the most adequate specification language for collaborative systems. Yet, XACML policies become complex and hard to manage for large scale systems, which decreases the performance of the decision process (i.e., policy evaluation process).

To address these two challenges, we propose a Cloud middleware that facilitates to its users the enforcement of their own policies specified in XACML. The middleware permits a translation of curl primitives into XACML requests. The XACML policies are evaluated against the requests using an automata-based approach which uses the model presented in section 4. The remainder of the chapter is organized as follows: Section 5.2 describes the architecture of the proposed middleware curlX. The middleware is composed of three components, but in this chapter we present only two of them: the curlX translator and X2Automata for policy evaluation. The
third component is presented in chapter 6. Section 5.3 provides implementation details of the policy evaluation process. It compares the evaluation performance to Balana (The open source implementation of XACML). Finally Section 5.4 concludes the chapter.

Parts of this chapter have already been published in [14, 15, 16].

5.2 curlX: a MiddleWare to Enforce Policies in OpenStack

A XACML request represents a subject’s request to perform a specific action on a specific resource. XACML assumes an architecture containing a PDP (Policy Decision Point) that searches in the policy repository for the appropriate policies that match the request. The PDP then sends a response to the requestor, which can be: Permit, Deny, Not Applicable or Indeterminate. Not Applicable is applied if no rule matches the request (or we say that the request does not satisfy the policy). While Indeterminate is applied if the system cannot interpret the request. This process is called decision process or policy evaluation process.

Commercial implementations of XACML, such as Sun XACML PDP, perform a brute force search to compare an access request against a set of rules in a XACML policy. This search technique decreases the performance of the policy evaluation process.

In this section, we present a novel approach to increase the policy evaluation performance of XACML. The approach is implemented as a cloud policy engine that we denote X2Automata which is a part of our proposed middleware curlX. X2Automata models the access control and the policy by an automaton using the synthesis procedure of chapter 4. Then, it combines both of them using the synchronous composition. X2Automata applies an evaluation procedure to the resulting automaton to decide whether an access request is granted or not. Thus, instead of checking all XACML policy rules in the XML file, we check only the final states of the automaton. Such states represent the pair \((a, u_A)\), where \(a\) is the requested action and \(u_A\) represents the decision of the matched rule. So the policy evaluation process is reduced to the verification of: \(a \in A\).

5.2.1 Architecture

Openstack is vendor nutral (i.e. the user can add middlewares and features in the original kernel), and hence flexible to extend existing functionalities. Since most of the openStack features are extended using middlewares, we extend the security policies’ analyses by a middleware denoted curlX [15, 14] presented in Fig. 5.1. Our proposed middleware consists of three components: (a) a translator [14]: responsible for translating the cloud client’s requests into XACML requests; (b) X2Automata: is responsible of evaluating the XACML policies using an automata-based approach; and (c) CPVS (Cloud Policy Verification Service) [18]: detects
and resolves anomalies in the XACML policies, and verifies their completeness. In this section, we present only the first two components (translator and X2Automata), while chapters 6 and 7 detail the third component: Cloud Policy Verification Service.

5.2.2 CurlX Translator

Swift’s clients usually communicate with the storage cluster via curl queries (Section 2.6). When a user wants to access an object, he requests an authentication from keystone:

```
```

Keystone provides the user by an authentication token and a storage url which are useful for any access request. The user requests the access to an object using a curl query as well. For instance, a doctor of $H_1$ wants to access a brain scan of patient$_1$ in hospital $H_2$. The syntax to get this object is expressed as follows:

```
curl -v -H GET /v1/Auth-account-patient1/Hospital2/BrainScan HTTP/1.1 -U doctor$_1$ : Hospital$_1$ -K passWord
```

The parameter passed after -U represents the user name, and the parameter that follows -K represents the password. Algorithm 2 translates the curl request into a XACML request.

The translation is done by splitting the curl request into three main parts namely; action, user’s credentials and storage url.

- Action: GET, PUT, DELETE, etc;
Algorithm 2 \texttt{curlX\_curl\_request}: translates a curl query into XACML request

\begin{verbatim}
Input: curl access request
Output: XACML request
1: for request do
2: \textit{Decompose\_curl\_request}(request\_access) \textbf{return} user, container, action, account
3: \textbf{sub} $\leftarrow$ user
4: \textbf{res} $\leftarrow$ container
5: \textbf{action} $\leftarrow$ action
6: \textbf{env} $\leftarrow$ account
7: \textit{Transform\_XACML}(sub, res, action, env) \textbf{return} XACML request
8: end for
\end{verbatim}

- User’s credentials: name and password (\textit{doctor}_1, passWord);
- The storage url: container’s name and account’s name (\textit{hospital}_2 and patient1).

The system then retrieves from each part a set of attributes to be adapted then to the XACML attributes syntax: subject, resource, action and environment. Afterwords, the request is written into the XACML format as shown in Listing 5.1.
Swift gives its users the possibility to share their objects with other users in other accounts by adding a meta-data to the container containing the objects. This meta-data is represented by a set of ACLs based on two primitives: X-Container-Read and X-Container-Write, which specify the read acl and the write acl respectively. curlX translator writes the acl queries into XACML rules and insert them in the policy using the Policy Administration Point (PAP). Algorithm 3 present the translation of an acl request into a XACML rule. The users’ requests are then evaluated by the X2Automata component using an automata-based approach.

5.2.3 X2Automata: An efficient policy evaluation engine

The objective of X2Automata component is to evaluate a XACML Policy against a given request. This necessitates to check whether a request satisfies a XACML policy; this process
Algorithm 3: AcltoPolicy : adds a rule to a policy

**Input:** curl acl request

**Output:** update the XACML policy

1: for acl do
2: Decompose acl(acl)
3: if X_container_write then return user, container, account, write
4: sub ← user
5: res ← container
6: env ← account
7: action ← write
8: else
9: return user, container, account, read
10: sub ← user
11: res ← container
12: env ← account
13: action ← read
14: end if
15: if container.xml exists then
16: update container.xml
17: add_policy (sub, res, action, env)
18: else
19: Create policy(container.xml)
20: end if
21: return container.xml
22: end for

is called XACML policy evaluation process. This section presents the automata-based approach applied for the policy evaluation. We use the following definitions:

**Definition 1** An access to an organization $x$ is any access $=(S, O, a)$ such that $O$ consists of objects of $x$.

**Definition 2** An access $Acc = (S, O, a)$ to an organization $x$ satisfies a security policy $SecPol_x$ if for every $s \in S$ and $o \in O$, $SecPol_x$ permits that $s$ has access to $o$ through the action $a$ of $Acc$.

From the fact that the security policy of an organization $x$ specifies restrictions uniquely on objects of $x$, our analysis is achieved by verifying whether the security policy $SecPol_x$ of each organization $x$ is satisfied by every access to $x$. Our objective can therefore be achieved by the following procedure:

Algorithm 4: Analysis Procedure

```plaintext
procedure
  for every organization $x$ do
    for every access $Acc$ of the collaborative session do
      if $Acc$ is an access to $x$ then
        verify if $Acc$ satisfies $SecPol_x$
      end if
    end for
  end for
end procedure
```

We now propose an automata-based approach to execute the “verify if $Acc$ satisfies $SecPol_x$” instruction of the analysis procedure. Let $A$ and $A_x$ be the automata modeling $Acc = (S, O, a)$ and $SecPol_x$, respectively. Recall that $A$ has a match state and possibly a no-match state, and $A_x$ has one or several match states. We have the following proposition:
Proposition 1 Consider a security policy SecPol\textsubscript{x} of an organization \textit{x} and an access Acc to \textit{x}. We have Acc satisfies SecPol\textsubscript{x} if for every (s, o):

\begin{itemize}
  \item (s, o) leads to the match state of \textit{A} (associated to the action \textit{a}) implies that
  \item (s, o) leads to a match state of \textit{A}\textsubscript{x} associated to a decision \textit{u}\textsubscript{A} that contains a, i.e., \textit{a} \in \textit{A}.
\end{itemize}

The verification of the implication of Proposition 1 is achieved by constructing the automaton that combines \textit{A} and \textit{A}\textsubscript{x}. The composition is guaranteed in two steps that are similar to the Steps 3 and 4 of the synthesis procedure of Section 4.2.

**Step a:** The automata \textit{A} and \textit{A}\textsubscript{x} may have distinct alphabets because their respective transitions are not necessarily labeled by the same sets of subjects and objects. To be able to combine these automata in Step b, the objective of Step a is to re-label the transitions of \textit{A} and \textit{A}\textsubscript{x} so that they have the same alphabet. This is achieved by splitting each transition labeled by a given set \textit{U} (of subjects or objects) into several transitions labeled by an adequate partition of \textit{U}. The states are not modified.

**Step b:** After their transformation in Step a, \textit{A} and \textit{A}\textsubscript{x} are then combined by the synchronous composition. The resulting automaton is denoted by \textit{A} \times \textit{A}\textsubscript{x}. Each of its states is a combination (r, q) of two states of \textit{A} and \textit{A}\textsubscript{x}, respectively. In particular, (r, q) is a final state of \textit{A} \times \textit{A}\textsubscript{x} if \textit{q} and \textit{r} are final states of \textit{A} and \textit{A}\textsubscript{x}.

We have the following proposition used as an automata-based procedure to execute “verify if Acc satisfies SecPol\textsubscript{x}” instruction of the analysis procedure.

Proposition 2 Consider a security policy SecPol\textsubscript{x} of an organization \textit{x} and an access Acc to \textit{x}. Let \textit{A} \times \textit{A}\textsubscript{x} be the synchronous composition of the automata \textit{A} and \textit{A}\textsubscript{x} modeling Acc = (\textit{S}, \textit{O}, \textit{a}) and SecPol\textsubscript{x}, respectively. We have Acc satisfies SecPol\textsubscript{x} if for every final state (r, q) of \textit{A} \times \textit{A}\textsubscript{x}:

\begin{itemize}
  \item \textit{r} is a match state of \textit{A} (hence \textit{r} is associated to the action \textit{a}) implies that
  \item \textit{q} is a match state of \textit{A}\textsubscript{x} associated to a decision \textit{u}\textsubscript{A} containing the action \textit{a}, i.e. \textit{a} \in \textit{A}.
\end{itemize}

As an example, lets consider that a generalist wants to create a personal record (PR). We apply the automata-based procedure to “verify if Acc satisfies SecPol\textsubscript{H\textsubscript{1}}” where Acc = (generalist, PR, create) and SpecPol\textsubscript{H\textsubscript{1}} represents the security policy of hospital \textit{H\textsubscript{1}}. Using the steps a and b, we obtain the automaton \textit{A} \times \textit{A}\textsubscript{H\textsubscript{1}} (Figure 5.2) that represents the combination of the policy’s and request’s automata. The automaton of Fig. 5.2 has a match state named (\textit{q}\textsubscript{1}\textsuperscript{2}, \textit{h}\textsubscript{2}\textsuperscript{1}) that corresponds to (create, \textit{p}\textsubscript{r}). Using Proposition 2, we deduce that Acc does not satisfies SecPol\textsubscript{H\textsubscript{1}} because “create” \in \{r\}. In this case, X2Automata returns NotApplicable as a result of the policy evaluation process.
To demonstrate the efficiency of our approach, we evaluate its performance against the XACML open-source implementation named Balana. The implementation and the performance measurements of our approach are based on Devstack release of OpenStack. Our experiments were performed on an Intel Core 2 Duo CPU 2 GHz with 3 RAM running on Windows 7. It is difficult to get a large number of real-life XACML policies, since access control policies are often deemed confidential for security reasons. Therefore, we have developed a random XACML policy and request generator. We generated 6 synthetic XACML policies of large sizes. In our experiments, we evaluated the impact of the policy size in terms of the number of rules varying from 10 rules to 10,000 rules.

Fig. 5.3 shows the pre-processing time versus the number of rules for X2Automata. The pre-processing time represents the time for constructing the policy’s automaton. We observe that there is an almost constant correlation between the number of rules and the pre-processing time of X2Automata for policies with hundreds of rules, which demonstrates that X2Automata is scalable in the pre-processing phase for policies with a small number of rules. However, it starts to grow for large size policies due to the multiplication of the number of rules. This increase is also related to the number of subjects and objects in each security policy.

This pre-processing step is executed only if the policies are modified: new rules are added, new policies are added to the collaboration, or rules are changed in a given policy. In the first case, the time of pre-processing almost does not change because the number of rules to be added is usually does not exceed hundreds. However, the two other cases increase the time of pre-processing. To illustrate this case, lets consider an organization \( x \) with a policy \( PolSec_x \).
containing 1000 rules. This organization is collaborating with another organization denoted $y$ with a policy $PolSec_y$ containing 2000 rules. The time to add the new policy $PolSec_y$ to $PolSec_x$ (i.e. the time of constructing and combining the policies’ automata) is around 9 seconds. This time includes the pre-processing time to generate the automaton of $PolSec_y$ and the time of the synchronous composition. In fact, the time of 9 seconds is dominated by the generation of the automaton of $PolSec_y$, because the time for the synchronous composition does not exceed 5 ms.

![Figure 5.3: Pre-processing (Automata construction) time on synthetic XACML policies.](image)

In terms of efficiency, we measured the policy evaluation processing time of X2Automata compared to Balana. For X2Automata, the processing time includes the time for constructing the synchronous composition of the request’s automaton and the policy’s automaton, and the time for finding the final state that matches the request. For Sun PDP (Balana), the processing time for the policy evaluation is the time for finding the decision in the XML file. Fig. 5.4 shows the difference between Sun PDP and X2Automata for the total processing time of different synthesized XACML policies. Note that the two lines in Fig. 5.4 are not close to each other. This figure shows that X2Automata outperforms Sun PDP by 10 times.

![Figure 5.4: Processing time difference between Sun PDP and X2Automata.](image)
5.4 Chapter Summary

This chapter has presented our proposed middleware curlX which permits to cloud users to enforce their own security policies. We adopted OpenStack as our Cloud environment because of its advantages (open source, vendor neutral, etc). Since OpenStack’s object storage communicates with the users using curl queries, curlX contains a translator which translates the curl primitives into XACML requests. curlX translates the swift ACLs into XACML rules, and adds them to the old policy. The proposed middleware contains a policy decision engine denoted X2Automata. X2Automata uses an automata-based approach in order verify if the access is permitted by the XACML policy or not (policy evaluation process). This chapter has demonstrated that X2Automata improves the policy evaluation time compared to Balana. Next chapter presents the automata-based approach to detect anomalies within a XACML policies.
Chapter 6

Anomalies detection in XACML policies

6.1 Introduction

e-Health permits the usage of modern communication infrastructures such as Cloud computing to improve health care services. Cloud computing ensures an inter-organizational collaboration via the multi-tenancy model which improves the quality of diagnoses. The problems that can be encountered during collaborative diagnoses are often related to security and privacy of data. To overcome these problems, each tenant (medical organization) enforces its own security policy. Their composition presents a global policy that regulates accesses during the collaboration. Yet, this composition may provoke policy anomalies such as similarities and inconsistencies. Another solution to insure confidentiality of data is to verify the completeness property. The policy is complete when it guarantees that each access request is either accepted or denied by the access control policy.

This chapter presents CPVS (Cloud Policy Verification Service) for the analysis and the verification of access control policies specified using XACML. The analysis process detects two categories of anomalies: intra-policy anomalies, and inter-policies anomalies, while the verification process verifies the completeness property. The rest of this chapter is organized as follows: In section 6.2, we describe the multi-tenancy model for collaborative diagnoses. Section 6.3 presents a categorization of security policy anomalies. Section 6.4 presents the automata-based approach to detect security policy anomalies and to verify the completeness property. In order to demonstrate the efficiency of our method, we also provide the time and space complexities in section 6.5. Implementation details regarding the anomalies detection modules of CPVS and experimental evaluations are presented in section 6.6. Section 6.7 summarizes the chapter.

Parts of this chapter have already been published in [18, 17, 16].
6.2 Multi-Tenancy Collaboration

Cloud computing offers most of its services under multi-tenancy environments. To facilitate cross-organizational collaborations, cloud computing offers multi-tenancy architectures. In such architectures, multiple customers, called tenants, transparently share the cloud’s resources and services [55]. Multi-tenancy permits cloud providers to establish collaborative relationships among cloud users [6], especially at the storage service level. Thereby, the data stored in the cloud are available not only to its original owners, but also to other people in different tenants. However, data sharing across collaborations raises significant challenges in terms of security and especially access control.

To satisfy security requirements among collaborating tenants, each tenant may define a set of access control policies to secure access to shared data (Fig. 6.1). Each organization, i.e., a tenant, manages its internal resources via an access control policy (intra-tenant control). Thus, a given medical file may be stored in two different organizations. For instance, in Fig. 6.1 a Magnetic Resonance Imaging (MRI) scan is stored in two different clusters belonging to two different organizations: HH and hospital 2. Therefore, cross-tenancy communication imposes an inter-tenant control to manage accesses to data stored outside the organization. This kind of control consists of combining the authorization policies in order to control the access during the collaboration. This combination may provoke different types of anomalies. Next section represents a categorization of these anomalies.
6.3 Anomalies Categorization

In collaborative distributed systems, each collaborative organization may enforce a set of rules to regulate the access to their own data. A request in the collaborative session is then evaluated using a global policy generated by the composition of the collaborators’ policies. However, the composition may produce a set of anomalies. Khoumsi et al. [59] categorize the anomalies into two sets: conflicting anomalies and nonconflicting anomalies. The first category occurs when a request matches several rules that have contradictory authorizations (conflicts). As for the second category, it occurs when the same request matches several rules that have the same authorizations (redundancies).

In this thesis, we consider, in addition to conflicts and redundancies, another category of non-conflicting anomalies that we call “ambiguities”. A redundancy occurs when removing a rule does not affect the policy’s behavior; whereas an ambiguity occurs when a request matches two rules with the same authorization permit or deny, but with different sets of actions $A_1$ and $A_2$ (e.g. \{read, write\} and \{read, delete\}). Intuitively, in such situation, we do not know if we should apply the authorization to $A_1$, $A_2$, $A_1 \cap A_2$ or $A_1 \cup A_2$. For instance, “The radiologists can read the electroencephalogram (EEG) of all the patients” and at the same time “The radiologists can read and write into the EEG of all the patients”. This situation presents an ambiguity, because both rules permit the radiologist to read EEG, while the second rule permits in addition the radiologist to write into EEG. So, ambiguity manifests itself by the following question: should we or should not we authorize the radiologists to write in the EEG of patients? A conflict occurs when negative and positive authorizations are assigned (by two different rules) to the same subject to perform an action over a given object. For example, “The generalist of hospital $H_1$ is permitted to read the medical record (MR) of the patient $x$ in hospital $H_2$” and also “The generalist of hospital $H_1$ is forbidden to read the medical record (MR) of all the patients of hospital $H_2$”.

Policy anomalies are defined at two levels namely: intra-policy and inter-policies. Intra-policy anomalies corresponds to anomalies within the same policy (redundancies, ambiguities, and
conflicts), whereas inter-policies correspond to anomalies between several security policies such as inconsistency and similarity. A summary of our anomalies’ categorization is presented in figure 6.2 (where the term *intra* refers to intra-policies anomalies, and the term *inter* refers to inter-policies anomalies).

Next section presents our proposed automata-based approach to detect the anomalies presented in this section. The approach uses the automata model generated in section 4.2. The automata-approach was implemented in a cloud middleware as a policy verification service (CPVS).

### 6.4 CPVS: Policy Verification Service

Cloud Policy Verification Service (CPVS) is a component of curlX middleware (Fig. 5.1) which permits the detection and resolution of anomalies and the verification of the completeness property. It consists of the following modules (see Fig. 6.3):

Figure 6.3: The three modules of CPVS (Cloud Policy Verification Service)

- Intra-policy anomaly detection: responsible for detecting redundant, ambiguous, and conflicting rules in a single security policy.

- Inter-policy anomaly detection: if the policy cache contains combined security policies, this module is used to detect inconsistency and similarities between the combined policies.

- Policy completeness verification: verifies the completeness for each policy stored in the cache.
Each of the three modules communicates with the policy cache via the \textit{Xparser} module that parses the XACML policies and extracts its components in a hierarchical way. Each module of \textit{CPVS} detects and resolves anomalies. However, in this chapter we only present the detection approaches, while the resolution approaches are presented in chapter 7. The details of anomaly detection modules are presented in the rest of the chapter as follows: Section 6.4.1 presents the anomalies detection process, and the policy properties verification module is presented in Section 6.4.2.

### 6.4.1 Anomalies Detection Process

With XACML, we can specify the policies of collaborative organizations. The policy of each organization is presented by the XACML component \textit{Policy}. The component \textit{PolicySet} specifies how different \textit{Policy} components are combined. We study therefore two types of anomalies: intra-policy anomalies correspond to anomalies between rules of a same policy (we can refer to intra-tenant control), and inter-policies anomalies correspond to anomalies between rules of different policies (we can refer to inter-tenants control). The first type occurs within the \textit{Policy} component, and the second type occurs within the \textit{PolicySet} component. In this thesis, we consider three types of intra-policy anomalies: redundancies, ambiguities, and conflicts; and two types of inter-policies anomalies: similarities and inconsistencies. Hereafter, we use the term “authorization” to refer to permission (permit) or prohibition (deny). We use the term “decision” to refer to the authorization $X$ (permission or prohibition) of a set $A$ of actions, which is denoted $X_A$.

**Intra-Policy Anomaly Detection Module**

Let us first consider intra-policy anomalies of a policy $F$ and show how they are detected by the automaton of $F$.

- **Redundancy Detection**

**Definition 3** \textit{Redundant rules:} In a policy $F$, a rule $R_j$ is redundant to a rule $R_i$ if the result of $F$ is not changed by removing $R_j$ and keeping $R_i$.

**Proposition 3** Consider a policy $F$ and its automaton $A_F$. A rule $R_j$ is redundant to a rule $R_i$ if for every match-state $x_{j_1,j_2...}$ of $A_F$:

1. the match-state has the index $j$ only if it has also the index $i$, and
2. $i$ and $j$ are associated to the same decision (i.e. same authorization and same set of actions).
Consider, for example, the policy of Table 2.3 and its automaton of Fig. 4.5. Rules 2 and 6 are mutually redundant to each other, because: 1) the indices 2 and 6 are in the state $h_{2,6}$ and there is no other state where the indices are not together, and 2) the same permission $P_{\text{read}}$ is associated to both indices. Therefore, we can remove either $R_2$ or $R_6$ without changing the result of the policy.

**Conflict Detection:**

**Definition 4** *Conflicting rules:* In a policy $F$, two rules $R_i$ and $R_j$ are conflicting if they can match the same subjects and objects $(s,o)$ and have different authorizations.

**Proposition 4** Consider a policy $F$ and its automaton $A_F$. Rules $R_i$ and $R_j$ are conflicting if there exists a match-state $x_{i,j}^2$ of $A_F$ such that:

1. the match-state has the indices $i$ and $j$, and
2. $i$ and $j$ are associated to contradictory decisions (i.e. different authorizations and at least one common action).

Consider, for example, the policy of Table 2.3 and its automaton of Fig. 4.5. The match-state $h_{2,4}$ implies that $R_3$ and $R_4$ are conflicting. Intuitively, $R_3$ permits radiologists to read scans while $R_4$ forbids it. Also, the match-state $h_{2,5,7}$ implies that $R_1, R_5, R_7$ contains pairs of conflicting rules. Intuitively, $R_1$ and $R_7$ permits generalists to read PR while $R_5$ forbids it.

**Ambiguity Detection:**

**Definition 5** *Ambiguous rules:* In a policy $F$, two rules $R_i$ and $R_j$ are ambiguous if they can match the same subjects and objects $(s,o)$ and have the same authorization with different actions.

**Proposition 5** Consider a policy $F$ and its automaton $A_F$. Rules $R_i$ and $R_j$ are ambiguous if there exists a match-state $x_{i,j}^2$ of $A_F$ such that:

1. the match-state has the indices $i$ and $j$, and
2. $i$ and $j$ are associated to the same authorization with different sets of actions.

To illustrate ambiguity detection, we consider the policy that regroups the rules of the policies of $H_1$ and $H_2$ of Tables 2.1 and 2.2. The automaton $A_{H_1,H_2}$ obtained from that policies is represented in Fig. 6.4. Each of states is a combination $(u_1, u_2)$ of states of the automata that model $H_1$ and $H_2$. Note that each match-state (a leaf of the tree) is associated to two authorizations. Consider the match-state $(h_{1,2}^2, r_{1,2}^7)$ which is reached for any request matched by
Figure 6.4: Automaton modeling the combination of $H_1$ and $H_2$ of Tables 2.1 and 2.2.

the rules $h_1$ and $r_1$ of policies $H_1$ and $H_2$. There is an ambiguity, because $h_1$ and $r_1$ have the same authorization (permit) and different sets of actions (read vs \{read, write\}). Intuitively, $r_1$ permits generalists to read and write PR while $h_1$ permits generalists to only read PR.

Algorithm 5 regroups the steps of detecting intra-policy conflicts. The algorithm consists in extracting the final states using the function $\text{getFinalNodes}$. For each state having both indices \(i\) and \(j\), the algorithm compares the authorization associated to the final state; if the authorizations are different, then the rules are considered as conflicting rules. Otherwise the algorithm compares the actions. If the actions are equal, the rules are redundant, otherwise they are ambiguous.

Inter-policies anomaly detection Module

Let us now consider inter-policies anomalies of two policies $F_1$ and $F_2$. To detect this type of anomalies, we first need to construct an automaton that models the combination of $F_1$ and $F_2$. This is done by defining a new policy \(\langle F_1, F_2 \rangle\) that regroups $F_1$ and $F_2$ in a same table and then constructing the automaton modeling \(\langle F_1, F_2 \rangle\). For example, for the policies $H_1$ and $H_2$ of Tables 2.1 and 2.2, we obtained the automaton $A_{H_1,H_2}$ of Fig. 6.4.

- Detecting similar policies

Definition 6 Two policies $F_1$ and $F_2$ are similar if in every situation, the decision of $F_1$ is similar to the decision of $F_2$.

Proposition 6 Consider two policies $F_1$ and $F_2$ and the automaton $A_F$ of the policy $F$ that regroups $F_1$ and $F_2$. $F_1$ and $F_2$ are similar if in $F$: every rule of $F_1$ is redundant to a rule of $F_2$, and every rule of $F_1$ is redundant to a rule of $F_2$. 

60
Algorithm 5 Intra-policy Anomaly Detection

Input: Policy Automaton
Output: Redundancy Set $\mathcal{RS}$, Ambiguity Set $\mathcal{Am}$ and Conflicts Set $\mathcal{CS}$

1: procedure IsRedundant(Automaton) \quad \triangleright verify if there are any redundant rules
2: nodes $\leftarrow$ getFinalNodes(Automaton) \quad \triangleright Extract the final states from the automaton
3: while node.size $\neq 0$ and nodes $\neq$ null do
4: for $i \leftarrow 0$, node.size do
5: for $j \leftarrow i + 1$, node.size do
6: if node.get(i) = node.get(j) then \quad \triangleright The authorizations of the final states are equal
7: $\mathcal{RS}$.add(i,j) \quad \triangleright Add both rules i and j to the redundant Set
8: end if
9: end for
10: end for
11: end while
12: return $\mathcal{RS}$
13: end procedure

14: procedure IsAmbigious(Automaton) \quad \triangleright verify if there are any ambiguous rules
15: nodes $\leftarrow$ getFinalNodes(Automaton) \quad \triangleright Extract the final states from the automaton
16: while node.size $\neq 0$ and nodes $\neq$ null do
17: for $i \leftarrow 0$, node.size do
18: for $j \leftarrow i + 1$, node.size do
19: if node.permission.get(i) = node.permission.get(j) then \quad \triangleright The permissions are equal
20: if node.action.get(i) $\neq$ node.action.get(j) then \quad \triangleright The actions are different
21: $\mathcal{Am}$.add(i,j) \quad \triangleright Add both rules i and j to the ambiguity Set
22: end if
23: end if
24: end for
25: end for
26: end while
27: return $\mathcal{Am}$
28: end procedure

29: procedure hasConflict(Automaton) \quad \triangleright Verifies if there are any conflicting rules
30: nodes $\leftarrow$ getFinalNodes(Automaton)
31: while node.size $\neq 0$ and nodes $\neq$ null do
32: for $i \leftarrow 0$, node.size do
33: for $j \leftarrow i + 1$, node.size do
34: if node.get(i) $\neq$ node.get(j) then \quad \triangleright The authorizations of the final states are not equal
35: $\mathcal{CS}$.add(i,j) \quad \triangleright Add both rules i and j to the conflict set
36: end if
37: end for
38: end for
39: end while
40: return $\mathcal{CS}$
41: end procedure

Proposition 6 implies that similarity can be verified by detecting redundancy between the rules of the policies using Proposition 3. Consider, for example, the combination of the two policies of Tables 2.1 and 2.2 modeled by the automaton of Fig. 6.4. The two policies are not similar, because their only redundant rules are $h_6$ and $r_6$.

- Detecting inconsistent policies:

Definition 7 Two policies $F_1$ and $F_2$ are inconsistent if there exists a situation where they have contradictory (i.e., different) decisions.

Proposition 7 Consider two policies $F_1$ and $F_2$ and the automaton $A_F$ of the policy $F$ that regroups $F_1$ and $F_2$. $F_1$ and $F_2$ are inconsistent if in $F$ there exist a rule of $F_1$ and a rule of $F_2$ which are conflicting.

Proposition 7 implies that inconsistency can be verified by detecting conflicting rules using Proposition 4. Consider, for example, the combination of the two policies of Tables 2.1 and
modeled by the automaton of Fig. 6.4. The state \((h_2^2; r_2^2)\) implies that the two policies are inconsistent since they have contradictory decisions illustrated by \(p_{r,w}\) and \(d_{r,w}\).

Therefore, the same logic of the two procedures of Algorithm 5 can be applied to detect inconsistency and policy similarity. The only difference is the input of the algorithm: instead of an automaton corresponding to one policy, the input is replaced by an automaton modeling the concatenation of two policies (Algorithm 6).

**Algorithm 6** Inter-policy Anomaly Detection

| Input: | Policy1, Policy2 |
| Output: | Inconsistency Set IS and Similarity Set SS |
| 1: | procedure InterPolicyAnalyzer(Automaton) |
| 2: | ProductAutomaton ← generateProductAutomaton(P1, P2) |
| 3: | nodes ← getFinalNodes(ProductAutomaton) |
| 4: | RedRules ← IsRedundant(ProductAutomaton) |
| 5: | if RedRules.size() = n then |
| 6: | SS ← RedStates |
| 7: | end if |
| 8: | IS ← hasConflict(ProductAutomaton) |
| 9: | Return IS and SS |
| 10: | end procedure |

### 6.4.2 Verification of the Completeness Property

Besides the intra- and inter-security policy anomalies, it is important to assure the evaluation of security properties to guarantee the correctness of access control policies. Most of the existing Cloud verification methods focus mainly on the system behaviour verification and do not take into consideration the security policies. Therefore, designing a dedicated tool that targets the verification of security properties in the Cloud is an important issue to be addressed [47]. In this section, we describe a formal method based on the automata-model presented in chapter 4 to detect and verify the completeness property. Completeness guarantees that each access request is either accepted or denied by the access control policy.

**Proposition 8** A security policy \(\mathcal{P}\) is complete if and only if the corresponding synthesized automaton \(\mathcal{A}_\mathcal{P}\) has no no-match state.

For instance, the security policy presented in Table 2.3 is incomplete because its corresponding automaton in Fig. 4.5 has a no-match state denoted \(q^\#\). The 3 paths leading to \(q^\#\) correspond to the following 3 situations:

- A radiologist requests access to a PR or EEG.
- A generalist requests access to a scan.
- A neurologist requests access to a PR or a scan that is not EEG.

Intuitively, the security policy of Table 2.3 does not take any decision in these 3 situations. Algorithm 7 presents the procedure `isComplete` that verifies if the input automaton has a no-match state.
Algorithm 7 Verification of the Completeness Property

Input: Policy Automaton
Output: Verification of Completeness (C)

1: procedure isComplete(Automaton)
2: nodes ← getFinalNodes(Automaton)
3: while node.size ≠ 0 and nodes ≠ null do
4:   for i ← 0, node.size do
5:     if node.get(i) = E then ▷ E represents a non-match state
6:       return The policy is not complete
7:     end if
8:   end for
9: end while
10: end procedure

6.5 Evaluation of Space and Time Complexities

The results of this section are adaptations of results from [59, 60]. Let n be the number of rules of a policy, and \(d_1\) and \(d_2\) be the numbers of bits to code the subjects and objects, respectively. Hence, the maximum possible number of subjects and objects are \(2^{d_1}\) and \(2^{d_2}\), respectively. Let \(D = d_1 + d_2\). We consider two notions called great fields and small fields defined by Khoumsi et al. [60]. A great field is a field whose domain contains more than \(n\) values, and a small field is a field whose domain contains at most \(n\) values. We then consider two variables: \(\mu\), the number of great fields; and \(\delta\), the sum of the number of bits to code the small fields. In the security policies presented in this thesis, we have two fields \(m=2\) representing subjects and objects. Therefore, \(\mu \leq 2\).

By adapting results of Khoumsi et al. [59, 60] to our context, we obtain Proposition 9 (the proof of this proposition is in Appendix A).

Proposition 9 The space and time complexities of automata construction and completeness detection are in \(O(n^{\mu+1} \times 2^\delta)\), which is bounded by both \(O(n^3)\) and \(O(n \times 2^D)\).

The bounds of the complexities for the procedures of policy analysis are obtained by multiplying the above values by \(n\) (proof is in Appendix B). Hence, we obtain the following proposition:

Proposition 10 The space and time complexities of redundancy, ambiguity and conflict detections are in \(O(n^{\mu+2} \times 2^\delta)\), which is bounded by \(O(n^4)\) and \(O(n^2 \times 2^D)\).

The latter result holds also for detecting similarity and inconsistency between two policies, but by replacing \(n\) by \(n_1 + n_2\), where \(n_1\) and \(n_2\) are the numbers of rules of the two policies.

As an example, we can consider a policy with \(n = 500\) rules where the maximum number of subjects is 256 (so the subjects are coded in 8 bits: \(2^8 = 256\)) and where the maximum number of objects is 131072 (hence the objects are coded in 17 bits: \(2^{17} = 131072\)). Hence:

- \(D = 25 = 8+17 = \text{total number of bits to code the great and small fields}\).
- \(\mu = 1 = \text{number of great fields}: \text{there is one great field which is "objects", because } 2^{17} > 500\).
• $2 - \mu = 1 = \text{number of small fields: there is one small field which is } '\text{subjects}', \text{ because } 2^8 \leq 500.$

• $\delta = 8 = \text{number of bits to code the small field } \text{subjects}$

If we use the expression $O(n^{\mu+1} \times 2^\delta)$ which depends on the great and small fields, we obtain:

$O(n^{\mu+1} \times 2^4) = O((500^2) \times (2^8)) = O(64 \text{ millions}).$ However, If we use the two expressions $O(n^2)$ and $O(n \times 2^D)$ which do not depend on the great and small fields, we obtain:

• $O(n^2) = O(500^3) = O(125 \text{ millions})$

• $O(n \times 2^D) = O(500 \times 2^{25}) = O(16.7 \text{ billions})$

From the example, we can conclude that by considering the great and small fields, we obtain a more precise estimation of the complexity. Note that:

• when the two fields are great, we obtain $O(n^{\mu+1} \times 2^\delta) = O(n^2)$,

• when the two fields are small, we obtain $O(n^{\mu+1} \times 2^\delta) = O(n \times 2^D)$.

From Prop. 9, the time and space complexities of automata construction and completeness detection are upper-bounded by $O(n^3)$ and $O(n \times 2^D)$. Let $Ns$ and $No$ be the maximum numbers of subjects and objects, respectively. We have $2^D = 2^{d_1} \times 2^{d_2}$, $Ns = 2^{d_1}$ and $No = 2^{d_2}$. Therefore, $O(n \times 2^D) = O(n \times Ns \times No)$. We deduce that our complexities of automata construction and completeness detection exceeds neither the order of the polynomial $n^3$ nor the order of $n \times Ns \times No$.

With the same reasoning on Prop. 10, we obtain that our complexities of redundancy, ambiguity and conflict detections exceeds neither the order of the polynomial $n^4$ nor the order of $n^2 \times Ns \times No$.

In conclusion, our complexities are at most polynomial in $n$ and linear in $Ns$ and $No$.

### 6.6 Implementation and Evaluation

We have implemented our policy analysis service CPVS (Cloud Policy Verification Service) in Java. This service is integrated into curlX (section 5.2). Based on our policy anomaly analysis mechanism, CPVS consists of three core components: Inter-policy anomaly detection module, intra-policy anomaly detection module, and policy property verification module.

CPVS makes use of the DOM API provided by the Sun XACML implementation in order to parse the XACML policies and extract the attributes. We have implemented a Domain Specific Language (DSL) to support the construction of automata. In order to evaluate the efficiency and effectiveness of the proposed solution, first we need large policy data sets. Unfortunately, no
one has been published due to confidentiality constraints. Hence, we have developed a random policy generator in order to generate a large number of XACML policies. The number of rules of each policy varies from 100 rules to 1000 rules.

For scalability, it is also important to note that creating subjects and objects with no semantic relationship (categorization) is an inefficient approach. It is better to regroup the subjects and objects in subsets or categories to reduce their sizes. For instance, we can have 10 objects: 4 files consisting of prescriptions, 3 scans (EEG, MRI, BrainScan), and 3 documents containing information about the patient. For this example, we have two categories: Scans and PR (prescriptions and documents). In this way, instead of having 10 atomic objects, we have only 2 objects where each object corresponds to several possible elements. This reduces the number of states in the final policy automaton, which then reduces the time of anomaly detection.

We evaluated the efficiency and effectiveness of CPVS for synthetic XACML policies using 10 synthetic generated policies of 100 to 1000 rules. Our experiments were performed on an Intel Core 2 Duo CPU 2.00 GHz with 3.00 RAM running on Windows 7. Our experiments are related to the three types of anomalies detections: (a) intra-policy anomaly detection, (b) inter-policy anomaly detection, and (c) completeness verification.

The time required by CPVS to detect anomalies, such as redundancy, ambiguity and conflicts, depends on the time of parsing and comparing the final states of the automaton. From Fig. 6.5, we can notice that the times of redundancy, ambiguity and conflict detections are quasi equal, which reflects the results of time complexity of Section 6.5.

Furthermore, we generated synthetic policies consisting of 100 rules, and we combined them using the synchronous composition operator. Figure 6.6 presents the performance of CPVS to detect inconsistency and similarity between different set of policies (2, 4 . . .10). The detection time increases when the number of policies grows. This is can be explained by the time consumed by the synchronous composition which is composed of two main steps: the construction of each policy’s automaton and the composition of the resulted automata.
The verification of completeness, which consists in finding the *no-match* state in the policy’s automata, depends on at which step the research algorithm finds the *no-match* state. The performance of such verification is quasi constant (Fig. 6.7). It remains 2 ms for policies that contain 100, 200, 300 and 400 rules, and then it goes to 3 ms for the four other policies. This result reflects the results of time complexity of Section 6.5. The time complexity of the incompleteness detection is related directly to the time of the automaton’s construction.

6.7 Chapter Summary

In this chapter, we proposed a formal approach based on automata to detect XACML policy anomalies and verify the policy completeness. The approach has been implemented in a Cloud service called *CPVS* (Cloud Policy Verification Service) integrated into *curlX* middleware. The advantage of our approach is that it detects several anomalies in XACML policies at two different levels using the same formal model. In fact, it can detect intra-policy anomalies such as conflicts and redundancies, inter-policy anomalies such as inconsistencies and similarities, and ambiguity anomalies. We evaluated the time and space complexities for anomalies detection.
Chapter 7

Anomalies resolution in XACML policies

7.1 Introduction

Many access control policy languages such as XACML [100], EPAL [12], and firewall policies [110] support the combination of multiple sub-policies. These languages resolve the conflicts by combining the effects of the sub-policies according to some algorithms. Among existing policy languages, XACML offers the most flexible approach [73]. XACML (extensible Access Control Markup Language) proposes four policy combining algorithms (PCA) to resolve conflicts between multiple policies and rules. These algorithms take, as input, the authorization decision from each policy matching the request and apply some standard logic to come up with a final decision.

The PCAs are currently chosen in advance by the policy administrator and hence they are static. In highly dynamic environments, there is a need to select the PCAs dynamically. In this direction, we propose a strategy to dynamically choose the adequate PCA based on the request’s context. The resolution of anomalies is based on Aspect-Oriented FSM (AO-FSM), where pointcuts and advices are used to adopt Domain-Specific Language (DSL) [80] state machine artifacts. The pointcuts define matching states and patterns representing anomalies that may occur in a security policy, while the advices define the actions applied at the selected pointcuts, which may consist in inserting new state’s label, as well as deleting existing one.

The rest of this chapter is organized as follows: Chapter 7.2 gives an overview of the Aspect Oriented Finite State Machine. In section 7.3, we present our anomalies resolution strategies. Section 7.4 applies the proposed anomalies resolution strategies to AO-FSM using the automata model generated in chapter 4. We also provide the time and space complexities in section 7.5. We provide implementation details in section 7.6. Finally section 7.7 summarizes the chapter.

Parts of this chapter have already been published in [17, 38].
7.2 Aspect Oriented Finite State Machine: AO-FSM

7.2.1 Aspect Oriented Programming

Aspect Oriented Programming (AOP) was first defined and developed by Kiczales et al. in 1997 [61]. It extends previous programming paradigms like Object-Oriented Programming (OOP), while introducing the concept of crosscutting concerns. In fact, OO technology offers a great ability for separation of concerns. However, it still cannot localize concerns which do not fit into a single program module [44]. Concerns may define both: (a) high level notions such as security and quality of service, and (b) low level notions such as logging and XML parsing. We say that two concerns crosscut if the methods related to those concerns intersect [43] (i.e., there exist some methods that appear in both concerns).

Aspect-oriented programming (AOP) separates the crosscutting concerns into single units called aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules [43]. Each aspect can be expressed in separate form, then the various aspects can be combined together into a final executable form.

There are several Aspect Oriented implementations. However, the most known one is AspectJ [66], which is an open source java implementation of AO. It extends java by adding the keyword aspect. An AOP language has three critical elements for separating crosscutting concerns: joinpoint, pointcut and advice. They are detailed below:

- **A join point**: is an identifiable point in the execution of a program. It represents the “hooks” where enhancements may be added. For instance, in AspectJ, the join point can be illustrated by a method call or an execution handler [67].

- **A pointcut**: is a program construct that selects join points and collects their context. Pointcuts can be composed using logical operators [67].

- **An advice**: is the code to be executed at a join point that has been selected by a pointcut. It represents the crosscutting functionality. It encapsulates the logic to be executed once a join point is reached. AspectJ defines three types of advices namely: (1) before advice, (2) around advice and (3) after advice [67].

The pointcut defines a set of join points where the advices are integrated into the code. Therefore, aspect-oriented programming eases the development of reusable and maintainable code. It can ensure the security of the developed system in different ways [107]: (a) it is an automatically log data that may be relevant to security, (b) it can be used to replace generic socket code with ssl socket code, (c) it can be used to specify security policies, etc.

Generally, developers are not very good in writing secure software. Therefore, it was necessary to separate security specification from the system’s code. In this context, aspect
oriented programming allows this separation enabling security experts to specify security policies independently from developers. However, aspect oriented programming does not permit the verification and analysis of the specified policies. The previous chapter (6) presents an automata-based approach to detect anomalies within security policies. The current chapter presents an approach to eliminate these anomalies. The approach uses a formalism denoted AO-FSM (Aspect Oriented Finite State Machine) which combines two paradigms: FSM (automata) and ASP.

7.2.2 Aspect-Oriented Modeling with State Machines

In this thesis, we extend the formalism defined in our previous work [38] to detect and dynamically resolve anomalies in security policies. The formalism is denoted aspect-oriented finite state machines (AO-FSM). An AO-FSM defines a set of partial FSM (states and transitions) used as matching patterns to describe a concern (repeated behavior). It is worth noting that an FSM pattern can miss some parts of the FSM. For instance, the FSM pattern can define start states and outgoing transitions of the FSM, but no final state. In this thesis, we use AO-FSM to define patterns describing anomalies within a security policy. The anomalies’ patterns are illustrated by a set of final states patterns with labels representing redundant, ambiguous, or conflicting authorizations. If the policy’s final states match the defined anomalies’ patterns, then a method (advice) is executed to eliminate and resolve the matched anomaly. Like in other aspect-oriented languages, in an AO-FSM, there are three parts: Join-points, pointcuts and advices.

An AO-FSM joinPoint represents points in the execution of an FSM program. In this thesis, we consider four joinpoints:

- **Starting state machine** is triggered when a state machine is started, e.g. the automaton $A_{H_1}$ of Fig. 4.3 starts at the initial state $h_0$. The state $h_0$ represents starting state machine joinpoint for $A_{H_1}$.

- **Entering state** could be reified (illustrated) once an FSM enters into a given state, e.g. the automaton $A_{H_1}$ of Fig. 4.3 enters the state $h_1$ after the execution the the transition Generalist. The Entering state joinpoint is illustrated directly after the execution of transition Generalist.

- **Exiting state** is reified when an FSM exits the given state. If we use the previous example, this joinpoint is illustrated after that $A_{H_1}$ leaves state $h_0$ and directly before the execution of Generalist

- **Reset state machine** is reified when the FSM goes back to the start state. For instance, when $A_{H_1}$ goes back to $h_0$, the reset state machine joinpoint is illustrated.
The composition of a set of joinpoints defines a pointcut. An AO-FSM pointcut defines a state-transition pattern (partial FSM pattern) that describes a repeated behavior of the system. We say that the pattern is triggered or matched when the system’s FSM recognizes the pattern. The pattern’s recognition process starts when a state of the system’s FSM matches the first state of the pattern’s FSM. Then, the process continues checking the following states and transitions and compare them with the pattern’s states and transitions. If the process enters the last state of the FSM pattern (it can be a final state), then the pattern is matched and the pointcut is triggered. For instance, we can describe the following pointcut pattern: an initial state $q_0$ linked to a state $q_1$ by the Subject transition, and $q_1$ is linked to a final state $q_2$ by the Object transition. This pattern matches a rule evaluation in the policy’s automaton. This pointcut can be used during the policy evaluation process.

The advice defines a method to be applied to the system’s FSM in order to correct a problem detected by the pointcut. It may insert new states and transitions as well as it may change states’ labels. The around advice can perform custom behavior before and after the method invocation. It is the most powerful kind of advice. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. The around advice may access the return value of a method and it can modify the value and returns back a different value. For instance the advice RemoveRedundancy is triggered around the redundancy pointcut. This advice permits to change the label of the final state matching the Redundancy pointcut. The RemoveRedundancy returns the final state with one authorization label (redundancy-free).

7.3 Anomalies Resolution Strategies

In distributed systems, each organization has its own Policy Decision Point (PDP) that decides which permission is granted to a given subject to perform a specific action on a given object. Therefore, in collaborative systems, they use master PDP which combines all the collaborating policies. Our proposed anomalies resolution strategy is usually associated to the master PDP. The resolution strategy depends on the type of the detected anomalies. In fact, the redundancies are resolved by removing one of the authorizations (removing one rule and keeping the second rule). For ambiguity, we apply prioritization of decisions strategy. However, if the anomaly is classified as a conflict, we apply the context-based approach. A workflow representing the conflict resolution strategy is presented in Figure 7.1.
7.3.1 Priorization of Decisions to Resolve Ambiguities

As it was mentioned in the previous chapter (6), two rules are ambiguous if they match the same request, they have the same authorization, and they have different actions. The automata-based model presented in chapter 4 represents ambiguities by final states with two or more identical authorization (permit or deny) with different actions. For instance, the automaton $A_{H_1,H_2}$ of Fig. 6.4 contains ambiguous rules represented by the following two final states: $(h_1^2; r_1^2)$ and $(h_2^2; r_4^2; )$. The approach presented in this section consists in selecting one of the two decisions that has the greatest priority. We propose two ways of prioritizing decisions presented in Table 7.1, where $>$ denotes has more priority than. The terms $p_a$ and $d_a$ are equivalent to $p_{r,w,c,d}$ and $d_{r,w,c,d}$, respectively. The prioritizing approach is used to select one decision among two. If the two decisions contain the Permit authorization, the approach applies the prioritization with permit. Otherwise, if the two decisions contain the Deny authorization, then the approach adopts the prioritization with deny.

<table>
<thead>
<tr>
<th>Prioritization with deny</th>
<th>$d_a &gt; d_{r,w} &gt; d_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prioritization with permit</td>
<td>$p_a &gt; p_{r,w} &gt; p_r$</td>
</tr>
</tbody>
</table>

Table 7.1: Prioritization of decisions.

For instance, to resolve the ambiguity detected in the match-state $(h_1^2; r_1^2)$ of the automaton $A_{H_1,H_2}$ (Fig. 6.4), we keep $p_{r,w}$ and eliminate $p_r$. As another example, in the match-state $(h_3^2; r_4^2)$, we eliminate $d_r$ and keep $d_{r,w}$. 

Figure 7.1: Workflow representing the anomaly resolution strategy
7.3.2 Context-Based Approach to Resolve Conflicts

In the case of conflict, we need to check the context of the request. According to the most frequent cases of healthcare, we propose three types of context: “emergency”, “critical” and “normal”. The first context corresponds to the case of emergencies where the patient needs a quick intervention of the doctors to save his life. The critical context corresponds to patients with sensitive political positions who need to keep their health state secret. The normal context corresponds to general health cases.

In the case of emergency, the master PDP chooses Permit-Overrides as the appropriate PCA: if one of the policies returns the permit decision, then the master PDP permits the access to the required resource. If the context is critical, the PCA chosen is Deny-Overrides. Finally, if the context is evaluated to normal, the PCA is Deny-Unless-Permit: i.e the master PDP denies all the required accesses unless there exists a policy that permits them. However, if the emergency context is combined with another context, we prioritize the emergency and we apply the Permit-Overrides. For instance, a very important political person that has been severely injured. Here we have both contexts emergency and critical. In order to save lives, we prioritize emergencies.

Next section presents the approach to detect and resolve anomalies within security policies using AO-FSM. The approach uses the automata model presented in chapter 4.

7.4 Resolving Anomalies using AO-FSM

To eliminate anomalies within a security policy, we define a set of FSM singularities (aspects) describing anomalies. There are three types of singularities: (1) A redundancy presented by a final state with identical decisions, (2) An ambiguity presented by a final state with identical authorizations (prohibition or permission) and different actions, and (3) A conflict presented by a final state with two contradictory decisions. This singularities are described by a set of pointcuts as shown in Fig. 7.2. Whenever the policy contains anomalies, the pointcut matching this anomaly type is triggered and then an advice is invoked.

![Different pointcut patterns](image)

Figure 7.2: Different pointcut patterns

There are three different kinds of pointcuts patterns: i) matches a final state that contains two identical decisions (redundancies), ii) matches a final state with two ambiguous decisions,
i.e. two identical authorizations with different sets of actions (ambiguities), and iii) matches a final state with two contradictory decisions (conflicts), where \( A \cap A' \neq \emptyset \).

To resolve these anomalies, we need to specify a set of resolution advices such as the ones presented in Fig 7.3. Each advice corresponds to one of the pointcut (i.e (i), (ii), and (iii)) and removes the detected anomaly from the Policy’s FSM. The advices implement the anomalies resolution strategies presented in section 7.3. To simplify, In this figure we use the term all to refer to all the actions (read, write, create and delete). Hence, \( A \subset \text{all} \) is equivalent to \( A \subset \{ \text{read}; \text{write}; \text{create}; \text{delete} \} \).

Figure 7.3: Different advice patterns.

There are 7 different kinds of advices represented by letters in figure 7.3. Each advice presents one anomaly’ resolution strategy. Advice (a) resolves the redundancies, advices (b), (c), (d), and (e) resolve the ambiguity anomaly as follows: (b) and (c) implement the deny prioritization of Table 7.1, while (d) and (e) implement the permit prioritization of Table 7.1. The advices (f) and (g) resolve the conflict anomaly. The advices change the final state’s label based on the adequate resolution strategy proposed in section 7.3. The changes are applied as follows:

(a) This advice removes one of the identical decisions.

For instance, the state \( (h_{62}^2; r_{6}^2) \) of the automaton \( A_{H_1,H_2} \) of Fig. 6.4 contains two redundant rules with the same decision \( (d_a; d_a) \). So to eliminate this redundancy, we keep only one of them. Hereafter, we use the final states of the automaton of Fig. 6.4 as examples for the rest of advices.

(b) If the final state contains two deny authorizations with two different actions read and read/write, then the advice removes the \( d_r \) decision and sets the state’s label to \( d_{r,w} \).

For instance, the state \( (h_{43}^2; r_{4}^2) \) of the automaton of Fig. 6.4 represents an ambiguity illustrated by the two decisions: \( d_r \) and \( d_{r,w} \). To resolve this ambiguity, advice (b) removes \( d_r \) and keeps \( d_{r,w} \). Hence each read or write request leading to this final state will be evaluated as deny.

(c) If the final state contains \( d_a \) (\( d_{r,w,c,d} \) and any other deny decision (\( d_r \) or \( d_{r,w} \)), then the advice sets the state’s label to \( d_a \). This means that all the actions leading to this final state
will be denied.

(d) If the final state contains two permit decisions that have two actions sets \{read\} and \{read, write\}, then the advice removes \(p_r\) and sets the state’s label to \(p_{r,w}\).

For example to eliminate the ambiguity that occurs in the state \((h^2_1, r^2_1)\), the advice removes \(p_r\) and keeps \(p_{r,w}\).

(e) If the final state contains \(p_a\) \((p_{r,w, c,d})\) and any other permit decision \((p_r\) or \(p_{r,w}\)), then the advice sets the state’s label to \(p_a\). This means that all the actions leading to this final state will be permitted.

(f) If the final state contains two contradictory decisions, then the advice chooses the permit decision. For instance, if we have the two decisions \((p_{r,w}; d_r)\), the advice chooses the decision \(p_{r,w}\). This implements the permit-overrides strategy (section 7.3) to resolve the conflict. Using the permit-overrides strategy, states \((h^2_2, r^2_2), (h^2_3, r^2_3), (h^2_4, r^2_4), \) and \((h^2_6, r^2_3)\) will have the following access permissions respectively: \(p_{r,w}, p_a, p_{r,w}, p_a, \) and \(p_r\).

(g) If the final state contains two contradictory decisions, then the advice chooses the deny decision. For instance, if we have the two decisions \((p_{r,w}; d_r)\), the advice chooses the decision \(d_r\). This implements the deny-overrides strategy (section 7.3) to resolve the conflict. With the deny-overrides strategy, states \((h^2_2, r^2_2), (h^2_3, r^2_3), (h^2_4, r^2_4), \) and \((h^2_6, r^2_3)\) will have the following decisions respectively: \(d_{r,w}, d_r, d_a, d_a, \) and \(d_a\).

Next section presents space and time complexities evaluation of the anomalies resolution strategies.

### 7.5 Evaluation of Space and Time Complexities

The results of this section are adaptations of results from \[59, 60\]. Let us consider the following parameters:

- \(n\) be the number of rules of a policy.
- \(d_1\) and \(d_2\) be the numbers of bits to code the subjects and objects, respectively.
- Maximum number of subjects and objects respectively are \(2^{d_1}\) and \(2^{d_2}\).
- \(\mu\), the number of great fields.
- \(\delta\), the sum of the number of bits to code the small fields.

The anomaly resolution strategy uses the results of anomalies detection process. Therefore, the time and space complexities of anomalies resolution equals the space and time complexities of
detecting the anomalies and resolving them. Therefore, we can obtain the following proposition (its proof is presented in appendix C):

**Proposition 11** The space and time complexities of resolving anomalies are in $O(n^{\mu+2} \times 2^4)$, which is bounded by $O(n^4)$ and $O(n^2 \times 2^D)$.

Next section presents implementation details and confirms the calculated complexities.

### 7.6 Implementation and Evaluation

To validate the resolution approach, we have implemented a prototype of AO-FMS in the Groovy language [62] using the POPART framework [37]. POPART allows the user to embed DSLs and to develop aspect-oriented extensions for those DSLs in form of plug-ins. Basically, the DSL language is implemented as a library, instead of implementing it with a parser and a compiler. Then, the DSL programs are evaluated by invoking this library. An example of groovy code representing the policy of Table 2.1 is presented in Listing 1 in the appendix D.

The joinpoint’s context capture some information that allows the definition of the joinpoint. For instance, `FinalStateJoinPoint` (see List. 2 of appendix D) get the names of the final states of the FSM. Another example of joinpoints is the `ExitStateJoinPoint` (see List. 3 of appendix D). When the system wants to exit a state, it has to capture information related to: the name of the current State, the name of owning FSM, the event that invoked the system to exit the state, the outgoing transition corresponding to the event, and finally the next state.

Each pointcut is implemented as a class that inherits the `Pointcut` class. In FSM-DSL, pointcut sub-classes match the current state parameters with the context of a corresponding points of execution in the base code (joinpoint). The pointcut returns “true” if the pointcut matches and “false” if not. For instance, the pointcut `RedundancyPC` (shown in List. 4 of appendix D) checks if the current state is a final state and contains redundancies. If it detects at least one redundancy, then the pointcut matches and returns “true”. For instance, when the system enters the final state $(r_6^2; h_6^2)$ of $A_H$, it recognizes the `RedundancyPC` pointcut pattern which returns true.

The last part of the implementation is the advice language. This language represents methods that react over the parts of the FSMs that match the pointcuts. The advice language of AO-FSM resolves the detected anomalies using a set of methods. For instance, the `removeRedundancyAdvice` (shown in List. 5 of appendix D) overrides the name of the final states and eliminates the detected redundancies.

We evaluated the efficiency and effectiveness of the policy resolution approach for XACML policies using 9 synthetic generated policies of 100 to 4000 rules. Our experiments were performed on an an Intel Core i5 CPU 2.3 GHz with 4 GB RAM. Fig. 7.4 presents the time
required for anomaly detection and resolution as a function of policy size. The graph explicitly shows that the execution time for detecting and resolving anomalies increases with the number of policy rules. This is due to the fact that on average, the number of anomalies increases with the number of rules. Obviously, the number of detected and resolved anomalies (presented in Fig. 7.5) impacts the detection and resolution time. For instance, with 100 rules, it takes 0.16 s to detect and resolve 13 anomalies, while it takes 0.62 s for 1000 rules to detect and resolve 398 anomalies, and 1.97 s for 4000 rules to detect and resolve 797 anomalies. These results demonstrate the performance efficiency of the analysis process for reasonable size of policies.

### 7.7 Chapter Summary

In this chapter, we have presented an anomaly resolution strategy based on a formal approach to detect and resolve different types of anomalies within a distributed software system. The approach is based on a formalism for aspect-oriented state machines (AO-FSM) and language implementation AO-FSM based on finite-state machines. The proposed resolution strategy consists of two different approaches: prioritization of permissions and a context-based approach. The selection of the appropriate approach depends on the type of the detected anomaly. The approach uses aspect-oriented state machines to intercept, prevent, and dynamically manipulate rules that cause conflicts. We presented time and space complexities evaluation which were confirmed by the experimental results. Next chapter presents the software architecture of the middleware that regroups the different proposed approaches presented in this thesis.
Chapter 8

Software Architecture

8.1 Introduction

The previous chapters described our proposed automata-based approaches which present the functional behavior of the middleware curlX. In this chapter, we describe the software architecture of the testbed that we rely on to evaluate the efficiency of the proposed approaches. More specifically, we describe in sect. 8.2 the global software architecture which presents the different features offered by the middleware and their interactions. In sections 8.3, 8.4, 8.5, 8.6, 8.7, and 8.8, we present in detail the components’ architecture of each feature. Finally sect. 8.9 summarizes the chapter.

8.2 Global Software Architecture of curlX Middleware

Most of the current Cloud architectures do not offer to the users the possibility to manage their own security policies. In fact, the cloud data migration imposes the outsourcing of security management. Therefore, in this thesis we proposed a cloud middleware denoted curlX that facilitates the XACML policies management within cloud environments. As it was presented in sect. 5.2.1, curlX consists of three main components: (1) the translator, (2) CPVS: cloud policy verification service, and (3) X2Automata for the policy evaluation process. These implemented components offer a set of features (functionalities) allowing the modeling, the study and the evaluation of the XACML security policies within a cloud environment based on the automata-based approaches studied in this document. The middleware offers eight main functionalities (Fig. 8.1), namely:

- **Curl primitives Translation (CPT):** It translates the two types of curl primitives (presented in chapter 2) into XACML syntax. The primitives include the curl requests to access cloud data, and the curl ACLs to add and update a policy associated to the data.
• **Policy Automata Generation** (PAG): It models the XACML policy by an automaton using the synthesis procedure described in chapter 4.

• **Access Automaton generation** (AAG): It is based on the synthesis procedure, the middleware can generate an automaton for each request. This automaton is used during the policy evaluation process of chapter 5.

• **Policy Evaluation** (PE): This feature evaluates the XACML policies based on the automata-based approach proposed in chapter 5.

• **Completeness Verification** (CV): It verifies if a XACML policy is complete using the automata-based approach proposed in chapter 6.

• **Intra-policy Anomalies Detection** (Intra-PAD): Using the automata-based approach of chapter 6, this feature detects possible anomalies (conflict, ambiguity and redundancy) in a XACML policy.

• **Inter-policies Anomalies Detection** (Inter-PAD): Using the automata-based approach of chapter 6, this feature detects anomalies (inconsistency and similarity) during the composition of several XACML policies.

• **Anomalies Resolution** (AR): It permits the resolution of both intra and inter-policies anomalies using the strategy proposed in chapter 7.

Figure 8.1 summarizes the main possible scenarios of use of the various features of the framework. This figure clearly shows that the Policy Automata Generation (PAG) feature is the key element to all other aspects studied in this thesis. In fact, the automaton model generated by this feature is the basis of all the proposed approaches related to policies’ studies including completeness verification, policy evaluation, anomalies detection and anomalies resolution.

**Curl primitive Translation** (CPT) feature writes XACML policies. The later are then modeled by an automaton using the PAG feature. The resulting automaton is used for all the policy’s studies and evaluation. The extraction of the automaton’s final states permits the verification of the completeness property, as well as the intra-policy anomalies detection. Furthermore, the composition of different generated automata facilitates the inter-policies anomalies detection. On the other hand, the composition of the policy’s automaton and the access’ automaton guarantees an efficient policy evaluation process.

Next sections present the components’ architecture of each feature shown in Fig. 8.1. The components’ architecture proposes usage scenarios of each feature, and describes the interaction between different features.
Figure 8.1: Global architecture of the middleware
8.3 Curl Primitive Translation (CPT)

The cloud user can be either a regular cloud user or a cloud administrator. The administrator is responsible for managing security policies associated to objects stored in the cloud using a set of curl ACLs. On the other side, a regular cloud user may send a request to access data stored in the cloud. This request is usually expressed using curl primitives (chapter 2). The Curl Primitive Translation (CPT) feature translates the users curl requests into XACML syntax. Figure 8.2 illustrates the main components of CPT and its interactions with its environment. The cloud user can be assigned to two different roles namely: Data Requestor (corresponding to the regular cloud user) and Administrator. The data requestor sends an access request to the curl2XACML Generator which translates the request into a XACML request to access a medical data stored in the cloud. As for the administrator, he sends ACLs to the Swift ACLs Generator which rewrites the ACLs into XACML syntax and then writes or updates the XACML policy file associated to the medical data.

![Figure 8.2: Curl Primitive Translation (CPT) and its interactions.](image)

8.4 Policy Automata Generation (PAG)

The Policy Automaton Generation feature permits the generation of automata modeling XACML security policies. This feature, shown in Fig. 8.3, is composed of three main components namely: Policy Retrieval, Rule Automaton Constructor, and Policy Automaton Constructor.

The feature takes as input a XACML policy file, then the Policy Retrieval extracts the important attributes from each rule in the file (i.e. subject, resource, action and effect). The extracted attributes of each rule are then used by the Rule Automaton Constructor to generate an automaton modeling a single rule. The automata of all rules are combined by the Policy Automaton Constructor which produces an automaton modeling the global XACML policy. This policy’s automaton is then used for the policies’ studies and evaluation offered by other features.
8.5 Policy Evaluation (PE) and Access Automata Generation (AAG)

The Policy Evaluation (PE) feature is always associated to Access Automaton Generation (AAG) feature. AAG feature consists of two main components: Attribute Retrieval and Access Automaton Constructor. While PE is composed of one component denoted Policy Decision Point. Figure 8.4 presents the interactions between the components of the two features. An access requester may send a XACML request to the Attribute Retrieval component of AAG which retrieves the most important attributes (i.e. subject, resource, action). The Attribute Retrieval sends the extracted attributes to the Access Automaton Constructor which constructs the corresponding automaton. The Policy Decision Point of PE uses both access's automaton and policy's automaton to take the decision about the requested access.

8.6 Completeness Verification (CV)

The Completeness Verification (CV) feature verifies if the policy given as input is complete or not. Figure 8.5 presents CV and its interactions. It consists of three main components namely: FinalStates Extractor, Non-Match State Detector, and Default-Rule Generator. The first component is served (takes as input) by the policy automaton generated by PAG, from which extracts its final states. The set of those final states then serve the Non-Match State
Detector. If the latter finds the "non-match state" among the set of final states, the Default-Rule Generator is triggered to make the XACML policy complete by adding a default rule to it.

Figure 8.5: Completeness Verification (CV) and its interactions.

8.7 Intra-Policy Anomalies Detection Intra-PAD

The Intra-Policy Anomalies Detection (Intra-PAD) feature permits the detection of three types of anomalies within XACML policies namely: conflicts, ambiguities and redundancies. Figure 8.6 presents the different components of Intra-PAD and its interaction with the Anomalies Resolution (AR) feature. The Intra-PAD is composed of four main components: (a) FinalState Extractor extracts the final states from the Policy’s Automaton, (b) Redundancy Detector analyses the extracted final states to detect the redundant rules, (c) Ambiguity Detector detects ambiguous rules using the final states, and (d) Conflict Detector detects the conflicting rules by analyzing the extracted final states as well. A list of redundant, ambiguous and conflicting rules are then generated and served to the first component of AR denoted AspectGenerator. The latter generates a set of joinpoints where the redundancy and conflict pointcuts occur. Then, the joinpoints trigger a set of methods provided by AdviceConstructor to resolve the detected anomalies. Finally, as an output of the AR feature, a new policy’s automaton is generated with no conflicts and no redundancies. This automaton is used to update the XACML policy in the policy repository.

Figure 8.6: Intra-Policy Anomalies Detection (Intra-PAD) and its interactions with PAG and AR.
8.8 Inter-Policies Anomalies Detection *Inter-PAD*

The Inter-Policies Anomalies Detection (*Inter-PAD*) feature detects anomalies during the collaboration of two or more security policies. It detects two types of anomalies: inconsistencies and similarities. The feature consists of four main components as shown in Fig. 8.7: (a) the *synchronous composition* component is responsible for composing the policies’ automata using the synchronous composition operator (chapter 4) to generate a global automaton, (b) *FinalStates Extractor* extracts the final states of the global automaton, (c) *Similarity Detector* uses the extracted final states to detect the similarity anomalies, and (d) *Inconsistency Detector* uses the final states to detect inconsistencies between the composed policies. The list of the detected anomalies is then sent to the *AR* feature which removes the anomalies and returns an anomaly-free policy’s automaton. The latter is then used to update the original XACML Policies.

![Figure 8.7: Inter-Policies Anomalies Detection (*Intra-PAD*) and its interactions with PAG and AR.](image)

8.9 Summary

In this chapter, we presented the software architecture of the suggested cloud middleware curlX. The architecture encompasses the features offered by the middleware and their interactions. Furthermore, we presented components’ architectures of each feature to describe the middleware usage scenarios. The architectures were validated using Archimate tool. The implementation of the proposed architecture was made using Java programming language and openstack core. The implementation is used for performance analysis of the proposed approaches.
Chapter 9

Conclusion and Perspectives

9.1 Conclusion

This thesis has presented an approach using XACML and automata formalisms to manage policies within e-health cloud. The approach consists in first describing policies in XACML and then translating the XACML models into automata. The latter are then used for different policy studies: (a) policy evaluation process, i.e. a process that checks whether a request satisfies at least one rule of a XACML policy, (b) policy anomalies detection and completeness verification, and (c) anomalies resolution. The approaches were integrated, as services, in a cloud middleware denoted curlX which is composed of: (1) curlX translator: translates curl primitives into XACML requests and rules, (2) X2Automata: permits the policies evaluation, and (3) CPVS is a cloud policy verification service that detects and resolves anomalies in policies. The anomalies resolution strategy uses Aspect Oriented Programming.

We started by reviewing existing works related to security policies specification. There are several policy specification languages that can be categorized into three main categories: logic-based languages (e.g. D2LP and RT), object-based languages (e.g. ponder), and XML-based languages (e.g. XACML). Even though there are several specification languages, XACML remains the most adequate language for distributed systems. Yet, in large scale systems such as Cloud computing, XACML becomes complex and hard to manage.

Several research works have been proposed to represent XACML policies differently in order to facilitate their management. They have been represented as: BDD (Binary Decision Diagram), UML, coq, graph representation, CTL (Computation Tree Logic), ect. However, we noticed that most of the XACML representations were proposed for a specific policy’s study. For instance, BDD permits the detection of anomalies within a single security policy (conflicting rules and redundancies). On the other hand, the coq technique permits only the detection of conflicting rules. Therefore we proposed a generic automata-based model which is used for different security policies’ studies.
Policy specification languages are ways to represent rules, but cannot respond to all the system’s security requirements. Therefore, there is always a need for access control models. In the literature, there are several access control models such as: MAC, DAC, TBAC, RBAC and ABAC. Therefore, the ultimate challenge is to associate an access model to a specification language. In this direction, there were research works that developed an RBAC profiles for XACML. These profiles map the most important components of RBAC into the XACML hierarchy, which facilitates the inter-organizations communications and data sharing. However, such collaborations may provoke policies anomalies such as similarities and inconsistencies.

The automata-based model consists in describing XACML policies by automata. The construction of the automaton is done in four steps. The first step extracts from each rule a set of attributes: effects, subjects, resources, and actions. These attributes are then used in the second step to construct the rule’s automaton consisting of four states: one initial state $x_0$, state $x_1^i$ and two final states: $x_2^i$ and $x^\#$ $i$. The transitions are labeled S and O (subjects and objects respectively), and finally the access permission $u_A$ is associated to the final state $x_2^i$. Since theses automata do not have the same set of alphabets, the third step unifies their alphabets. Then the automata are composed in the fourth step to have global policy’s automaton. This model is used then to manage security policies: policy evaluation process and anomalies detection and resolution process.

XACML policy evaluation process is one of security policies management processes. It consists in checking whether the requests attributes match a rule in the policy. Our automata-based approach for the policy evaluation process consists in modeling the request by an automaton, and compose it with the policy’s automaton. The policy evaluation process is reduced to the comparison of the requested action and the action imposed by the policy. The comparison occurs in the final state level. If the action is granted (included) by the access permission, then we say that the request satisfies the policy. The approach was integrated in the curlIX middleware as a policy decision engine denoted X2Automata. Our experimental results proved a significant improvement in terms of time evaluation compared to XACML implementation Balana.

Detecting anomalies within a security policy is considered as a policy management process. In this thesis, we propose two anomalies’ categorizations. The first categorization splits the anomalies into three types: conflicting anomalies (conflicts) and non-conflicting anomalies (redundancies and ambiguities). The second categorization is based on the number of security policies: intra-policy detects anomalies within a single policy (e.g. redundancies, ambiguities, and conflicts), and inter-policies detects anomalies within multiple policies (e.g. similarities and inconsistencies). We proposed and automata-based approach to detect these anomalies. Final states consists of authorizations under the form $u_A$ where $u$ presents the authorization and
A presents the action that needs to be performed. The automata-based approach consist in analyzing the final states of the policy’s automaton. The advantage of this model is that it can detect several types of anomalies using the same model. The detection process was implemented as a cloud Service in the proposed middleware. The service is denoted CPVS (cloud policy verification service).

The automata-based model was useful for the resolution of anomalies as well. For the resolution strategy, we proposed a dynamic strategy that selects the adequate XACML policy combining algorithm based on: the type of the anomaly and the context of the request. The resolution strategy was implemented using Aspect Oriented Finite State Machine (AO-FSM). We define a set of pointcuts (final state with two authorizations) representing the policies anomalies. Then we define a set of advices (methods) in order to eliminate these anomalies (remove a label or add a label).

Finally, we presented the curlX middleware software architecture that describes the different functionalities and features offered by the middleware and their interactions. Each feature represents an implementation of one of the proposed automata-approaches. The implementation of the different system’s features allowed us to obtain a sort of a validation of the theoretical concepts, and on the other hand, it is considered as an exploitable end product.

## 9.2 Perspectives

The work of this thesis can be extended to several future research directions. Regarding theoretical aspects, an interesting one to consider consists in managing the delegation of privileges in e-health collaborative systems [58]. Delegation is often used in inter-organizational communications. It depends usually on the role of a user within the organization. It permits to grant access rights under certain temporal conditions. However, it needs to be controlled in order to guarantee the integrity and confidentiality of medical data. During the delegations, the security policies are often changing, which may raise several conflicts and anomalies.

Furthermore, the era of Internet of Things (IoT) can promote very interesting extensions of this work. For instance, the security analysis in the fog computing [4] can be a continuity of this thesis. Substantially, most of the current works [27, 97] propose to push the cloud storage to the edge devices in IoT, which raises several challenges in terms of security and untrusted supply chains. AO-FSM approach can be adapted to provide an on the fly analysis and management to overcome these challenges.

Smart transportation is another direction to extend this work. In fact, replacing the driver by autonomous cars pushes actually the software to manage different exceptions [63] such as: weather catastrophes (storms, tornadoes, ...), traffic violations (driving red lights), or animals
hazard (deer, armadillos, caws). The behavior of these exceptions is mostly common and known, hence AO-FSM can be an interesting solution to manage these exceptions and prevent accidents. AO-FSM can be also used for the management of cars’ collusion and road conflicts.

The detection and the resolution of anomalies in the context of big data is another extension. We are working on a software to manage large scale data using the clustering technique presented by [20, 41, 40]. The technique consists in gathering similar rules in a single cluster, then the anomalies are eliminated using the suppression of the conflicting rule.
Bibliography


Appendices

A Proof of Proposition 9

The proof of Proposition 9 corresponds to the proof of Prop. 12 of [59], where we take parameter m (number of fields) equal to 2. We use the notation $\Psi_i = \min(2^{d_i}; n)$. We omit the complexity of Step 1 because it needs a fixed, and finite time $O(1)$.

A.1 Complexity of Step 2

The space and time complexities to construct one state or one transition of $\mathcal{A}_i$ are in $O(1)$. Each $\mathcal{A}_i$ contains 4 states and a limited number of transitions from each state. Hence, the space and time complexities to construct each $\mathcal{A}_i$ are in $O(1)$. Since we have to construct $n$ automata, the space and time complexities of Step 2 are in $O(n)$.

A.2 Complexity of Step 3

This step consists in replacing each set of objects and subjects by the corresponding transitions. The number of transitions from $r_i^j$ of $\mathcal{A}_i^*$ is bounded by both $O(2^{d_j})$ and $O(n)$ which means $O(\Psi_i)$. The bound $O(2^{d_j})$ is because $2^{d_j}$ is the number of possible values of either subjects or objects, which is necessarily $\geq$ than the number of transitions from $r_i^j$. Hence, the space and time complexities to construct all the transitions of $\mathcal{A}_i^*$ are in $O(\Psi_0 + \Psi_1)$. Therefore, the space and time complexities to construct all the $\mathcal{A}_i^*$ are in $O(n \times (\Psi_0 + \Psi_1))$.

A.3 Complexity of Step 4

Let us consider the construction of $\mathcal{A}_F$ in Step 4 level by level, where the states of level $i$ are those reached after $i$ transitions from the initial state. At each level $i$, the transitions links level $i-1$ to level $i$. The space and time complexities to construct a state $r = \langle r_1; \ldots; r_n \rangle$ of $\mathcal{A}_F$ are in $O(n)$, because we need to construct and store the $n$ components of the state. The space and time complexities to construct a transition between two constructed states of levels $i$ and $i+1$ are in $O(1)$, because we need to store the label of the transition.
**Level 0:** The unique state is the initial state \( r^0 = <q_1^0; \ldots; q_n^0> \). The space and time complexities of its construction are in \( O(n) \).

**Level 1:** Using the same reasoning as in the proof of Step 3, the number of transitions from \( r^0 \) is in \( O(\Psi_0) \). Hence, the number of states at level 1 is in \( O(\Psi_0) \). Therefore, the space and time complexities to construct all the transitions from level 0 to level 1 are in \( O(\Psi_0) \), and the space and time complexities to construct all the states at level 1 are in \( O(n \times \Psi_0) \).

**Level 2:** The number of transitions from each state of level 1 is in \( O(\Psi_1) \). Since the number of states of level 1 is in \( O(\Psi_0) \), we obtain that the number of states at level 2 and the number of transitions from level 1 to level 2 are in \( O(\Psi_0 \times \Psi_1) \). Therefore, the space and time complexities to construct all the states at level 2 are in \( O(n \times \Psi_0 \times \Psi_1) \), and the space and time complexities to construct all the transitions from level 1 to level 2 are in \( O(\Psi_0 \times \Psi_1) \). At each level \( j \), the number of states is also bounded by \( 2^n \), because each state is defined by \( n \) 2-value components \( r_i \) (\( r_i = q_i^j \) or \( r_i = E_i \), for \( i = 1 \ldots n \)). But this bound has no influence due to the assumptions \( n > D \) and \( 2^n > n^2 \).

**All levels:** By adding the complexities of all levels, we obtain that the space and time complexities of constructing \( A_F \) are in \( O(n \times \Psi_0) + O(n \times \Psi_0 \times \Psi_1) \).

From \( d_i \geq 1 \) and \( n > D \), we obtain \( \Psi_i \geq 2 \), from which we deduce that \( \Psi_0 + (\Psi_0 \times \Psi_1) \leq 2 \times (\Psi_0 \times \Psi_1) \). Hence, the space and time complexities of constructing \( A_F \) are in \( O(n \times \Psi_0 \times \Psi_1) \).

**Associating permissions:** It remains to compute complexities of associating permissions to the match states of \( A_F \). The space complexity of associating a permission to a match state of \( A_F \) is in \( O(1) \), because we only need to store the permission associated to the state. The time complexity of associating permissions to all match states of \( A_F \) is in \( O(n) \), because we may need to consult the \( n \) components of the state.

**A.4 Total complexity**

Since Steps 1 to 3 are less complex than Step 4, we obtain that the space and time complexities for constructing \( A_F \) are in \( O(n \times \Psi_0 \times \Psi_1) \). By definition of \( \mu \) and \( \delta \), we obtain \( n \times \Psi_0 \times \Psi_1 = n^{\mu+1} \times 2^\delta \), which can be easily shown to be smaller than both \( n^3 \) and \( n \times 2^D \).

**B Proof of Proposition 10**

As we mentioned in section A, the number of states at level 2 which represents final states of the automaton is in \( \Psi_0 \times \Psi_1 \). During the detection of anomalies (conflicts and redundancies), we compare for each final state two labels (i and j). Hence, the space and time complexities of detecting anomalies within a single final state are in \( O(n^2) \). As result, the space and time complexities of detecting anomalies in the automaton are in \( O(n^2 \times \Psi_0 \times \Psi_1) \). By definition of
\( \mu \) and \( \delta \), we obtain
\[ n^2 \times \Psi_0 \times \Psi_1 = n^{\mu+2} \times 2^\delta, \]
which can be easily shown to be smaller than both \( n^4 \) and \( n^2 \times 2^D \).

C Proof of Proposition 11

The space and time complexities of associating a decision to a state of an automaton are in \( O(1) \), because we only need to write and store the decision associated to the state once. Since there is the possibility to have anomalies in all the final states, then the space and time complexities of inserting a new label for the anomaly-prone states are in \( \Psi_0 \times \Psi_1 \) (number of final states).

The time and space complexities of detecting anomalies are in \( O(n^2 \times \Psi_0 \times \Psi_1) \). Therefore, the time and space complexities of the anomalies resolution are in \( O(n^2 \times \Psi_0 \times \Psi_1) = n^{\mu+2} \times 2^\delta \).

We obtain: \( O(n^{\mu+2} \times 2^\delta) \), which is bounded by \( O(n^4) \) and \( O(n^2 \times 2^D) \).

D Automata-Domain Specific Language

D.1 Domain-Specific (DSL) to present Policy’s FSM

Listing 1 presents an example of the groovy code representing the hospital security policy of Table 2.1.
Listing 1: the groovy FSM representing the hospital’s policy of Table 2.1.
D.2 Domain-Specific JoinPoint Language

Listing 2 presents the `FinalStateJoinPoint` code that allows the context to gather the information about the final states of the names of the final states of the FSM.

```java
protected void prolog() {
    if (DEBUG) println("INSTRUMENTATION (MOP): \t prolog \${instrumentationContext.args[0]}")
    joinPointContext = new HashMap();
    StateMachine thisStateMachine = (StateMachine)instrumentationContext.receiver;
    joinPointContext.FinalStateNames = thisStateMachine.getFinalStatesNames();
    joinPointContext.FinalStates = thisStateMachine.getFinalStates();
    joinPoint = new StartingStateMachineJoinPoint("", joinPointContext);
    joinPointContext.thisJoinPoint = joinPoint;
}
```

Listing 2: Groovy code of the FinalStateJoinPoint

Listing 3 presents the `ExitStateJoinPoint` code that allows the context to gather information related to: current state, owning FSM, triggering event, outgoing transition, and finally next state.

```java
protected void prolog() {
    if (DEBUG) println("INSTRUMENTATION (MOP): \t prolog \${instrumentationContext.args[0]}")
    joinPointContext = new HashMap();
    StateMachine thisStateMachine = (StateMachine)instrumentationContext.receiver;
    String event = (String)instrumentationContext.args[0];
    joinPointContext.thisStateMachine = thisStateMachine;
    joinPointContext.event = event;
    joinPointContext.thisCurrentState = thisStateMachine.getCurrentState();
    joinPointContext.nextState = thisStateMachine.getCurrentState().handleEvent(event).fire();
    joinPointContext.thisTransition = thisStateMachine.getCurrentState().getTransitions().get(event);
    joinPoint = new EventReceivingJoinPoint("", joinPointContext);
    joinPointContext.thisJoinPoint = joinPoint;
}
```

Listing 3: Groovy code of the ExitStateJoinPoint

D.3 Domain-Specific Pointcut Language

Listing 4 presents the `RedundancyPC` pointcut that detects redundancies in the FSM.

```java
public class RedundancyPC extends Pointcut {
    public RedundancyPC() {
        super("pRedundancy");
    }

    @Override
    public boolean match(JoinPoint jp) {
        return (jp instanceof FinalStateMachineJoinPoint) &&
            (((FinalStateMachineJoinPoint) jp).getFinalStateNames().toString().matches("(?i).*p_.+;p_.+||(?i).*d_.+;d_.+"));
    }
}
```

Listing 4: Excerpt of RedundancyPC to define final states with redundancies
D.4 Domain-specific Advice Language

Listing 5 advice that eliminates the redundancies within final states in the FSM. The advice removes one of the access authorizations according to the resolution strategy of Table 7.1.

```java
@After("RedundancyPC")
public void removeRedundancyAdvice(FinalStateMachineJoinPoint finalState){
    RedundancyPC testRedundancyPC = new RedundancyPC();
    if(testRedundancyPC.match(finalState)){
        for (int i = 0; i < finalState.getFinalStateNames().size(); i++)
            switch (finalState.getFinalStateNames().get(i))
            {
                case "p_r;p_r,w":
                    finalState.getFinalStateNames().set(i,"p_r,w");
                    break;
                case "p\emptyset;p_r":
                    finalState.getFinalStateNames().set(i,"p_r");
                    break;
                case "d\emptyset;d_r":
                    finalState.getFinalStateNames().set(i,"d_r");
                    break;
                case "d_r;d_r,w":
                    finalState.getFinalStateNames().set(i,"d_r,w");
                    break;
                default:
                    return proceed();
            }
    }
}
```

Listing 5: Implementation of removeRedundancyAdvice