PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Exploring Dynamic Constraint Enforcement and Efficiency in Access Control

Fatih Türkmen

Advisor: Prof. Dr. Bruno Crispo

February 2012

Abstract

Dynamic constraints such as Separation of Duty (SoD) prevent the possibility of frauds and enable flexible protection of sensitive resources appearing in active contexts. They are enforced in various ways depending on the access control model and the application. Role based access control (RBAC) employs restrictions on the activation of roles and the exercise of permissions by individuals for enforcing the constraints. However, whether a constraint specification correctly enforces a given dynamic policy is an open research question. This is mainly due to the nature of the dynamic constraint enforcement: a constraint satisfied at a state can be violated at a future state as a result of the event sequences occurred in between. Moreover, the support of dynamic enforcement usually imposes low level extensions to the implementation, which in return requires another level of verification. In the approaches that tackle this problem at run-time, efficiency is a key concern.

In this dissertation, we present two approaches for analyzing and enforcing dynamic constraints. The first is employed off-line *and is based on software testing features available in software model checkers. The relevant components of an access control system are modeled as a software and the execution of this software mimics the RBAC run-time. A software model checker is used to check some properties that represent constraint specifications and the actual authorization policies encoded in eXtensible Access Control Language (XACML). We demonstrate our approach by using an open source software model checker, Java Path Finder (JPF) [Cena], and its sub-projects for different testing scenarios. In this first approach, efficiency is not the main concern but coverage is.*

The second approach relies on a propositional satisfiability (SAT) based run-time procedure to replace the conventional policy evaluation in RBAC systems. Efficiency and flexibility are the prominent features of this approach. Efficiency is obtained by dividing the steps involved in policy evaluation into on-line *and* off-line. *On-line steps correspond to request answering in conventional policy evaluation and have to be done at run-time. Off-line steps can be performed as pre-processing or post-processing of the on-line steps and have no effect on policy evaluation performance. We experimentally show that our approach is efficient and scales well in realistic scenarios.*

The final chapter of the thesis presents an extensive study of XACML policy evaluation performance. Policy evaluation corresponds to a function, $Eval(Policy, Request)$, that takes a policy and a request as input, and produces an access control decision. Our experimental results show that the Eval function can create a bottleneck in application domains where the number of policies and rules is large. We present a list of optimization techniques that can speed up the evaluation performance.

Keywords
[Access Control, Run-time, Constraint Enforcement, Verification, XACML Performance]

Acknowledgements

First of all, I would like to thank my advisor, Bruno Crispo, for his guidance and patience during all years of my PhD. Without his support, this work would not be possible.

A lot of details on the first part of the thesis have been developed from discussions in the Java Path Finder (JPF) mail group. My prototype implementation extensively uses JPF and its sub-projects. I would especially like to thank JPF maintainers from NASA, Darko Marinov and Steven Lauterberg for their patient answers to my, somewhat, naive questions. This work could not be realized without their helps.

Much gratitude goes to security groups at the University of Trento and Fondazione Bruno Kessler (FBK). In particular, the contributions of Alessandro Armando and Silvio Ranise from FBK have been priceless. The development of a constraint aware run-time decision procedure would not be possible without our intensive discussions as my understanding of boolean satisfiability (SAT) have been deeply influenced from them. I wish to thank Daniel Le Berre for his support in the use of SAT4J in my implementation.

Discussions with Lalana Kagal and Xiaoqian Jiang helped me a lot to understand the XACML policy evaluation process. I am also grateful to Seth Proctor for his comments on my XACML policy evaluation experiments. I also appreciate the useful discussions with Mario Lischka during my time at NEC. I improved a my policy generator to a more sophisticated tool.

I wish to thank my thesis committee members; Fausto Giunchiglia, Alessandro Armando and Benjamin Aziz .

I am extremely lucky to have persistent support of my family. I can not remember how many times a simple talk with them at a difficult moment turned into a fresh start.

Most importantly, I am grateful to Arianna Bisazza for standing on my side during the course of my PhD and the thesis writing. I can not thank you enough for your patience, love and encouragement. You made this tough path a joy.

Majority of this work have been financially supported by funding from University of Trento and FBK.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today the Internet has become a commodity rather than a privilege allowing people to share information, governments to provide services to their citizens and companies to perform commercial activities. The larger it grows the more information is exposed to it. Nowadays, it is not a surprise to hear the governmental secrets are unleashed [Wik] or the private information of an individual is misused. The detection of frauds or unauthorized access to sensitive information has changed the pace of security in this decade. The owners of sensitive information have been long concerned with the question “How do we manage authorizations under certain constraints?”

Historically, organizations employed security measures for the regulation of data access or completion of a sensitive job function in two perspectives: a strong model of access control and an expressive policy specification language. Without a proper model, the organizational needs cannot be captured and without an expressive language the rules of access cannot be encoded properly. Extensive research has been conducted on both aspects leading to a certain level of maturity. Role based access control (RBAC) has emerged as the dominant model, and eXtensible Access Control Markup Language (XACML) as the language. In fact, a dedicated line of literature and various prestigious conferences and journals have been established for their research.

The well known security principles (e.g. least privilege or complete mediation) play a key role in the design of authorization systems, when choosing the associated components and the underlying machinery. Domain-specific security measures, on the other hand, require certain extensions for their support as some of them exist independently of the model. In the literature, these measures usually appear as security policies or constraints. Examples of such constraints include Dynamic Separation of Duty (DSoD), Chinese Wall (CW) and constraints with spatio-temporal attributes. Together with the security principles, security constraints cause many authorization problems to be recast as optimization problems. In fact, many techniques available in constraint satisfaction research have found a successful application in security system research and development. For instance, supporting SoD constraints in a system where the least privilege is applied can be considered as a constraint optimization problem.

The correctness of the mapping between an authorization model and its encoding in a policy has been the subject of many papers, as well as the analysis of properties that represent static constraints. Also referred to as static analysis, this verification deals with the problem of whether a policy captures all semantics of the authorization model and entails a set of static constraints. Static analysis methods have been an important motivation for the development of many formal systems in security research. An important application area of static verification has been change impact analysis, in which a new version of a policy is analyzed with regards to an old version. However, there are security constraints that are dynamic in nature and their verification cannot be done with static analysis. This is mainly due to the following reasons:

– A dynamic constraint c that is satisfied at a system state s may not hold at a future state s' during system execution.

– Dynamic constraints are enforced through *mechanisms* available in the authorization model or the specification language and these mechanisms usually require run-time knowledge and implementation support.

– The mapping between a constraint and the mechanism that implements it may be incorrect.

The *first* reason requires to model an authorization system as a state transition system, where the verification of dynamic constraints is done in the execution context (or an abstraction of it). Access control decisions that involve such constraints use parameters and the context in which they are evaluated changes over time. Hence, some constraints require continuous monitoring of relevant events happening during system execution and the occurrence of an event may effect future decisions. These events can be considered as abstract actions that cause a state transition. The *second* reason originates from the fact that many constraint definitions involve parameters that change over time. The exercise of a permission or the activation of a role may change the satisfiability of a dynamic constraint. The *third* reason is related to the errors in model specifications and the way a mechanism enforces a constraint. Indeed, there might be semantic gaps between the requirements of a constraint and the actual functionality provided by the mechanism. For instance, it is now widely accepted that RBAC's core mechanism, "mutual exclusion" among the roles, does not correctly support SoD. It is difficult to discover such problems without actually running the mechanism and verifying the execution states with regards to the constraints that are specified as properties.

Intuitively, empirical methods such as those used in software testing can be used to verify properties of an authorization system. They can generate instances of particular states and check if a certain property holds. Empirical methods are efficient but they do not give any guarantee on the system behavior for all different input values. They only verify one execution path at a time, determined by the input set. In many authorization systems, the size of the state machine representing all possible execution paths is large and the constraints need to be analyzed dynamically for all possible evaluation contexts.

Thanks to the recent developments in automated reasoning techniques (e.g. model checking), an abstract model of a system (e.g. a software or a protocol) can now be verified rigorously by using available tools and different formalizations. Nevertheless, the application of these techniques to access control systems in a dynamic context has been limited. Such an approach allows certain security guarantees in the support of constraints during policy *design*. Moreover, even though dynamic analysis can lead to state explosion, many optimizations can be applied. In most of the cases, these optimization techniques provide significant performance gains by eliminating unnecessary state search. This enables model checking techniques to be efficient alternatives to their empirical counterparts under many scenarios.

Problem Statement

From a broad perspective, the two research problems addressed by this thesis can be described as follows: First, given an access control model and its encoding in a policy together with a set of dynamic constraints, how can we exploit the existing verification tools and automated reasoning techniques for efficient verification of dynamic constraints while enhancing the authorization decision process? Second, is the performance a problem in the current implementations of an access control language if a constraint aware decision procedure is to be integrated to the policy evaluation process?

Contributions

There are at least two ways of verifying the enforcement of a dynamic constraint. In the first, an abstract model of the security system is developed, some of the run-time parameters are approximated and finally exhaustive search methods are applied to the system state space. The principal advantage of this approach is that it can be applied in an off-line manner and off-the-shelf software model checkers can be employed to automate the process. Consequently, an access control policy and the abstract model of the authorization system can be iteratively developed. However, off-line analysis works on an abstract model of the system under investigation and it heavily relies on the correctness of the abstraction. Moreover, the verification process can be subject to state explosion even in the case of optimizations. In the second, a decision procedure is developed to be used at run-time. Such a procedure works on a snapshot of the system that propagates the necessary decision parameters to the current state incrementally. More importantly, because the majority of the components are static, they can be processed in an off-line manner. A run-time decision procedure brings further requirements to the development, such as efficiency. Because it is employed at run-time, the response time of the procedure is a critical concern. The optimizations used to reduce run-time overhead play a key role in the development of the approaches presented in this thesis. Thus, the contributions of this thesis regarding the two research questions presented above are the followings:

– We present an approach for design-time analysis of dynamic constraints that borrows ideas from software verification.

– We develop an efficient constraint-aware decision procedure based on propositional satisfiability for policy evaluation at run-time.

– We provide an extensive study of policy evaluation performance for XACML by identifying the steps in access control decisions.

Structure of the Dissertation

In Chapter 2, we present some background information and create a common terminology for the concepts discussed in the thesis. In Chapter 3, we present an applied approach for off-line analysis of dynamic constraints based on software model checking. In Chapter 4, we propose a boolean satisfiability (SAT) based constraint-aware decision procedure for user authorization queries in RBAC and discuss its applicability at run-time. In Chapter 5, we empirically analyze the performance of decision procedures tailored to XACML based authorization systems and proposes possible optimizations to improve the decision process. Chapter 6 provides a review of the relevant literature while Chapter 7 concludes the thesis and presents future research directions.

Chapter 2

Background

2.1 Propositional Satisfiability

Propositional satisfiability (SAT) is an NP-complete problem that dates back to 1920s. It played a key role in defining, understanding and solving many computationally complex problems. A boolean satisfiability problem amounts to finding an assignment of all boolean variables $\{x_1, \ldots, x_n\}$ available in a propositional formula φ such that the formula φ will be true. If no such assignment is found the formula is said to be unsatisfiable. A boolean formula can constitute of a single variable x, a set of variables $\{x_1, \ldots, x_n\}$ or other formulae that are associated with each other using propositional connectives, $\neg, \wedge, \vee, \implies, \iff$. Parenthesis can be used to modify precedence among the operators. Most of the SAT solvers accept Conjunctive Normal Form (CNF) formatted boolean formulae for solving. In CNF, a formula φ is composed of a conjunction of clauses ω.

$$\omega_1 \wedge \ldots \wedge \omega_n$$

Each ω is either single propositional variable (i.e. unit clause) or a disjunction of them.

$$x_1 \vee \ldots \vee x_k$$

Although falling into NP-complete class, many real world SAT problems can be solved efficiently. In fact, through the years a large number of SAT solvers appeared that are well engineered in solving very complex SAT instances with millions of clauses. The research on SAT is an active area of research in both theory and practice. Many different SAT variants and extensions have been proposed. A particularly interesting for this thesis is MaxSAT. MaxSAT is a generalization of SAT that tries to find the maximum number of clauses of a propositional formula that can be satisfied by an assignment of variables. MaxSAT is NP-hard and it is mainly used for solving optimization problems.

SAT also plays a key role in the development of new approaches for solving problems with more complex structures. For instance Satisfiability Modulo Theories (SMT) is a new research area that tackles problems with a first-order nature. SMT can be considered as a SAT instance in which some of the variables are replaced by predicates. The predicate is a boolean function defined by non-binary variables.

2.2 Role Based Access Control

RBAC has been the predominant access control model with a wide industry support and a wealth of literature. It eases the administration of permissions by grouping them as roles (i.e. job function) and assigning those roles to the users. Many variants of RBAC have appeared since its initial proposal. Among the various versions RBAC96[SCFY96] found a wide acceptance and lead into a standardization[oSN04] of the model. The standard RBAC distinguishes between three different RBAC models: core, hierarchical and constrained. Core RBAC includes the basic features of RBAC with users (U), roles (R), permissions ($PRMS$) and sessions (S). Hierarchical RBAC adds role hierarchies to core RBAC in two forms, general and limited. In general hierarchy model, the hierarchy relation forms a partial order and supports multiple inheritance between roles. Limited role hierarchy restricts the multiple inheritance by imposing $\forall_{r,r_1,r_2 \in R},\ r \succeq r_1 \wedge r \succeq r_2 \implies r_1 = r_2$. Constrained RBAC adds Separation of Duty (SoD) constraints to hierarchical RBAC in two contexts, static and dynamic SoD. Static SoD (SSoD) exploits the user assignment (UA) relation for supporting constraints. Dynamic SoD (DSoD) provides a more relaxed approach and prevents simultaneous role activations in the same session. DSoD is of particular interest in this thesis because of the way it is enforced.

Unless it is explicitly mentioned we will refer to constrained RBAC, when we use the acronym RBAC throughout the thesis. The features of constrained RBAC can be summarized as follows:

- Permissions are defined as pairs $PRMS \subseteq OBS \times OPS$ where OPS is the set of operations and OBS is the set of objects.

- Roles in a set R associate permissions in a set P to users in a set U by using *user-assignment relation* $UA \subseteq U \times R$ and a *permission-assignment* relation $PA \subseteq R \times P$. If $(u, r) \in UA$, then we say that *user u is a member of role r*.

- The set of roles R is endowed with a hierarchy relation, i.e. a partial order $\succeq \subseteq R \times R$ where $r_1 \succeq r_2$ means that r_1 is *more senior than* r_2 for $r_1, r_2 \in R$.

- Users are mapped to a subset of their assigned roles ($rset \subseteq$ ASSIGNEDROLES(U)) in a

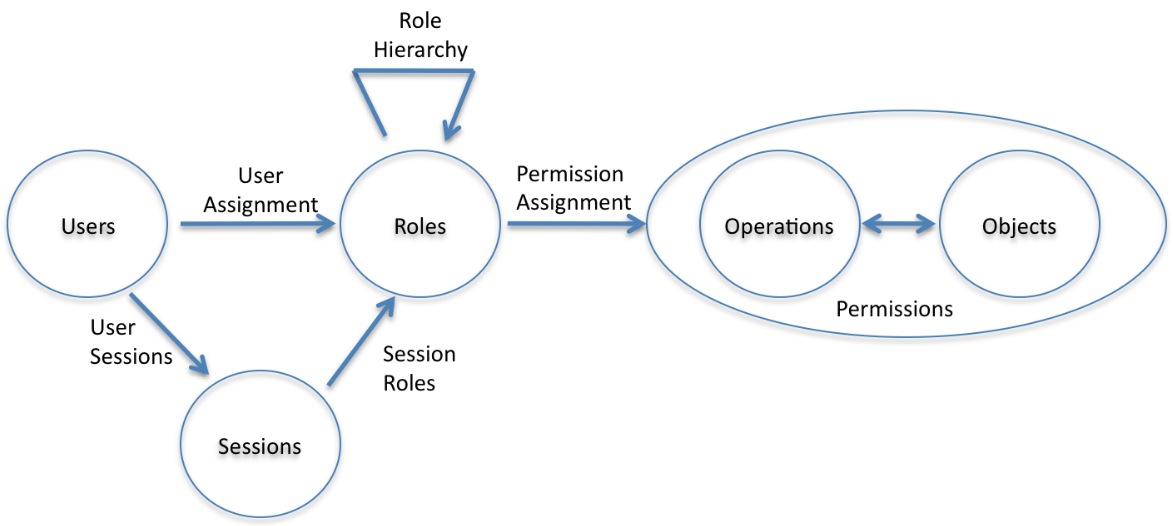

Figure 2.1: RBAC Model Components

session[1] ($s \in SESSIONS$). According to functional specification in the RBAC standard documentation, a user can exercise her rights in a session by activating an initial set of roles during session creation or activating further roles in an active session. Figure 2.1 summarizes the RBAC model components and the relations between them.

– Static and dynamic Separation of duty (SoD) constraints are specified on sets of roles: SSoD, DSoD $\subseteq (2^{Roles} \times N)$, such that an RBAC SoD constraint (rs, n) where rs is defined by a subset of roles ($rs \subset R$) and a natural number, $n \geq 2$. As we will see in the following chapters, the way a DSoD constraint is enforced (i.e. per session) is widely considered as underspecification.

A user u *has permission* p iff there exist roles $r, r' \in R$ such that $r \succeq r'$, $(u, r) \in UA$, and $(r', p) \in PA$. Conventionally, an *RBAC policy* is a tuple $RP = (U, R, P, UA, PA, \succeq)$. All the RBAC policies considered in this thesis are assumed to be *finite*, i.e. U, R, and P have finite cardinality (and thus UA, PA, and $\succeq$ have finite cardinality too).

2.2.1 Separation of Duty (SoD)

SoD[CW87] has been considered as a mechanism to eliminate errors or accidents in the performance of a sensitive function (i.e. task). Known as two man rule, it regulates the completion of a sensitive task by putting constraints on the users performing it. SoD has been studied extensively and solutions have been proposed at different levels of access control objects. A role based SoD defines SoD policies in terms of roles, a permission

[1]The functions in capitals (e.g. ASSIGNEDROLES) are standard RBAC functions

based SoD defines SoD policies in terms of permissions and so on. We should note that the original definition (aka Separation of Privilege [SS75]) referred to the permissions of a user while the most recent SoD research defines SoD policies in terms of users.

2.3 Security Policies, Principles and Mechanisms

In this section, we discuss some of the well-known security policies to support security motivations and a set of principles that should be taken in the design and deployment of any security system. A rather pragmatic approach is followed when choosing the terms for a security element and specifying the category it belongs to. For instance, the separation of privileges is considered as a principle in various resources and a policy in some others.

Apart from conventional use in access control systems (i.e. authorization policy), a *security policy* in access control refers to a security measure that regulates certain functionalities, mainly via restrictions, for security motivations including *conflict of interest*, *prevention of fraud*, *error detection*, *information integrity* and *confidentiality.* In the literature, they are sometimes named as objectives [LTB07a], constraints [SZ97] or concerns. We will often use the term *constraints* to refer to security policies. However, when possible, a clear distinction is preserved between authorization and security policies in this thesis. The former is considered to encode usually, but not necessarily, static information and does not need state notion, while the latter might be dynamic and requires state maintenance. Security policies might be application dependent and a custom specification language might be needed for their specification. They are usually authorization model transparent and exist indepedently of the underlying security model. Among the others, SoD policies are widely employed and dicsussed in the literature and are the main policies that are considered in this thesis.

Security policies are enforced through *mechanisms.* Mechanisms are methods or approaches for supporting the requirements of security policies in a particular authorization model. Hence, they are usually model specific. The nuance between authorization policies and security policies lies in their evaluation results. The evaluation of an authorization policy results either with a negative authorization A^- (i.e. denial of the request) or with a positive authorization A^+ (i.e. set of privileges P_{found}). If the result is A^+, then the evaluation of security policies can further retrict the availability of these privileges. Specifically, the set of privileges requested (P_{req}) can be either a subset of the set of privileges ($P_{constrained}$) that does not violate the security policy ($P_{req} \subseteq P_{constrained}$) and preserving a positive authorization A^+ or a superset ($P_{req} \supset P_{constrained}$) that turns a positive authorization into a negative one. The way how a security policy is enforced and the authorization policy is evaluated is a design consideration for a security system. For

Motivation	**Policy**	**Mechanism**
Conflict of Interest, Prevention of Fraud, Confidentiality	Separation of Duty	Mutually Exclusive Roles (MER) Role enablement Minimum Cardinality on Usersets
	Chinese Wall	Conflict of Interest Classes

Table 2.1: Example Motivations, Policies and Mechanisms

the sake of simplicity, we will consider an intuitive approach as described above: security policy enforcement follows the authorization policy evaluation. Table 2.1 presents two constraints (i.e. SoD and Chinese Wall[BN89]) and a set of mechanisms that can be used to enforce the security policy.

Security *principles* are best practices that help a security system to be feasible. They provide common considerations that need to be taken into account during the design and deployment of the system. In fact, many security decision problems can be cast as optimization problems in their existence. For instance, the well-known principle of least privilege would require $P_{constrained} \setminus P_{req}$ to be minimized. This thesis covers the cases for three types of such principles: least privilege, economy of mechanism and complete mediation. These principles set the guidelines for security system design.

Least Privilege: Least privilege principle requires that a user should be provided only the permissions that are necessary to complete her task and not more[Bis04]. It is also known as need-to-know principle in mandatory access control [Gol06]. Least privilege has been widely applied as a best practice in security design for eliminating frauds and errors. It is also used as a motivation for various security enforcement mechanisms. For instance it serves as a strong motivation for dynamic mutual exclusive role specifications in the context of RBAC.

Complete Mediation: Complete mediation is a generic principle that has a particular impact on the performance of a security mechanism. It requires that every access to a sensitive resource must be checked for authorization. It implies that in an authorization system, the number of access requests is quite likely to be very high. The support of complete mediation requires efficient approaches to authorization policy evaluation and constraint checking.

Economy of Mechanism: In order a security system to be feasible it must cost-effective. Economy of mechanism principle makes sure that a security mechanism serves its purpose in the most efficient way. It implies that the mechanism must be efficient in terms of time, resources while maintaining its primary purpose.

A we have seen above, each of these principles has impact not only on the system design but also on the run-time performance.

2.4 Enforcement of Constraints

The constraints considered in this thesis can be named as authorization constraints following the terminology introduced by [BFA99] and the mechanisms to enforce them are dynamic. That means their enforcement involves state information. A dynamic constraint that is satisfied at a state, might still be violated at a future state because of events in the system. Some of the literature also consider the order in the enforcement of constraints such that in an order dependent constraint, a specific event order must be followed. In an order independent sceheme, the order of events is immaterial. However, in many access control systems such a distinction is unnecessary. For instance, a cheque must be created for being signed is a business or application requirement, rather than authorization, in the course of events: *create* and *sign* rather than a constraint enforcement scheme.

Majority of the time, enforcement mechanisms involve implementation support which requires additional analysis. One mechanism or its implementation may or may not guarantee the correct enforcement of a constraint. From an architectural point of view, there are two important steps involved in the enforcement of constraints: verification and monitoring. Historically, verification dealt with the correctness of the specification, either an authorization policy or a constraint. Monitoring on the other hand captured the necessary information for mainly auditing purposes. Recently, these two steps are tightly integrated to each other so that a clear distrinction is difficult to make. A better classification can be made based on the time the enforcement is done: design time and run time. Design time enforcement involves, also called as policy testing and usually exhaustive, the analysis of a given system. For design time enforcement, performance is not big concern but the correctness. Run time enforcement, on the other hand employs efficient decision procedures inside the authorization decision process. For run time enforcement both the performance and the correctness are fundamental. In both enforcement methods, history plays an important role as we will see.

Chapter 3

A New Approach for Design-time Analysis of RBAC Systems with Dynamic Constraints

While the security analysis of RBAC based authorization policies have been well studied, it has been mostly abstraction based. In abstraction based techniques, an abstract model of the authorization policy is created in a custom modeling language and properties are analyzed on the created model. The knowledge of the custom language and the functionalities provided by the underlying analysis tool are key elements in abstraction based analysis. Moreover, majority of the analysis work were tailored to static access control elements (e.g. user-role assignments in RBAC). Design time information governed by the administrative operations were enough to build an abstract model. The subject of the analysis was the relations between elements such as *a query that would yield the state of these relations*. For instance, asking whether a user can perform an operation through one of her roles could ensure that she can perform her daily job function within this context. In this approach, the state is defined by the static RBAC relations and it changes through administrative events such as assigning a new role to a user or changing the definition of a role. From a dynamic constraint point of view, the run-time state of an access control system changes also through some other events that are not administrative such as past user actions and dynamic run time context. In fact, as summarized in Chapter 1 an enforcement mechanism to support dynamic constraints works in an active context that frequently changes. Hence, majority of the dynamic constraint enforcement mechanisms require a *state* definition and proper handling of state changes.

XACML[XO05a], as the predominant language for access control policy specification, comes with an architecture for policy evaluation, making it is beyond a policy language.

However, XACML run-time provided by the architecture represents a stateless system. While we can use the history of exercised permissions and past role activations of a user at the RBAC model level, at the language level XACML does not provide the necessary means. XACML components need to be updated for a stateful decision process. Several proposals, in the form of implementation support, appeared to enrich XACML with stateful policy evaluation. These proposals required new approaches for testing access control policies that are closer to the implementation level and that can test the policy dynamically so that the provided implementation support can also tested.

It is difficult to analyze the dynamic constraints of an access control system and verify its correctness at design time. In particular obtaining good test cases is challenging. This is mainly due to the fact that the authorization policy and the constraint specifications are part of an active system and the context which they are verified changes frequently. However, similarly to what is done in software verification, it is possible to statically give some approximations of the run-time behavior and analyze a finite number of possible configurations that represent both the user behavior and the decision parameters in a run-time context. The use of software verification tools can be handy in generating the test cases. In this way, not only constraints (e.g. history based) can be analyzed dynamically but they can also be analyzed in combination so that their co-existence can be projected. However, using software model checking requires careful considerations on two aspects: optimizations to avoid state explosion and understanding of RBAC run-time events that cause a state change. For the former, we discuss some of the optimization techniques available in the software model checker we use during demonstrations. For the latter, a detailed analysis of RBAC operational semantics that result with clear definitions of relevant run-time events are presented.

3.0.1 Contributions and Organization

In this section of the thesis, we present a new approach to dynamic constraint analysis, in particular verifying the enforcement of Dynamic Separation of Duty (DSoD) policies. The authorization systems we consider are RBAC systems designed according to XACML RBAC profile[XO05b]. Our approach exploits the operational semantics of RBAC standard and the functionalities provided by an explicit state software model checker, Java Path Finder[Cena]. A shorter version of this chapter appeared in [TJC11].

Because our approach models the RBAC system as a mix of programming language constructs, it does not require the knowledge of specific modeling languages. The presented approach differs from state-of-the-art in ways as the following:

– Employs a software verification tool, which takes program code as input (i.e. execution based). We believe that writing program code is easier than learning custom languages/notations required by model checkers[1].

- Tests also the implementation extensions required by the constraints and automates the test generation process.
- Provides insights about a running RBAC system with concurrency in mind and the constraints by exhaustively submitting requests to an actual XACML policy and it does this in an off-line manner.

In Section 3.1 and 3.2, we define the problem with relevant background information and present a running example respectively. Section 3.3 surveys standard RBAC specification [oSN04] and presents our findings from analysis of RBAC operational semantics. Analyzing operational semantics of RBAC provides a functional view to RBAC and helps us to define event instances that change a state in a running RBAC system. In Section 3.4 we present our approach for dynamic analysis and Section 3.5 demonstrates various testing cases and possible optimizations by using the software model checker we employ. Section 3.6 concludes the chapter and presents a list of future work.

3.1 Problem Definition

The RBAC model used in this chapter is the one described in XACML RBAC profile [XO05b]. The model supports all hierarchical RBAC features of standard RBAC. In addition, it allows the specification of a set of attributes whose values are only available at run-time to enable further dynamicity. It does not fully support DSoD constraints without extensions to the XACML architecture as we will see. In this profile, roles are represented as sets of subject (and resource in some cases) attributes and role hierarchies as policy references. Figure 3.1 shows how an RBAC query is responded through the available policies. Three policies are defined in the profile:

- **Permission PolicySet (PPS)**: Contains actual permissions by specifying resources and actions to be performed on resources.
- **Role PolicySet (RPS)**: Specifies roles by associating role attributes with a given Permission PolicySet. RPS handles the PA relations of RBAC.
- **Role Enablement Authority (REA) Policy** : Manages role assignments to subjects. REA handles the user assignment (UA) relations of RBAC.

[1]Note that model checkers input a custom language specification while software model checkers input source code

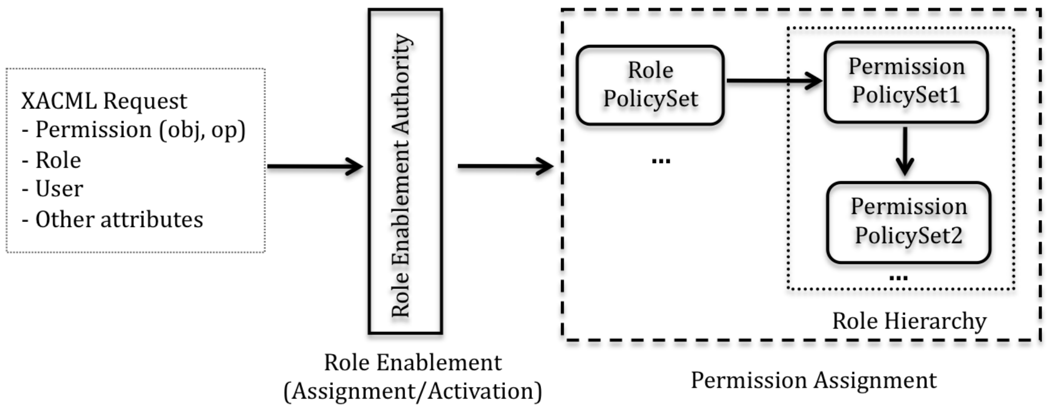

Figure 3.1: RBAC Authorization Queries in XACML

Among these policies, REA policies are of particular importance for us as they contain the UA relations. The architectural components for enforcing policies (see Chapter5 for more details) in XACML include: 1. *Context handler* acts as a communication bus by creating a mutual context between components, 2. *Policy decision point (PDP)* makes an access control decision by evaluating the request against the available policies, 3. *Policy enforcement point (PEP)* forwards the received requests to PDP and enforces the obligations returned from PDP, 4. *Policy administration point (PAP)* makes the policies available to PDP, 5. *Policy information point (PIP)* retrieves the attribute values requested by context handler. Among these components, PEP and PDP are of particular importance for us as they are the components in which state information can be maintained.

3.1.1 Defining States

Conventionally, the state of an RBAC system with static constraints is defined by the tuple $\langle U, R, P, UA, PA, RH \rangle$ where U, R, P are the set of users, roles and permissions respectively, UA is the user assignment relation, PA is the permission assignment relation and RH is the role hierarchy. In RBAC with dynamic constraints, the state must be extended, $\langle U, R, P, UA, PA, RH, H, R_p \rangle$ with dynamic information obtained from the past events (H) and the run time parameters R_p whose values obtained from the execution environment. Thus, we first need to identify what constitutes the past events relevant to dynamic analysis of constraints. We call such events as state changing events. Further we define some of the information that are provided from the execution environment and show how they can be extracted from policies. We show that, in practical terms, these information are just a set of attributes in XACML policy specifications and can be represented as program constructs.

3.1.2 Analyzing Dynamic Constraints and Their Enforcements

In standard RBAC terms, we consider only DSoD motivated constraint definitions, specifically mutually exclusive roles (MER). In the context of RBAC specified using XACML, constraints on role activations and restrictions on exercising permissions are also supported. For the enforcement of DSoD motivated constraints, several approaches have been proposed [FB09, Cra05]. In each proposal, the *implementation support* that extends the XACML enforcement architecture plays a key role. Table 3.1 summarizes the XACML architectural components to be extended for supporting DSoD enforcement mechanisms introduced in these proposals. The first row refers to MER specifications which implies a maximum of k roles can be activated at the same time from the set $\{r_1 \ldots r_n\}$. The second row refers to a high level SoD specification which implies there must be at least k users exercising the permissions $\{p_1 \ldots p_n\}$. In both rows $n \geq k$.

In general, verifying whether the constraint specifications (e.g. MER) correctly enforce the security policies (Table 2.1 of Chapter 2) behind them is difficult. With the addition of implementation support, the situation gets more complicated as there can also be errors in the extensions. Thus the aim of this chapter is first, *trying to find an approach to analyze if the security policies are correctly enforced with the implementation support* and second, *observing the behavior of the whole authorization system in a realistic environment where the authorization sessions can exist concurrently.*

Specification	Extended XACML Component
$(\{r_1 \ldots r_n\}, k)$	PEP
$(\{p_1 \ldots p_n\}, k)$	PEP + PDP

Table 3.1: Constraint Types and Effected Access Control components

3.1.3 Java Path Finder

While there are many freely available tools for the execution based verification of software (i.e. that takes source code as input rather than an abstract model), we illustrate the proposed approach by using Java Path Finder (JPF) [Cena] because of its wide user support and stable development cycle. JPF is an explicit state model checker that analyzes programs written in Java. It works on top of a custom built Java Virtual Machine (JVM). JPF accepts Java programs and property specifications as input, and reports whether the property holds by analyzing all possible execution paths. It employs several optimization techniques including state matching, i.e. whether the state has already been visited, or partial order reduction to reduce the size of state space. Although, JPF works as an explicit model checker when used alone, there are many subprojects that extend its functionality with different testing schemas including symbolic execution and actor testing.

3.2 Example Policy

Throughout the chapter we will be using an example policy designed for a banking system dealing with money transfer operations. The money transfer requested by a customer passes several steps before being realized. Based on customer's request the bank teller initiates a transaction on customer's name. The initiated transaction together with the customer data is validated through a security check that is done by an auditor. A valid transaction is finally approved by a manager for the actual transfer.
The system can be modeled with the following RBAC elements:
$U = \{alice, bob, seth\}$
$P = \{p_1(initiate, transaction),\ p_2(approve, transaction),\ p_3(validate, transaction)\}$
$R = \{manager, teller, auditor\}$
$PA = \{(teller, p_1),\ (manager, p_2),\ (auditor, p_1)\},\ (auditor, p_3)\}$
$UA = \{(alice, teller), (bob, auditor), (bob, manager), (seth, teller)\}$

In such a system, there are several rules that regulate the process:

r1 A transaction can occur only during the week days (Monday - Friday) and between the hours (08 - 12).

r2 A money transfer process must involve at least two different people, that is $(\{initiate$ $, approve,\ validate\}, 2)$.

There are various ways of encoding these rules. For rule $r1$ we use a role enablement[2] constraint that permits the activation of a role only under certain circumstances. For rule $r2$ we use a MER specification that prevents the simultaneous activation *auditor* and *manager* roles. More concretely, the following constraint specifications are available in the system:
$RE = \{(teller, time \in (08 - 12am)), (teller, day \in (Monday - Friday))\}$
$MER = \{(\{auditor, manager\}, 2)\}$

[2]A role enablement policy contains only one role in the definition and restricts the sequential activation of the role with a given predicate.

3.2.1 Enforcing Dynamic Constraints

As can be noted, each of the constraint specifications of the example requires various degrees of run-time support in their enforcement. MER specification relies on the history information whether one of the roles in {auditor,manager} has been activated. RE specification is defined by using the run-time values.

Intuitively, the REA policy of Figure 3.1 is a point of defense where the relevant restrictions can be applied at the policy level. One method to specify *simultaneous role activation* restrictions is the use of an additional "deny" rule in the role enablement policy that denies a request asking the activation of *auditor* and *manager* at the same time. In fact, earlier versions of XACML RBAC profile included a set of policies, called SoD policies, that prevents the simultaneous activation of a given role set $rs = \{r_1 \dots r_n\}$. In order such a rule to be effective, a request must contain a set of roles rs_R such that $rs \subseteq rs_R$. This approach may not be feasible out of the box because the role activations correspond to a duration in most of the authorization systems. However, with an implementation support, this drawback can be overcome by an extension to PEP that allows keeping track of already active roles and prevents the new ones that contradict the constraint definitions. The necessary information (e.g. history) can be added as variables to the implementation. To illustrate the concepts, we will be mainly referring to the use of REA policies for constraint enforcement, even though it is not the best approach to follow.

3.3 State Changing Event Instances

Standard RBAC provides the "session" notion that encapsulates the state changing events. For that reason, they are central in understanding the operational semantics of RBAC and implementing a software system that represents RBAC dynamics. However they are defined only in very generic terms in the standard. In fact, standard RBAC defines a session as "a mapping between a user and the *activated* subset of roles (rs) that are assigned to him/her" [3]: $\textsc{SessionRoles}(s) \subseteq \textsc{AuthorizedRoles}(u)$ s.t. $s \in \mathit{user\_sessions}(u)$. The definition of sessions and the standard RBAC functions (see Appendix 8 for a list of available functions) have the following implications when there are dynamic constraints (C) available:

1. Each session is associated with only one user and each user is associated with one or more sessions.
2. Sessions are *active* once they are created or *inactive* once they are deleted. Being

[3]The functions in capital letters are from RBAC standard and we assume their semantics are self evident from their names.

active is a precondition for all other RBAC events at run-time. Beware that the events happened in an inactive session may need to be stored in order to enforce some historic constraints.
3. A user can add (and drop) multiple roles rs in an active session as long as the dynamic constraints are not violated: $rs \subseteq \textsc{AuthorizedRoles}(\text{u}) \wedge c$, for each $c \in C$.
4. A user can exercise a permission p in an active session s if there is an active role r and the dynamic constraints are not violated: $(p, r) \in PA \wedge r \in \textsc{SessionRoles}(s) \wedge c$, for each $c \in C$.

There are three events, namely *session creation*, *role activation* and *permission exercise* that are significant when run-time state changes are considered. The first two events also have their closing events, i.e. session deletion and dropping of a role. Another important point to mention is the chronological dependency relation between them such that a session has to be created before a role is activated or a role has to be activated before a permission invoked.

Session Creation. According to the requirements of the application domain, an RBAC user can create (or be associated with) multiple sessions each having a unique identifier. Each session is instantiated with an initial set of roles and enables the user to perform various activities related to her job function. Because it involves automatic activation of some roles it is also a relevant event in dynamic constraint analysis. RBAC standard provides two functions for the rudimentary creation/deletion of sessions: CreateSession and DeleteSession.

Role Activation. The discussion of sessions as a run-time aspect has been mainly centered around role activations. Note that role activation does not only refer to initial activation of roles during session creation but also the dynamic activation/dropping of roles, denoted as R_A and R_D respectively, through functions AddActiveRole and DropActiveRole. These functions can be invoked in an *active session*. For instance for the example in Section 3.2, the user *bob* can create two sessions and activate *auditor* and *manager* roles in each session respectively. Such a configuration would not be prevented by the standard RBAC.

The literature suggests that there are two types of R_A in a session: single (sR_A) and multiple (mR_A). mR_A enables further flexibility but also adds more complexity in the run-time permission management. In a multiple session enforcement setting, an R_A may create a restriction on the activation of roles in other sessions of a user if there is a dynamic constraint specification. In general, role activation is a point of contention where the main RBAC criticism originates. Among the very arguments listed in [LBB07], (mR_A) is considered to be in conflict with the *economy of mechanism* principle [SS75]. In particular, the support of least privilege principle with mR_A requires additional measures

Event	**RBAC Supporting Functions**
Session Creation	$CreateSession()$, $DeleteSession()$
Role Activation	$AddActiveRole()$, $DropActiveRole()$
Permission Exercise	$CheckAccess()^{+}$

Table 3.2: Run-time state changing events and corresponding RBAC functions

(e.g. dynamic separation of duty of RBAC) and this causes mR_A to be more expensive than sR_A. Moreover, the control required for checking all combinations of roles (i.e. their permissions) is more complicated than the one for a single role in a session. Although our primary focus would be on mR_A in what follows, the analysis can be easily restricted to a case where only sR_A sessions are considered.

Exercising Permissions. While role activation plays a key role in the discussion of dynamic constraint enforcement, it is the exercise of permissions that makes role activations meaningful in practice. RBAC standard provides a function CheckAccess to request a permission in a given session. Conventional RBAC constraint mechanisms work at the role level and enforcement is done per session. If permission level granularity is not considered, then role level enforcement requires the assumption that a role activation entails the exercise of all associated permissions of the role activated. In reality, it is the permission p_1 of the role *auditor* in a session of *bob* that again effects *bob* to undertake role *manager* role or exercise p_2. Hence, we also consider the exercise of permissions as atomic events that can change a state in order to accommodate permission level constraint specifications.

It is easy to see that the events listed above correspond to *supporting* functions of RBAC standard as summarized in Table 3.2. We will be using these events as class methods in one testing scenario and in the other as instances of a class declaration. In both cases, they have access to relevant information such a role activation event can can access to the associated role.

3.4 Approach: Policy Testing as Software Testing

We assume that the policy authors describe their access control requirements (e.g. privileges) in XACML according XACML RBAC profile and the person in charge of deploying the authorization system provides an enforcement logic (i.e. implementation support) for the dynamic constraints. Apart from the XACML policy, the testing system we will present is provided a set of inputs that constitutes an active RBAC system. These elements include:

Constraint Enforcement Mechanism: We consider the constraints as functions that

constrain the user request. Any constraint enforcement mechanism that can be encoded as a Listener (see Section 3.1.2) function can be used.
Domains of Run-time Parameters (PD). In the first testing scenario we will present, we require a finite set of parameter values used in policy evaluation and constraint definitions. A more automated approach to generate these domains is also presented at later section as an optimization.

In order to analyze an access control system automatically, we first extract some attribute values from a given policy. These attribute values correspond to attributes that are only available at run-time summarized previously. We call them as run-time parameters in what follows. We further create a software system that simulates an RBAC system and stores the necessary information representing the extended RBAC state as variables. For instance the set of active roles in a session is represented by a variable, *activeRoles*, of Java vector structure. During the operation the software emits some event instances. When doing this it makes use of the attribute values obtained from previous step to form proper XACML requests. The generated event instances are state changing events listed in Section 3.3 and they carry a system from an initial state, s_0 to a future state, s_n. Some of these events may cause a constraint violation that can be detected at run-time. We represent these events as program constructs that are instantiated during the policy load. The source of state changing events is the unknown user behavior, that means, their occurrence is indeterministic. In our approach they are generated by the software model checker that wraps our software and executes it for all possible execution paths according to chosen testing strategy. The events corresponding to indeterministic user behavior forms an access request and is forwarded to the policy decision point for evaluation. Beware that, it is the actual policy as opposed to an abstract representation of it in a formalization (e.g. Datalog program), that is used for the evaluation. Figure 3.2 depicts our approach at a high level. We believe that the simulator component can be replaced with a wrapper implementation (e.g. API) of the authorization system software with a minor effort. This leads to systematic testing of actual authorization systems as a whole.

The context in which a policy is evaluated or a constraint is checked is called a *decision context* and corresponds to state notion in formal analysis. In addition to the static information contained in the policy, a decision context is composed of two types of dynamic information as explained before: *execution history* (H) and *run-time parameters* (P). Both information can be represented as variables in the RBAC software implementation: Execution history is a sequence of state changing event instances, $H = \{e_1 \dots e_n\}$. Many dynamic constraint enforcement mechanisms rely on historic information as the future permissions of an individual may depend on the permissions she had before. The

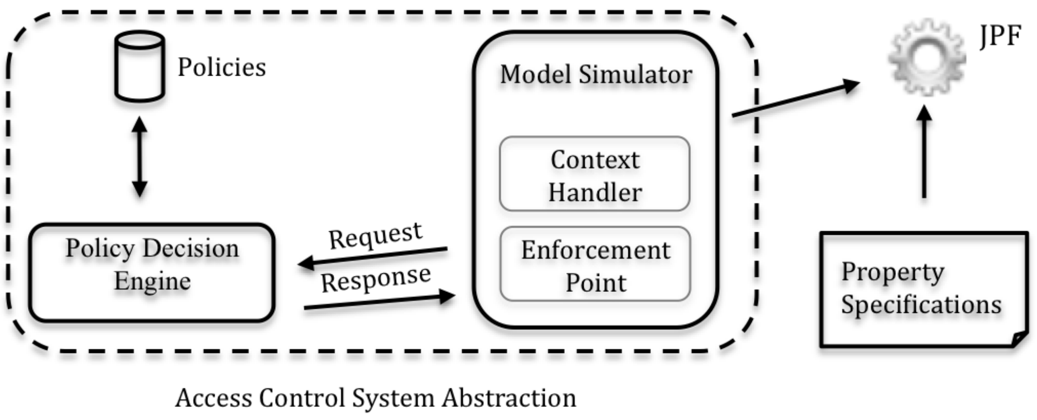

Figure 3.2: Overall Approach

history grows by the addition of new events and provides background information for the evaluation of dynamic constraints. Optimizations can be applied for reducing the size of history. A run-time parameter is a variable that is bound to a value at run-time and used during enforcement of constraints (or the evaluation of actual access requests) with a manipulation logic. As we will see in the following section, we map the run-time parameters to a restricted set of attribute values in XACML.

3.4.1 Environment Attributes as Run-time Parameters

XACML enables the specification of various attributes that are either retrieved from the request or from the execution environment at run-time. We call the attributes that are defined in REA policies as run-time parameters. Run-time parameters regulate the activation of roles and are retrieved from the execution environment. Finding run-time parameters and their representation in an authorization language usually requires custom mining algorithms. However, as it is in XACML, the majority of authorization languages provide special constructs or expressions where such parameters are specified. Hence, we conveniently assume that the discovery of run-time parameters amounts to finding these constructs and forming a top level recursive function $f(X)$ where X has the form :

$$X ::= p_1 \dots p_k \mid v_1 \dots v_n \mid f'(X)$$

The elements $p_1 \dots p_k$ denote the set of run-time parameters, $v_1, \dots v_n$ denote a set of values (usually hardcoded in the policy) used in computation of f, and $f'(X)$ is a subfunction. As can be seen from the function form, the result of one function can be a parameter to another function.

In the case of XACML, some of the *AttributeDesignator (AD)* elements represent the run-time parameters which appear in *Apply* elements of XACML conditions. *Apply* elements act as a container for them and provide a manipulation logic through rich XACML functions. An *Apply* element also contains a set of *AttributeValue(AV)* elements used in the computation of the function. As you can see in the function form presented above, *Apply* elements can be nested. Accordingly, a condition containing run-time parameters can be represented in the following form in XACML ($ConFunc$ represents the function of top level *Apply* element):

$$ConFunc(X) \to \{true, false\}$$
$$X ::= AD_1 \ldots AD_k \mid AV_1 \ldots AV_n \mid Apply(X)$$

Algorithm 1 presents an algorithm ($TransformExpression$) to generate this flat form from a given XACML condition element. Note that, the top level function ($ConFunc(X)$) must be a boolean function. While XACML allows the specification of different run-time

```
Data: Condition expression (ex)
Result: Expression of the form { XACMLFunction funcId; Expression[] parameters; Object[] values}
Initialize expr and set its function id;
foreach child $\in$ ex do
    switch child.type do
        case Apply
            Expression newEx $\longleftarrow$ transformExpression(child);
            if expr.parameters $= \emptyset$ then instantiate expr.parameters;
            expr.parameters.Append(newEx);
            return;
        case AttributeValue
            if expr.values $= \emptyset$ then instantiate expr.values;
            expr.values.Append(child);
            return;
        case AttributeDesignator
            instantiate expNew $\in$ Expression, ;
            expNew.funcId $\longleftarrow$ (attributeId $\in$ child) ;
            expNew $\longleftarrow \emptyset$ ;
            if expr.parameters $= \emptyset$ then instantiate expr.parameters;
            expr.parameters.Append(expNew);
            return;

    endsw
end
return expr;
```

Algorithm 1: TransformExpression

parameters by leaving attribute semantics to implementers, there are attributes associated with the execution environment. In the standard specification, the following attributes are explicitly specified as parameters that needs to be handled by a context handler at run-time:

- *urn:oasis:names:tc:xacml:1.0:environment:current-time*: Represents the current time in the execution environment.
- *urn:oasis:names:tc:xacml:1.0:environment:current-date*: Represents the current date in the execution environment.

In what follows we will consider only these attributes as run-time parameters for simplicity even though the concept can be easily generalized to other types of attribute values.

3.4.2 Defining Constraint Check Points

In order to verify the enforcement of dynamic constraints, we need to define checkpoints (e.g. execution locations) where the current state is verified. JPF provides various ways of specifying properties that represent the constraint specifications in dynamic analysis. One of the simplest ways is to use program assertions in the program (i.e. software) under test. In this method, the violation of the constraint specification, i.e. bad state, is typically implemented as a function whose return values are used in assertion checking. An alternative approach is the use of property extensions provided by the model checker. Property extensions provide a more granular view to the verification process by intercepting lower level events and we will be using them. While it should be relatively easy to extract bad states for SoD specifications from a policy, we assume that the properties representing them are implemented by the policy author. We believe, this gives the policy author a better understanding of the policy content during refinement process. Figure 3.3 presents an example JPF listener that intercepts the relevant events and evaluates them according to given enforcement logic. For instance, a subsumption check, whether a $MER(\{r_1, \ldots, r_n\}, k)$ specification correctly enforces the given SoD policy $(p_1, \ldots, p_t)$ can be checked in this listener.

3.5 Demonstration

In this section we demonstrate how different software testing techniques can be applied to test access control systems. XACML related functionalities including PDP, run-time parameter extraction and request wrapping have been implemented in Java and Sun XACML PDP implementation [Sun] has been used for policy evaluation. An event generated by

```
public class MERPropertyCheck extends PropertyListenerAdapter{
Session session;
public boolean check(Search search, JVM vm){
    Instruction insn = vm.getLastInstruction();
    //An "assignment" instruction to a variable
    if (insn instanceof PUTFIELD){
        PUTFIELD getInsn = (PUTFIELD) insn;
        FieldInfo fi = getInsn.getFieldInfo();
        ElementInfo ei = getInsn.getLastElementInfo();
        //Is the variable changed "activeRoles" in a session?
,,      if (fi.getName().equals("activeRoles") && ei.getClassInfo().getName().equals("Session")){
        session = (Session)ei.getFieldAttr(fi);
        /*Check whether the permissions in SoD({p_1,...p_t})
        are subsumed by the set of permissions available to active roles*/
        ....
        }
    ....
    }
    return false;
}
}
```

Figure 3.3: Example Listener for Constraint Enforcement

JPF is forwarded to PDP engine after generating a request. We will now demonstrate how JPF and its sub-projects can be exploited for testing the specifications and the implementation support. When doing this, we will use the example policy introduced in Section 3.2.

3.5.1 Exhaustive State Space Traversal

An access control policy can be exhaustively verified for analyzing a given property that represents dynamic constraint. By approximating the values of the given run-time parameters, an RBAC policy can be simulated as a whole to obtain insights about a running system. Approximation is necessary for variables of large domains such as *time* and effect the possible number of test cases. The system events are generated randomly by JPF according to the testing environment. Each relevant user event (i.e. request) is evaluated whether it is compatible with the provided constraint enforcement mechanism. The events are also provided with a a list of parameter values (the result of Algorithm 1) randomly selected from possible parameter combinations. For example, a time parameter can be selected from a set of values $\{09:00, 18:00, ...\}$ for the *time* variable of the example policy. We use a data structure *Param*, to denote a list of these parameters which are necessary for the generation of XACML request (i.e. the policy to be applicable for the request) and they are concrete values in this section provided by the user.

JPF has a special class, $Verify$, [VPP06] that acts as one possible type of interfaces to its verification features. We use some of the methods available in this class:

- *beginAtomic*() ... *endAtomic*() creates a code block that will be executed atomically.
- *random*(n) returns a random value between 0 and n.
- *ignoreIf*(*cond*) forces the model checker to backtrack when *cond* evaluates to true.

The analysis cases that can be performed by exhaustive state space traversal includes activity overlaps (simultaneous role activations) among sessions and the verification of simple dynamic constraints. Figure 3.4 illustrates one possible testing environment in which the provided enforcement implementation is called after each function call to *createSession*(), *activateRole*() and *checkAccess*(). It is executed by JPF for all possible combinations of the values selected in *Verify.random*() method. The number M is used to obtain different method call sequences of size K such as: [*createSession*, *addActiveRole*, *createSession* ...]. A listener implementation similar to Figure 3.3 intercepts all low level instructions that modify a set of variables. In our case, these variables represent the RBAC state (e.g. activeRoles) and their modification would mean a state change. A check whether a constraint is violated is performed (e.g. activeRoles) by using the requested modification instruction. The class *RBAC* encapsulates the functionality required by an RBAC system (see 3.3) its constructor inputs an XACML policy. The method *obtainParameterCombinations*() generates parameter lists from user provided set of parameter values.

The main problem of this approach is the number of combinations to be analyzed can be very large leading to state explosion problem. Moreover, obtaining combinations of parameter values can be cumbersome in this approach as the user needs to provide a set of discrete value ranges for the variables. JPF supports various optimization techniques for defending against state explosion and generating test inputs for unknown variable domains. For example, to eliminate redundant state visits it employs *state matching.* State matching can be enabled by calling the function, *Verify.IgnoreIf(CheckSubsumptionAndStore*
(system)) after the line 42 in Figure 3.4. Furthermore, some of the variables with concrete values can be replaced with the symbolic variables to decrease the size of value combinations. In the following sections, we focus on two testing techniques: symbolic execution and actor based testing.

```
//Obtain the list of users, roles, permissions and parameters from policy
RBAC system = new RBAC(policy);
//Get the run-time parameter values provided by the user
Parameter[Parameter[]] pCombinations = obtainParameterCombinations();
int M; /*The number of method calls to obtain
public static void main(String[] args){
  for {int i = 0; i < K; i++}{
     Verify.beginAtomic();
     Parameter[] param = pCombinations[Verify.random(pList.length)];
     switch{Verify.random(4)}{
        case 0: User u = Verify.random(system.users.length - 1);
                 Role[] rset;
                 for (int j = 0; j < u.roles.length;j++)
                    if (Verify.random(1) == 1)  rset.add(u.roles[j]);
                system.createSession(u,rset,param); break;
        case 1: //precondition: check there are active sessions
                    Session s = system.sessions.get(
                    Verify.random(system.sessions.length - 1));
                    system.deleteSession(sDrop);
        case 2: //precondition: check there are active sessions
                    s = system.sessions.get(
                    Verify.random(system.sessions.size() - 1));
                    //check there are active roles
                    Role r = s.activeRoles.get(
                    Verify.random(s.activeRoles.size() - 1));
                    system.activateRole(s,r,param); break;
        case 3: //precondition: check there are active sessions
                    s = system.sessions.get(
                    Verify.random(system.sessions.size() - 1));
                    //check there are active roles
                    r = s.activeRoles.get(
                    Verify.random(s.activeRoles.size() - 1));
                    system.dropRole(s,r,param);break;
        case 4: //precondition: check there are active sessions
                    s = system.sessions.get(
                    Verify.random(system.sessions.size() - 1));
                    //check there are active roles
                    p = s.permissions.get(
                    Verify.random(s.permissions.size() - 1));
                    system.checkAccess(s,p,param); break;
     }
     Verify.endAtomic();
  }
}
```

Figure 3.4: Test Environment with Explicit Exhaustive Search

3.5.1.1 Symbolic Execution

Some of the variables in a given RBAC system can be replaced with symbolic values to ease the testing process and to gain improvements in generation of method call sequences. We consider the run-time parameters and the manipulation logic provided by them as the functions (i.e. $ConFunc$ of Section 3.4.1), to be symbolically executed. By employing symbolic execution, we can eliminate the requirement of run-time parameter domains (PD) and create a line of defense for state explosion problem. As we will see in the example, some of the variables representing the run-time parameters can be replaced with symbolic variables in order reduce the size of state space to be explored. We use Symbolic Path Finder (SPF)[Cenb] for implementing symbolic execution tests. SPF is a project that works on top of JPF and performs symbolic execution of Java bytecode by combining symbolic execution techniques with model checking and constraint solving. Several different constraint solvers and decision procedures are integrated to SPF for finding execution paths in a program.

SPF supports various program constructs such as integer variables or data structures that are replaced with symbolic variables. Hence, the RBAC class as a whole can also be used for symbolic execution. However, for the sake of simplicity, we consider only the function $ConFunc(X)$ to be symbolically executed and replaced its non-integer parameters with integer ones. There are tools that can automatically instrument code to be used in SPF. Symbolic execution requires the code under test to be instrumented. In particular, the symbolic expressions obtained from execution must be supported by the underlying decision procedure. SPF supports a large number of decision procedures for solving symbolic expressions, but it is still restricted to certain types of expressions. In addition to compliance with decision procedure, some of the method definitions such as the ones that update variables must be revised in order to be processed by SPF.

Figure 3.5 illustrates the code that corresponds to the condition of our policy example. It contains the run-time parameters *time* and *date* as *int* parameters and represents the predicates accompanying based on the functions provided in XACML conditions. The function $ConFunc$ can be executed symbolically by replacing some (or all) parameters by symbolic variables. The result of the symbolic execution is a predicate defined by using the parameters represented symbolically. This predicate can then be fed to a decision procedure SPF supports to obtain value ranges which are the only ones that makes the predicate true. For instance, the right side of the Figure 3.5 shows one of the possible snapshot of running both parameters of $ConFunc$ symbolically. It shows that if the time (represented by the variable $time\_1\_SYMINT[15]$) is $14:00$ (i.e. constant $CONST\_15$) then the function is true during the week day "Friday" (i.e. variable $date\_2\_SYMINT[5]$ and the value is $CONST\_5$ that maps to one of the week days), false for the other cases.

```
ConFunc(int time, int date){
//maps to time_in_range function of XACML
//conditions
boolean time_in_range = false;
for (int i = 0; i < timeList.length; i++)
    if (time == timeList[i]) {
        time_in_range = true; break;
    }
}
//maps to type_is_in function of XACML
//conditions
boolean type_is_in = false;
for (int i = 0; i < dayList.length; i++){
    if (date == dayList[i]){
        type_is_in = true; break;
    }
}
//if (!time_in_range || !type_is_in) assert(false);
//and function
return (time_in_range && type_is_in);
}
```

```
1. PC is:constraint # = 14
2. date_2_SYMINT[5] == CONST_5 &&
3. date_2_SYMINT[5] != CONST_4 &&
4. date_2_SYMINT[5] != CONST_3 &&
5. date_2_SYMINT[5] != CONST_2 &&
6. date_2_SYMINT[5] != CONST_1 &&
7. time_1_SYMINT[15] == CONST_15 &&
8. time_1_SYMINT[15] != CONST_14 &&
9. time_1_SYMINT[15] != CONST_13 &&
10. time_1_SYMINT[15] != CONST_12 &&
11. time_1_SYMINT[15] != CONST_11 &&
12. time_1_SYMINT[15] != CONST_10 &&
13. time_1_SYMINT[15] != CONST_9 &&
,,14. time_1_SYMINT[15] != CONST_8 &&
15. time_1_SYMINT[15] != CONST_7
16. Return is: CONST_1
**********************************
Allowed

Result for PC# 14
conFunc(14,1) → Return Value: 1
conFunc(14,2) → Return Value: 1
conFunc(14,3) → Return Value: 1
conFunc(14,4) → Return Value: 1
conFunc(14,5) → Return Value: 1
conFunc(14,0) → Return Value: 0
```

Figure 3.5: Symbolically executed ConFunc() of policy example and a state example from its symbolic execution with SPF

With symbolic execution, only the relevant value sets for run-time parameters that can cause the function *ConFunc* to return *true* are obtained. Symbolic execution can reduce the number of states to be visited and provide a set of input values for the domains of our run-time parameters.

3.5.2 Actor Based Testing

As shown in the previous sections, a simulation environment can be generated for exhaustive analysis of access control systems. However, an access control system is a more complex entity with the following abstract components: *User* front-end (e.g. enforcement point), *Decision engine*, *Reference monitor* and *Session Coordinator*. In such a system, sessions exist concurrently and managed in a variety of ways by a scheduler. An activity occurred in a session may effect another session and this requires analysis of sessions with

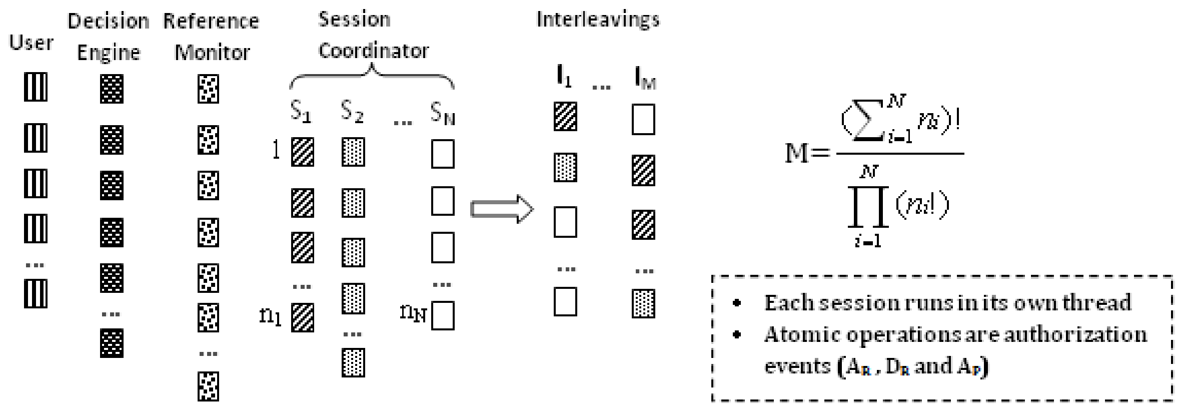
User
Decision Engine
Reference Monitor
Session Coordinator
S_1 S_2 ... S_N
1
n_1
n_N
Interleavings
I_1 ... I_M
$M = \frac{(\sum_{i=1}^{N} n_i)!}{\prod_{i=1}^{N}(n_i!)}$
Each session runs in its own thread
Atomic operations are authorization events (A_R, D_R and A_P)

Figure 3.6: RBAC system with Concurrent Sessions

different interleavings between them. Figure3.6 provides a view to RBAC with sessions existing concurrently. The possible role activations and the requests for the permissions of those roles create different interleavings. The calculation of total number of interleavings (M) between concurrent sessions is given by the formula [4]. It involves the length of a session (N, the number of atomic operations such as role activation) and the total number of sessions (n). In order to verify all possible cases of session interleavings the number M represents the upper bound.

There are similarities between the way an actor system works and an RBAC system operates. In real-world applications, the authorization systems contain multiple active components (e.g. subjects) that exist and act concurrently. These subjects might be actual users or programs acting on behalf of them. Hence, we propose an actor based [Agh86] approach to model user behaviour. An actor is a concurrent process that interacts with other actors via asynchronous message passing. In response to a message, an actor can create other actors, send messages to known actors or change its behavior. The benefits of actor model and its message passing paradigm over shared-memory threads with locks is summarized in [HO09]. Actor model is potentially more efficient than threads and more secure than shared-memory systems because of its race-free mail box mechanism. Eliminating low level problems caused by race conditions (e.g. concurrent policy requests) and allowing autonomous dynamic behavior, actor model enables us to capture the semantics of dynamicity in authorization system operation.

Once an actor based model of the authorization system is obtained and implemented as part of the software model previously described, Basset [LKDM10] can be used to analyze constraints. Basset is an extension to JPF framework for analyzing actor programs. It systematically tests an actor program by exploring different message arrival schedules

[4]The formula is inspired from JPF's computation of thread interleavings [Cena]

during actor communication. Basset requires a simple test driver application to obtain initial configurations to create actors and messaging. Note that, as actors are autonomic agents, a randomized behaviour can be added to the content of actor implementation. At an abstract level, some of the messages exchanged between actors represent state changing event instances that are regulated by the policy. Compliant with the complete mediation principle, every event goes through policy evaluation and the result of evaluation determines state transition. This enables the systematic exploration of states for the given authorization system. However, because of intrinsic dynamicity of the problem, the state space can be large for exploration.

3.5.2.1 Implementing Actor Testing

We implemented a prototype of the presented approach by using Java and the ActorFoundry actor library [Lab12]. In XACML terms actor implementation mainly maps to PEP functionalities.

We define four actors with the following functionalities:
Session: It provides an encapsulation mechanism for the user activities in a session. In case, symbolic evaluation is used, it generates finite number P_A, R_A and R_D events.
User: By imitating the user behavior in a given test case, it creates a finite number of sessions.
Coordinator: Enables the communication between all other actors. In terms of access control, it performs an enforcement functionality.
Authorizer: Produces decisions for the requests that are forwarded from the coordinator actor (i.e. P_E and R_A).

The interaction between the actors is depicted in the Figure 3.7. Every actor message representing an access control event (i.e. state changing event) goes through the actor "Coordinator". The "Coordinator" actor prepares the XACML request by wrapping it with relevant attributes and forwards to "Authorizer" actor which makes an access control decision. If the property to be analyzed is encoded as an assertion then the verification happens on "Coordinator" actor as all the relevant requests pass through it.

3.5.2.2 Component Communication Patterns

The communication between the run-time elements is achieved asynchronously with a mailbox system. Java annotations are used for matching between a message from the mail box and an action to be performed by the actor. For instance, P_E, R_A, R_A and session creation requests can be matched as the following message patterns.

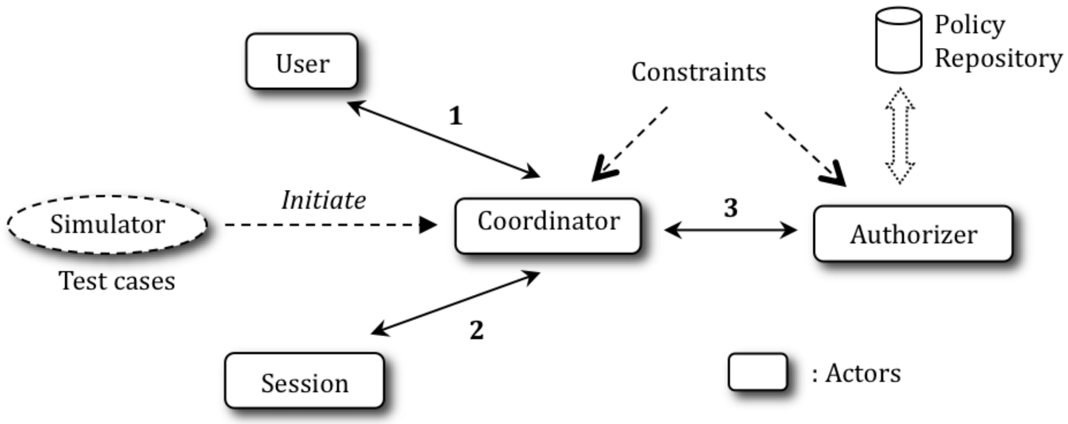

Figure 3.7: Actor Interactions

$$
\begin{aligned}
case \;\; event \; : \; P_E \;\; &=> \;\; checkAccess(event) \\
case \;\; event \; : \; R_A \;\; &=> \;\; activateRole(event) \\
case \;\; event \; : \; R_D \;\; &=> \;\; dropRole(event) \\
case \;\; s \; : \; Session \;\; &=> \;\; createSession(s)
\end{aligned}
$$

While the system can be extended, the following messages are exchanged between the actors (see Figure 3.7):
(1) User - coordinator: User actor sends *Session* requests to coordinator while the coordinator responds with either *Positive* or *Negative* results.
(2) Session - coordinator: Sessions encapsulate the activities performed by users and they send P_E, R_A and R_D requests to coordinator. Coordinator responses can be one of the followings: *Permit*, *Deny*.
(3) Authorizer - coordinator: Authorizer actor performs decision functions by only considering the user rights encoded in a policy. The authorizer sends *Permit* or *Deny* messages to coordinator based on the policy evaluation. While the coordinator forwards the requests (P_E and R_A) received from sessions.

3.5.2.3 Running Basset

Similar to the listener presented in Figure 3.3, we define the constraint enforcement mechanism for DMER constraints as a subsumption check between the permissions in SoD policy and the permissions available to active roles in a session. This way we can verify whether the addition of a new role (i.e. R_A) to the set of active roles causes a violation of the SoD policy. As each positive access request is stored in the history as a variable, obtaining the set of active roles is straightforward.

Once the constraint to be verified is encoded as a listener, a driver implementation (i.e. simulator) that instantiates the actors can be fed to Basset with the following command:

```
..\jpf-core\bin\jpf gov.nasa.jpf.actor.Basset Simulator
```

Basset analyzes different execution patterns by sending messages until it finishes all possible states. In case it detects a violation it reports back to the console with the relevant details.

3.6 Conclusions and Future Work

In this chapter, we have described a new approach to testing the enforcement of dynamic constraints in RBAC with XACML. The RBAC policies encoded in XACML can contain certain parameters that are available only at run-time. The key idea of our approach is to create a software system that mimics the run-time of RBAC and analyzing the resulting software with an explicit state software model checker, Java Path Finder (JPF), for properties. The operational semantics of the analyzed software are obtained from supporting functions of standard RBAC (Appendix 8) that may lead to state changes. The properties represent the bad states in which the constraints are violated. We have shown how various testing scenarios (i.e. exhaustive, symbolic and actor testing) can be implemented in order to verify both the policy and the implementation support for enforcing dynamic constraints.

The presented approach can be used by security administrators to gain insights about a running access control system. As a future work, we plan to analyze the performance of our approach with a set of experiments to see how it performs with real world policies. Even though the performance is not the main concern of design-time analysis, the experimental analysis can be useful to provide indications for developing such tools. In addition to JPF and Basset, we currently investigate the ways of extending our approach to run-time verification tools, i.e. Clara [BLH10] and tracematches [BSH08], which can be used to generate run-time monitors for just in time verification. Clara enables the development of residual run-time monitors from monitoring aspects written in AspectJ [KHH+01]. It converts these aspects into residual monitors that watches events triggered by program locations where the analyses failed to prove safe at compile time.

Chapter 4

An Efficient Authorization Decision Procedure for RBAC User Authorization Queries

In this section, we present an authorization decision procedure tailored to RBAC based systems. While off-line analysis for dynamic constraints checks whether there is any system state, under the given policy configuration, that violates the given constraint, the procedure described here presents an alternative to actual access control decision procedure used at run-time. Such a procedure must make sure that the available dynamic constraints are not violated when responding an access control request. In standard RBAC terms, it corresponds to CheckAccess function with a slightly modified signature. However, it avoids the restrictions related to single session enforcement and considers role activation history in certain types of constraints. The procedure tackles a specialized form of policy evaluation problem, namely, *User Authorization Query (UAQ)*, at the model level and employs boolean satisfiability (SAT) tools for solving.

The UAQ for RBAC [ZJ08] is the problem of determining a set of roles to be activated in a given session in order to achieve a given set of permissions while satisfying all constraints regulating the activation of roles (e.g. mutual exclusion of roles). As it might not be possible to find an exact match between the requested set of permissions and the available roles, one is usually interested in finding an optimal solution, i.e a solution that minimizes (or maximizes) the set of active roles or permissions in accordance with the least privilege principle and the availability of the requested permissions.

Initial approaches to solving the UAQ problem are presented in [ZJ08]. A first approach is based on a greedy search for a set of roles covering the needed permissions that also tries to minimize the additional permissions these roles may have. If any solution is found, then it checks whether it satisfies all constraints. If the check succeeds, then the set of roles is returned as a solution, otherwise the request is rejected. The approach is very efficient, but incomplete since the greedy search algorithm does not explore the space of possible solutions but stops as soon as one is determined. A complete approach, based on a simple generate-and-test strategy is also discussed. The idea is to enumerate all the subsets of the set of roles assigned to the user and stop as soon as one is found that provides the needed permissions and satisfies all the available constraints. The problem with this second approach is that in the worst-case, the first step can be asked to generate 2^n solutions, where n is the number of roles assigned to the user associated with the session.

By borrowing ideas from constraint satisfaction, [WQL09] puts forward a number of alternative, more efficient procedures: a variety of search algorithms based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and a procedure based on a reduction of the UAQ problem to the MAXSAT problem that leverages off-the-shelf SAT solvers. The same paper provides also a comparative experimental analysis between the their proposed procedures. The experiments indicate that the greedy search procedure proposed in [ZJ08] is very likely to reject requests that have a solution. Moreover, the experiments indicate that the SAT-based procedure scales better than the DPLL-based procedures when optimal solutions are sought, whereas the DPLL-based procedures scale better than the SAT-based procedure when an exact solution is wanted. However, the experiments indicate that the time needed to solve the UAQ problem is in the order of seconds even for relatively simple problems. For instance, finding a minimal solution to UAQ problem with 33 roles takes more than 7 seconds in average. These results seem to question more than confirming the practical usefulness of techniques for solving the UAQ problems that are available in the literature.

In this chapter of the thesis, we present a new SAT-based procedure that can replace the existing access control decision procedures at run-time. In addition to its unique features, our procedure improves the state-of-the-art for solving UAQ problems [WQL09] in a number of ways:

1. We show that the procedure can be extended to handle a wider class of constraints spanning over the session history as well as over multiple sessions belonging to the same user.

2. We demonstrate that most of the encoding into SAT need not to be generated at run-time, but can be computed once for all as a pre-processing step. Specifically, a preliminary definition of the corresponding SAT problem is generated off-line. It is refined according to the run-time information and fed to the solver. As we will see later in the chapter, this has important consequences on the performance and hence

on the practical usability of the approach.

3. We have implemented the SAT-based procedure presented in this chapter using state-of-the-art SAT solvers. Furthermore, we have carried out a thorough experimental analysis obtained by running our implementation against a wide range of UAQ problems. The results indicate that procedure not only tackles a wider class of constraints, but also outperforms the procedure presented in [WQL09]. More importantly, the experiments indicate that our technique can quickly solve UAQ problems of real-world complexity: problems with 300 roles are solved in less than 1 second in average.

Organization. Section 4.1 briefly recalls the basic notions underlying RBAC, defines the types of dynamic constraints that we consider in the context of UAQ problem. Section 4.3 explains how to reduce instances of the UAQ problem to (variants of) the SAT problem, how propositional assignments can be mapped back to solutions of the original UAQ problem instance, and how all this is integrated with some key optimizations in our use of the solver. Section 4.4 discusses our experimental settings, the generation of UAQ problem instances, the results of an implementation of our SAT-based solver, and a comparison with the approach in [WQL09]. Section 4.6 concludes the chapter.

4.1 Problem Definition

As explained in Section 2.2, RBAC regulates access through roles. Two relations *user-assignment relation* and *has permission* determine the availability of permissions to a user. A user u is said to be an *explicit* member of role r if there exists $(u, r) \in UA$ and an *implicit* member of r if there exists $r' \in R$ such that $r' \succeq r$ and $(u, r') \in UA$.

A key notion in RBAC is that of *session*. The session concept has been generally neglected in the scientific literature even though their critical importance in constraint enforcement. A user can create multiple sessions in which he can initially activate a subset of the roles that he is assigned to according to the UA relation. More importantly, he can activate new roles and drop already active roles in an active session. Hence, sessions play an important tole in the operational semantics of an RBAC deployment.

Formally, let S be a set of sessions, $\mathit{user} : S \to U$ is a function that associates each session $s \in S$ with the corresponding user, and a *state* is a function $\rho : S \to 2^R$ that associates each session with a subset of the roles assigned to $\mathit{user}(s)$ by the UA relation, i.e. if $r \in \rho(s)$ then $(\mathit{user}(s), r) \in UA$. If $r \in \rho(s)$, then we say that *role r is active in session s at state ρ*. If $u \in U$, then S_u denotes the set of sessions associated with u, i.e. $S_u = \{s \in S : \mathit{user}(s) = u\}$. We assume, for the sake of simplicity, that sessions pre-exist and that the user associated with each session is known in advance. A *history* of an RBAC policy RP is a sequence $H = [\rho_0, \ldots, \rho_k]$ of states of RP, where $k \geq 0$ and ρ_i is obtained from ρ_{i-1} by activating or deactivating one or more roles in some session $s \in S$ and all the remaining sessions are left unmodified (i.e. $\rho_i(s') = \rho_{i-1}(s')$ for all $s' \in (S \backslash \{s\})$, for $i = 1, \ldots, k$). If $H = [\rho_0, \ldots, \rho_{k-1}]$, then $H@\rho_k$ denotes $[\rho_0, \ldots, \rho_{k-1}, \rho_k]$.

4.1.1 Supported Constraints and Their Enforcements

As explained in Section 2.3, MER constraints present one way of enforcing SoD policies. They are also motivated with the least privilege principle so that a user can assume the least amount of privileges at a time. Thus RBAC policies are often enriched with mutually exclusive role constraints. In the literature, MER constrains are usually classified as static MER (SMER) and dynamic MER (DMER). SMER constraints ensure that a user is not assigned conflicting roles and hence constrain the applicability of administrative actions affecting the user-assignment relation UA. DMER constraints ensure that a user does not activate conflicting roles. In this chapter of the thesis we focus on DMER constraints as they are dynamic and therefore we will not consider SMER constraints any more.

DMER constraints are defined on a role set (rs) and are variants of the same constraint type which vary depending on the way they are enforced. Formally, given an RBAC policy $(U, R, P, UA, PA, \succeq)$ and a set $rs \subseteq R$, we define the following types of constraints:

– SS-DMER(rs, n): single-session dynamic MER;

– MS-DMER(rs, n): multi-session dynamic MER;

– SS-HMER(rs, n): single-session history-based MER; and

– MS-HMER(rs, n): multi-session history-based MER.

Figure 4.1 summarizes the relationship between our MER constraint definitions. The arrow implies the subsumption of one constraint by the other. For instance, an MS-DMER constraint definition $\{r_1, \ldots r_n\}$ covers also the prohibited permission space of an SS-DMER constraint having the same signature, SS-DMER$(\{r_1, \ldots r_n\}) \subseteq$ MS-DMER$(\{r_1, \ldots r_n\})$. In fact this is reflected in the SAT encoding we will present.

We also consider role activation constraints referring to cardinality restrictions on the concurrent or sequential activations of a given role. Formally, let $r \in R$ and $t \geq 2$, we define

– CARD(r, t): cardinality constraint.

Given an RBAC policy $(U, R, P, UA, PA, \succeq)$, a history $H = [\rho_0, \ldots, \rho_k]$, and a constraint c, we say that H *satisfies* c (in symbols, $H \models c$) iff

$$\begin{tikzcd}\text{SS-DMER} \arrow[r] \arrow[d] & \text{MS-DMER} \arrow[d] \\ \text{SS-HMER} \arrow[r] & \text{MS-HMER}\end{tikzcd}$$

Figure 4.1: Relations between MER constraints

– $[\rho_0, \ldots, \rho_k] \models \textsc{ss-dmer}(rs, n)$ iff for all $s \in S$, $|rs \cap \rho_i(s)| < n$ for $i = 0, \ldots, k$;

– $[\rho_0, \ldots, \rho_k] \models \textsc{ms-dmer}(rs, n)$ iff for all $u \in U$, $|rs \cap \bigcup_{s \in S_u} \rho_i(s)| < n$ for $i = 0, \ldots, k$;

– $[\rho_0, \ldots, \rho_k] \models \textsc{ss-hmer}(rs, n)$ iff for all $s \in S$, $|rs \cap \bigcup_{i=1}^{k} \rho_i(s)| < n$;

– $[\rho_0, \ldots, \rho_k] \models \textsc{ms-hmer}(rs, n)$ iff for all $u \in U$, $|rs \cap \bigcup_{i=1}^{k} \bigcup_{s \in S_u} \rho_i(s)| < n$;

– $[\rho_0, \ldots, \rho_k] \models \textsc{card}(r, t)$ iff $|\{s \in S : r \in \rho_i(s)\}| < t$ for $i = 0, \ldots, k$.

Given a finite set C of constraints, we say that H *satisfies* C (in symbols, $H \models C$) iff H satisfies c, for each $c \in C$.

Given an RBAC policy $(U, R, P, UA, PA, \succeq)$ and a finite set C of constraints of the types listed above, we will speak of an *RBAC policy with constraints* (or, simply, an *RBAC policy*) to denote the tuple $(U, R, P, UA,\ PA, \succeq, C)$. As we also discussed in 2, the RBAC policy is different from an SoD policy in that it encodes the rules of access rather than restricting the available access. A history H is *valid with respect to the RBAC policy* $(U, R, P, UA,\ PA, \succeq, C)$ iff $H \models C$; the RBAC policy is usually omitted when it is clear from the context. In the rest of this chapter, we will consider valid histories only.

4.2 The User Authorization Query Problem

Given an RBAC policy $(U, R, P, UA,\ PA, \succeq, C)$, define $\pi(r) = \{p \in P |$ there exists $r \succeq r'$ and $(p, r') \in PA\}$, i.e. the set of permissions assigned to role r. If $Q \subseteq R$, then we define $\pi(Q) = \bigcup_{r \in Q} \pi(r)$.

Definition 4.2.1. *Let $RP = (U, R, P, PA, UA, C)$ be an RBAC policy and $H = [\rho_0, \ldots, \rho_{k-1}]$ be a history of RP. A* User Authorization Query (UAQ) *for RP is a tuple $(s, P_{lb},\ P_{ub}, obj)$, where $s \in S$, $P_{lb} \subseteq P_{ub} \subseteq P$, and $obj \in \{\mathsf{any}, \mathsf{min},\ \mathsf{max}\}$. The* UAQ Problem *associated with (s, P_{lb}, P_{ub}, obj) and H is the problem of extending H to a new (valid) history $H@\rho_k$, where the RBAC state ρ_k of RP is called the* solution (of the UAQ problem)*, such that $P_{lb} \subseteq \pi(\rho_k(s)) \subseteq P_{ub}$ and*

Table 4.1: Definition of $\chi_k(rs, s)$

Constraint	$\chi_k(s, r)$	$\overline{\chi}_k(s, r)$
SS-DMER	$r \in \rho_k(s)$	$\overline{r}$
MS-DMER	$r \in \rho_k(s)$ or $r \in \rho_{k-1}(s')$ for some $s' \in S_{user(s)} \setminus \{s\}$	$\overline{r} \vee \bigvee_{s' \in S_{user(s)} \setminus \{s\}} \overline{Y_r(s')}$
SS-HMER	$r \in \rho_k(s)$ or $r \in \rho_{<k}(s)$	$\overline{r} \vee \overline{O_r(s)}$
MS-HMER	$r \in \rho_k(s)$ or $r \in \rho_{<k}(s')$ for some $s' \in S_{user(s)}$	$\overline{r} \vee \bigvee_{s' \in S_{user(s)}} \overline{O_r(s')}$

– *if obj* $=$ min*, then for every RBAC state* ρ'_k *such that* $P_{lb} \subseteq \pi(\rho'_k(s)) \subseteq P_{ub}$ *we have* $\pi(\rho_k(s)) \subseteq \pi(\rho'_k(s))$*;*

– *if obj* $=$ max*, then for every RBAC state* ρ'_k *such that* $P_{lb} \subseteq \pi(\rho'_k(s)) \subseteq P_{ub}$ *we have* $\pi(\rho'_k(s)) \subseteq \pi(\rho_k(s))$*.*

Notice that $\rho_k(s') = \rho_{k-1}(s')$ *for all* $s' \in S \setminus \{s\}$*.*

If Q is a set of propositions, then $\sum Q < n$ is a proposition that holds iff at most $n-1$ of the propositions in Q hold.

Theorem 4.2.1. Let $RP = (U, R, P, PA, UA, C)$ be an RBAC policy and $H = [\rho_0, \dots, \rho_{k-1}]$ a history of RP. A state ρ_k of RP is a solution of the UAQ (s, P_{lb}, P_{ub}, obj) for RP iff it satisfies the following conditions:

1. for each $r \in R$, if $r \in R$ and $(user(s), r) \notin UA$, then $r \notin \rho_k(s)$;
2. for each $p \in P$, if $p \in P_{lb}$, then $p \in \pi(\rho_k(s))$;
3. for each $p \in P$, if $p \in (P \setminus P_{ub})$, then $p \notin \pi(\rho_k(s))$;
4. for each $r, r' \in R$ and $p \in P$, if $r \in \rho_k(s)$ and $(p, r') \in PA$ with $r \succeq r'$, then $p \in \pi(\rho_k(s))$;
5. for each $p \in P$, if $p \in \pi(\rho_k(s))$, then there exist $r, r' \in R$ such that $r \in \rho_k(s)$, $r \succeq r'$, and $(p, r') \in PA$;
6. if $MER(rs, n) \in C$, where MER is SS-DMER, MS-DMER, SS-HMER, or MS-HMER, then $\sum\{\chi_k(r, s) : r \in rs\} < n$, where $\chi_k(r, s)$ is defined in the second column of Table 4.1.
7. if CARD$(r, t) \in C$ and $|\{s' \in S \setminus \{s\} : r \in \rho_{k-1}(s')\}| = t - 1$, then $r \notin \rho_k(s)$.

Additionally, if $obj = \mathsf{min}$ ($obj = \mathsf{max}$), then consider only those $\rho_k(s)$ such that $\pi(\rho_k(s))$ is minimal (maximal, resp.) w.r.t. set inclusion.

Proof. Proof of Theorem 4.2.1 is the following: A solution $S = v_1, \dots, v_m$ to a valid $UAQ(s, P_{lb}, P_{ub}, obj)$ problem is defined by the positive (i.e. true) assignment of boolean variables $(v_1, \dots, v_n)$ that represent the permissions and roles associated with the session s such that; Step 1 ensures that only the variables that represent user's assigned roles $(user(s), r)$ are set to *true* in the solution by encoding UA relation for the user u, Step 2 ensures that all the variables that represent the permissions in P_{lb} are set to *true* in the solution, Step 3 ensures that all the variables that represent the permissions that are outside of P_{ub} are set to $false$ in the solution, Step 4 ensures that if a variable that represents a role is assigned *true* then all the relevant variables representing its permissions also set to *true* in the solution, Step 5 ensures that if a variable representing a permission is set to *true* in the solution then there is at least one variable representing a role which has the permission is also set to *true*, Step 6 and 7 ensure that the cardinality formulae denoting the MER and CARD constraints are also used in solving such that a solution produced by the solver satisfies them. Hence any solution S produced by the solver can not contain a variable that does not meet the conditions imposed by the Steps 1-7 and must contain the variables that are required by the conditions defined in Steps 1-7. □

4.3 A SAT-based Procedure for Solving UAQ Problems

Let $RP = (U, R, P, PA, UA, C)$ be an RBAC policy, $H = [\rho_0, \dots, \rho_{k-1}]$ a history of RP, and (s, P_{lb}, P_{ub}, obj) a UAQ problem for RP. Since RP is finite (i.e. the set U of users, R of roles, and P permissions are finite) and only finitely many sessions are active at any given time instant, the UAQ problem can be reduced to a SAT problem as follows. Preliminarily, we introduce the following propositional variables:

- $\overline{p}$ to represent the statement "$p \in \pi(\rho_k(s))$," for each $p \in P$;
- $\overline{r}$ to represent the statement "$r \in \rho_k(s)$," for each $r \in R$;
- $\overline{Y_r(s)}$ to represent the statement "$r \in \rho_{k-1}(s)$" (called *Yesterday statement*, or *Y-statement* for short) for each $r \in R$, $s \in S$, and $k \in \mathbb{N}$;
- $\overline{O_r(s)}$ to represent the statement "$r \in \rho_{<k}(s)$" (called *Once statement*, or *O-statement* for short) for each $r \in R$, $s \in S$, and $k \in \mathbb{N}$, where $\rho_{<k}(s)$ abbreviates $\bigcup_{i=1,\dots,k-1} \rho_i(s)$.

The key idea of the approach rests on the observation that most of the clauses in the encoding can be computed off-line and only a small number of clauses need to be computed

```
procedure add-YO-clauses()
 foreach $s \in S$ and $r \in R$
  if Y[s,r] then
   HC := HC$\cup\{\overline{Y_r(s)}\}$;
  else
   HC := HC$\cup\{\neg\overline{Y_r(s)}\}$;
  if O[s,r] then
   HC := HC$\cup\{\overline{O_r(s)}\}$;
  else
   HC := HC$\cup\{\neg\overline{O_r(s)}\}$;
```

Figure 4.2: $add - YO - clauses()$ Procedure

at runtime. Specifically, in every new state, the information regarding *Y-statement* and *O-statement* are updated to reflect the changes. Beware that we do not store all states $k = 1, \ldots, n$ for *O-statement* but propagate the fact that the relevant role has been activated at some point in time ($rho_{<k}(s')$) within a session s'. That is because the RBAC system works incrementally when dealing with the constraints. In practical terms, *Y-statement* and *O-statement* correspond to map structures whose entries are a session s and a role r, and used for non-history (e.g. MS-DMER) and history (e.g. SS-HMER) based constraints respectively. As a result, the set of clauses that can be statically generated is $\mathcal{C}(s)$, defined as the smallest set of propositional clauses such that

1. for each $r \in R$ if $(user(s), r) \notin UA$ then $\neg\overline{r} \in \mathcal{C}(s)$;
2. for each $p \in P$ and $r \in R$ such that $(p, r') \in PA$ with $r \succeq r'$, $(\neg\overline{r} \vee \overline{p}) \in \mathcal{C}(s)$;
3. for each $p \in P$, $(\neg\overline{p} \vee \bigvee\{\overline{r} : \text{exists } r' \in R, r \succeq r', (p, r') \in PA\}) \in \mathcal{C}(s)$;
4. for each $MER(rs, n) \in C$, where MER is SS-DMER, MS-DMER, SS-HMER, or MS-HMER, the CNF of the following propositional formulae is in $\mathcal{C}(s)$:
   (a) $\sum\{\overline{\chi_k(r,s)} : r \in rs\} < n$,
   (b) $(\overline{\chi_k(r,s)} \leftrightarrow \overline{\chi}_k(r,s))$,

   where $\overline{\chi_k(r,s)}$ is a new propositional variable and the formula $\overline{\chi}_k(r,s)$ is defined in Table 4.1;
5. if CARD$(r, t) \in C$ and $|\{s' \in S \setminus \{s\} : r \in \rho_{k-1}(s')\}| = t - 1$, then $\neg\overline{r} \in \mathcal{C}(s)$.

This propositional encoding is used in the procedure UAQ-solve of Figure 4.3 to solve instances of the UAQ problem. It consists of a preprocessing phase (lines 2-7) followed by a

```
program UAQ-Solve()
 foreach $s \in S$
  C[$s$] := $\mathcal{C}(s)$; // initialization
 foreach $r \in R$ and $s \in S$
  Y[$s$,$r$] := false; // Yesterday
  O[$s$,$r$] := false; // Once
 add-YO-clauses();
 // beginning of on-line phase
 while(true)
  read($s, P_{lb}, P_{ub}, obj$); // reading UAQ problem
  HC := C($s$); // hard constraints
  SC := $\emptyset$;    // soft constraints
  foreach $p \in P_{lb}$
   HC := HC $\cup \{\overline{p}\}$;
  foreach $p \in (P \setminus P_{ub})$
   HC := HC $\cup \{\neg\overline{p}\}$;
  if $obj$=min then
   foreach $p \in (P_{ub} \setminus P_{lb})$
    SC := SC $\cup \{\neg\overline{p}\}$;
  if $obj$=max then
   foreach $p \in (P_{ub} \setminus P_{lb})$
    SC := SC $\cup \{\overline{p}\}$;
  Res := PMAX-SAT-Solve(HC,SC);

  if Res=UNSAT then
   print "No solution.";
  else
   print-solution(Res);
   foreach $r \in R$
    if $\overline{r} \in$ Res then
     Y[$s$,$r$] := true;
     O[$s$,$r$] := true;
    else
     Y[$s$,$r$] := false;
  add-YO-clauses();
```

Figure 4.3: SAT-based procedure for the UAQ Problem

loop that reads one UAQ problem at a time (line 10). The procedure employs an external PMAX-SAT solver (line 23), which is capable of finding a propositional assignment of variables defined in the problem that satisfies all those clauses labeled as *hard* together with the maximum (or minimum) number of clauses marked as *soft*. The use of a PMAX-SAT solver allows us to handle the UAQ instances (s, P_{lb}, P_{ub}, obj) where $obj \in \{\mathsf{min}, \mathsf{max}\}$. These instances require to find the minimal and maximal, respectively, set of permissions associated to session s between P_{lb} and P_{ub} (lines 11-22). Indeed, when no clauses are marked as soft, the PMAX-SAT solver behaves as a "standard" SAT solver and it is possible to handle those UAQ problem instances for which $obj = \mathsf{any}$. It is possible to extract a solution of the original UAQ problem from a propositional assignment satisfying the set of clauses sent to the PMAX-SAT solver (line 23 of Figure 4.3) in UAQ-solve(). To see how, define $\rho_k(s) = \{r \in R : \overline{r(s)}$ is in $\pi\}$ and $\rho_k(s') = \rho_{k-1}(s')$ for every $s' \in S \setminus \{s\}$. This functionality is encapsulated in the procedure print-solution (at line 28) in Figure 4.3. The correctness of UAQ-solve() derives from Theorem 4.2.1 and the following observations. First, conditions 1.–7. of Theorem 4.2.1 can be effectively translated into propositional logic by quantifier instantiation of the universal and existential quantifiers over roles and permissions as they range over the finite sets R and P, respectively. Second, items 1, 2, 3, and 5 of the encoding used to generate the set $\mathcal{C}(s)$ of clauses are the propositional encoding of conditions 1, 4, 5, and 7 of Theorem 4.2.1, respectively. Third, item 4 of the encoding takes into account condition 6 of Theorem 4.2.1 via the definition of $\overline{\chi}_k(r, s)$ in Table 4.1 (see also procedure add-YO-clauses() in Figure 4.2). Fourth, the remaining conditions (namely, 2 and 3) of Theorem 4.2.1 are handled at lines 24-27 in Figure 4.3. The last observation concerns the relationship between the propositional assignments satisfying the clauses generated by our encoding and the states in a history which are also solutions of instances of the UAQ problem. It is not difficult to extract a solution of the original UAQ problem from a propositional assignment π satisfying the set of clauses sent to the PMAX-SAT solver (line 23 of Figure 4.3) in UAQ-solve(). To see how, define $\rho_k(s) = \{r \in R : \overline{r(s)}$ is in $\pi\}$ and $\rho_k(s') = \rho_{k-1}(s')$ for every $s' \in S \setminus \{s\}$. This functionality is encapsulated in the procedure print-solution (at line 38) in Figure 4.3.

Theorem 4.3.1. *Let*

- $RP = (U, R, P, PA, UA, C)$ *be an RBAC policy,*
- UAQ-solve() *be in a state obtained by submitting a sequence* $[q_1, \ldots q_n]$ *of UAQ problem instances* $(n \geq 1)$*, and*
- $\hat{H} = [\hat{\rho}_1, \ldots, \hat{\rho}_n]$ *be a sequence of states such that* $\hat{\rho}_i$ *is the solution returned by the procedure to the UAQ problem instance* q_i*, for each* $i = 1, ..., n$.

UA	PA

alice {doctor}
bob {doctor}
seth {doctor,auditor,nurse}
john {auditor,nurse}
	doctor {p0,p1,p2,p4,p5,p6}
	auditor {p3,p7}
	nurse {p2,p6}

Table 4.2: RBAC Relations for the example

Then, $\hat{H}$ is a (valid) history of RP. Moreover, if a new UAQ problem instance q is submitted to UAQ-solve()*, then the procedure returns a solution $\hat{\rho}$ iff $\hat{\rho}$ is a solution to q.*

Proof. The proof of Theorem 4.3.1 follows from the proof of the theorem 4.2.1 such that every acceptable solution from the procedure leads to a consistent state s such that $s \models C$ and the set of consistent states builds a valid history. □

Although we have assumed that the set S of sessions is fixed over histories, we believe that our technique can be easily extended to handle the dynamic creation of sessions.

4.3.1 Illustration

We now illustrate our encoding on an example use case with MER constraints. The example is about a drug dispensation process in a hospital where there are three roles, $R = \{Doctor, Nurse, Auditor\}$, two prescription objects, $OBJ = \{Pre_1, Pre_2\}$ and four operations, $OPS = \{Prescribe, Validate, Dispense, Audit\}$. The permissions are obtained from the Cartesian product of objects and operations ($PERMS = OBJ \times OPS$). For the sake of simplicity we consider that there are four users available in the authorization system, $U = \{alice, bob\ seth\ john\}$. The UA and PA relations of RBAC are summarized in Table 4.2.

The described system has the following MER constraints specified independently from each other (i.e. subsumption has not been taken into account) without considering the relation between their restriction space:

```
MS-DMER_0 ({doctor,auditor,nurse}, 2)
MS-HMER_0 ({doctor,auditor}, 2)
SS-DMER_0 ({doctor,auditor}, 2)
SS-HMER_0 ({doctor,auditor}, 2)
```

Each user of the system has two sessions with no active role initially (i.e. State 0 of Table 4.3). One of the valid sequences of three states ($|H| = 3$) history is shown in Table 4.3. In State 2, the user *seth* activates *auditor* role in session s_2.

State 1	**State 2**	**State 3**
s_1 (alice) {}	s_1 (alice) {}	s_1 (alice) {doctor}
s_2 (seth) {}	s_2 (seth) {auditor}	s_2 (seth) {auditor}
s_3 (alice) {}	s_3 (alice) {}	s_3 (alice) {}
s_4 (john) {}	s_4 (john) {}	s_4 (john) {}
s_5 (seth) {}	s_5 (seth) {}	s_5 (seth) {}
s_6 (john) {}	s_6 (john) {}	s_6 (john) {}

Table 4.3: History for the example

Now, by using the given configuration we can generate a set of CNF clauses that represent an UAQ instance for a given session, p_{lb} and p_{ub} setting. For instance, Figure 4.4 presents the set of clauses that have been generated for a query with $session = s_5$, $p_{lb} = \emptyset$ and $p_{ub} = P$. Beware that there also some intermediate propositional variables between the lines 22 - 41 (e.g. SS-DMER_3) necessary to encode the constraint definitions. Moreover, because of the selections of p_{lb} and p_{ub}, the objective corresponds to *any* case in which no soft clauses are generated.

4.4 Experimental Evaluation

We present a thorough experimental evaluation of our approach obtained by running an implementation of UAQ-Solve procedure (Figure 4.3) on several synthetic UAQ problems.

In the literature (see, e.g., [KTZ11]), several dimensions have been identified to specify RBAC policies, such as the number of users, roles, permissions, and sessions. In our procedure, also the length of histories plays a significant role for the evaluation of authorization queries because of HMER constraints. Since there are several reasonable values for each one of the parameters listed above, we explain the rationale underlying our choices in Section 4.4.1. Then, we explain our method to generate valid histories of increasing length in Section 4.4.2 where we genuinely use the capability of solving UAQ problems to ensure the validity of histories. Finally, we discuss our findings on several synthetic instances of the UAQ problem and compare with the approach presented in [WQL09] when considering only SS-DMER constraints in Section 4.4.3.

4.4.1 Dimensions of the UAQ problem

Recall from Definition 4.2.1 that a UAQ problem (s, P_{lb}, P_{ub}, obj) is defined in terms of a RBAC policy $RP = (U, R, P, PA, UA, C)$ and a (valid) history $H = [\rho_0, \ldots, \rho_{k-1}]$ of RP. Thus, there are several dimensions that should be taken into account to generate

```
−1 (doctor) 4 (p0)
−1 (doctor) 5 (p1)
−1 (doctor) 6 (p2)
−1 (doctor) 8 (p4)
−1 (doctor) 9 (p5)
−1 (doctor) 10 (p6)
1 (doctor) −4 (p0) −5 (p1) −6 (p2) −8 (p4) −9 (p5) −10 (p6)
−2 (auditor) 7 (p3)
−2 (auditor) 11 (p7)
2 (auditor) −7 (p3) −11 (p7)
−3 (nurse) 6 (p2)
−3 (nurse) 10 (p6)
3 (nurse) −6 (p2) −10 (p6)
−4 (p0) 1 (doctor)
−5 (p1) 1 (doctor)
−6 (p2) 1 (doctor) 3 (nurse)
−7 (p3) 2 (auditor)
−8 (p4) 1 (doctor)
−9 (p5) 1 (doctor)
−10 (p6) 1 (doctor) 3 (nurse)
−11 (p7) 2 (auditor)
1 (doctor) −2 (auditor) 20 (SS–DMER_3)
−1 (doctor) 2 (auditor) 20 (SS–DMER_3)
−1 (doctor) −2 (auditor) 21 (SS–DMER_4)
−21 (SS–DMER_4)
1 (doctor) 2 (auditor) −3 (nurse) 16 (MS–DMER_4)
1 (doctor) −2 (auditor) 3 (nurse) 16 (MS–DMER_4)
−1 (doctor) 2 (auditor) 3 (nurse) 16 (MS–DMER_4)
−1 (doctor) −2 (auditor) −3 (nurse) 16 (MS–DMER_4)
−1 (doctor) −2 (auditor) 17 (MS–DMER_5)
−1 (doctor) −3 (nurse) 17 (MS–DMER_5)
−2 (auditor) −3 (nurse) 17 (MS–DMER_5)
−17 (MS–DMER_5)
1 (doctor) −2 (auditor) 22 (SS–HMER_3)
−1 (doctor) 2 (auditor) 22 (SS–HMER_3)
−1 (doctor) −2 (auditor) 23 (SS–HMER_4)
−23 (SS–HMER_4)
1 (doctor) −2 (auditor) 18 (MS–HMER_3)
−1 (doctor) 2 (auditor) 18 (MS–HMER_3)
−1 (doctor) −2 (auditor) 19 (MS–HMER_4)
−19 (MS–HMER_4)
```

Figure 4.4: Set of hard clauses generated for the example

Table 4.4: Experimental Settings

$\|U\|$	$\|R\|$	$\|P\|$	$\|S_u\|$	$\|C_t\|$	$\|H\|$
100	40 - 300	80	10	5	100
80	50	60	2 - 50	5	100
100	60	80	10	5	10 - 100
100	60	100 - 1000	10	5	50

problem instances for UAQ problem. Here, we discuss various dimensions that determine the underlying RBAC policy RP, the length of history H, and the choice of session s with the set P_{lb} and P_{ub} of permissions.

Table 4.4 provides the values for various components of an RBAC policy RP (namely the number of users $|U|$, roles $|R|$, permissions $|P|$, sessions per user $|S_u|$, constraints per type $|C_t|$) and the length $|H|$ of histories that we have considered in the experiments. In the columns $|R|$, $|S_u|$, and $|H|$, a range $m - M$ of values (where m and M are the minimum and maximum values, respectively) corresponds to three plots that will be discussed in Section 4.4.3, where the performances for increasing numbers of roles and sessions, and longer histories, respectively, are measured. Since $|S_u|$ represents the number of sessions per user, in order to compute $|S|$, it is sufficient to multiply it by the number $|U|$ of users (hence, there is a linear dependence between the number of users and the number of sessions). Similarly, in order to obtain the total number of constraints C in the underlying RBAC policy RP, one must multiply the value in column $|C_t|$ by 4, which is the number of distinct MER constraint types considered in this work. (We have not considered CARD constraints in our experiments for the sake of simplicity; their addition is straightforward and does not change our findings in Section 4.4.3.) For example, a setting with $|U| = 100$, $|S_u| = 2$, and $|C_t| = 4$ gives $|S| = 200$ and $|C| = 16$ for the RBAC policy RP. Each one of the four types of MER constraints has the form MER(rs, n), where rs is a sub-set of the set R of roles and n is a positive integer which should be less than or equal the cardinality of rs. In our experiments, rs is a randomly selected sub-set of R containing 3 roles and n is randomly chosen among the values of 2 and 3. The relation PA in the RBAC policy RP is a randomly chosen subset of $R \times P$ while the generation of the relation UA has been designed so as to augment the chances to violate some of the MER constraints as follows. First, we randomly extract a sub-set rs_{sub} of the set rs of roles of a randomly selected MER constraint among those available in C. Then, we associate a user u to all the roles in rs_{sub} plus some randomly selected roles from $R \setminus rs_{sub}$. We repeat these two steps for each user u in U. The role hierarchy $\succeq$ of the RBAC policy RP is not considered in our experiments since it can be compiled away in a pre-processing step by distributing the permissions to each role.

For the RBAC policy RP, the values of the first five columns in Table 4.4 were inspired from those discussed in [KTZ11]. Unfortunately, no value for the number of sessions is provided although [KTZ11] multiple sessions are taken into consideration.

The generation of valid histories takes a substantial amount of time because, as we will describe in Section 4.4.2, it requires to solve a number of instances of UAQ problem which is at least equal to the length of a history. As a consequence, to keep the amount of time required by history generation reasonable, we were able to consider RBAC policies categorized as "Literature" in [KTZ11]. Notice also that [KTZ11] considers only the problem of enforcing RBAC policies and not the UAQ problem as done here.

We are left with the discussion of our choices for (s, P_{lb}, P_{ub}, obj) to complete the description of UAQ dimensions. The session s is randomly selected from the set S of sessions. Concerning P_{lb} and P_{ub}, we randomly select a set $perms \subseteq P_u$ of permissions and then let

1. $P_{lb} = P_{ub} = perms$ if $obj = \mathsf{any}$,
2. $P_{lb} = perms$ and $P_{ub} = P_u$ if $obj = \mathsf{min}$, and
3. $P_{lb} = \emptyset$ and $P_{ub} = perms$ if $obj = \mathsf{max}$;

where P_u is the set of permissions that can be acquired by the user u, associated to session s, if all his/her roles are activated, i.e. $P_u = \{p \mid \exists r \in R \text{ s.t. } (u, r) \in UA \text{ and } (r, p) \in PA\}$. Case 1 means that user u wants to activate a set of roles associated with the exact set *perms* of permissions. Case 2 implies that at least the permissions in *perms* should be available in the session s of user u. Case 3 is used when the principle of least privilege must be ensured, as no permission outside *perms* should be available to user u in session s. Case 1 is called "exact match" in [WQL09] where it is shown equivalent to the UAQ problem defined in [ZJ08]. Cases 2 and 3 were also solved in [WQL09] albeit considering only what we call SS-DMER constraints in our work (for a comparison with [WQL09], see Section 4.4.3).

4.4.2 Generation of valid histories

Since sessions and histories play a key role in the satisfaction of MER constraints, we are required to generate histories with multiple sessions in a flexible way to evaluate the efficiency of our approach. We have implemented a procedure for the generation of valid histories containing multiple sessions as follows. The core of the procedure is a function generate which takes as input a session s and a sub-set *perms* of the set P of permissions, while using the sets U, S, and P together with the relations PA and UA (these last elements are assumed to be generated as explained in Section 4.4.1).

```
function OneSession($s$)
1  $P_u \leftarrow \{p \mid \exists r \in R$ s.t. $(u,r) \in UA$ and $(r,p) \in PA\}$
           where $u$ is the user of session $s$
2  let $perms$ be a randomly selected sub-set of $P_u$
3  return generate($s, perms$)

function AllSessions($S_a$)
1  if $S_a = \emptyset$ then return fail
2  else begin
3      randomly pick a session $s \in S_a$
4      $\mu \leftarrow$ OneSession($s$)
5      if $\mu \neq$ fail then return $\mu$
6      else $S_a{}' \leftarrow S_a \setminus \{s\}$; return AllSessions($S'_a$)
7  end

function OneStep($H@\rho$)
1  $\mu \leftarrow$ AllSessions($S$)
2  if $\mu =$ fail then
3      if $H = []$ then return fail
4      else return OneStep($H$)
5  else return $H@\rho@(\rho \oplus \mu)$

function AllSteps($H, b$)
1  if $|H| < b$ then
2      return AllSteps(OneStep($H$), $b$)
3  else return $H$
```

At the top level, invoke AllSteps($[\rightarrow \emptyset], n$) with $n \geq 1$

Figure 4.5: History generation

The function generate solves the UAQ problem instance $(s, perms, perms, \mathsf{any})$ by using a modified version of the procedure UAQ-Solve in Figure 4.3, called UAQ-Solve^r, and returns either fail, when the problem is unsolvable, or a new state (i.e. a finite mapping $\rho : S \rightarrow 2^R$ associating each session to a set of roles), when the problem is solvable. The main difference between UAQ-Solve^r and UAQ-Solve is that, when the UAQ problem is solvable, the former does not compute the new state by using the first solution returned by the SAT solver as the latter would do. Rather, UAQ-Solve^r computes the new state from a randomly selected solution among those available. This was easy to implement since most of the available SAT solvers support mechanisms to enumerate one after the other all satisfying assignments of a set of clauses. The reason behind the use of solutions to have some variety in the generated histories, i.e. every pairs of consecutive states in a generated history should differ in the set of roles that are activated in a given session whereas all the others remain the same.

Before describing in more detail all the functions in Figure 4.5, we introduce some notions. The singleton mapping associating a session s with a set rs of roles is written as $\{s \mapsto rs\}$, the mapping ρ' such that $\rho'(s_1) = \rho(s_1)$ for each session $s_1 \neq s$ and $\rho'(s) = rs$ (for some set rs of roles) is denoted by $\rho \oplus \{s \mapsto rs\}$, and $\rightarrow \emptyset$ is the abbreviation for the mapping that associates each session with the empty set of roles. The variable μ ranges over singleton mappings. The empty history is written as $[\,]$, and the length of a history H is denoted by $|H|$ (indeed, $|[\,]| = 0$).

To generate a history of length n, we invoke the function AllSteps on the history containing just $\emptyset$ and the second parameter set to n. This is a recursive function that returns a history of length n after n calls to itself. At each recursive invocation, the function OneStep is invoked on the actual history $H@\rho$ (notice that initially we have $[]@(\to\ \emptyset)$) which tries to return a history with one additional state appended at the end, by recursively invoking itself and the function AllSessions. The latter is capable of returning either fail or a singleton mapping $\mu = \{s \mapsto rs\}$ for some session s (among those in input) such that $\rho(s) \neq rs$ for a given state ρ. Then, it is tested (line 2) if AllSessions has returned fail in which case OneStep backtracks and tries to extend history H (instead of $H@\rho$) when H is not empty (line 4); otherwise, it reports failure (line 3). If AllSessions has not returned with failure, then μ is a singleton mapping and OneStep appends at the end of $H@\rho$ the mapping $\rho \oplus \mu$. In this way, the recursive call of AllSteps has extended the input history with one new state that introduces a change (e.g., a new active role) to one of the sessions.. We now consider the function AllSessions which, in turn, calls the function OneSession. We start to describe the latter, which takes as input a session, establishes to which user u is belongs (line 1), computes the set P_u of permissions associated to u according to the relation UA, and then randomly selects a sub-set *perms* of P_u. At this point, generate is invoked on s and *perms*. This allows us to compute valid histories that correspond to activations and deactivations of roles by a certain user u in a given session s as explained above. We emphasize that the use of randomization tries to make histories more heterogeneous, considering several possible activations and deactivations patterns. Finally, we describe the function AllSessions which takes a set $S_a \subseteq S$ of sessions as input and randomly picks one among them, say s when S_a is not empty (line 3); otherwise it reports failure (line 1). Then, OneSession is invoked on s and if it returns a singleton mapping μ, this is also returned by AllSessions (line 6); otherwise (i.e. when OneSession returns fail) AllSessions is invoked recursively on the set of remaining sessions, i.e. all those in S_a except s.

In our implementation, we have used the Sat4J [1] library to provide the SAT solving capability (i.e. implementation of generate). This is so because we can sacrifice a bit the efficiency of SAT solving in favor of a seamless integration (via the available API) within the application that implements the functions in Figure 4.5; thus making it very easy to compute a new state out of a propositional assignment.

4.4.3 Results

We have implemented the procedure UAQ-solve in Figure 4.3 in Java. Third party tools have been integrated to leverage state-of-the-art SAT encoding and solving techniques. Concerning the former, we have used the routines described in [Sin05] for the compact

[1]http://www.sat4j.org/

encoding of Boolean cardinality constraints derived from the various types of DMER constraints, which put restrictions on the number of Boolean variables that are allowed to be true at the same time. This has been used in the implementation for the generation of the initial set $\mathcal{C}(s)$ of clauses at line 14 of UAQ-solve. Concerning SAT solving, we have chosen the QMaxSAT solver [Kos11] to implement the PMAX-SAT-solve function invoked at line 34 of UAQ-solve. We have chosen QMaxSAT because it performed quite well in the latest MaxSAT evaluation and because of its efficiency on the sets of clauses generated by our encoding.

Our implementation was run on a large set of synthetic UAQ problem instances obtained by randomly generated RBAC policies (see Section 4.4.1) and randomly generated histories (see Section 4.4.2). According to the values in Table 4.4, we consider four scenarios: (a) increasing number of roles (first line of the table) from 40 up to 300 with a step of 10, (b) increasing number of sessions (second line of the table) from 160 to 4000 (recall that $|S| = |S_u| \times |U|$) with a step of 2, (c) increasing history length (third line of the table) from 10 to 200 with a step of 5 and (d) increasing number of permissions (fourth line of the table) from 10 to 1000 with a step of 30.

All the experiments have been conducted on a computer with Intel Xeon 3.20 GHz CPU and 4 GB RAM running Linux. The three plots below show the behavior of our implementation on the three scenarios. In all plots, the x-axis reports increasing values of one of the dimension in Table 4.4—namely, $|R|$ for (a), $|S_u|$ for (b), and $|H|$ for (c) whereas the y-axis shows the timings of our implementation of UAQ-solver in milliseconds. For the two plots corresponding to scenario (b) and (c), the y-axis adopts a standard scale to report the timings. Instead, for the plot describing the behavior in scenario (a), the y-axis adopts a log-scale. In all experiments, each point in the y-axis is obtained from the median value of the overall timings of UAQ-solve over 10 randomly selected pairs of values for P_{lb} and P_{ub} for given RBAC policy, history, and session s. This choice was made to reduce the variance in solving an instance (s, P_{lb}, P_{ub}, obj) of the UAQ problem when P_{lb} and P_{ub} are randomly selected. This phenomenon is particularly acute in scenario (a) when obj is either min or max. All the points in the plots include the time needed to generate the set of propositional clauses and that taken by the solver to establish satisfiability or unsatisfiability. Now, we analyze each of the four scenarios in detail. In the plots, lines labeled with 'EXACT' refer to UAQ instances (s, P_{lb}, P_{ub}, obj) where $P_{lb} = P_{ub} = perms$ and $obj = \mathsf{any}$, for those with 'MAX' we set $P_{lb} = perms$, $P_{ub} = P_u$, and $obj = \mathsf{min}$, and for those with 'MIN' we set $P_{lb} = \emptyset$, $P_{ub} = perms$, and $obj = \mathsf{max}$ for some randomly selected sub-set $perms$ of P_u, which is the whole set of permissions that can be acquired by the user u associated to session s. (The choice of P_{lb} and P_{ub} for the various objectives of the UAQ problem was explained in Section 4.4.1.)

(a) Increasing number of roles. Figure 4.6 shows the plot for scenario (a), i.e. when the number of roles increases from 40 to 300. The time necessary for encoding and solving

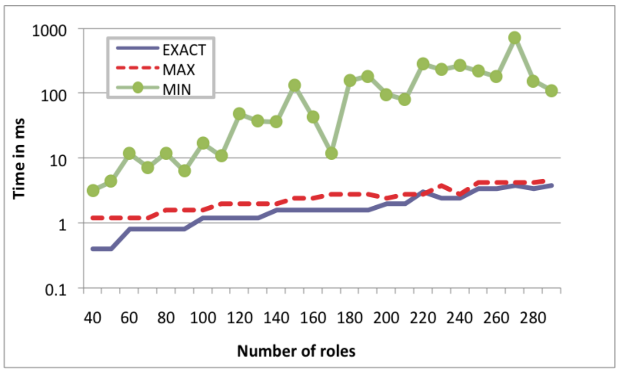

Figure 4.6: Increasing number of roles

each UAQ instance is less than 10milliseconds up to 300 roles for $obj \in \{\mathsf{any}, \mathsf{max}\}$. When $obj = \mathsf{min}$, there is a degradation in performance since, while the time taken for the generation of the clauses remains the same, the time taken for the solution of the generated PMAX-SAT problem increases significantly, as expected. However, we observe that the timings in this case are still below 1 second. Furthermore, notice that for most of the RBAC policies classified as 'Literature' by the authors of [KTZ11], the number of roles is not larger than 250 and our choice of 300 as the maximum number of roles is compliant with this indication. Interestingly, for such a number of roles, our technique gives a performance of about 200 milliseconds.

(b) Increasing number of sessions. Figure 4.7 shows the plot for scenario (b), i.e. when the number of sessions per user increases from 2 to 50 (and the corresponding number of sessions goes from 160 up to 4000). In this case, regardless of the type of satisfiability

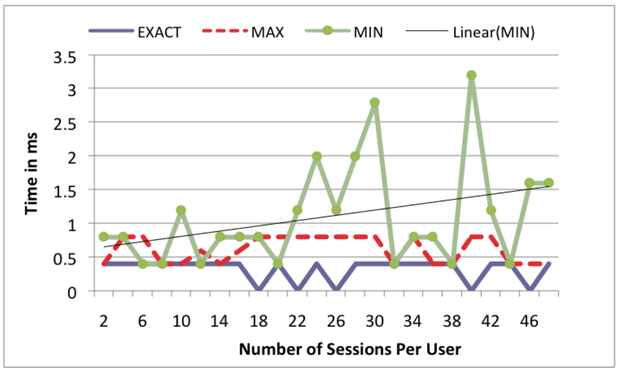

Figure 4.7: Increasing number of sessions per user

problem that must solved, the performances stay below 3.5 milliseconds and there is no

significant degradation when $obj = \mathsf{min}$. We can conclude that increasing the number of sessions per user does not have a significant impact on the performance of our technique.
(c) Increasing history length. Figure 4.7 shows the plot for scenario (c), i.e. when the history length increases from 10 to 200. In this case, regardless of the type of satisfiability

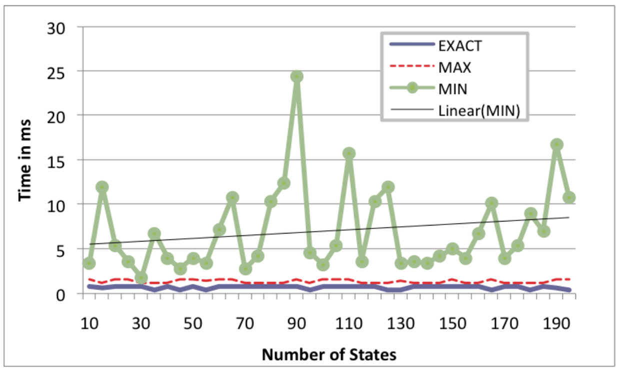

Figure 4.8: Increasing history length

problem that must solved, the performances stay below 25 milliseconds and there is only a slight degradation for $obj = \mathsf{min}$. In fact, solving the UAQ problem instances always takes less than 2 milliseconds for $obj \in \{\mathsf{any}, \mathsf{max}\}$ and goes up to (almost) 25 milliseconds when $obj = \mathsf{min}$. We can conclude that increasing the history length has almost no impact on the performances of our technique.
(d) Increasing number of permissions. Figure 4.9 shows the plot for scenario (d) in logarithmic scale, i.e. when the number of permissions increases from 10 to 1000. In

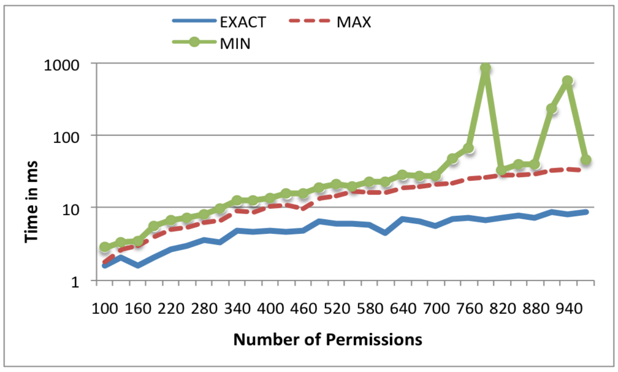

Figure 4.9: Increasing number of permissions

general, the encoding time for UAQ problems with varying number of permissions is less than 15 ms. Total time required to solve UAQ instances with 1000 permissions remains under 1s. However, as can be noted from the figure, the number of permissions has an important impact in solving the SAT instance. This can be explained from the fact that

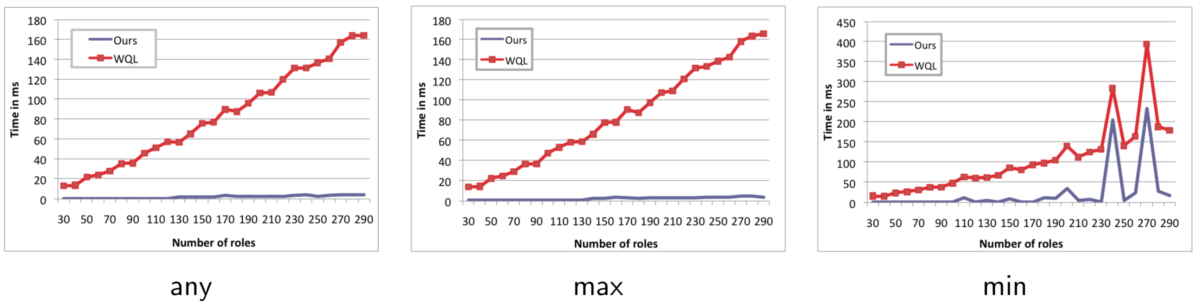

Figure 4.10: Comparison with the SAT-based procedure of Wickramaarachchi, Qardaji, and Li (2009)

many permissions can construct a clause by themselves. In particular, the steps 2 and 3 of Theorem 4.2.1 and its encoding (lines 9 - 23 of $UAQ - Solve$ procedure) at run-time involves permissions and generates unit clauses for them.

4.4.3.1 Comparison with the SAT-based procedure of Wickramaarachchi, Qardaji, and Li

Since the UAQ problem considered in [WQL09] can be seen as an instance of the one defined here when only SS-DMER and CARD constraints are considered, it is interesting to compare the performances of the two approaches. To this end, we have implemented the SAT encoding proposed in [WQL09] and used the same MaxSAT solver (QMaxSAT) for the sake of a fairness in comparison. Moreover, the set of roles used in the comparison represents all the roles of the system R rather than the user's roles in the UA relation and the first step (i.e. item 1 in Section 4.3) is applied by both techniques. We believe that this not only makes two approaches in line but also provides an additional optimization in SAT solving by providing more information to the solving process. This can be easily observed from the figures presented in this section and the ones available in [WQL09].

We have considered a similar set of experiments as the ones performed in [WQL09] with some larger number of roles. More precisely, we considered $30 \leq |R| \leq 300$, $|U| = 50$, $|P| = 80$, $|C| = 5$ (recall that we only consider SS-DMER constraints), and used the approach described in Section 4.4.1 to randomly generate a suitable RBAC policy. We did not report the number of sessions and the length of the history since they are not relevant to establish the satisfiability of SS-DMER constraints. Similar to the experiments in Section 4.4.3, we take the median value over 10 randomly selected pairs of values of P_{lb} and P_{ub} for the given RBAC policy.

Figure 4.10 reports the results of the experiments for any (corresponding to the 'exact match' case of [WQL09]), max, and min cases as defined at the end of Section 4.4.1 for our approach (lines labelled "Ours") and that of [WQL09] (lines labelled "WQL"). In the any and max cases, the timings of our approach are less than 20 milliseconds and in the min case, it is less than 250 milliseconds. Instead, the approach presented in [WQL09] grows linearly up to around 160 milliseconds in the any and max cases and up to 400 milliseconds for the min case. Clearly, our technique outperforms that in [WQL09] in the any and max cases while the difference is less striking for the min case. As for the results reported above, the degradation in the performance in our approach is due to the longer running times taken by the solver to handle min case. The time needed to generate the encoding remains negligible in our approach while it grows almost linearly for increasing number of roles [WQL09]. The key to explain the superiority of our approach is in the observation that most of the clauses in solving UAQ problems can be computed off-line (line 14 of Figure 4.3) and only a small number of (unit) clauses need to be computed at runtime (see procedure add-YO-clauses of Figure 4.3 and the definition of $\overline{\chi}_k(r, s)$ in Table 4.1).

4.5 Supporting Separation of Duty (SoD)

As an extension of our approach for UAQ problems we gave initial thoughts about supporting SoD constraints. This allows the specification of constraints for supporting SoD from a user perspective (i.e. user cardinality). We left its experimental analysis as future work however we believe that major part of the encoding can be handled in an off-line manner as discussed before. While there are different ways of specifying SoD constraints we adopt the notion of Role-based Separation of Duty from [LTB07a] because of its seamless integration with our framework. The original definition of RSoD constraints put restrictions on the assignment of roles to users and they are transformed into SMER constraints for enforcement. In our approach, we interpret RSoD constraints as restrictions on the simultaneous activation of roles such that enforcement spans through multiple users. This provides a less restrictive approach to role assignment. A user can be assigned a role that her job function requires but can not acquire the permissions associated with it under certain circumstances.

A *t-n* RSoD *(t-out-of-n Role Separation of Duty)* policy is written as

$$rsod(rs, t)$$

where rs is a set of roles of cardinality $|rs| = n$, n and t are integers such that $1 < t \leq n$. Intuitively, $rsod(rs, t)$ means that there should not exist a set of fewer than k users that together have all the roles in rs. In other words, the roles in a *t-n* RSoD policy are those roles needed to carry out a sensitive task, and the policy guarantees that at least t users are needed to successfully complete it.

4.5.1 Encoding RSoD Constraints

It is not difficult to see that the propositional formulae encoding the condition in Theorem 4.2.1 that correspond to $rsod(rs,t)$ are

$$\bigvee_{r \in rs} \bigwedge_{s \in S_X} \neg\overline{r \in \rho_k(s)}$$

for every X such that $X \subseteq U$ and $|X| = t-1$. Since U can be very large, the number of these formulae can be dramatically high since it is equal to the combination of $n = |U|$ objects taken $t-1$ at a time, i.e. $n!/(t-1)! \cdot (n-t+1)!$. Thus, the encoding above is too naive to scale up for large RBAC policies. Fortunately, there is still room for some optimizations:

– Some parts of the encoding can be generated in an off-line manner similar to MER constraints.

– In every $|X| = t-1$, the user u, who is the owner of the session in the query must be contained, $u \in X$. This decreases the size of the possible combinations to some extent. Specifically, the size of the possible combinations becomes

$$\binom{|U|}{t} - \binom{|U|-1}{t}$$

– Finally, we need not consider all users in U and we can instead focus on those users of the active sessions only.

We still have the problem of converting (4.1)—which is in disjunctive normal form—into a set of clauses that can be processed by the solver. Fortunately, the following procedure transforms (4.1) into an equi-satisfiable set of clauses in linear time (it is an adaptation of existing standard techniques [PBZ03]). Introduce a fresh propositional variable sod_r^X for each $r \in rs$ and compute the the set $CL^X_{rsod(rs,t)}$ of clauses:

$$\begin{array}{l} \neg sod_r^X \vee \neg\overline{r \in \rho_k(s)} \text{ for each } s \in S_X \\ \bigvee_{r \in rs} sod_r^X. \end{array}$$

It is easy to see that $CL^X_{rsod(rs,t)}$ is equi-satisfiable to (4.1). Notice that the clauses associated to each $s \in S_X$ are obtained from the clausification of $sod_r^X \rightarrow \bigwedge_{s \in S_X} \neg\overline{r \in \rho_k(s)}$. We did not use the biconditional here because we are interested in generating an equi-satisfiable set of clauses rather than a logically equivalent one.

4.6 Conclusions and Future Work

We have described a carefully tuned SAT encoding to solve instances of the UAQ problem that outperforms and extends the existing proposals. In particular, our approach brings the time necessary to solve an UAQ instance to the level milliseconds which is 100 times more efficient than that of state-of-the-art [WQL09]. We also extended the types of authorization constraints by considering the multiple sessions and role activation histories of the same user. An important feature of our reduction to SAT is the generation of large part of the propositional clauses in a pre-processing phase and leave only the addition of simple (unit) clauses at run-time so that the time required for the encoding is greatly reduced.

We also presented a novel algorithm to instantiate consistent UAQ instances in several dimensions including history and random behavior in assignment relations. We believe that the presented generation algorithm may help researchers in obtaining problem instances with varying sizes. We have experimentally evaluated our procedure to show its practical viability and compared it with the approach presented in [WQL09].

As a future work, we would like to experimentally analyze the encoding we introduced for Role Based Separation of Duty (RSoD) constraints. We also intend to investigate further optimizations of our encoding to handle very large RBAC policies containing thousands of users, roles, and permissions as those categorized as ‘Synthetic’ in [KTZ11].

Chapter 5

Performance Analysis of XACML Policy Decision Point (PDP) Implementations

In the previous chapters of the thesis we first presented an approach to analyze access control policies in an off-line manner and then proposed a constraint aware run-time procedure that can be used as a decision procedure with abstract access control policies and requests. Among the various policy specification languages we chose XACML for our analysis.

XACML has become the *de facto* standard for specifying access control policies and has found a wide industrial acceptance. Especially many commercial products that fall into the Service Oriented Architecture (SOA) category, integrated XACML in their authorization components. Together with the language, the Organization for the Advancement of Structured Information Standards (OASIS) XACML technical committee defined the architecture and the components for the creation, enforcement and management of XACML policies. One of the components specified in the OASIS XACML architecture is the *Policy Decision Point (PDP)*. The goal of the PDP is to evaluate the access control requests received as input against a set of XACML policies and to return a decision (e.g. permit or deny). The PDP is a crucial component of the whole architecture since it is involved in any access control decision and therefore potentially invoked at every request. Optimizations such as caching decisions can contribute to reduce the number of evaluations, but they don't change the fundamental fact that PDP is a critical component for the performance of the entire access control system.

Another important workload factor is the average number of policies per application that PDP needs to evaluate, and this number grows for several reasons. One reason can be explained in the context of Web Services and how they are used in application development. Often services are composed of multiple independent peer services (e.g. mash-ups) each with its own security policy. These policies may need to be combined with the need of specifying fine-grained policies for different business contexts. This, in turn, leads to a substantial growth in the number of security policies a PDP needs to evaluate.

The situation is even worse in those scenarios where the PDP is provided as a third party service to multiple organizations. In these kind of settings, the number of security policies can be very high. For instance, the Amazon Simple Storage Service (S3) [Ama08] allow service providers to specify security policies for their hosted services. These security policies are evaluated and enforced when customers request access to the services. Such evaluation is performed by S3 on behalf of the service provider. Clearly, in this scenario, scalability is an issue and this is probably one of the reasons why S3 supports only very simple security policies (e.g. access control lists). With more Web Service hosting platforms similar to S3 appearing over the Internet [Dat08], a rigorous study on how well current implementations of PDP cope with high workload is an interesting question.

In this chapter of the thesis, we fill this gap by performing an extensive testing on three open source implementations of XACML PDP: Sun XACML Engine [Sun], XACMLight [XACb] and XACML Enterprise [XACa]. Our analysis showed that all tested implementations perform poorly as the number of XACML policies increases (> 100) or policies are slightly more complex than the very basic ones (e.g., only one rule) when the the policy evaluation is considered as a whole. We will show how this is mainly due to two factors: the way that the XACML policies are made available and represented in memory (i.e. memory loading) and how the policy rules are evaluated. Our study indicates that more efficient PDP implementations are needed to support scalability. In fact, several efficient engines such as XEngine [AXL08] appeared with this motivation. Although XEngine represents the state-of-the-art for efficient XACML policy evaluation and we could not include it in the experiments, we provide a summary of their approach.

The contributions of this chapter can be summarized as the following: First, we analyze how these engines work internally during the policy evaluation in order to study the feasibility of integrating an intelligent decision procedure such as the one presented in the previous chapter. In particular, we identify the steps that are involved in access control decisions and that constitute a decision procedure. Second, we perform a set of syntactic experiments to compare the performance of the considered engines. The output of our work can be used for the selection of a suitable engine for the system deployment. In our tests, we defined several usage patterns such as large number of policies and large number of rules that can raise performance bottlenecks in real life situations. As a byproduct of our performance analysis, we built a test suite that could constitute a possible benchmark also for developers of future PDP implementations. Finally, we provide some suggestions for the optimization of such engines. A preliminary version of the ideas discussed in this chapter appeared in [TC08] and [TMCB08].

5.1 eXtensible Access Control Markup Language (XACML)

XACML is an XML-based access control language with a simple but expressive syntax. Figure 5.1 presents a reduced grammatical representation of XACML language (i.e. policy and context) we produced by analyzing the specification document. It is encoded in Extended Backus-Naur Form (EBNF). EBNF representation summarizes how an XACML policy is generated from a set of atomic attribute id-value pairs and demonstrates its rich expressivity. XACML follows an attribute based approach (ABAC) to represent access control components: subjects, resources, permissions and the constraints. Although this approach allows the specification of policies for many different access control models and diverse applications, it also brings a penalty on the performance. Moreover, as we will see XACML does not have formal semantics and it requires additional techniques to provide security guarantees in terms of constraints.

An XACML *policy* is basically composed of three components: Policy *target*, *rule/s* and *obligation/s*. The Policy Target section is used to seek the applicability of a policy for a given request. It can include sets of *Subject/Resource/Action/Environment* elements (abbreviated as S/R/A/E in what follows) elements inside. XACML Policies can include different numbers of *rules* whose effects are combined with a *combination algorithm*. A Rule specifies a decision (e.g. Permit), a target element similar to Policy target and may contain a *condition* that a request must further satisfy for the rule to be applicable (e.g. a request sent between 8pm and 8am is applicable). Rule target together with policy target (i.e. p.target $\wedge$ r.target), specifies additional attributes for request-rule matching. Conditions are rich set of functions introducing computational constraints (e.g. temporal) on additional attributes of S/R/A/E. Some of these functions include equality predicates (e.g. string-equal), logical and comparison functions (e.g. and/or operators). As also explained in Chapter 3, conditions play a key role in the specification and enforcement of policies. An XACML rule results with either "permit" or "deny" if it is applicable and its condition is satisfied. When there is more than one rule in a policy, there might be conflicts among their decisions. To resolve possible conflicts in a policy, XACML provides five rule combining algorithms: *(Permit/Deny)_overrides*, *Ordered(Permit/Deny)_overrides* and *First_Applicable*. *Obligations* define the post actions to be performed after exercising the permissions defined in the allowed request. Within the context of this chapter, we leave

```
PolicySet ::= (PolicySet)* PolicyCombiningAlgId  Target  (Policy)* Obligations?
PolicyCombiningAlgId ::=  ("deny-overrides" | "permit-overrides" |
          "first-applicable" | "only-one-applicable" | "ordered-deny-overrides"
          | "ordered-permit-overrides")
Target ::= Subjects?  Resources?  Actions?  Environments?
Policy ::= RuleCombiningAlgId  Target  ( … | (Rule)+)  Obligations?
RuleCombiningAlgId ::= URI ("deny-overrides" | "permit-overrides" |
          "first-applicable" | "ordered-deny-overrides"
          | "ordered-permit-overrides")
Rule ::= RuleId  ("Permit" | "Deny")  Description?  Target?  Condition?
Obligation ::= ObligationId  ("Permit" | "Deny")  (AttributeAssignment)*
Subject ::= (AttributeValue (SubjectAttributeDesignator | AttributeSelector))+
Resource ::= (AttributeValue (ResourceAttributeDesignator | AttributeSelector))+
Action ::= (AttributeValue (ActionAttributeDesignator | AttributeSelector))+
Environment ::= (AttributeValue (EnvironmentAttributeDesignator
          | AttributeSelector))+
Expression ::= Apply | AttributeSelector | AttributeValue | Function
          | VariableReference | (Expression | AttributeDesignatorType)
          | (Expression | AttributeDesignatorType) | (Expression  |
          (AttributeDesignatorType SubjectCategory?)) | (Expression |
          AttributeDesignatorType)
Condition ::= Expression
AttributeAssignment ::= AttributeId AttributeValue
AttributeValue ::= Expression | (Any)* DataType
Apply ::= FunctionId (Expression)*
AttributeSelector ::= RequestContextPath  DataType MustBePresent?
(S/R/A/E) AttributeDesignator ::= Expression | AttributeDesignatorType

Request ::= Subject+ Resource+ Action Environment
```

Figure 5.1: XACML Syntax in EBNF

obligations out of scope as they are handled on the site (e.g. client application). Figure 5.2 shows an example of a policy defined using XACML. In particular, the policy is composed of two rules: the first rule, called (Report_Access), states that only Managers affiliated to Sales Department are permitted to modify the Sales Report, whereas the second rule, called (FinalRule), denies all other cases.

```
<Policy PolicyId="Policy0" RuleCombiningAlgId="Permit-Overrides">
<Description>Sales Report Policy</Description>
<Target/>
<Rule RuleId="Report_Access" Effect="Permit">
<Target>
  <Subjects>
     <Subject>
       <SubjectMatch MatchId="string-equal">
           <AttributeValue DataType=" string">Manager</AttributeValue>
           <SubjectAttributeDesignator AttributeId="subject-id" DataType="string"/>
       </SubjectMatch>
     </Subject>
  </Subjects>
  <Resources>
     <Resource>
       <ResourceMatch MatchId="string-equal">
           <AttributeValue DataType="string">Sales Report</AttributeValue>
           <ResourceAttributeDesignator AttributeId="resource-id" DataType="string"/>
       </ResourceMatch>
     </Resource>
   </Resources>
   <Actions>
      <Action>
       <ActionMatch MatchId="string-equal">
           <AttributeValue DataType="string">Modify</AttributeValue>
          <ActionAttributeDesignator AttributeId="action-id" DataType="string"/>
       </ActionMatch>
      </Action>
   </Actions>
</Target>
<Condition>
   <Apply FunctionId="string-equal">
      <Apply FunctionId="string-one-and-only">
         <SubjectAttributeDesignator AttributeId="Division" DataType="string"/>
      </Apply>
     <AttributeValue DataType="string">Sales Department</AttributeValue>
    </Apply>
</Condition>
</Rule>
<Rule RuleId="FinalRule" Effect="Deny"/>
</Policy>
```

Figure 5.2: Example policy encoded in XACML

To provide better organization of Policies, XACML supports *PolicySets*. PolicySets act as a container for different Policies or other PolicySets of an organization to create physical or logical units. For example, an enterprise can express resource (e.g. service, hardware) access rules of all divisions with one Policy per department in a PolicySet and provide a high level view to authorization. While this approach gives high flexibility, it also increases the possibility of having conflicts among policies. To eliminate possible policy conflicts XACML provides policy combining algorithms which are similar to rule combining algorithms and an additional algorithm that makes sure there is only one applicable policy : *Only_One_Applicable.*

Because XACML allows the specification of policies that contain many rules and policy sets that contain many policies, policy management in XACML can easily become a complicated task. This demystifies the rationale behind a large body of XACML static formalization works up-to-date [Kol08, KHP07, HB08, FKMT05, RLB^{+}08] trying to provide a formal grounding to XACML for management and analysis.

5.1.1 Policy Evaluation Architecture

In order to analyze the performance of policy evaluation in XACML, we have to understand the interactions between XACML architectural components. The specification document presents the evaluation architecture in relatively generic terms. Depending on the access control model and the application domain, the components that take part in policy evaluation can be tightly or loosely coupled. This allows administrators or developers to customize the system entities according to their requirements. Moreover, the provided architecture enables the sharing of system components such as PDP among different organizations for cost-efficiency. Relevant to the work presented here, Figure 5.3 illustrates one possible approach to implementation. There are four components:

– **Policy Decision Point (PDP)**: makes an access control decision for the given request by evaluating it against the available policies/policysets.

– **Policy Enforcement Point (PEP)**: forwards the access requests to PDP and enforces the obligations returned as a result of positive decision.

– **Context Handler**: generates a context from the request and the attribute stores made available by PIP, forwards it to PDP.

– **Policy Information Point (PIP)**: provides the attributes requested by context handler.

There is also an additional component, policy administration point (PAP) that makes the policies available to PDP. However, majority of the time PAP is integrated to PDP or Context Handler. In fact, the PDP implementations we tested integrated PAP into PDP and it was difficult to isolate its functionality. Thus, we will assume that it is part of PDP in the folowing sections.

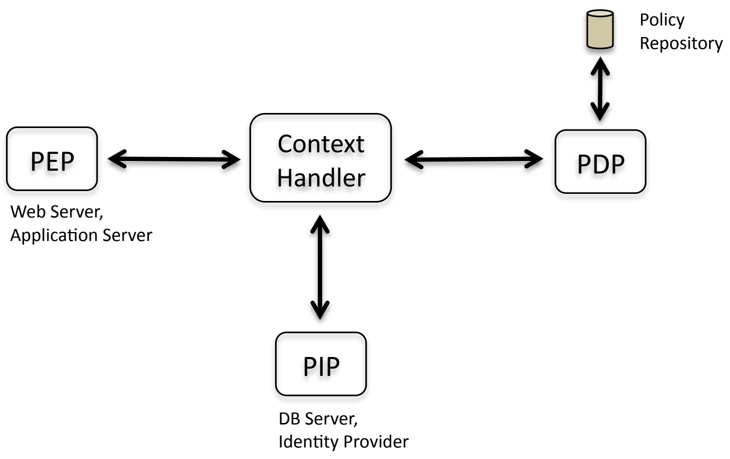

Figure 5.3: XACML Evaluation Architecture

5.2 PDP Implementations

At the time of our writing, there were only four available open source implementations of XACML v2.0: XEngine, Sun XACML, XACMLight and XACML Enterprise. While the ideas presented in XEngine [AXL08] represent the state-of-the-art we could not include XEngine in our tests because of the following reasons. First, while the authors claim that XEngine supports XACML v2.0 in [AXL08], the implementation supports only v1.2. In particular XEngine uses an external library to parse XACML policies and this library parses only XACML v1.2 policies[1]. Second, XEngine considers conditions as uninterpreted functions (Con) that are resolved at a later phase. Specifically, the overall policy evaluation function $Eval(r)$ for a request r is defined as $Match(r) \wedge Con(r)$ and XEngine internal data structure handles the $Match$ function. Hence we can conclude that XEngine implementation can not handle conditions. We provide a summary of the techniques introduced in XEngine. There is also a .NET implementation of XACML named XACML.NET but it does not support XACML v2.0 policies. Moreover, having all engines implemented in Java enabled us to isolate the implementation logic from the environmental factors such as the varying performance of underlying virtual machines. For these reasons we excluded both XEngine and XACML .NET from our experiments. What follows is a brief description of the tested PDP implementations in our experiments.

5.2.1 Sun XACML

[Sun] is the reference implementation of XACML and it is by far the most widely used engine both in commercial applications and research projects. It has been implemented in

[1]We contacted the authors and made them aware of the issue.

Java and follows the *jaxax.xml.parsers* interfaces for XML operations. Any XML parser that is compatible with these interfaces can be used for XML operations. We used Apache Xerces2 2.9.0 [Xer] during the experiments. Many of the existing open source or proprietary XACML solutions are based on Sun XACML. This proves that it has been widely tested against bugs with different usage patterns. Sun XACML claims to be XACML v2.0 compliant. It has a modular design so that developers can extend the implementation by their own modules such as for policy loading and optional XACML functions. In our Sun XACML experiments, we have extended sample policy finder modules of the distribution for policy load. The implementation comes also with some examples for different ways of obtaining policies such as local file access or Web retrieval.

5.2.2 XACMLight

[XACb] has been designed as a Web service with the necessary components (i.e. PDP an) for policy evaluation. It is a Java implementation and uses XMLBeans to access the XML elements. The distinguishing character of XMLBeans from other XML processing models is it maps (almost) all features of the given XML document into equivalent Java contructs and types. This leads to important performance gains in XML processing. XACMLight was originally conceived designed as a Web service for authorization decisions and supports XACML v2.0. However, to eliminate the overhead caused by Web service mechanisms and messaging, we have executed XACMLight as a console application. XACMLight is still under active development and provides with useful features not available in existing implementations. For instance, it supports multiple results obtained from the evaluation of a single request containing multiple resources. If there are multiple independent policies for evaluation it creates a PolicySet on the fly to wrap all policies.

5.2.3 XACML Enterprise

[XACa] is the third PDP engine we considered and also the most advanced in terms of policy evaluation. It has been designed as efficiency in mind. A simple API is also provided for PAP. XACML Enterprise has been implemented in Java and uses Xerces for XML operations. It supports XACML v2.0. XACML Enterprise has many mechanisms for efficient policy evaluation such as target indexing and caching of policies and results. Target indexing significantly speeds up the target matching for a given request. Because of evaluation result caching, XACML Enterprise can bring important performance gains by avoiding redundant policy evaluations.

Summary of XEngine XEngine has performance gains in order of magnitudes (e.g. for some cases 103) compared to Sun XACML engine. It employs efficient data structures and special pre-processing techniques to improve the evaluation performance. In XEngine the XACML policies are tranformed into a multi valued decision diagram (MDD) via two steps: numericalization and normalization. The former converts all string values (which are expensive to process) into corresponding integer values for efficient number processing. The latter converts an XACML policy with a hierarchical structure into a flat structure by doing some preprocessing such as resolution of references. Moreover, an XACML policy with four rule/policy combining algorithms is transformed into a policy with only one rule combining algorithm (i.e. First-Applicable) as a result of the normalization step. This way unnecessary applicability checks of requests can be avoided. These two steps together result with an MDD that is efficient to process. A request evaluation become a reachability check on this MDD.

5.3 Performance Evaluation

In this section, we summarize our analysis of XACML PDP implementations and present the experimental results. There are two important stages when making access control decisions in XACML based authorization systems:

– Loading of policy/policies from disk (or external source) to main memory,

– Request evaluation against loaded policies

The first phase prepares the policies for evaluation. The load of a policy is basically an XML load operation composed of the following steps in the context of XACML: 1. Parsing and validation of the policy file: the given policy file(s) is parsed with an XML parser and a validation is performed using the XACML schema document. XACML Technical Committee provides a set of XML Schema Document (XSD) files in which the syntactical details of XACML policies and requests are described. 2. Creation of an memory model of the policy: this step highly depends on the XML library used. If the library uses Document Object Model (DOM) for internal representation then a DOM tree is created in the memory. The parsing results are placed into a DOM structure which can be serialized back to the original document. The time spent in this phase is proportional to the number of policies considered in the evaluation. Briefly, the more policies are evaluated, the longer it takes in this phase. There is also the time necessary for loading requests into memory. However, this time is negligible compared to policy loading due to the size and complexity of requests. We expect that the requests would be much smaller in size and simpler in structure compared to policies.

The second phase includes evaluation of the received request against available policy/ies. This phase is when the PDP implementations can be distinguished from each other as it heavily depends on the data structures used to represent XML objects (e.g. DOM) and their respective manipulation in memory. The request evaluation can result in four states: Deny, Permit, Not Applicable or Indeterminate. Certain optimizations can be obtained by restructuring the policy representation in memory. For instance, the policy can be re-organized according to combining algorithms so that a more efficient processing can be obtained. However, one must be careful to preserve the semantics of the evaluated policy when doing such preprocessing. For instance, the post actions (i.e. *Obligations*) must be preserved if a set of policies is combined into a single one without careful handling. Along with the obligations, the resulting decision produced by request evaluation phase is enforced by the PEP component of XACML.

Apart from the above mentioned factors effective in the performance of an XACML engine, there is also Java Virtual Machine (JVM) state effecting the performance. To mitigate this effect we have run all experiments several times and computed their average to get the final result. In the following sections, we will present the details of our test suite and the experimental results obtained from the experiments.

5.3.1 Policy Test Suite

To gather a close proximity to the real world usage patterns for XACML, we have designed an experimental schema with different elements representing different usage scenarios. Each item in the schema provides a view to the access control problem on diverse environments. For instance, in a service oriented system, we expect that it would be the number of policies that will increase to large numbers where the policies draw the organizational boundries. In an authorization system used within the boundaries of a single enterprise, however, we would expect that the number of rules can increase to large numbers. Finally, it is highly probable that the policies of independent parties would be similar to each other as they work within the same context. For example, there can exist similar specification patterns such as the use of a service restricted in a rule such that all methods accessing to a special database can only be invoked within a given time interval or a given resource can only be accessed by the given subject provided its connection type is a particular one (e.g. SSL connection). Thus, there can definitely be some overlaps among different policies and significant performance improvements can be obtained if similar policies are grouped together. For instance, [LRBL07] presents a new approach for policy similarity among different organizations during the access to a shared resource. Their approach simply groups the similar components of policies and evaluates the similarity on hierarchy and numerical distance. In what follows, we present the details of our experimental settings:

- Large Number of Policies: In these type of systems, there is a large number of parties eager to specify their access control requirements, and a single PDP (maybe bought as a third party service) is shared among all those parties. In our experiments, we have tested 4 different cases of large policy numbers: 10, 100, 1000 and 10000 policies. All these policies are used for the evaluation of the given request. All policies consist of 4 rules and the rule combining algorithm for each policy is selected randomly.

- Large Number of Rules: A single organization with large number of users and resources having a single point of control can have many rules within a single policy. We tested 5 cases : 10, 50, 100, 500 and 1000 rules in a single policy. For each case, there were 10 policies used in each evaluation run meaning 10 policies with 10 rules each and so on. This also ensures diversity in testing.

- Policy Similarity: We also have policy similarity tests to see if there is any mechanism embedded in the engines for the similarity analysis of policies. Our similarity tests included 10 policies comprising 10 rules each. We tested different degree of similarity among these rules (e.g., 20% means that all these policies has 2 out of 10 rules that are the same)

Another important factor on the performance of policy evaluation is the effect of dimension of each element. XACML policies can contain different number of rules, subjects, resources, actions, conditions and obligations. They can also contain varying size of attribute values within these S/R/A/E elements. This, in turn, is reflected as variable computing requirements (e.g. memory and CPU) during the representation of policies in the memory. Under normal circumstances, the content of attribute elements are not expected to be very high as attribute references can be used instead of putting all content of a resource into the policy. However, unlike the subject and action sections, resource sections can be large in size. To mitigate this effect, we generated the policy contents with the same pattern for all engines in our experiments. For instance, if a text value is generated then its size would be fixed.

For all policy settings, the number of Subjects, Resources, Actions has been selected as 4, 2, 2 respectively in order to have a simple but common XACML element size. Each rule contains a Condition on the subject specified in rule target "subjects" section. The attribute values of Subject,Resource and Action sections are generated randomly to avoid conflicts. The attributes in condition elements are also generated randomly either as an integer or a string value.

5.3.2 Testbed

The testing environment is a Windows Vista desktop computer with the following hardware configuration: 3.4 GHz Pentium IV CPU, 2GB RAM, 160 GB Serial ATA (7200 rpm) HDD. During the tests only the essential system services were allowed to run. JVM has been enabled to allocate between 256 MB and 1024MB of heap space.

5.3.3 Experiment Results

All PDP implementations have been tested with the same set of policies that have been generated with our generator. Moreover, each request used in the experiments has been generated by using a request template.

5.3.3.1 Large Number of Policies

In large number of policies test, 1 XACML request has been evaluated against 10, 100, 1000 and 10000 policies respectively. The graphical representation of the results is summarized in Figure 5.4 and 5.5. We could not perform 10000 policies test with XACMLight as it requires more memory than the other engines and we did not have that much memory on our testbed.
Load: In the loading phase, XACML Enterprise performed worst in all cases as it does special indexing for efficient evaluation. It took 168 seconds to load 10000 policies with XACML Enterprise while it was 126 seconds with Sun XACML. While the number of policies increased, the differences between load times increased almost linearly. XACMLight performed best in all cases when loading. We believe this is mainly because it uses XMLBeans for XML operations and XMLBeans performs better than DOM oriented approach when accessing XML documents.
Evaluation: Because of additional indexing mechanisms and caching of evaluation results, XACML Enterprise performs very efficiently compared to other engines during evaluation. It took 72 miliseconds to evaluate 10000 policies with XACML Enterprise. Sun XACML processed the same number policies within almost 1 second. XACMLight, however, performed worst in evaluation. It took almost 3 seconds to process 1000 policies with XACML Light. As can be seen in 10000 policies case Sun XACML performs worse than XACML Enterprise.

5.3.3.2 Large Number of Rules

In large number of rules experiments, 10 XACML policies were used at a time. The number of rules in have been 10, 50, 100, 500 and 1000 in each testing case for varying

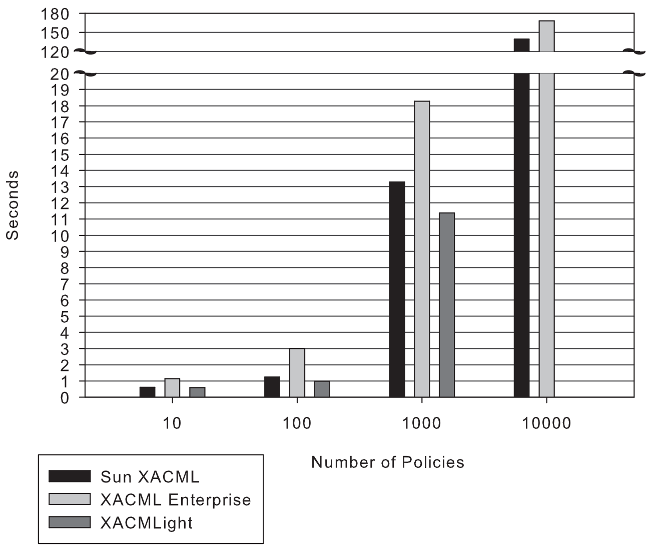

Figure 5.4: Large Number of Policies (Load)

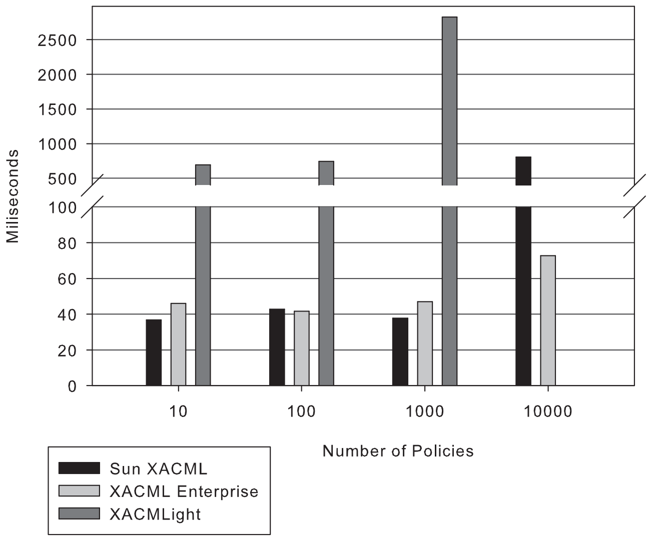

Figure 5.5: Large Number of Policies (Evaluation)

number of rules. The obtained results are shown in Figure 5.6 and 5.7.
Load: XACML Enterprise performed worst again in loading. It took around 34 seconds to load 10 policies with 1000 rules while it was 14 seconds with Sun XACML. XACMLight performed best in loading with almost 3 seconds for 10 policies each having 1000 rules.
Evaluation: The effect of rule number up to 100 was not so obvious as it only changed in the orders of 10 miliseconds for Sun XACML. After 100 rules, the evaluation time became

proportional to the number of rules in the policy. For XACML Enterprise, we did not observe a clear effect of rule number because of the effect of result caching and indexing. It evaluated 10 policies with 1000 rules within 125 miliseconds. XACML Light reacted almost proportionally to the number of rules. It completed processing 10 policies with 1000 rules within 6 seconds.

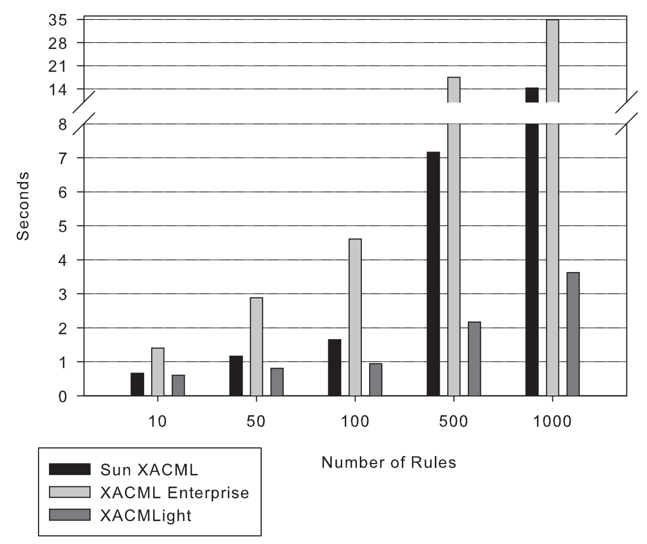

Figure 5.6: Large Number of Rules (Load)

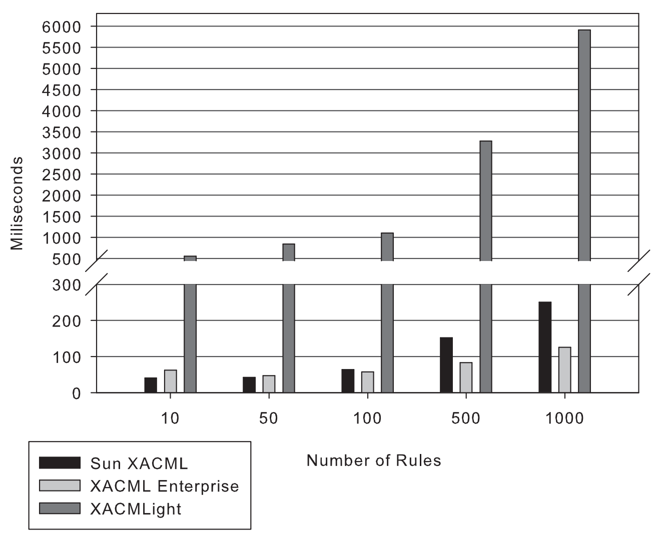

Figure 5.7: Large Number of Rules (Evaluation)

5.3.3.3 Policy Similarity

We have conducted extensive tests on policy similarity among 10 policies. There are 5 scenarios (i.e. %20, %40, %60, %80, %100) in the similarity of policies and 5 scenarios (i.e. %20, %40, %60, %80, %100) on the similarity of content. For example, the scenario *%20 of the policies are %60 similar to each other* would make 2 of the policies have 6 rules that are the same in their contents in a total of 10 policies with 10 rules inside. We wanted to investigate two issues with these experiments: first, to see if there is an intelligent mechanism that anticipates the similarity at the level of policies and the rules, and avoiding unnecessary evaluation by grouping similar policies and rules together as suggested in [CMBS05]. Second, to analyze what kind of similarities we can have in policies at a very high level.

As a result of our experiments, we couldn't find any similarity analysis of policies and their content supported by the engines. Figure 5.8 and 5.9 which are very similar to the 10 policies case in large number of policies tests, summarize the obtained results.
Load: In any of the XACML PDP implementations we could not find any similarity analysis. As a consequence, all engines gave similar loading times on all tests among different similarity cases. However, the loading results obtained in large number of policies scenario came up into prominence for each individual engines.
Evaluation: In addition to the case that there was no similarity analysis during policy loading, there was no special grouping of similar policies in the memory. Hence, there was no major change in evaluation times of all engines in teh case of similarity.

5.3.4 Testing Response Times

As a final test case, we have also tested the stability and response times of XACML PDP implementations with different number of requests arriving simultaneously. As Figure 5.10 shows the response times of XACML Enterprise and Sun Engine were very close to each other. XACMLight was almost 5 times higher in general. The response time curve of XACMLight did not follow the same pattern of the other engines as its performance deteriorate quickly with a number of requests higher than 50.

5.4 Discussion

Our analysis showed that each engine has its own strengths and weaknesses. In general, policy loading is expensive with XACML Enterprise because of additional mechanisms such as indexing and caching. However, once the policies are loaded it is paid off quite well with an efficient evaluation phase. Sun XACML performed in the middle with reasonable

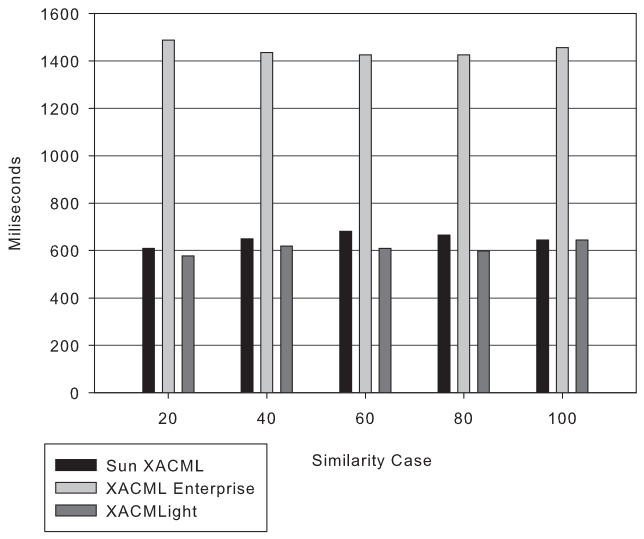

Figure 5.8: Similarity Tests(Load)

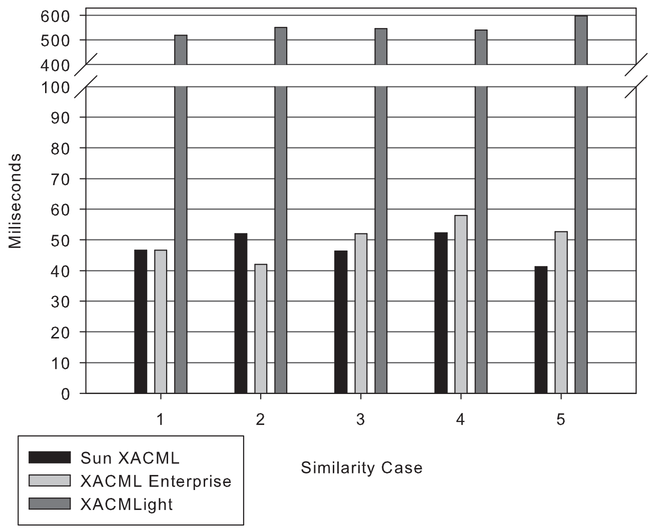

Figure 5.9: Similarity Tests(Evaluation)

timings both in loading and evaluation. However, it lacks of additional mechanisms such as result caching and target indexing available in XACML Enterprise to speed up the evaluation. This pushed back Sun XACML behind XACML Enterprise in evaluation. XACMLight performed quite well during the load as it is using XMLBeans for XML operations. However, overall XACMLight performances were much worse than the other two. As the number of policies and rules increase, XACMLight performed an order of

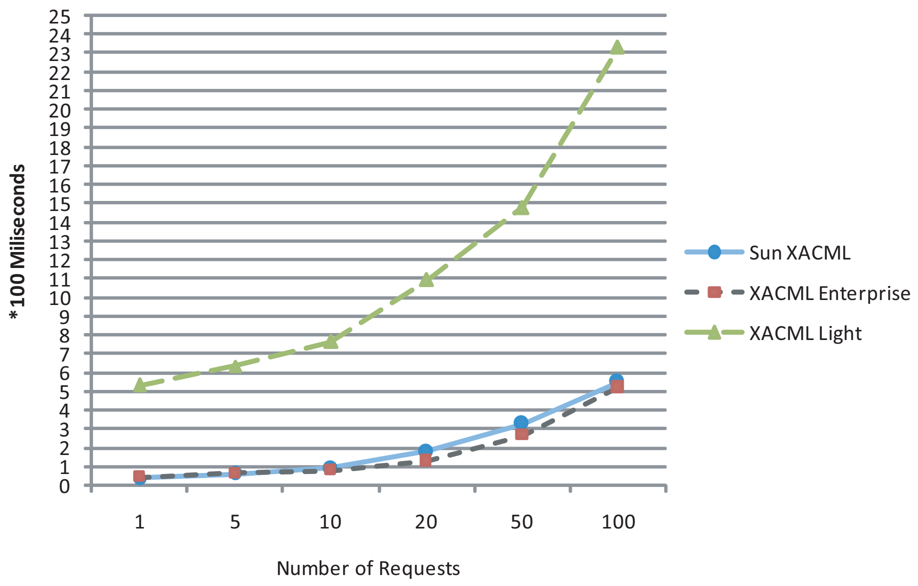

Figure 5.10: Request/Response Times

magnitude worse than the others. Summarizing our findings, we can say that if the frequency of policy loadings to the memory is not very high then XACML Enterprise is definitely a better choice. In extremely dynamic environments in which the policy repositories are modified very frequently and the number of simultaneous requests is low then XACMLight can be a good choice. However XACMLight does not scale well with large number of policies because of large memory requirements. Sun XACML performs reasonably well in all cases but there is definitely a possibility of optimization.

5.5 Possible Optimizations

During our experiments, we observed some possible optimization points to improve the efficiency of PDP implementations. As there are two stages (i.e. policy loading and evaluation) when making an access control decision in XACML, we will discuss different optimization possibilities for these two phases.

Loading: The verbose syntax caused by XML makes the policy loading very expensive (i.e. orders of magnitude more time than the evaluation). Efficient XML technologies can be used to store policies in the memory. For instance, an XML database can efficiently store policies and provide indexing services for faster access times. Frequently received requests can be profiled and a caching mechanism can eemployed. Moreover, efficient XML representation technologies such as binary XML[2] format can serve as an alternative to

[2]http://en.wikipedia.org/wiki/Binary_XML

formatting. For instance, there is one promising binary XML based effort, Efficient XML Interchange (EXI)[Gro11], that encodes a given text XML in a binary form. Documents encoded in binary form have smaller sizes and such a technology could help XACML find usage in computing devices with limited resources. However, this may require the assistance of tools that generate a binary XML document from a given high level policy specification. Moreover, additional libraries might be necessary for processing the binary encoded policies.

Evaluation: Although, according to our experiments, policy loading plays a key role in the overall performance of access control decisions, the policies can reside in memory for longer periods most of the time. For example, in an authorization system where the policies do not change frequently and enough hardware resources are available, the policies can be loaded periodically instead of at every request arrival. In these cases, custom data structures and algorithms for policy evaluation can improve the performance. Decision diagrams, such as MDDs can present an efficient alternative to encode policies and rules in the memory. If the policies or the policysets do not contain obligations, then the policies can be restructured so that unnecessary policy and rule evaluations are avoided. For instance, two combining algorithms *Permit_Overrides* and *Deny_Overrides* can be cast as *First_Applicable* as proposed by XEngine. Once a policy is restructured, analogically similar but moreefficient search algorithms for implementing the First_Applicable combining algorithm can be found in different research areas such as of query processing and decision procedures. We also consider that the boolean satisfiability (SAT) based techniques such as Satisfiability Module Theories (SMT) can be employed if a constraint-aware policy evaluation is sought. Such an approach allows also reasoning about certain properties at run-time. On the other hand, all these mechanisms require a preprocessing of the policies for converting them to acceptable formats and one must be careful whether the proposed appraoch pays off at all.

5.6 Conclusions and Future Work

In this chapter, we presented an empirical study of XACML PDP implementations performance. The cost of evaluation is given by two operations: loading the policy from disk to memory and then the actual evaluation of the submitted request against the loaded policies. Between the two phases, the loading is the most expensive in the case of large number of policies (> 100). Loading gets more expensive, almost proportionally, by increasing the policy number. Among the tested implementations, XACML Enterprise performed best in terms of evaluation time. It was the worst in policy loading. Sun XACML had reasonable performance but it does not implement any of the optimizations of the other two engines. XACMLight had scalability problems with large number of policies. At the time of writing, we discovered the availability of another open source XACML engine, namely Herasaf [Her], but could not include it in our experiments. As a future work, we are planning to extend our XACML engine performance studies with XEngine and Herasaf and enrich our test policy suite with real examples of XACML policies.

Chapter 6

Related Work

Many approaches have been proposed to analyze whether important security requirements and constraints are satisfied in *policy design*. However the majority of these proposals are limited to static security constraints such as static separation of duty (SSoD) or focused on administrative features of RBAC. These approaches are usually unaware of the run-time context and fail to verify the security constraints associated with the run-time. Moreover, dynamic constraints usually require implementation support adding another layer to their analysis [FB09]. To our knowledge, only a few papers exist on run-time enforcement of dynamic constraints [BFA99, WQL09, DFK06]. In this thesis, we first provide an approach to analyze dynamic constraints in RBAC with XACML in an offline manner. We further present a decision procedure for RBAC-based authorization systems that integrates reasoning to policy evaluation at run-time. In the following sections, we provide a systematic overview of the relevant scientific literature.

6.1 Off-line Analysis of RBAC Systems

Because RBAC has been standardized and widely deployed, it has attracted the attention of many researchers. It has played a key role in the development of access control research in terms of models, languages and technologies. For the same reason, it enjoys a wealth of literature and a large community support. Many RBAC variants have been proposed with different expressive features when it comes to constraints. Examples include RBAC models with temporal constraints on role activations in [BBF01] and a generalization of [BBF01] to a wider class of constraints in [JBLG05], spatial constraints in [DBCP07, NKZ10] and spatio-temporal constraints in [TR11]. RBAC has been expressed in many different formal and informal notations [SA00, FCAG03, XO05b]. In this thesis we use some of the constraints presented in these works regarding the specification of DMER and DSoD constraints.

6.1.1 Static Analysis

In the context of more expressive languages and more complex security requirements, the need of static analysis of RBAC policies has arisen. [LT06] proposed to use state transition systems to analyze an RBAC system for simple properties. An access control scheme, denoted as $\langle \Gamma, Q, \vdash, \Psi \rangle$ where Γ is the set of states, Q is the set of queries, $\vdash$ is the query checking function and Ψ is the state transition rule, constitutes a framework for analyzing whether the given set of queries is satisfied. [JLT^{+}08] extended the same framework by including the principals A and the actions Σ in the scheme, and supported the user-action association in the analysis. The motivation of these works was to reduce the administrative overhead. [SWL^{+}11] investigated the complexities of similar analyses and presented a list of complexity results and supporting methods for various constraint types. Their results and methods can be used to ease the administration of a given RBAC configuration.

6.1.2 Dynamic Analysis

There is a reasonable amount of research on dynamic analysis in RBAC. One of the early works on dynamic analysis of RBAC systems is presented in [GGF98] where the separation of duty policies are enforced based on the system states and the transitions between them. The authorization function, $auth =: STATES \times ROLES \times OBS \rightarrow 2^{OPS}$, maps each role (in conjunction with run-time states and objects) to a set of operations. The transitions between states are modeled (i.e. partitioned) and pre/post-conditioned according to the composed application *plans*. These plans can be considered as snapshots of time during system execution where an access control evaluation is required. A similar approach is followed in [KMPP02]: the system states are represented as graphs and the constraints are verified by a set of graph transformations. The graph nodes represent users, roles (and hierarchies), and sessions. A transition from one state to the other is regulated by a set of transformation rules which are simply the proper encodings of run-time constraints.

Datalog-based approaches have constituted the majority of research in dynamic analysis. A role based trust management with dynamic features, namely RT, is presented in [NL02]. RT combines some features of RBAC and trust management systems for the specification and analysis of distributed applications. It employs authorized attributes (i.e. credentials) to make access control decisions in a dynamic setting. In a similar vein, [BS04] propose an access control system, Cassandra, that contains a Datalog-based policy language and a formal semantics for access control policy evaluation. By using the access control semantics, they define clear semantics for the conditions and the consequences of role activation/deactivation and permission requests. [Bec09] work on a similar system but focus on the analysis of specifications to ensure the given properties hold. [BN10] propose a logic to support the state changing events in an RBAC setting. In this logic, the history of allowed access requests is taken into account (i.e. state update) when making an access control decision. This is done by an extension to Datalog that contains statements for state modification and negation. The ideas presented in this thesis span both theory and implementation of an RBAC-based authorization system with dynamic constraints.

6.2 Constraint Enforcement for RBAC

Many constraints described in the literature such as *DSoD* and their associated mechanisms such as *MER*, require some form of run-time support [AS00, Ber06]. For example, imagine that a bank has a MER policy to prevent fraud on transactions. A transaction comprises three steps: creation, validation and checkout. No employee can perform alone all three operations required to complete a transaction (dynamic MER constraint). The bank can also set up a policy that limits the total amount of transactions approved by one teller and put day time limits on the operations. A teller can act as a supervisor only for transactions that are not created, validated and checked out by himself/herself. These dynamic constraints cannot be enforced statically because the access control system evolves with the actions occurring in the system. Moreover, the satisfaction of many constraints (e.g. time of day) can not be not known a priori.

Because SoD has been a prominent feature of RBAC via MER specifications, it has been mainly discussed in the context of RBAC. However, SoD constraints are generated and specified independently of the underlying access control model. Recent research has focused on addressing constraints and security design principles such as least privilege individually [CC07a, DJ06, CC07b] with run-time context ([XWZC09]). [CC07a] try to support the least privilege principle as a role mapping problem between different domains. [DJ06] and [CC07b] focus also on the least privilege in a distributed setting. [DJ06] define the problem of finding the set roles in a domain (i.e. inter-domain role mapping (IDRM)) that are authorized for a set of permissions requested from an external domain. [CC07b] provide a solution to the same problem based on set cover problem. A more general constraint enforcement framework is presented in [CK09] in which the enforcement of constraints is transformed into an authorization checking problem by using RBAC states. They employ relational algebra whether a derived state (after a set of events) is among the permitted states (or prohibited states). In our approach, we have enabled the use of existing SAT tools for constraint enforcement by reducing the problem of user authorization queries to a model checking problem. Our proposal has supported 4 types of MER constraints and role activation constraints, and presented a preliminary encoding for role based SoD constraints [LW08]. We have put ”sessions” together with the history as core concepts in our model.

Li et at. [LTB07b] show that even the enforcement of SSoD constraints is intractable (coNP-complete), and proposed to convert SSoD requirements into RBAC SMER. Our approach is in the same vein except it uses DMER to support DSoD. Xu et al. [XWZC09] proposed to use XACML in administrative RBAC systems to specify how administrative changes in the policies (user assignment and permission assignment) can be reflected at run-time. They investigated the use of lock mechanisms to handle concurrency of sessions and provided XACML solutions for the constraint enforcement.

The prominent work of Li and Wang [LW08] presented a SoD specific solution (SoD algebra - SoDA) to address the shortcomings of RBAC in addressing SoD constraints [LW08]. SoDA not only provides a formalism for constraint specification but also presents a methodology for secure workflow design. On the other hand, it does not exploit the notion of sessions in RBAC. Two approaches are offered for the enforcement: static enforcement via static safety checking (SSC) and dynamic enforcement via term satisfiability (TSAT). A SoDA policy regulating a ”payment transaction” process with two steps *sign* and *create* can be specified as follows

$$(Employee \cup Manager) \otimes Manager$$

Each step of a workflow, independent of the objects and ownerships, is assigned to a userset. Thus, SoDA can be too restrictive in terms of permissions that can be exercised by a single user. Indeed, if a userset and the corresponding permissions are not compatible with a high level security policy defined in SoDA, they are not allowed to do any action but denied completely. In order to have a tighter integration with the workflow systems [BBK09] provide a mapping of SoDA terms to workflows specified in communicating sequential processes (CSP) expressions.

The main motivations of these works are the inherent limitations RBAC standard, the flexibility of policy design provided by run-time policy enforcement and the dynamicity of contemporary applications. Another motivation is to enable the use of access control policies in diverse application domains such as workflows. In fact, there has been a line of research dedicated to the enforcement of constraints in workflow systems [BE01, BFA99, KPF01]. A permission to a task also implies relevant permissions to the objects effected by the task. For that reason, a design principle can not only be addressed at design time but must also be supported at run time with extensions to the implementations.

The RBAC standard defines a set of high level functions to model the requirements of applications while providing a bird-eye view to authorization. However, the use of these functions requires careful consideration for correctly enforcing dynamic constraints and supporting well-known security design principles (e.g. least privilege). Within this context, sessions enable a finer control on the management of permissions as they enable the "role activation/de-activation". They are also a point of contention where the main RBAC criticism originates [LBB07]. Recent research showed that there is a gap between the static RBAC model and its deployment with execution support when it comes to the enforcement of constraints. In particular, sessions are underspecified (e.g. sessions) [LTB07b, LW08] in the RBAC standard. Many papers investigated the issues related to RBAC run-time. As one of the remarkable works that proves this observation, [LTB07b] show that the RBAC standard lacks proper functionality in supporting SoD. Indeed, in the standard RBAC, the constraints are enforced per session and users can have multiple sessions and thus there is no mechanism to prevent a user to do illegitimate actions in multiple sessions.

More recently, Zhang and Joshi [ZJ08] formulated the User Authorization Query (UAQ) problem in two steps: a *role mapping* step for the set of requested permissions and an *activation checking* for the obtained roles when some run-time constraints are in place. They provided three algorithms for the role mapping step and one algorithm for the activation checking step. Addressing the same problem, Wickramaarachch et al. [WQL09] propose two solutions. The first solution employs a back-tracking search algorithm while the second is an approximation algorithm that reduces the problem to a MAXSAT problem. This allows the use of off-the-shelf SAT solvers for answering UAQs. However, both approaches only considered single sessions for authorization queries. More importantly, they have not included the history in decision making. In RBAC, a user can have multiple sessions and a constraint spans multiple sessions if not the whole authorization run-time. Yet, the activation of a role by a user at an instance of time may effect the activation of another role at another time by the same or other user/s.

[KTZ11] presented an empirical analysis of RBAC policy enforcement approaches in a distributed setting. Subsequently, they introduced a component, Secondary Decision Points (SDP), to the conventional policy enforcement architecture which is composed of policy decision point (PDP) and policy enforcement point (PEP). SDPs aid PEP in maintaining state information when enforcing access control rules and constraints. Whie their work focuses on the architectural level for the enforcement the approach presented in this thesis is closer to the model level. However, we used some of the experimental settings they presented.

6.3 XACML

Extensive research has been conducted on static and dynamic analysis of XACML policies. In particular, logic-based formalisms have been a supporting tool to do off-line security analysis by reasoning on authorization policies ([FKMT05, KHP07, HB08, RLB^{+}08]).

Another very relevant formalism is provided by Hughes et al. in [HB08] with a mathematical model and pre-defined partial orders among the access control policies to compare policy constraints. Combinators and partial orders are used to form policies and capture the semantics of the XACML combination algorithms. Attributes of XACML are encoded into boolean predicates and environments are defined as the Cartesian product of possible attributes. Such an encoding can have scalability problems especially for unbounded attribute values.

Based on Binary Decision Diagrams (BDD) [Bry86], Margrave [FKMT05] is used for analysis and verification of XACML access control policies. Margrave provides a fault model for access control policies and a structural testing of policy-request pairs. Policy-request pairs are obtained from mutation testing, and in two scenarios: 1) Solving Single-Rule Constraints in which each XACML rule is considered in isolation for evaluation, 2) Change-Impact Analysis in which the policies are fed to Margrave for version comparison among them. The authors do not concentrate on policy conflict detection but rather on conformance checking if the actual policies are those intended by the policy authors.

More recently, Description Logic (DL) based solutions became popular for security analysis. DL is a decidable subset of first-order logic, covering only "safe" rules to preserve decidability. Moreover, it has wide community support with many freely available tools such as Pellet[SP04]. Kolovski[KHP07, Kol08] et al. use Defeasible Description Logics (DDL) to formalize XACML policies. They map each XACML rule into a DDL rule and analyze it together with formally defined XACML combination algorithms. For each policy, they define a specific effect literal to build the head of the destination rules. In order to use the state-of-the-art DL reasoners, the derivability of effect-literals has to be reduced to a description logic concept satisfiability problem.

Finally, close to the the ideas presented in this thesis, [PMT08] propose a policy testing methodology for testing access control requirements, test generation for policies and their evaluation. The authors also generate the test inputs from policies in one case. However, they follow a different approach in testing, and do not consider the constraints. In our approach, instead, we follow more of a verification approach to testing by employing software model checker in our approach.

6.3.1 SoD with XACML

[FB09] presented an approach to support RBAC policies in XACML based on Web Ontology Language (OWL). They demonstrated how various types of SoD requirements can be handled as consistency checks in ontologies and extensions to the XACML language.

An alternative approach to support SoD in XACML is the use of *blacklists* [Cra03] proposed by Crampton. In this approach, constraint specifications are used to define bad sets and the requests falling into these sets are denied. Formally, a constraint specification $(\{p_1 \ldots p_n\}, k)$ defines an exclusion among the permissions such that a user can not exercise k permissions from the set $ps = \{p_1 \ldots p_n\}$. The enforcement of these constraints is achieved through blacklist policies for each user. After a user exercises $k-1$ permissions, ps_{k-1}, from $ps_{k-1} \subset ps$, his/her blacklist policy is updated by adding the rest of the permissions $(ps \setminus ps_{k-1})$ to a "deny" rule in the blacklist policy. The enforcement of blacklist approach also requires implementation support.

6.3.2 Performance

There has been some research on performance of access control policy evaluation. The few papers published so far about this problem mainly focus on the effect of the number of rules in a policy evaluation. [AXL08] presents a new XACML PDP engine designed with the specific goal of efficiency in evaluating policies with a high number of rules. Unfortunately, during our experiments presented in the Chapter5, the code for that engine was not available so we could not include it in our analysis. With this new engine, before the actual evaluation, policies are preprocessed through two phases: *numericalization* in which the string values of a policy are converted into integers, and *normalization* in which the hierarchical structure of a policy is flattened with some re-organization in the combining algorithms. The resulting engine is several orders of magnitude more efficient than the Sun XACML engine. However, the performance experiments conducted in [AXL08] are limited to the large number of rules case and only the evaluation is considered. Besides, because of two additional phases (i.e. numericalization and normalization), the loading of policies is expected to take longer than the other engines.

Finally, in [LHX08], the authors propose the following approach for the detection of erroneous access control decisions: three XACML engines are invoked with the same policy-request pairs and their output are compared. The majority output obtained from the engines is accepted as the correct decision. The focus of that work is the correctness of XACML engine output (i.e. XACML response) instead of performance.

Chapter 7

Conclusions and Future Work

The analysis of dynamic constraints requires either design-time projections of run-time events or integration of efficient run-time procedures to the actual policy evaluation process. The former implies exhaustive state generation for covering the largest number of possible cases and value approximations for the parameters that determine the access control decision context. The latter, instead, imposes efficiency considerations due to the feasibility requirements of run-time deployments. To address the first issue, we have presented a generic framework that mimics a real-time access control system in operation. Such an execution based approach has two main motivations: 1. Checking whether a constraint is enforced correctly is difficult without actually running the whole authorization system. 2. An enforcement mechanism for a dynamic constraint usually needs implementation support and this implementation can also be erroneous. Most of the existing approaches rely on the fact that, once an abstract model of an authorization policy (and the relevant constraint mechanisms) is verified, the correctness of enforcement is a matter of careful mapping between the model and its implementation. Analyzing the access control system by exhaustive querying has its own drawbacks too. In particular, if necessary measures are not taken, it is prone to the well known "state explosion" problem. As a result, certain parts of the execution for exhaustive analysis can be supported by symbolic representation of the policy so that the state space is reduced. Our framework includes an example of such mixed approach. It can be used by the security administrators to gain insights about an access control system as a whole and in operation. Certain configuration options are provided in the form of properties and parameters to test specific authorization scenarios and, more importantly, constraints. Finally, the proposed approach can also be used as a base for building more advanced verification methodologies.

If the size of the analyzed RBAC system grows, e.g. with a large number of users, the execution based approach can perform badly and further optimizations might be needed. As an alternative, we have described a run-time procedure that is both efficient and constraint aware. This procedure involves a SAT encoding to solve instances of the UAQ problem and overcomes the main limitations of previously available techniques. It also extends the types of authorization constraints so that the scope of applicability can accommodate multiple sessions and histories. One of the prominent features of this procedure is the way the propositional clauses are generated. Our observation is that a large part of the clauses can be generated off-line and this can reduce run-time overhead greatly. An extensive experimental evaluation showing the practical viability of our procedure has also been presented.

As future work, we would like to analyze the performance of our preliminary encoding to support SoD constraints in the context of authorization problems that appear in workflow management systems. We also intend to further optimize our encoding to handle very large RBAC policies containing thousands of users, roles, and permissions as those categorized as 'Synthetic' in [KTZ11].

At the last section of the thesis, we presented an empirical study of XACML PDP implementations in order to gain insights about run-time performance of XACML authorization systems. Our analysis served us to investigate ways of integrating efficient constraint checking to the evaluation process while improving existing evaluation schema. In general, the cost of evaluation is characterized by two operations: loading of the policy from disk and then the actual evaluation of the submitted request against the loaded policies. Between the two phases, the loading is the most expensive in the case of large number of policies (> 100). Specifically, loading time increases almost proportionally with the number of policies. We believe that our experimental results can be used by system administrators and developers to select which implementation suits best for their use case. As future work, we are planning to extend our studies with the state-of-the-art engine (XEngine [AXL08]) and to enrich our test policy suite with real examples of XACML policies. We believe this would lead us to a new authorization schema that is both efficient and constraint-aware.

Although there are still many unsolved issues in the enforcement of dynamic constraints, in this thesis we have addressed some of the most significant problems in this area and provided diverse approaches to solve them.

Chapter 8

Appendix A: Summary of Standard RBAC Functions

Function Name	Core RBAC	Hierarchical RBAC		SSD Relations			DSD Relations		
		General Role Hierarchies	Limited Role Hierarchies	Core RBAC	General Role Hierarchies	Limited Role Hierarchies	Core RBAC	General Role Hierarchies	Limited Role Hierarchies
(A)AddUser	X								
(A)DeleteUser	X								
(A)AddRole	X								
(A)DeleteRole	X								
(A)AssignUser	X		X	X	X				
(A)DeassignUser	X								
(A)Grantpermission	X								
(A)Revokepermission	X								
(S)CreateSession	X	X					X	X	
(S)DeleteSession	X								
(S)AddActiveRole	X	X					X	X	
(S)DropActiveRole	X								
(S)CheckAccess	X								
(R)AssignedUsers	X								
(R)AssignedRoles	X								
(R)RolePermissions	X								
(R)UserPermissions	X								
(R)SessionRoles	X								
(R)SessionPermissions	X								
(R)RoleOperationsOnObject	X								
(R)UserOperationsOnObject	X								
(A)AddInheritance		X	X		X	X			
(A)DeleteInheritance		X							
(A)AddAscendant		X							
(A)AddDescendant		X							
(R)AuthorizedUsers		X							
(R)AuthorizedRoles		X							
(R)RolePermissions		X							
(R)UserPermissions		X							
(R)RoleOperationsOnObject		X							
(R)UserOperationsOnObject		X							
(A)CreateSsdSet				X	X				
(A)AddSsdRoleMember				X	X				
(A)DeleteSsdRoleMember				X					
(A)DeleteSsdSet				X					
(A)SetSsdSetCardinality				X	X				
(R)SsdRoleSets				X					
(R)SsdRoleSetRoles				X					
(R)SsdRoleSetCardinality				X					
(A)CreateDsdSet							X		
(A)AddDsdRoleMember							X		
(A)DeleteDsdRoleMember							X		
(A)DeleteDsdSet							X		
(A)SetDsdSetCardinality							X		
(R)DsdRoleSets							X		
(R)DsdRoleSetRoles							X		
(R)DsdRoleSetCardinality							X		

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* PhD thesis, 1986.

[Ama08] Amazon. Amazon Simple Storage Service (S3) Developer Guide, 2008. http://sourceforge.net/projects/xacmllight/.

[AS00] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3, 2000.

[AXL08] Fei Chen Alex X. Liu. Xengine: A fast and scalable xacml policy evaluation engine. Technical Report MSU-CSE-08-2, Department of Computer Science, Michigan State University, East Lansing, Michigan, March 2008.

[BBF01] Elisa Bertino, Piero A. Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[BBK09] David Basin, Samuel J. Burri, and Gunter Karjoth. Dynamic enforcement of abstract separation of duty constraints. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[BE01] Reinhardt A. Botha and Jan H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3):666–682, 2001.

[Bec09] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *CSF*, pages 203–217, 2009.

[Ber06] Elisa Bertino. Access control models, 2006. `http://www.cs.unibo.it/ricerca/grad/biss2006/ACS/security-4.pdf`.

[BFA99] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. In *ACM Transactions on Information and System Security, Vol. 2, No. 1*, 1999.

[Bis04] Matt Bishop. *Introduction to Computer Security.* Addison-Wesley Professional, 2004.

[BLH10] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: a framework for partially evaluating finite-state runtime monitors ahead of time. *International Conference on Runtime Verification (RV)*, 2010.

[BN89] David Brewer and Michael Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, 1989.

[BN10] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS04] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *CSFW*, pages 139–154, 2004.

[BSH08] Eric Bodden, Reehan Shaikh, and Laurie J. Hendren. Relational aspects as tracematches. In *AOSD*, 2008.

[CC07a] Liang Chen and Jason Crampton. Inter-domain role mapping and least privilege. In *Symposium on Access Control Models and Technologies (SACMAT)*, 2007.

[CC07b] Liang Chen and Jason Crampton. Inter-domain role mapping and least privilege. In *SACMAT*, pages 157–162, 2007.

[Cena] NASA Ames Research Center. Java path finder (jpf). http://babelfish.arc.nasa.gov/trac/jpf.

[Cenb] NASA Ames Research Center. Symbolic path finder (spf). http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[CK09] Jason Crampton and Hemanth Khambhammettu. A framework for enforcing constrained rbac policies. In *CSE (3)*, pages 195–200, 2009.

[CMBS05] Bruno Crispo, Pietro Mazzoleni, Elisa Bertino, and S. Sivasubramanian. P-hera: scalable fine-grained access control for p2p infrastructures. In *11th International Conference on Parallel and Distributed System*, 2005.

[Cra03] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *SACMAT*, pages 43–50. ACM, 2003.

[Cra05] Jason Crampton. Xacml and role-based access control. In DIMACS Workshop on Secure Web Services and e-Commerce, August 2005. http://dimacs.rutgers.edu/Workshops/Commerce/slides/crampton.pdf.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, 1987.

[Dat08] Data hosting instead of data portability, 2008. http://ideas.4brad.com/data-hosting-instead-data-portability.

[DBCP07] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information Systems and Security*, 10, 2007.

[DFK06] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, pages 632–646, 2006.

[DJ06] Siqing Du and James B. D. Joshi. Supporting authorization query and inter-domain role mapping in presence of hybrid role hierarchy. In *SACMAT*, pages 228–236, 2006.

[FB09] Rodolfo Ferrini and Elisa Bertino. Supporting RBAC with XACML+OWL. In *ACM Symposium on Access Control Models and Technologies*, 2009.

[FCAG03] David F. Ferraiolo, Ramaswamy Chandramouli, Gail-Joon Ahn, and Serban I. Gavrila. The role control center: features and case studies. In *SACMAT*, pages 12–20, 2003.

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering (ICSE)*, 2005.

[GGF98] Virgil D. Gligor, Serban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *IEEE Symposium on Security and Privacy (S&P)*, 1998.

[Gol06] Dieter Gollmann. *Computer Security 2nd Edition*. Wiley, 2006.

[Gro11] World Wide Web (W3C) Efficient XML Interchange Working Group. Efficient xml interchange (exi), March 2011. http://www.w3.org/XML/EXI/.

[HB08] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *STTT*, 10(6):503–520, 2008.

[Her] Holistic enterprise-ready application security architecture framework (herasaf). http://www.herasaf.org/heras-af-xacml.html.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410, 2009.

[JBLG05] James B.D. Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17, 2005.

[JLT$^+$08] Somesh Jha, Ninghui Li, Mahesh V. Tripunitara, Qihua Wang, and William H. Winsborough. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable Sec. Comput.*, 5(4):242–255, 2008.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[KHP07] Vladimir Kolovski, James A. Hendler, and Bijan Parsia. Analyzing web access control policies. In *World Wide Web Conference (WWW)*, 2007.

[KMPP02] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security*, 5, 2002.

[Kol08] Vladimir Kolovski. *Logic-based Framework for Web Access Control Policies*. PhD thesis, 2008.

[Kos11] Miyuki Koshimura. Qmaxsat: Q-dai maxsat solver. In *http://sites.google.com/site/qmaxsat/*, 2011.

[KPF01] Myong H. Kang, Joon S. Park, and Judith N. Froscher. Access control mechanisms for interorganizational workflow. In *6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.

[KTZ11] Marko Komlenovic, Mahesh V. Tripunitara, and Toufik Zitouni. An empirical assessment of approaches to distributed enforcement in role-based access control (rbac). In *CODASPY*, pages 121–132, 2011.

[Lab12] Open Systems Laboratory. Actorfoundry, January 2012. http://osl.cs.uiuc.edu/af/.

[LBB07] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ansi standard on role-based access control. *IEEE Computer Society, Security and Privacy Magazine*, 2007.

[LHX08] Nuo Li, JeeHyun Hwang, and Tao Xie. Multiple-implementation testing for xacml implementations. In *TAV-WEB*, pages 27–33, 2008.

[LKDM10] Steven Lauterburg, Rajesh K. Karmani, and Gul Agha Darko Marinov. Basset: A tool for systematic testing of actor programs. *In the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2010.

[LRBL07] Dan Lin, Prathima Rao, Elisa Bertino, and Jorge Lobo. An approach to evaluate policy similarity. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 1–10, 2007.

[LT06] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur.*, 9(4):391–420, 2006.

[LTB07a] Ninghui Li, Mahesh V. Tripunitara, and Ziad Bizri. On mutually exclusive roles and separation-of-duty. *ACM Trans. Inf. Syst. Secur.*, 10, May 2007.

[LTB07b] Ninghui Li, Mahesh V. Tripunitara, and Ziad Bizri. On mutually exclusive roles and separation-of-duty. *ACM Transactions on Information and System Security (TISSEC)*, 2007.

[LW08] Ninghui Li and Qihua Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *Journal of the ACM (JACM)*, 2008.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[NL02] William H. Winsborough Ninghui Li, John C. Mitchell. Design of a role-based trust management framework. In *IEEE Symposium on Security and Privacy*, 2002.

[oSN04] National Institute of Standards and Technology (NIST). Role-based access control. *American National Standards Institute, Inc.*, 2004.

[PBZ03] David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified boolean formulae. *Discrete Applied Mathematics*, 130(2), 2003.

[PMT08] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-based tests for access control policies. In *ICST*, pages 338–347, 2008.

[RLB$^+$08] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. EXAM - a comprehensive environment for the analysis of access control policies. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2008.

[SA00] Michael E. Shin and Gail-Joon Ahn. Uml-based representation of role-based access control. In *WETICE*, pages 195–200, 2000.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sin05] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, pages 827–831, 2005.

[SP04] Evren Sirin and Bijan Parsia. Pellet: An owl dl reasoner. In *Description Logics*, 2004.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63, 1975.

[Sun] Sun's XACML Implementation. http://sunxacml.sourceforge.net/.

[SWL$^+$11] Yuqing Sun, Qihua Wang, Ninghui Li, Elisa Bertino, and Mikhail J. Atallah. On the complexity of authorization in rbac under qualification and security constraints. *IEEE Trans. Dependable Sec. Comput.*, 8(6):883–897, 2011.

[SZ97] Richard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *CSFW*, pages 183–194, 1997.

[TC08] Fatih Turkmen and Bruno Crispo. Performance evaluation of xacml pdp implementations. In *Proceedings of the 5th ACM Workshop On Secure Web Services, in Conjunction with ACM Conference on Computer and Communication Security (CCS)*, pages 37–44, 2008.

[TJC11] Fatih Turkmen, Eunjin Jung, and Bruno Crispo. Towards run-time verification in access control. In *IEEE International Symposium on Policies for Distributed Systems and Networks(POLICY)*, pages 25–32, 2011.

[TMCB08] Fatih Turkmen, Pietro Mazzoleni, Bruno Crispo, and Elisa Bertino. P-cdn: Extending access control capabilities of p2p systems to provide cdn services. In *ISCC*, pages 480–486, 2008.

[TR11] Manachai Toahchoodee and Indrakshi Ray. On the formalization and analysis of a spatio-temporal role-based access control model. *Journal of Computer Security*, 19(3):399–452, 2011.

[VPP06] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. Test input generation for java containers using state matching. In *ISSTA*, 2006.

[Wik] Wikileaks. http://en.wikipedia.org/wiki/WikiLeaks.

[WQL09] Guneshi T. Wickramaarachchi, Wahbeh H. Qardaji, and Ninghui Li. An efficient framework for user authorization queries in rbac systems. In *SACMAT*, pages 23–32, 2009.

[XACa] XACML Enterprise. http://code.google.com/p/enterprise-java-xacml/.

[XACb] XACMLight. http://sourceforge.net/projects/xacmllight/.

[Xer] Apache xerces2 java parser. http://xerces.apache.org/xerces2-j/.

[XO05a] XACML and OASIS Security Services Technical Committee. eXtensible Access Control Markup Language (xacml) committee specification 2.0, Feb 2005.

[XO05b] XACML and OASIS Security Services Technical Committee. Xacml role based access control profile, Feb 2005.

[XWZC09] Min Xu, Duminda Wijesekera, Xinwen Zhang, and Deshan Cooray. Towards session-aware RBAC administration and enforcement with XACML. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2009.

[ZJ08] Yue Zhang and James B. D. Joshi. Uaq: a framework for user authorization query processing in rbac extended with hybrid hierarchy and constraints. In *SACMAT*, pages 83–92, 2008.