An Efficient Framework for User Authorization Queries in RBAC Systems

Guneshi T. Wickramaarachchi
Purdue University
305 N. University Street, West Lafayette, IN 47907, USA
gwickram@purdue.edu

Wahbeh H. Qardaji
Purdue University
305 N. University Street, West Lafayette, IN 47907, USA
wqardaji@cs.purdue.edu

Ninghui Li
Purdue University
305 N. University Street, West Lafayette, IN 47907,USA
ninghui@cs.purdue.edu

ABSTRACT

The *User Authorization Query (UAQ) Problem* for RBAC, introduced by Zhang and Joshi [9], is to determine the set of roles to be activated in a single session for a particular set of permissions requested by the user. This set of roles must satisfy constraints that prevent certain combinations of roles to be activated in one session, and should follow the least privilege principle. We show that the existing approach to the UAQ problem is inadequate, and propose two approaches for solving the UAQ problem. In the first approach, we develop algorithms that use the backtracking-based search techniques developed in the artificial intelligence community. In the second approach, we reduce the problem to the MAXSAT problem which can be solved using available SAT solvers. We have implemented both approaches and experimentally evaluated them.

Categories and Subject Descriptors

D.4.6 [**Security and protection**]: Access controls

General Terms

Security

Keywords

Role Based Access Control, Constraints

1. INTRODUCTION

Role-based access control (RBAC) systems have the notion of sessions. In each session, a user can activate a subset of the roles that the user is authorized for. In [9], Zhang and Joshi introduced the *User Authorization Query (UAQ) Problem*, defined as "*determining the set of roles to be activated in a single session for a particular set of permissions requested by the user*". When determining which set of roles should be activated in a session, there are several competing concerns. First, one has to ensure that the desired permissions are covered by the roles and hence are available to the session. Second, one has to ensure that the set of roles satisfy all constraints governing role activations, such as those forbidding certain combinations of roles to be activated together in one session. Third, while satisfying the first two conditions, one wants to minimize the set of permissions that are available to the session, so as to better achieve the least privilege principle.

While RBAC constraints have been studied extensively in the literature, little work has been done regarding the UAQ problem. The work by Zhang and Joshi [9] is the only work that we are aware of. Zhang and Joshi proposed a two-step algorithm for the UAQ problem. In the first step, the algorithm uses a greedy search to select a set of roles that cover the desired permissions while attempting to minimize extra permissions these roles have. This first step does not consider the constraints. In the second step, the algorithm checks whether the set of roles selected in the first step satisfy all the constraints. If one of the constraints is not satisfied, then the algorithm denies the user's request. We note that even when there exists a set of roles that both satisfies the constraints and covers the desired permissions, it is very likely that the two-step UAQ algorithm in [9] will deny the request, because the greedy search algorithm does not consider the effect of any constraint and may choose a set of roles violating some constraint. Zhang and Joshi also briefly discussed a naïve brute-force algorithm that goes through every subset of the set of roles that the user is authorized for, and checks whether each subset both satisfies the constraints and provides the desired permissions. If a user is authorized for n roles, then such a naïve brute-force approach needs to consider 2^n sets of roles; this can be greatly improved upon.

We recognize that the UAQ problem is similar in nature to constraint satisfaction problems that have been studied extensively in the artificial intelligence literature over the past several decades. Thus we can borrow ideas from this existing literature, using backtrack-based search algorithm that uses the constraints to guide the search. For example, if there is a constraint requiring that the no three roles in $\{r_1, r_2, r_3, r_4\}$ cannot be activated at the same time, then on the search path that r_1 and r_2 have been chosen, one can remove r_3 and r_4 from the roles to be considered.

In this paper we introduce two approaches for the UAQ problem. In the first approach, we extend the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving the CNF-SAT problem. In the second approach, we reduce the UAQ problem to the MAXSAT problem, which asks for the maximum number of clauses which can be satisfied by any assignment in a propositional formula. After the reduction, we use optimized off-the-shelf SAT solvers such as zChaff. We compare both solutions in terms of efficiency.

The rest of this paper organized as follows: In Section 2, we define the UAQ problem. We describe our approaches to the UAQ problems in Section 3, and the implementation details and with experimental results in Section 4. Related work is discussed in Section 5. We conclude in Section 6.

2. PROBLEM DEFINITION

The User Authorization Query problem takes as input a set of permissions that a user requests to have in a session, and tries to find an optimum set of roles to activate so that the set satisfies the user's permissions request and all role constraints within the system. Ideally, the chosen set of roles should activate *only* the requested permissions. However, this is not always possible. Hence we also consider a more general definition of UAQ where the input includes both a lower bound and an upper bound of the requested permissions. Solving the UAQ problem means outputting a set of roles that have permissions between the lower bound and upper bound, while satisfying all constraints. In this general definition, we consider two possible optimization objectives, one is to prefer a set of roles that have permissions as close to the lower bound as possible, and the other is to prefer a set that have permissions as close to the upper bound as possible. The choice between the two optimization objectives depends on the nature and objective of the request.

2.1 Formal Definition of UAQ

More formally, we define the User Authorization Query(UAQ) problem as taking the following three groups of inputs:

1. RBAC State Information: $(\mathcal{R}, \mathcal{P}, perms)$ where

- $\mathcal{R}$ is the set of roles that the user is authorized for, i.e., the set of roles the user can activate. For any RBAC state, it is straightforward to find all roles that a user is authorized for. This set includes both roles that a user is directly assigned to, and the roles the user can activate because of inheritance.
- $\mathcal{P}$ is the set of all permissions in the system.
- $perms : \mathcal{R} \rightarrow 2^{\mathcal{P}}$ gives the set of permissions each role in $\mathcal{R}$ has.

2. The Permission Request information: (P_{lb}, P_{ub}, obj) where

- $P_{lb} \subseteq \mathcal{P}$ is the lower bound for the set of requested permissions. Permissions in P_{lb} *must* be available for the session.
- $P_{ub} \subseteq \mathcal{P}$ is the upper bound for the set of requested permissions. Permissions not in P_{ub} *must not* be available for the session.

  We require that $P_{lb} \subseteq P_{ub}$.
- $obj \in \{\mathsf{any}, \mathsf{max}, \mathsf{min}\}$ indicates the optimization objective, where any means no optimization is needed and any set of roles that have permissions between P_{lb} and P_{ub} is fine, max means the more permissions the better, and min means the fewer permissions the better.

3. Constraints: C represents the set of constraints that limit which roles in $\mathcal{R}$ can be activated together. We consider the following two kinds of constraints, which are the same as those considered in [9].

- DSoD (Dynamic Separation of Duty) constraints: $\langle\{r_1, r_2...r_n\}, t\rangle$ means that no single user can activate within a single session t or more roles from the n roles listed.
- Cardinality constraints: a constraint $\langle r, t\rangle$ specifies that there can be at most t activations for the particular role r. Such a constraint is easy to handle. One counts how many activations of r there currently are, if r is already activated in t sessions, then the role r should be removed from $\mathcal{R}$, i.e., should not considered in the search process. If r is activated in no more than $t-1$ sessions, then r can be considered.

The UAQ problem should output a set $R \subseteq \mathcal{R}$ such that the following three conditions hold:

1. $P_{lb} \subseteq perms(R) \subseteq P_{ub}$
2. R satisfies all constraints in C.
3. When $obj = \mathsf{max}$, then for any R' that also satisfies the above two conditions, we have $|R| \geq |R'|$. When $obj = \mathsf{min}$, then for any R' that also satisfies the above two conditions, we have $|R| \leq |R'|$.

We note that the special case of UAQ where $P_{lb} = P_{ub}$ is NP-complete. This was observed and proved in [2, 9] by reducing the *Minimal Set Cover* (MSC) problem with uniform cost to it. The UAQ problem in general is thus NP-hard.

2.2 Comparison with Previous Definitions

Our definition of the UAQ problem is inspired by that in [9], and generalizes it. In [9], Zhang and Joshi use only one set of permissions P_{req} as input, as compared to our usage of P_{lb} and P_{ub}. They consider two cases. One is when least privilege is the primary concern, and one wants to ensure that no permission outside P_{req} is available to the session. In this case one wants to find R such that $perms(R)$ is a subset of P_{req} but is as large as possible. We note that this is a special of our definition where

$$P_{lb} = \phi;\ P_{ub} = P_{req};\ obj = \mathsf{max}.$$

The other case is when the availability is the major concern, and one wants to ensure that all permissions in P_{req} are available to the session. In this case, one wants to find R such that $perms(R)$ is a superset of P_{req} and is as small as possible. We note that this is a special case of our definition where

$$P_{lb} = P_{req};\ P_{ub} = \mathcal{P};\ obj = \mathsf{min}.$$

Our definition of using both an upper bound and a lower bound provides more flexibility, and we believe it is more practically useful, since often times the system security policy would want to ensure that at least some minimal permissions are available for the session, and some sensitive permissions are off the limit for the session.

2.3 Role Hierarchies

When generating an UAQ instance in an RBAC system with role hierarchies, one must exercise care with respect to the semantics of the role hierarchy. A role hierarchy will affect the determination of $\mathcal{R}$, the set of roles a user is authorized for, as well as $perms$, which determines the permissions each role has.

In [4], Joshi et al. introduced three types of role hierarchies in RBAC systems. The *inheritance-only hierarchy* (*I-hierarchy* denoted by $\geq_i$), the *activation-only hierarchy* (*A-hierarchy* denoted by $\geq_a$) and the *inheritance-activation hierarchy* (*IA-hierarchy* denoted by $\geq$). The I-hierarchy allows a senior role to acquire all the permissions of its junior roles, but does not allow any user assigned to a senior role to activate any of the junior roles. The A-hierarchy allows a user assigned to a senior role to activate its junior roles, but does not propagate permissions to the senior roles. Finally, the IA-hierarchy is the combination of both. It is the most commonly used and traditional type of role hierarchies in RBAC systems. In [9], Zhang and Joshi describe how their UAQ framework applies to hybrid hierarchies where the three hierarchy types co-exist.

These hierarchies can be straightforwardly handled when generating a UAQ instance. When computing the set of roles a user is authorized for (i.e., $\mathcal{R}$), one should consider the union of the A-hierarchy and the IA-hierarchy. When computing the set of permissions available to a role r in $\mathcal{R}$ (i.e., $perms(r)$), one should consider the union of the I-hierarchy and IA-hierarchy. In the rest of this paper, we work with UAQ instances and assume that issues of hybrid hierarchies are handled in the generation of the UAQ instances.

In [5], Li et al. considered yet another kind of role inheritance semantics beyond I-hierarchy and A-hierarchy. That r_1 is senior to r_2 may mean that whenever r_1 is activated in one session, r_2 is also automatically activated. To see why this is desirable, suppose that r_1 is senior to r_2, and r_3 is senior to r_4, and there is a constraint saying that no user can activate r_2 and r_4 in one session. Without automated activation of junior roles, a user can activate r_1 and r_3, obtaining all permissions of r_2 and r_4, without violating the constraint forbidding the simultaneous activation of r_2 and r_4. One solution to avoid this problem is to ensure that the set C of constraints has the property that whenever there is a constraint forbidding the activation of R, there is also a constraint forbidding the activation of R' if R' can be obtained by replacing some roles in R with a common senior role. If this solution is adopted, then one does not need to use the automated activation of junior roles interpretation. We point out that the algorithms developed in this paper can, however, be easily adapted to deal with automated activation of junior roles. For example, whenever a senior role is selected in the search process, the algorithm can automatically add the junior roles, and evaluate the resulting solution.

2.4 Three Cases of UAQ

We divide the UAQ problem into three subcases and provide algorithms for each of the three sub-cases.

Case 1 - Exact Match: $P_{lb} = P_{ub}$. This means that one wants a set of roles with an exact set of permissions.

Case 2 - Minimal Match: $obj = \mathsf{min}$. This means that one wants the minimal number of permission such that at least the lower bound permissions are activated.

Case 3 - Maximal Match: $obj = \mathsf{max}$. This means that one wants as many permissions as possible within the upper bound, while at least covering the lower bound.

For the case where $P_{lb} \subset P_{ub}$ and $obj = \mathsf{any}$, this means that any solution within the bounds is acceptable. One can use the algorithm for either the minimal match case or the maximal match case and stops after the algorithm finds the first acceptable solution. When there are optimization objectives, these algorithms will continue to search for better solutions.

3. ALGORITHMS FOR UAQ

In this section we propose two approaches for solving the UAQ problem. The first approach (section 3.1) is based on general recursive search techniques, and the second approach (section 3.2) transforms a UAQ instance to a SAT instance and solving it using a SAT solver. In both approaches, one first performs some processing to prune the search space. For example, all cardinality constraints are checked and any role that has already reached its cardinality limit is removed from $\mathcal{R}$. Furthermore, if any role has permissions outside P_{ub}, the role is removed. Finally, if we are finding exact matches or minimal matches, we can also remove any role that has no permission in P_{lb}. For the maximal match case, such role can nonetheless be useful.

The first approach uses ideas from the DPLL algorithm for SAT [7]. The algorithms do a backtracking based search where the search tree is traversed recursively. In each recursion, it performs the following steps:

1. Check feasibility and add mandatory roles. For each permission not yet covered, check how many candidate roles can cover the permission. If no candidate role covers it, backtrack. If only one candidate role covers it, add the role to the selected set.

2. If new roles are selected from the last step, then perform the following steps.

   (a) Check whether the current set of selected roles is a solution, i.e., covers all necessary privileges. If so, check whether it is a better solution than the stored current best solution. If so, record these roles as the new best solution.

   (b) Determine whether continuing the search from the currently selected roles can lead to a better solution than the best solution found so far. If not, then backtrack. For example, if the goal is to have minimal number of permissions, and the currently selected roles already have more permissions than the current best solution, then it is not necessary to continue.

   (c) Prune the search space. Remove from the candidate set any role that does not provide new permission not covered by the currently selected roles. Also, prune the search space using constraints. For example, if a DSoD constraint requires that no t roles in $\{r_1, r_2, \ldots, r_n\}$ can be activated in one session, and one has already chosen $t-1$ roles from $\{r_1, r_2, \ldots, r_n\}$, then one can remove all other roles in $\{r_1, r_2, \ldots, r_n\}$ from the candidate set. If any role is pruned due to constraints, go back to step 1.

3. Heuristically select the next role from the candidate roles.

4. Recursively call itself with the next role selected.

5. Recursively call itself with the next role not selected and removed from the candidate set. (This considers the case of not using this role.)

Such an algorithm dynamically prunes the search space and hence improves the efficiency of the search process. We note that if one just wants any solution rather than an optimal one, the algorithm can stop after finding the first solution, rather than continuing the search process. We also note that while the search for the optimal solution may take a long time, one can easily set a time limit on the search, stops when the limit is reached, and uses the best solution found so far.

The second approach we propose is to transform a UAQ instance to a MAXSAT instance and solves it using a SAT solver. This would involve formalizing each role and permission as a SAT variable and constructing propositional clauses to represent role-permission relationships as well as constraints. This approach was inspired by the approach used by Li et al. in [6] for the verification of static mutually exclusive roles. Here we extend the techniques to deal with dynamic role assignments.

3.1 Proposed Algorithms

Algorithm 1 implements the exact match sub case, where system grants permissions where $P_{ub} = P_{req} = P_{lb}$. The requested permissions will be exactly matched with a set of available roles in this case. In order to improve the performance of the search, the roles with extra permissions than the requested permissions are eliminated in the beginning of search. The algorithm finds the best solution among all possible solutions. The set of selected roles which satisfy the requested permissions is denoted as R_{sel}. We define P_{rem} as $P_{req} \setminus P(R_{sel})$ which defines the permissions that have not been covered by the selected roles. Algorithm 2 shows the minimal match sub case where $P_{lb} \leq P_{req} \leq P_{ub}$ and system satisfies at least P_{lb} of the request. We define P_{extra} as the extra permissions than P_{lb} and this should be minimized. The algorithm returns from the current recursion path if current R_{sel} has more extra permissions than the best solution R_B's extra permissions. The next role will be selected based on two factors.

Algorithm 1 ExactMatch($P_{req}, R_{avail}, C_{all}$)

Input: R_{avail} - set of roles; P_{req} - requested permissions; C_{all} - all constraints
Output: R_B - set of roles which can satisfy the exact match requirement

```
1:  $R_B \leftarrow \phi$
2:  $R_{sel} \leftarrow \phi$
3:  $P_{rem} \leftarrow P_{req}$
4:  for all $r \in R_{avail}$ do
5:      if $(P(r) \nsubseteq P_{rem})$ then
6:          remove $r$ from $R_{avail}$
7:      end if
8:      if $(P(r) \cap P_{lb} = \phi)$ then
9:          remove $r$ from $R_{avail}$
10:     end if
11: end for
12: selectRoles$(P_{rem}, R_{sel}, R_{avail})$;
13: ____________________________________________
14: selectRoles$(P_{rem}, R_{sel}, R_{avail})\{$
15: if $P_{rem} = \Phi$ then
16:     if $R_B = \Phi$ then
17:         $R_B = R_{sel}$
18:     else if $\mid R_B \mid > \mid R_{sel} \mid$ then
19:         $R_B = R_{sel}$
20:     end if
21:     return
22: end if
23: for all $C_i \in C_{all}$ do
24:     if $(\mid R_{sel} \cap R(C_i) \mid \ = t - 1)$ then
25:         $R_{avail} = R_{avail} \setminus (R(C_i) \setminus (R_{sel} \cap R(C_i)))$
26:     end if
27: end for
28: if $R_{avail} = \Phi$ then
29:     return
30: end if
31: for next $r$ in $R_{avail}$ do
32:     if $P(r) \cap P_{rem} = \Phi$ then
33:         $R_{avail} = R_{avail} \setminus r$
34:     end if
35: end for
36: selectRoles $(P_{rem} \setminus P(r), R_{sel} \cup r, R_{avail} \setminus r)$
37: selectRoles $(P_{rem}, R_{sel}, R_{avail} \setminus r)$
38: }
```

- minimizing the new extra permissions provide by the selected role.
- maximizing the permissions covered by the selected role within the remaining requested permissions.

Algorithm 3 implements the maximal match sub case explained in previous section, where system grants at most the upper bound of permissions. P_{needed} is the permissions required to reach P_{ub} (i.e. those that are within P_{ub} and not within the permissions associated with the currently selected set of roles) and this should be minimized. We eliminate all the roles from the search space which has extra permissions than the upper bound permissions in order to improve the efficiency of the search process. The next role will be selected based on two factors.

- minimizing needed permissions to reach P_{ub} (i.e. P_{needed}) by the selected role.
- maximizing the permissions covered by the selected role within the remaining requested permissions.

The aforementioned algorithms mimic a tree search approach which considers all possible combinations of roles to activate. Each branch of the tree is considered and pruned based on whether or not any constraints are violated. With each call to select roles, the following is guaranteed:

- The roles in the current solution (R_B) are always updated to the more optimum solution.
- The next role to be selected always satisfies the constraints because all violating roles are removed from R_{avail} before a role is selected.
- An alternative solution resulting from removing the currently selected role is always considered. The current solution would again be updated to the more optimum solution even if it involves choosing an alternative set of roles.

This guarantees that the algorithms always generate the optimum solution that satisfies all constraints.

The user can request set of roles to be activated during a session as well. Algorithm 4 shows how to authorize those requests.

3.2 Translation to SAT

In this section we show how the UAQ problem can be translated into a SAT problem (or one of its variants) which can be solved using off-the-shelf SAT solvers (see appendix A for more information about SAT). Although the SAT problem is NP-complete, a lot of research has been done in optimizing SAT solvers. Hence, SAT solvers, such as zChaff [1] are available that can produce accurate results efficiently.

The first case where an exact set of permissions is required ($P_{lb} = P_{RQ} = P_{ub}$) can be reduced to a SAT instance as follows:
Given:

- P: the set of all possible permissions in the system
- R: the set of all roles in the system that are available to the user
- PA: the set of permission assignments $\{(r_k, p_i) | r_k \in R, p_i \in P$ and p_i is assigned to role $r_k\}$
- RH: the role hierarchy in the form $\{r_i \geq r_j | r_i, r_j \in R\}$

Algorithm 2 MinimalMatch($P_{req}, R_{avail}, C_{all}, P_{lb}, P_{ub}$)

Input: R_{avail} - set of roles; P_{req} - requested permissions; C_{all} - all constraints; P_{lb} - lower bound permissions; P_{ub} - upper bound permissions

Output: R_B - set of roles which can satisfy the minimal match requirement; R_{sol} - set of solutions

```
1:  $P_{ub}, P_{lb}$
2:  $R_B \leftarrow \phi$
3:  $R_{sol} \leftarrow \phi$
4:  $R_{sel} \leftarrow \phi$
5:  $P_{rem} \leftarrow P_{req}$
6:  $P_{extra} \leftarrow \phi$
7:  for all $r \in R_{avail}$ do
8:      if $(P(r) \nsubseteq P_{ub})$ then
9:          remove $r$ from $R_{avail}$
10:     end if
11:     if $(P(r) \cap P_{lb} = \phi)$ then
12:         remove $r$ from $R_{avail}$
13:     end if
14: end for
15: selectRoles $(P_{rem}, R_{sel}, P_{extra}, R_{avail}, P_{lb}, P_{ub})$;
16: ______________________________________________
17: selectRoles $(P_{rem}, R_{sel}, P_{extra}, R_{avail}, P_{lb}, P_{ub})\{$
18: if $P_{rem} = \Phi$ then
19:     if $R_B = \Phi$ then
20:         $R_B = R_{sel}$
21:     else if $| P(R_B) | > | P(R_{sel}) |$ and $P(R_{sel}) \subseteq P_{ub}$ and
        $| R_B | \geq | R_{sel} |$ then
22:         $R_B = R_{sel}$
23:         $R_{sol} = R_{sol} \cup R_{sel}$
24:     end if
25:     return
26: end if
27: if $| P_{extra} | \geq | P(R_B) \setminus P_{lb} |$ and $| P(R_B) | > 0$ then
28:     return
29: end if
30: for all $C_i \in C_{all}$ do
31:     if $(| R_{sel} \cap R(C_i) | = t - 1)$ then
32:         $R_{avail} = R_{avail} \setminus (R(C_i) \setminus (R_{sel} \cap R(C_i)))$
33:     end if
34: end for
35: if $R_{avail} = \Phi$ then
36:     return
37: end if
38: Select next $r$ s.t. 'value' is maximum where,
39: value= $\frac{|P(r) \cap P_{rem}|}{|((P(r) \setminus P_{lb}) \setminus P_{extra})|}$
40:
41: selectRoles $(P_{rem} \setminus P(r), R_{sel} \cup r, P_{extra} \cup (P(r) \setminus P_{lb}), R_{avail} \setminus r, P_{lb}, P_{ub})$
42: selectRoles $(P_{rem}, R_{sel}, P_{extra}, R_{avail} \setminus r, P_{lb}, P_{ub})$
43: $\}$
```

Algorithm 3 MaximalMatch($P_{req}, R_{avail}, C_{all}, P_{lb}, P_{ub}$)

Input: R_{avail} - set of roles; P_{req} - requested permissions; C_{all} - all constraints; P_{lb} - lower bound permissions; P_{ub} - upper bound permissions

Output: R_B - set of roles which can satisfy the maximal match requirement;

```
1:  $P_{ub}, P_{lb}$
2:  $R_B \leftarrow \phi$
3:  $R_{sel} \leftarrow \phi$
4:  $P_{rem} \leftarrow P_{req}$
5:  $P_{needed} \leftarrow P_{ub}$
6:  for all $r \in R_{avail}$ do
7:      if $(P(r) \nsubseteq P_{ub})$ then
8:          remove $r$ from $R_{avail}$
9:      end if
10: end for
11: selectRoles $(P_{rem}, R_{sel}, P_{needed}, R_{avail}, P_{lb}, P_{ub})$;
12: ______________________________________________
13: selectRoles $(P_{rem}, R_{sel}, P_{needed}, R_{avail}, P_{lb}, P_{ub})\{$
14: if $R_B = \Phi$ then
15:     $R_B = R_{sel}$
16: else if $| P(R_B) | < | P(R_{sel}) |$ and $P_{lb} \subseteq P(R_{sel})$ then
17:     $R_B = R_{sel}$
18: end if
19: if $| P_{needed} | \geq | P_{ub} \setminus P(R_B) |$ and $| P(R_B) | > 0$ then
20:     return
21: end if
22: for all $C_i \in C_{all}$ do
23:     if $(| R_{sel} \cap R(C_i) | = t - 1)$ then
24:         $R_{avail} = R_{avail} \setminus (R(C_i) \setminus (R_{sel} \cap R(C_i)))$
25:     end if
26: end for
27: if $R_{avail} = \Phi$ then
28:     return
29: end if
30: Select next $r$ s.t. 'value' is maximum where,
31: value = $\frac{|P(r) \cap P_{rem}|}{|(P_{ub} \setminus P(r)) \cap P_{needed}|}$
32: selectRoles $(P_{rem} \setminus P(r), R_{sel} \cup r, P_{needed} \setminus ((P(r) \cap P_{needed})), R_{avail} \setminus r, P_{lb}, P_{ub})$
33: selectRoles $(P_{rem}, R_{sel}, P_{needed}, R_{avail} \setminus r, P_{lb}, P_{ub})$
34: $\}$
```

Algorithm 4 ($R_{RQ}, R_{avail}, C_{all}$)

Input: R_{avail} - set of roles; R_{RQ} - requested roles; C_{all} - all constraints;

Output: true/false;

```
1: if $| R_{RQ} \setminus (R_{RQ} \cap R_{avail}) | > 0$ then
2:     return false
3: end if
4: for all $C_i \in C_{all}$ do
5:     if $(| R_{RQ} \cap R(C_i) | = t)$ then
6:         return false
7:     end if
8: end for
9: return true
```

- C: the set of all constraints of the form $\langle\{r_i,\dots,r_j|r_i,\dots,r_j \in R\},t\rangle$. For the single user assignment problem, cardinality constraints can be converted to this form as described earlier in the paper.

Formalization:

- For each $p_i \in P$, let p_i^v denote a SAT variable which is true if and only if p_i is activated.
- For each $r_i \in R$, let r_i^v denote a SAT variable which is true if and only if r_i is activated.
- For each $r_i \in R$, construct the set $H_{r_i} = \{r|r_i \geq r\}$. This will denote all the roles r_i is superior to.

1. $S = \emptyset$ // the initial set of clauses
2. For each $p_i \in P_{RQ}$, mark p_i^v as a non-relaxable [1] clause and add it to S
3. For each $p_i \in P \setminus P_{RQ}$, mark $\neg p_i^v$ as a non-relaxable clause and add it to S
4. For each role $r_k \in R$, construct an equivalence propositional clause of the form
$$r_k^v \leftrightarrow (p_i \wedge p_j \wedge \dots \wedge p_l \ \wedge \ r_i \wedge r_j \wedge \dots \wedge r_l)$$
where $(r_k,p_i),(r_k,p_j),\dots(r_k,p_l) \in PA$ and $r_i,r_j,\dots r_l \in H_{r_k}$, mark it as non-relaxable and add it to S.
5. For each permission $p_k \in P$, construct a clause of the form
$$p_k^v \rightarrow (r_i \vee r_j \vee \dots \vee r_l)$$
where $(p_k,r_i),(p_k,r_j),\dots(p_k,r_l) \in PA$, mark it as non-relaxable and add it to S.
6. For each constraint $\langle\{r_i,\dots,r_j\},t\rangle \in C$. Construct a clause of the form $\neg r_k^v \vee \dots \vee \neg r_l^v$ for all roles in each t-sized subset of $\{r_i,\dots,r_j\}$. Mark the clause as non-relaxable and add it to S
7. Solve S using a SAT solver
8. If the solver returns SATISFIABLE, then activate each role $r_i \in R$ such that r_i^v is set to true

Steps 2 and 3 of the formulation ensure that only permissions in the request set are satisfied and all others are not. Alternatively, as an optimization, all requests which contain permissions outside the permission set can be initially removed before converting the problem to a SAT instance. Step 4 makes sure that a role r is activated if and only if all its children are activated. Step 5 confirms that a permission is not activated unless it belongs to an activated role. Note that this formulation, unlike the algorithmic approach, does not allow a role to remain inactive if all its permissions are activated. If, however, system designers view this as a requirement, then role clauses generated in step 4 can be marked as relaxable and the PMAX-SAT approach described later in this section can be applied.

In the case where constraints are unsatisfiable, there are two possibilities for role assignments:

[1] a non-relaxable clause is a propositional clause that is hard, i.e. it must be satisfied if the instance as a whole is satisfiable

- Favor the maximal solution by activating as many roles as possible such that all permissions in P_{lb} as well as the maximum number of permissions in Pub are satisfied.
- Favor the minimal solution by activating the smallest number of roles such that all permissions in P_{lb} are satisfied.

These two cases can be easily translated to a partial maximal satisfiability problem. Partial maximal satisfiability (PMAX-SAT) is a special variation of the MAX-SAT problem whereby certain clauses are relaxable and others are non-relaxable. PMAX-SAT aims to satisfy all non-relaxable clauses as well as a maximum number of relaxable ones [3]. To translate the role assignment problem to a PMAX-SAT problem, a SAT instance is created which is satisfied if and only if the maximum (or minimum) number of roles is activated while satisfying all constraints. All permissions in P_{lb} are translated into non-relaxable propositional clauses; all relaxable permissions ($P_{ub} \setminus P_{lb}$) are translated into *relaxable* propositional clauses.

To satisfy the maximum number of permissions in P_{ub}, all permission variables in $P_{ub} \setminus P_{lb}$ are initially set to true. Since PMAX-SAT aims at satisfying the largest number of relaxable clauses and given that the only relaxable clauses here are permissions, then using PMAX-SAT on this formulation will give the SAT instance that would satisfy the maximum number of wanted permissions, vis-a-vis the maximum number of wanted roles, while also satisfying all constraints.

To satisfy the minimum number of permissions in P_{ub}, all permission variables in $P_{ub} \setminus P_{lb}$ are initially set to false (i.e. $\neg p_i^v$). Since PMAX-SAT aims at satisfying the largest number of relaxable clauses then a minimum number of these permissions will be set to true in the SAT instance.

The translation can be formalized as follows:

- For all permissions $p_i \in P_{lb}$, mark p_i^v as a non-relaxable clause and add it to S
- To favor the minimal solution: apply the previous SAT formulation, but for each $p_i \in P_{ub} \setminus P_{lb}$, mark $\neg p_i^v$ as a relaxable clause and add it to S.
- To favor the maximal solution: apply the previous SAT formulation, but for each $p_i \in P_{ub} \setminus P_{lb}$, mark p_i^v as a relaxable clause and add it to S.

Fu and Malik proposed an approach for solving the PMAX-SAT problem [3]. The algorithm they presented was successfully implemented in solving the PMAX-SAT role activation problem using the above formulation.

The following example illustrates the above formulations for the exact match case:
$P = \{p_1,p_2,p_3,p_4\}$
$R = \{r_1,r_2,r_3\}$
$PA = \{(r_1,p_1),(r_1,p2),(r_2,p_2),(r_2,p_3),(r_3,p_4)\}$
$RH = \{r_3 \geq r_2\}$
$P_{RQ} = P_{ub} = P_{lb} = \{p_1,p_2,p_3\}$
$C = \{\langle\{r_1,r_3\},2\rangle\}$

The input to the SAT solver would be:
p_1^v
p_2^v
p_3^v
$\neg p_4^v$
$r_1^v \leftrightarrow (p_1^v \wedge p_2^v)$
$r_2^v \leftrightarrow (p_2^v \wedge p_3^v)$

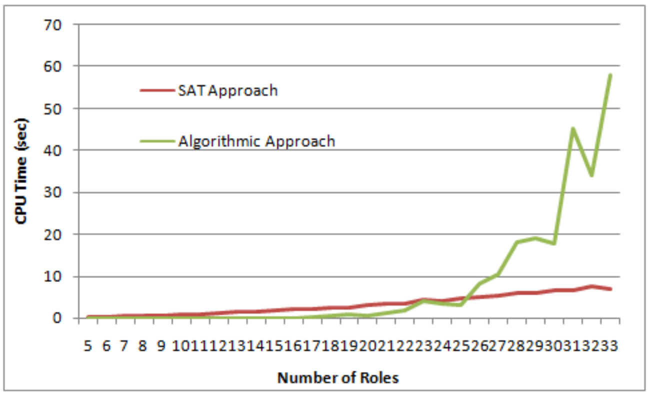

Figure 1: CPU Time for Minimal Match Case

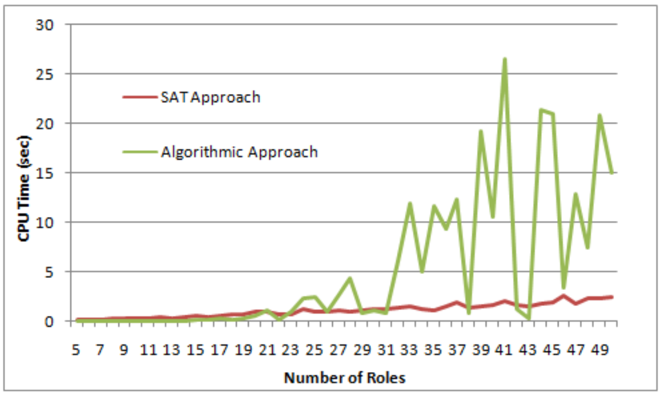

Figure 2: CPU Time for Maximal Match Case

$r_3^v \leftrightarrow (p_4^v \wedge r_2^v)$
$p_1^v \rightarrow (r_1^v)$
$p_2^v \rightarrow (r_1^v \vee r_2^v)$
$p_3^v \rightarrow (r_2^v)$
$p_4^v \rightarrow (r_3^v)$
$\neg r_1^v \vee \neg r_3^v$

4. IMPLEMENTATION AND RESULTS

In this section we present our experimental results for the UAQ framework presented in the previous section. We implemented the SAT approach using the zChaff SAT solver [1] following the maximal satisfaction algorithms introduced in [3] to implement the minimal and maximal satisfaction cases. We also compare our results with those from the greedy framework introduced in [9].

In order to run the experiments, we generated random test cases with a varying number of roles. The ratio of roles to permissions (1:12), roles to constraints (5:1) and roles to number of permissions requested (1:1) were kept constant in all the experiments in order to facilitate averaging and comparison. For each role, 10 randomly generated test cases were run and the results were used to generate the graphs.

The following experimental results show the CPU time taken by each program. For the SAT approach, this includes the time taken to construct the proper input files to the SAT solver. If the SAT solver is implemented in a real system, this conversion process would be more transparent and thus the performance would be slightly better. Furthermore, the efficiency of the SAT approach is dependent upon the SAT solver used. While for the purposes of this experimentation we found zChaff to be most suited for efficiently generating unsat cores for use in maximal satisfaction, other commercially available solvers may prove to be more efficient.

4.1 Minimal Match Case

Figure 1 shows the result of running the experiments for the minimal match case averaged over 10 times per role. For smaller numbers of roles, permissions and constraints, the two approaches produce comparable results, with the algorithmic approach performing slightly better. However, as the number of roles increases, the time taken for the algorithmic approach increases exponentially making it impractical for implementation in dynamic systems. The CPU time taken by the SAT approach did not exceed a few seconds, even for a larger number of roles and permissions. This makes the SAT approach acceptable in all cases for the minimal match approach.

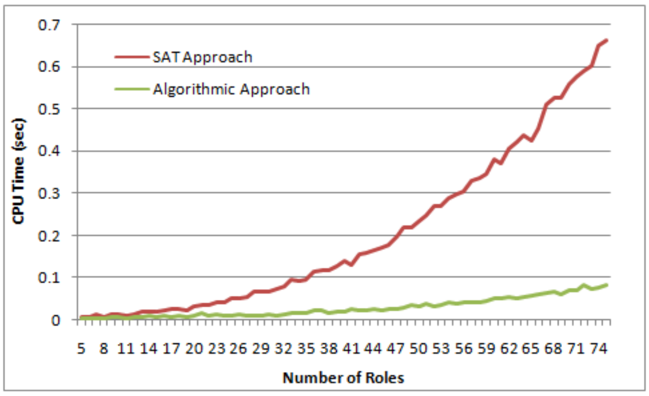

Figure 3: CPU Time for Exact Match Case

4.2 Maximal Satisfaction Case

Figure 2 shows the result of running the experiments for the maximal match case. In this case, the algorithmic approach performs quite well for smaller numbers of roles, permissions and constraints. As the number of roles increases, the overall trend in time taken increases exponentially. The time taken for the SAT approach, on the other hand, provides better overall results for larger number of roles. The figure shows certain "dips" in CPU time that deviate from the trend even though the results in the graph are the average of running the experiment multiple on the with random test cases generated with the same number of roles. This is suggestive of the effectiveness in heuristically choosing a particular role with certain test cases as opposed to others. If a role is *appropriately* chosen, the overall search tree will be *well-pruned* and the overall running time would be better. Finding better ways for heuristically choosing the next role in the search in order to optimize the algorithms may prove to be an interesting area or research albeit beyond the limited scope of this paper.

4.3 Exact Match Case

Figure 3 shows the result of running the experiments for the exact match case. In this case, both algorithms performed very well with results for large roles taking less than one second of CPU time. The algorithmic approach consistently out-performs the SAT approach in all cases. It is worth noting here that a large number of experiments for this case produced no solution due to the random

P(r0) =$\{p14, p10, p17, p13, p7, p3\}$
P(r1) =$\{p10, p1, p2\}$
P(r2) =$\{p18\}$
P(r3) =$\{p10, p8, p2, p12, p15\}$
P(r4) =$\{p13, p10, p8, p11, p6, p16, p18, p9\}$
P(r5) =$\{p14, p1, p10, p6, p8, p13, p5, p16, p15\}$
P(r6) =$\{p16\}$
P(r7) =$\{p8, p10, p15\}$
P(r8) =$\{p10, p11, p14, p13, p4\}$
P(r9) =$\{p12, p17, p1, p5, p14\}$
P(r10)=$\{p18, p19, p2, p16, p1, p3, p7\}$
P(r11)=$\{p9, p10, p3\}$
P(r12)=$\{p5, p12, p0\}$
P(r13)=$\{p18, p15, p12\}$
P(r14)=$\{p19, p10, p13, p4, p12, p14\}$
P(r15)=$\{p15\}$
P(r16)=$\{p2, p12\}$
P(r17)=$\{p1, p5, p4, p13, p7, p10\}$
P(r18)=$\{p11, p13, p10\}$
P(r19)=$\{p13, p11\}$
Constraints:
$(\{r2, r3, r6, r7, r12, r13, r15, r18\}, 2)$

Figure 4: Role-permission assignments and DSOD constraints

generation of constraints. In general, when there is no solution, the algorithmic approach out-performs the SAT approach. This explains the difference in performance shown in the graph in figure 3.

4.4 Comparison with Greedy Approach

To compare our results with the UAQ approach proposed in [9] we implemented the greedy algorithms proposed and ran them on the same test cases we used for our algorithms according to the formulation introduced in 2. The UAQ approach in [9] may produce a sub-optimal solution or a solution which fails the activation checking module. We show such an example where the solutions provide by [9] do not satisfy the constraints and ultimately the user request get rejected. Figure 4 represents the role-permission relationships and DSoD constraints considered for this example. The results are summarized in table 1. The approach defined in this paper, on the other hand, always produces a better solution while satisfying the constraints. The following summarizes the results out of approximately 1000 test cases with varying numbers of roles and permissions:

- For the approach favoring availability (maximal match in our framework), 94% of the test cases produced an incorrect solution violating the constraints. The remaining 6% produced *sub*-optimal or optimal solutions.

- For the approach favoring least privilege (minimal match in our framework), 39% of the test cases produced an incorrect solution violating the constraints. The remaining 61% produced *sub*-optimal or optimal solutions.

The approach proposed in [9] uses greedy approach and does not need to search through the solution space to find a solution that satisfy all the constraints. Hence, direct time comparison between the approach in [9] and our approach is not meaningful.

The solutions provided by the role mapping module of the UAQ framework of [9] violate the DSoD constraints whereas our proposed algorithms suggest much better solutions while satisfying the constraints.

Table 1: Comparison of proposed approaches with UAQ

Scenario	Input(P_{RQ})	Algo.	UAQ [9]
Exact Match	p11, p13, p10, p8, p2, p12, p15	(r3, r19)	(r3, r18):Reject
Minimal match	p18, p15, p12, p1, p6, p4, p13, p7, p10, p8, p11, p0, p14, p9	(r4,r5, r17,r12)	(r4,r9,r15, r17,r12):Reject
Maximal match	p18, p15, p12, p1, p6, p4, p13, p7, p10, p8, p11, p0, p14, p9	(r8, r13)	(r8,r13,r7) Reject

5. RELATED WORK

There are a number of user authorization frameworks in the literature which follow different approaches for authorizing user requests in RBAC systems. A solution which tackles the authorization query for Inter Domain Mapping Problem in the presence of hybrid hierarchies was introduced in a paper by Du et al. in [2]. The paper proves that that the problem is NP-complete and provides a greedy algorithm for solving it. This work, however, does not deal with constraints. The UAQ framework in [9] expands on the previous approach and introduces two further cases of the problem which deal with availability and least-privilege. The paper proposed a two-step algorithm for the UAQ problem. In the first step, the role mapping module uses a greedy search to select a set of roles that cover the desired permissions while attempting to minimize extra permissions these roles have. In the second step, the activation checking subsystem checks whether the set of roles selected in the first step satisfy all the constraints. If one of the constraints is not satisfied, then the algorithm denies the user's request. The problem of this approach is that it unnecessarily denies many requests, because the greedy search algorithm does not consider the effect of any constraint and may choose a set of roles violating some constraint.

In [6], Li et al. tackled the reduction of the verification of a static authorization problem to a SAT instance. Their approach considers a set of static, mutually exclusive role constraints. Our SAT approach builds on their findings and generalizes the problem for use in dynamic scenarios.

Other related problems to user authorization query include SAAM [8], which deals with the problem of determining whether a user's request should be authorized considering all the role memberships of the user. The approach caches the allowed and denied request/responses, and approximates future access control decisions based on the cached data. This primarily deals with the situation when there are no sessions and all roles are used in authorization.

6. CONCLUSION AND FUTURE WORK

In this paper we propose efficient algorithms for user authorization in RBAC systems. The algorithms we explained are based on optimized recursive search and transforming user authorization framework queries to a SAT instance and solve it using a SAT solver. We proposed three algorithms for each approach which exactly match the requested permissions, minimally match where at least lower bound permissions are satisfied and maximally match where at most upper bound permissions are satisfied. We also talk about how this system applies when there are hierarchical role assignments and DSOD constraints in RBAC systems. We compared our algorithms with the UAQ framework in [9] which is a sub case of our general algorithms and found that our algorithms always provide a precise solution if there is a possible solution.

The recursive algorithmic approach for authorizing user request works efficiently in the exact match case and when there is lesser number of available roles in the maximal match and minimal match cases. Our immediate focus is on optimizing this algorithm for larger numbers of roles and do more pruning during the search process. Also we will be integrating our implementations with real RBAC systems to test with real scenarios. Another important path of future research involves combining the recursive algorithmic approach and SAT based approach. Since the SAT approach can be very efficient when there is a small number of clauses and the algorithmic approach can isolate no-solution scenarios, both can be combined to give an overall more efficient approach. The recursive algorithmic approach can be customized so that it can calculate whether the user request can be granted and whether any of the clauses are irrelevant and can be safely removed from the clause set; then the SAT based approach can proceed further and find the solution.

7. REFERENCES

[1] zchaff. `http://www.princeton.edu/~chaff/zchaff.html`.

[2] S. Du and J. B. D. Joshi. Supporting authorization query and inter-domain role mapping in presence of hybrid role hierarchy. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 228–236, New York, NY, USA, 2006. ACM.

[3] Z. Fu and S. Malik. On solving the partial max-sat problem. In A. Biere and C. P. Gomes, editors, *Proceedings of Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, August 2006.

[4] J. B. D. Joshi, E. Bertino, and A. Ghafoor. Temporal hierarchies and inheritance semantics for gtrbac. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 74–83, New York, NY, USA, 2002. ACM.

[5] N. Li, J. Byun, and E. Bertino. A critique of the ANSI standard on role based access control. *IEEE Security and Privacy*, 5(6):41–49, Nov. 2007.

[6] N. Li, M. V. Tripunitara, and Z. Bizri. On mutually exclusive roles and separation of duty. *ACM Transactions on Information and System Security*, 10(2), May 2007.

[7] C. Sinz. Visualizing sat instances and runs of the dpll algorithm. *J. Autom. Reason.*, 39(2):219–243, 2007.

[8] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in rbac systems. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 63–72, New York, NY, USA, 2008. ACM.

[9] Y. Zhang and J. B. D. Joshi. Uaq: a framework for user authorization query processing in rbac extended with hybrid hierarchy and constraints. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 83–92, New York, NY, USA, 2008. ACM.

APPENDIX

A. SAT

A boolean expression ϕ is either a boolean variable, a negation of a variable $(\neg\phi)$, a conjunction of boolean expressions $(\phi_1 \wedge \phi_2)$, or a disjunction of boolean expressions $(\phi_1 \vee \phi_2)$. Each variable is a Boolean that can be assigned true or false. SAT is the problem of identifying an assignment of variables of a certain boolean expression in order to make the formula evaluate to true. If such an assignment exists, the formula is said to be satisfiable. The assignment of the variables in the boolean expression is said to be a SAT instance.