Efficient Run-time Solving of RBAC User Authorization Queries: Pushing the Envelope

Alessandro Armando
DIST Univ. of Genova, Italy &
Fondazione Bruno Kessler,
Trento, Italy

Silvio Ranise
Fondazione Bruno Kessler,
Trento, Italy

Fatih Turkmen
DISI Univ. of Trento, Italy &
Fondazione Bruno Kessler,
Trento, Italy

Bruno Crispo
DISI Univ. of Trento, Italy

ABSTRACT

The *User Authorization Query (UAQ) Problem* for Role-Based Access Control (RBAC) amounts to determining a set of roles to be activated in a given session in order to achieve some permissions while satisfying a collection of authorization constraints governing the activation of roles. Techniques ranging from greedy algorithms to reduction to (variants of) the propositional satisfiability (SAT) problem have been used to tackle the UAQ problem. Unfortunately, available techniques suffer two major limitations that seem to question their practical usability. On the one hand, authorization constraints over multiple sessions or histories are not considered. On the other hand, the experimental evaluations of the various techniques are not satisfactory since they do not seem to scale to larger RBAC policies.

In this paper, we describe a SAT-based technique to solve the UAQ problem which overcomes these limitations. First, we show how authorization constraints over multiple sessions and histories can be supported. Second, we carefully tune the reduction to the SAT problem so that most of the clauses need not to be generated at run-time but only in a pre-processing step. Finally, we present an extensive experimental evaluation of an implementation of our techniques on a significant set of UAQ problem instances that show the practical viability of our approach; e.g., problems with 300 roles are solved in less than a second.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, verification*

General Terms

Security, Algorithms, Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODASPY'12, February 7–9, 2012, San Antonio, Texas, USA.
Copyright 2012 ACM 978-1-4503-1091-8/12/02 ...$5.00.

1. INTRODUCTION

The *User Authorization Query (UAQ) Problem* for Role-Based Access Control (RBAC) 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 governing the activation of roles (e.g., mutual exclusion of roles) [6]. In many cases, one is 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.

Approaches to solving the UAQ problem are given in [6]. 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 is checked 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 solution 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 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, [5] 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 and leverages off-the-shelf propositional satisfiability (SAT) solvers. The same paper provides also a comparative experimental analysis between the proposed procedures. The experiments indicate that the greedy search procedure proposed in [6] 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 problems

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 problem that are available in the literature.

In this paper we improve the SAT-based procedure of [5] 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. As we will see later in the paper, 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 paper 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 the improved procedure described in this paper not only tackles a wider class of constraints, but also outperforms the procedure presented in [5]. 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.

Plan of the paper. Section 2 briefly recalls the basic notions underlying RBAC, defines the types of dynamic constraints that we consider in the context of UAQ problem. Section 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 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 [5]. Section 5 concludes the paper.

2. PROBLEM DEFINITION

RBAC [3] regulates access through roles. Roles in a set R associate permissions in a set P to users in a set U by using a *user-assignment relation* $UA \subseteq U \times R$ and a *permission-assignment* relation $PA \subseteq R \times P$. If $(u, r) \in UA$, then a *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 $r_1, r_2 \in R$. A user u *has permission* p iff there exist roles $r, r' \in R$ such that $r \succeq r'$, $(u, r) \in UA$, and $(r', p) \in PA$. An *RBAC policy* is a tuple $RP = (U, R, P, UA, PA, \succeq)$. All the RBAC policies considered in this paper are assumed to be *finite*, i.e. U, R, and P have finite cardinality (and thus UA, PA, and $\succeq$ have finite cardinality too).

A *session* allows a user to activate a sub-set of the roles that the user is assigned to according to the UA relation. Formally, let S be a set of sessions, $user : S \to U$ 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 $user(s)$ by the UA relation, i.e. if $r \in \rho(s)$ then $(user(s), r) \in UA$. If $r \in \rho(s)$, then a *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. $S_u = \{s \in S : user(s) = u\}$. 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 $s' \in (S \setminus \{s\})$, for $i = 1, \ldots, k$). If $H = [\rho_0, \ldots, \rho_{k-1}]$, then $H@\rho_k$ denotes $[\rho_0, \ldots, \rho_{k-1}, \rho_k]$.

2.1 Constraint Specifications

Mutual Exclusive Role (MER) constraints are used to reduce the possibility or impact of frauds. Thus RBAC policies are often enriched with mutually exclusive role constraints. Usually MER constrains are classified as static MER (SMER) and dynamic MER (DMER) based on their enforcement type. 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 paper we focus on DMER constraints and therefore we will not consider SMER constraints any more.

DMER constraints are defined on a set rs of roles and are variants of the same constraint type which vary depending on the way they are enforced. Formally, given an RBAC policy $(U, R, P, UA, PA, \succeq)$, a set $rs \subseteq R$, and $1 \leq n < |rs|$, 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.

We will 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

- $[\rho_0, \ldots, \rho_k] \models$ SS-DMER(rs, n) iff for all $s \in S$, $|rs \cap \rho_i(s)| < n$ for $i = 0, \ldots, k$;
- $[\rho_0, \ldots, \rho_k] \models$ MS-DMER(rs, n) iff for all $u \in U$, $|rs \cap \bigcup_{s \in S_u} \rho_i(s)| < n$ for $i = 0, \ldots, k$;
- $[\rho_0, \ldots, \rho_k] \models$ SS-HMER(rs, n) iff for all $s \in S$, $|rs \cap \bigcup_{i=1}^{k} \rho_i(s)| < n$;
- $[\rho_0, \ldots, \rho_k] \models$ MS-HMER(rs, n) iff for all $u \in U$, $|rs \cap \bigcup_{i=1}^{k} \bigcup_{s \in S_u} \rho_i(s)| < n$;
- $[\rho_0, \ldots, \rho_k] \models$ CARD(r, t) iff $|\{s \in S : r \in \rho_i(s)\}| < t$ for $i = 0, \ldots, k$.

We say that H *satisfies* a finite set C of constraints (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 $(U, R, P, UA, PA, \succeq, C)$. 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 paper, we will consider valid histories only.

2.2 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 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*

- *if $obj = \mathsf{min}$, 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 $obj = \mathsf{max}$, 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 2.1. Let $RP = (U, R, P, PA, UA, C)$ be an RBAC policy and $H = [\rho_0, \ldots, \rho_{k-1}]$ 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 Table 1.
7. if $\textsc{card}(r, t) \in C$ 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.

3. A SAT-BASED PROCEDURE FOR SOLVING UAQ PROBLEMS

Let $RP = (U, R, P, PA, UA, C)$ be an RBAC policy, $H = [\rho_0, \ldots, \rho_{k-1}]$ a history of RP, and (s, P_{lb}, P_{ub}, obj) a UAQ problem for RP. Since RP is 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 $\bigcup_{i=1,\ldots,k-1} \rho_i(s)$.

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 (unit) clauses need to be computed at runtime. 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 $p \in P$, $(\neg\overline{p} \vee \bigvee\{\overline{r} :$ 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 1;
5. if $\textsc{card}(r, t) \in C$ 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 program UAQ-solve of Figure 1 to solve instances of the UAQ problem. It consists of a preprocessing phase (lines 13-18) followed by a loop that reads one UAQ problem at a time (line 21) and tackles it by calling an external PMAX-SAT solver (line 34), which is capable of finding a propositional assignment that satisfies all those clauses labelled 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 22-33). Indeed, when no clauses are marked as soft, the PMAX-SAT solver behaves as a "standard" SAT

Table 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 $s' \in S_{user(s)}$	$\overline{r} \vee \bigvee_{s' \in S_{user(s)}} \overline{O_r(s')}$

solver and it is possible to handle those UAQ problem instances for which obj = 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 34 of Figure 1) in UAQ-solve(). To see how, define $\rho_k(s) = \{r \in R : \overline{r(s)} \text{ is in } \pi\}$ 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 1.

The correctness of UAQ-solve() derives from Theorem 2.1 and the following observations. First, conditions 1.–7. of Theorem 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 2.1, respectively. Third, item 4 of the encoding takes into account condition 6 of Theorem 2.1 via the definition of $\overline{\chi}_k(r, s)$ in Table 1 (see also procedure add-YO-clauses() in Figure 1). Fourth, the remaining conditions (namely, 2 and 3) of Theorem 2.1 are handled at lines 24-27 in Figure 1.

THEOREM 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* $i = 1, ..., n$.

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.

Although we have assumed that the set S of sessions is fixed over histories, our technique can be extended to handle the dynamic creation of sessions. For lack of space, we do not show here the changes in the encoding to accomodate this extension.

4. EXPERIMENTAL EVALUATION

We present a thorough experimental evaluation of our approach obtained by running an implementation of UAQ-Solve in Figure 1 on several synthetic UAQ problems.

In the literature (see, e.g., [1]), several dimensions have been identified to characterize RBAC policies, such as the number of users, roles, permissions, and sessions. In our framework, 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 4.1. Then, we explain our method to generate valid histories of increasing length in Section 4.2 where we 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 [5] when considering only SS-DMER constraints in Section 4.3.

Table 2: 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

4.1 Dimensions of the UAQ problem

Recall from Definition 2.1 that a UAQ problem instance (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 problem instances for the UAQ problem. Here, we discuss various dimensions that determine the underlying RBAC policy RP, the length of the history H, and the choice of session s with the sets P_{lb} and P_{ub} of permissions.

Table 2 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 the three plots that will be discussed in Section 4.3, where the performances for increasing numbers of roles and sessions, and longer histories, respectively, are measured. We do not show a plot for increasing numbers of users since few (unit) clauses are added to $\mathcal{C}(s)$ because of the set U of users (recall item 1 at the beginning of Section 3). 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

```
1   procedure add-YO-clauses()
2    foreach s ∈ S and r ∈ R
3     if Y[s,r] then
4      HC := HC∪{overline(Y_r(s))};
5     else
6      HC := HC∪{¬overline(Y_r(s))};
7     if O[s,r] then
8      HC := HC∪{overline(O_r(s))};
9     else
10     HC := HC∪{¬overline(O_r(s))};
11
12  program UAQ-Solve()
13   foreach s ∈ S
14    C[s] := 𝒞(s); // initialization
15   foreach r ∈ R and s ∈ S
16    Y[s,r] := false; // Yesterday
17    O[s,r] := false; // Once
18   add-YO-clauses();
19   // beginning of on-line phase
20   while(true)
21    read(s, P_lb, P_ub, obj); // reading UAQ problem
22    HC := C(s); // hard constraints
23    SC := ∅;    // soft constraints
24    foreach p ∈ P_lb
25     HC := HC ∪ {overline(p)};
26    foreach p ∈ (P \ P_ub)
27     HC := HC ∪ {¬overline(p)};
28    if obj=min then
29     foreach p ∈ (P_ub \ P_lb)
30      SC := SC ∪ {¬overline(p)};
31    if obj=max then
32     foreach p ∈ (P_ub \ P_lb)
33      SC := SC ∪ {overline(p)};
34    Res := PMAX-SAT-Solve(HC,SC);
35    if Res=UNSAT then
36     print "No solution.";
37    else
38     print-solution(Res);
39     foreach r ∈ R
40      if overline(r) ∈ Res then
41       Y[s,r] := true;
42       O[s,r] := true;
43      else
44       Y[s,r] := false;
45    add-YO-clauses();
```

Figure 1: SAT-based solver for the UAQ Problem

4, which is the number of distinct MER constraint types considered in this paper. (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 4.3.) 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 $\text{MER}(rs, n)$, where rs is a sub-set of the set R of roles and n is a positive 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 2 were inspired from those discussed in [1]. Unfortunately, no value for the number of sessions is provided although [1] takes into consideration multiple sessions.

The generation of valid histories takes a substantial amount of time because, as we will describe in Section 4.2, it requires to solve a number of instances of the UAQ problem which is 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 [1]. Notice also that [1] 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 the dimensions of the UAQ problem. 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 [5] where it is shown equivalent to the UAQ problem defined in [6]. Cases 2 and 3 were also solved in [5] albeit considering only what we call SS-DMER constraints in this paper (for a comparison, see Section 4.3).

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 to do this as follows; see also Figure 2. The core is the 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 toghether with the relations PA and UA (these last elements are assumed to be chosen as explained in Section 4.1). 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 1, called UAQ-Solve^r,

```
function OneSession(s)
1   P_u ← {p | ∃r ∈ R s.t. (u, r) ∈ UA and (r, p) ∈ 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 = ∅ then return fail
2   else begin
3       randomly pick a session s ∈ S_a
4       μ ← OneSession(s)
5       if μ ≠ fail then return μ
6       else S_a' ← S_a \ {s}; return AllSessions(S'_a)
7   end

function OneStep(H@ρ)
1   μ ← AllSessions(S)
2   if μ = fail then
3       if H = [] then return fail
4       else return OneStep(H)
5   else return H@ρ@(ρ ⊕ μ)

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

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

Figure 2: History generation

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 is easy to implement since most available SAT solvers support mechanisms to enumerate one after the other all satisfying assignements of a set of clauses. This is done with the aim of having 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 2, 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. The function OneStep is invoked on the actual history $H@\rho$ (notice that, initially, $[]@(\rightarrow \emptyset)$) 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 a session as input, establishes to which user u 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 library[1] to implement generate. This is so because we can sacrifice a bit the efficiency of SAT solving in favour of a seamless integration (via the available API) within the Java application implementing the functions in Figure 2

4.3 Results

We have implemented the procedure UAQ-solve in Figure 1 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 [4] for the compact 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 [2] 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

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

UAQ problem instances obtained by randomly generated RBAC policies (see Section 4.1) and randomly generated histories (see Section 4.2). According to the values in Table 2, we consider three 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, and (c) increasing history length (third line of the table) from 10 to 200 with a step of 5.

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 behaviour of our implementation on the three scenarios. In all plots, the x-axis reports increasing values of one of the dimension in Table 2—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 behaviour 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 analyse each of the three scenarios in detail. In the plots, lines labelled 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 4.1.)

(a) Increasing number of roles. Figure 3 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 each

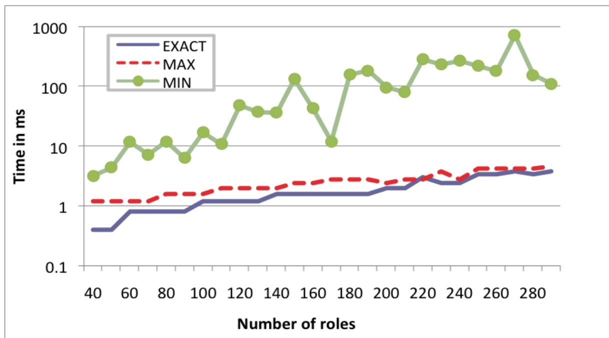

Figure 3: Increasing number of roles

UAQ instance is less than 10 milliseconds 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 [1], 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 shows the plot for scenario (b), i.e. when the number of sessions per user

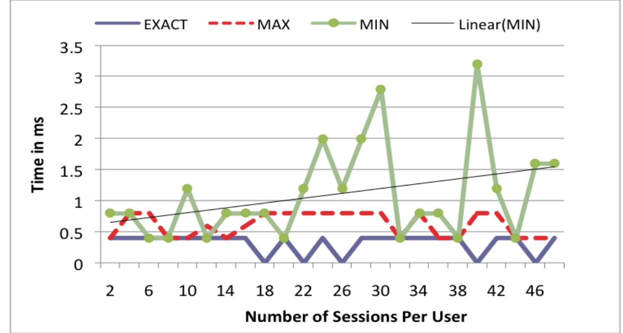

Figure 4: Increasing number of sessions per user

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 problem that must be 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 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

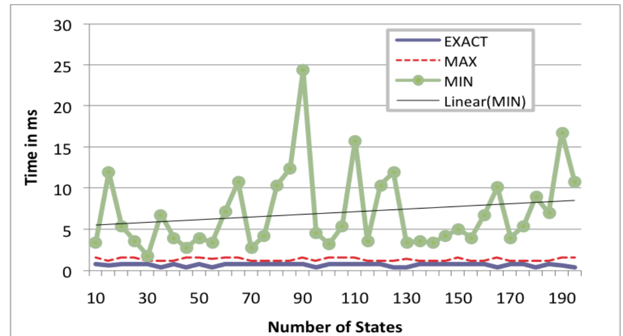

Figure 5: Increasing history length

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 performance of our technique.

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

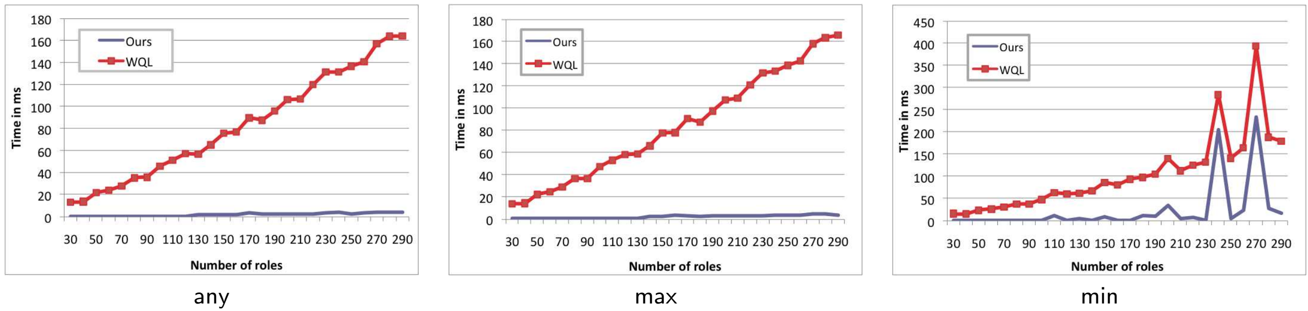

Figure 6: Comparison with the SAT-based procedure of Wickramaarachchi, Qardaji, and Li (2009)

Since the UAQ problem considered in [5] 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 [5] and used the same MaxSAT solver (QMaxSAT) previously mentioned for the sake of a fair 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 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 [5].

We have considered a similar set of experiments as the ones performed in [5] with some larger number of roles. More precisely, we consider $30 \leq |R| \leq 300$, $|U| = 50$, $|P| = 80$, $|C| = 5$ (recall that we only consider SS-DMER constraints), and use the approach described in Section 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 4.3, we take the median value over 10 randomly selected pairs of values of P_{lb} and P_{ub} for the given RBAC policy.

Figure 6 reports the results of the experiments for any (corresponding to the 'exact match' case of [5]), max, and min cases as defined at the end of Section 4.1 for our approach (lines labelled "Ours") and that of [5] (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 [5] 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 [5] 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 the min case. The time needed to generate the encoding remains negligible in our approach while it grows almost linearly for increasining number of roles [5]. 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 1) and only a small number of (unit) clauses need to be computed at runtime (see procedure add-YO-clauses of Figure 1 and the definition of $\overline{\chi}_k(r, s)$ in Table 1).

5. CONCLUSIONS

We have described a carefully tuned SAT encoding to solve instances of the UAQ problem which overcomes the main limitations of previously available techniques. In particular, we have extended the types of authorization constraints that can be taken into consideration to DMER constraints whose scope of applicability can accommodate multiple sessions and histories. An important feature of our reduction to SAT is the generation of a large part of the propositional clauses in a pre-processing phase and leave at run-time only the addition of simple (unit) clauses so that the time required for the encoding is greatly reduced. We have also presented an extensive experimental evaluation that show the practical viability of our technique and its superiority to the one presented in [5].

Acknowledgements. This work was partially supported by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7). We thank Maurizio Festi and Andrea Mongera for the information about Trento Uni. authorization system.

6. REFERENCES

[1] 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.

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

[3] R. Sandhu, E. Coyne, H. Feinstein, and C. Youmann. Role-Based Access Control Models. *IEEE Computer*, 2(29):38–47, 1996.

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

[5] 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.

[6] 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.