Rule-Based RBAC with Negative Authorization

Mohammad A. Al-Kahtani
Computer Department of Saudi Air Defense
mohammad_abdulla@yahoo.com

Ravi Sandhu
George Mason University & NSD Security
sandhu@gmu.edu

Abstract

RBAC has proven to be a flexible and useful access control model in practice. Rule-Based RBAC family of models was developed based on RBAC to overcome some of its limitations. One particular model of this family, which we call RB-RBAC-ve, introduces the concept of negative authorization to the RBAC arena. This paper provides a more detailed analysis of RB-RBAC-ve. The analysis includes user authorization, conflict among rules, conflict resolution polices, the impact of negative authorization on role hierarchies and enforcement architecture.

1. Introduction.

Role-based access control (RBAC) has emerged as a widely deployed alternative to classical discretionary and mandatory access controls [1, 2 and 3]. Since roles in an organization are relatively persistent with respect to user turnover and task re-assignment, RBAC provides a powerful mechanism for reducing the complexity, cost, and potential for error of assigning users permissions within the organization. Conventional RBAC was designed with a closed-enterprise environment in mind where a team of security officers manually assign users to roles. However, the landscape of business and information technologies has changed dramatically in recent years. An increasing number of service-providing enterprises make their services available to their users via the Internet. There has been some work to extend present RBAC models so they can be used to manage users' access to the enterprise services and resources over the Internet [4,5, and 6].

Also, many enterprises have users (i.e. workers and/or clients) whose numbers can be in the hundreds of thousands or millions [7]. Typical examples are banks, utility companies, insurance companies and popular Web sites, to name a few. For such enterprises, manually assigning users to roles may not be feasible, especially in case of external users, i.e. the enterprise customers and business partners.

Moreover, RBAC is being supported by software products designed to serve large number of clients, such as popular commercial database management systems, e.g. Oracle, Informix, and Sybase [8].

All of these factors mentioned above render the manual user-to-role assignment a formidable task because maintaining user-role assignment up-to-date is both costly and error-prone. Besides, automated assignment gives the enterprise an edge by extending its user-consumer business partnership.

In fact, some enterprises with large customer bases have already implemented systems that assign and revoke users automatically [7], and many of them have achieved 90-95% automation of administration [9]. Rule-Based RBAC (RB-RBAC) Family of models was suggested to provide a sound conceptual basis for the automation process and sets a benchmark for software implementations of the process [10, 11 and 16]. RB-RBAC provides the specification needed to *automatically* assign users to roles based on a finite set of authorization rules defined by the enterprise, hence the name Rule-Based RBAC or RB-RBAC for short. The RB-RBAC family introduces negative authorization, represented by negative roles, to the RBAC world. The central contribution of this paper is to explore and analyze different aspects of negative authorization in RB-RBAC context.

This paper is organized as follows. Section 2 provides an overview of related research. In section 3, RB-RBAC is revisited. In section 4, we introduce the RB-RBAC-ve model i.e. RB-RBAC with negative authorization. Section 5 concludes the paper including a discussion of issues that we have not explored in this paper, though they are closely related to the topic discussed.

2. Related Work.

In the real world of access control, there are two well-known decision policies [12]:

a. Closed policy: This policy allows access if there exists a corresponding positive authorization and denies it otherwise.
b. Open policy: This policy denies access if there exists a corresponding negative authorization and allows it otherwise.

Bertino et al. contends that the closed policy approach has a major problem in that the lack of a given authorization for a given user does not prevent this user from receiving this authorization later on. They therefore proposed an explicit *negative* authorization as blocking authorizations. Whenever a user receives a negative authorization, his positive authorizations become blocked [13].

Negative authorization is typically discussed in the context of access control systems that adopt open policy. There is an extensive amount of work in this regard, see for example [14] and [13]. The introduction of negative authorization brings with it the possibility of conflict in authorization, an issue that needs to be resolved in order for the access control model to give a conclusive result. The types of conflicts brought about by the negative authorization and conflict resolution polices are discussed in abundance outside RBAC literature. For example, Jajodia et al. suggest a model that is based on a logical authorization language that allows users to specify, together with the authorizations, the policy according to which access control decisions are to be made [15]. The key components of the model are objects, subjects, actions, and rules. Subjects who may be authorized to perform actions on objects include user, roles and groups. The unit of authorization is an action on an object. The authorization language expresses the policy by means of rules of different types. One type of rule is used to explicitly authorize users, roles or group. Another type of rule is used to derive further authorization based on those provided by the first type of rule. Any conflict that might arise with respect to authorization derivation is resolved using a third type of rule. Several types of conflicts and conflict resolution policies are suggested. RB-RBAC utilizes some of these policies as well as some new conflict resolution policies specified in this paper for the first time. In another work, Jajodia et al. provide formal definitions for several policies for authorization propagation and conflict resolution [12].

Negative authorization is rarely mentioned in RBAC literature, mainly because RBAC Models such as RBAC96 and the proposed NIST standard model are based on positive permissions that confer the ability to do something on holders of the permissions [2]. This is different from the semantics given to this concept in RB-RBAC, as will be discussed in section 4.

Al-Kahtani has proposed a family of models which can be used to dynamically assign users to roles based on a set of authorization rules defined by the enterprise. These rules take into consideration users' attributes and any constraints set forth by the enterprise's security policy. The Rule-Based RBAC (RB-RBAC) models provide a family of languages (*Authorization Specification Languages* or ASL for short) to express these rules. The models also define relations among rules, provide specification for derived induced hierarchies among the roles, and allow constraints specification. Figure 1 shows members of the RB-RBAC family. Model A is the most basic among the family. This model allows the specification of a set of authorization rules that can be used to assign users to roles based on users' attributes. Model B extends Model A to allow the specification of negative authorization (Model B_1) and mutual exclusion (Model B_2) by extending the ASL_A language. The extended language is called ASL_{B1} and ASL_{B2}, respectively. Model C extends Model A to allow constraints specification. In the following two sections we will briefly discuss model A which is the basic model and then we introduce negative authorization to RB-RBAC which yields model B_1 which we name RB-RBAC-ve in this paper.

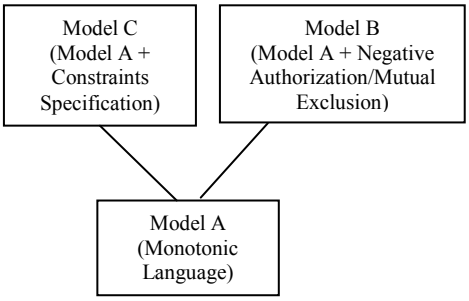

Figure 1 :RB-RBAC Family

3. RB-RBAC Model A.

3.1 Model A Basic Concepts.

This model is discussed in [10,11 and 16]. The main components of the RB-RBAC model A are the sets U, AE, R, and P which represent users, attribute expressions, roles, and permissions respectively (Figure 1).

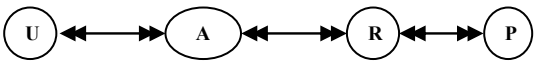

Figure 2: RB-RBAC Main Components

The U, R, and P sets are imported from RBAC96. In RB-RBAC, the security policy of the enterprise is expressed in the form of a set of authorization rules. Each rule takes as an input the attributes expression (a member of AE set) that is satisfied by a user (a member of U set) and produces one or more roles (a member of R set). An attribute expression is a well-formed formula in

propositional logic that specifies what combination of attributes values a user must satisfy in order to be authorized to roles specified in the rule. The attributes expressions can be stated using the language provided by the model. Syntactically, a rule has two parts:

a. The left hand side (LHS) of a rule is an attribute expression.
b. One or more role(s) in the right hand side (RHS).

If *u* satisfies the attribute expression, *u* is authorized to the role(s) specified in RHS of the rule. The following is an example of a $rule_i$:

$$ae_i \Rightarrow r_g$$

where ae_i is the attribute expression and r_g is the produced role. If user *u* satisfies ae_i, then *u* is authorized to all the roles in the right hand side of $rule_i$. To maintain user-role authorization the set URAuth is defined as follows:

$$\text{URAuth} = \{(u,r) | \ (\exists rule_i)[u \text{ satisfies } ae_i \wedge r \in RHS(ae_i)\}$$

If $(u,r) \in$ URAuth then this means that *u* is authorized to role *r*. This set is the key component of RB-RBAC since it captures the semantics of user-role assignment in the models. Only a user who has authorization on roles that are specified in RHS can activate these roles. Activating a role enables the user to execute the permissions assigned to that role. A user can activate one or more of his authorized roles in a session. Different sessions belonging to the same user can have different roles.

There is an implicit "OR" among the rules. If *u* satisfies one or more rules that produce different roles, then he is authorized to activate any combination of these roles. Upon receiving a user request of a role, the system that implements RB-RBAC searches the authorization rules set to find a rule which the user satisfies such that the rule yields that requested role. As a user satisfies more rules, the set of roles that he is authorized to assume does not diminish. Thus Model A is *monotonic*.

3.2 User States.

A user can be in any of several states *wrt* a specific role. For a given role *r*, we distinguish the following user's states:

a. Potential (P): user *u* is authorized to role *r* but has not activated it yet.
b. Revoked (R): user *u* has activated role *r* at least once but is not currently authorized to activate it.
c. Not-candidate (N): user *u* has not activated role *r* and is not currently authorized to activate it because he does not have the required attributes for assuming *r*, i.e. *u* is not authorized to *r*.
d. Deleted (Del): user *u* has been deleted from the system by an authorized individual such as the System Security Officer (SSO).
e. Active (Act): refers to the state where the user is currently active in the role.
f. Dormant (D): After deactivating a role, the user becomes dormant with respect to that specific role, i.e. in "D" state.

The importance of this distinction among different states of users becomes clear when specifying constraints and enforcing policies like the Chinese Wall. Figure 2 shows the state diagram of a user with respect to a single role.

To express authorization rules, RB-RBAC provides ASL_A a language based on a context-free grammar. The language syntax and semantics are detailed in [10].

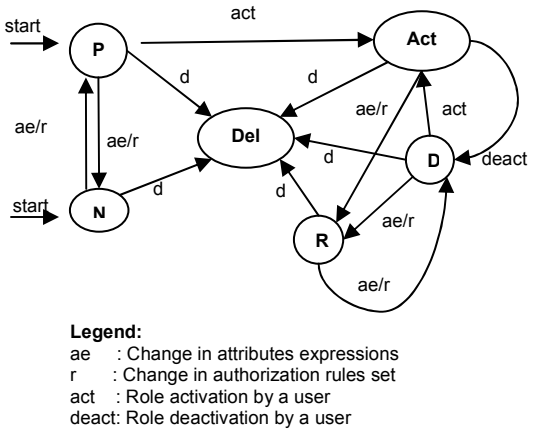

Figure 3: User's State Diagram with Sessions

3.3 Seniority Among Authorization Rules.

Seniority can be determined among the rules based on attributes expressions on their left hand sides. The "≥" symbol, read "is senior to", represents seniority relation among rules:

$$rule_i \geq rule_j \leftrightarrow (ae_i \rightarrow ae_j)$$

where ae_i and ae_j are the LHS of $rule_i$ and $rule_j$ respectively. This implies that users who satisfy $rule_i$ also satisfy $rule_j$ and, hence, are authorized to the roles produced by $rule_j$. The seniority relation on authorization rules, i.e. among attributes expressions forming the LHS of the rules, induces a hierarchy among the roles forming the RHS of these rules. This induced role hierarchy (IRH) captures inheritance of user-role assignment. If r_i is senior to r_j then the users who satisfy the LHS of the rule that yields r_i will also satisfy the rules that yield r_j. As a result, the set of r_i users is a subset of r_j users. In other words, user inheritance flows downwards in the IRH graph, that is, a junior role in IRH inherits all the users assigned to its seniors. In general IRH is a quasi-order, i.e., it is reflexive and transitive.

3.4 Alternative Ways to Gain Authorization.

There are three approaches to assign roles to users:

1. Implicit Assignment: Based on certain criteria, users are automatically assigned to roles. This is what Model A does thus far.
2. Hybrid Assignment: Besides the automatic assignment, the SSO can manually assign users to roles.
3. Explicit Assignment: In this approach a person with proper authority such as the SSO manually assigns users to roles. This is what traditional RBAC follows.

In [16] Al-Kahtani argues that there are situations where pure implicit assignment is not flexible enough. Thus, to provide flexibility, the concept of *can_assume* was introduced. The SSO may use *can_assume* relation to explicitly authorize users who are authorized to a role, say r_g, to another role, r_h, for a certain duration *d* starting at a specific time *t*. The SSO specifies the duration and the starting time. As a result, the user(s) in role r_g is authorized to activate role r_h at time *t* for duration of *d*. The motivation and specification of *can_assume* relation is detailed in [16]. Appendix A provides a summary of relevant definitions extracted from RB-RBAC model.

4. RB-RBAC-ve Model.

4.1 Introduction.

RB-RBAC-ve extends Model A to allow the specification of negative authorization (called Model B_1 in [16]). This extension has an impact on user authorization, formally represented by URAuth set and it may cause conflict among rules. RB-RBAC-ve is the first RBAC model that provides detailed analysis of different aspects of negative authorization in an RBAC context. In this section, we analyze this conflict and present several novel conflict resolution polices. The definition of URAuth is modified to accommodate the semantics of negative authorization. The new definition takes into consideration conflict resolution policies in effect. We also discuss the impact of negative authorization on URAuth, IRH, and RB-RBAC enforcement architecture.

To specify a negative authorization we use the ASL_{B1} language which imports the syntactic constructs of ASL_A (Appendix B) but it modifies the syntax of Roles as follows [16]:

Roles ::= [¬] Role

role-set ::= Role | Role||,||role-set

The syntax above allows specifying negative authorization on roles such as the following:

$$ae_k \Rightarrow \neg r_i$$

The rule above states that once a user satisfies ae_k the system that implements RB-RBAC prohibits that user from assuming r_i.

4.2 Motivation.

The motivations to use negative authorization are not immediately apparent in environments where RBAC is applied. Even though user-role assignment could be decentralized [17], it is not left to users' discretion to assign other users to roles. Instead a small number of individuals (e.g. SSOs) are entrusted with applying the enterprise security policy regarding user-role assignment. However, since RB-RBAC automates this process, negative authorization provides an extra safeguard, since it is not always easy to foresee all possible combinations of roles a user can assume based on his attributes, which change over time. Negative authorization helps in blocking any user whosoever satisfies certain criteria (expressed as attributes expression) from assuming certain roles. Also, it can be used to block receiving authorization of certain roles via *can_assume* and *can_delegate* relations. The SSO can use *can_assume* relation to explicitly authorize users who are authorized to a role, say r_g, to another role, r_h, for a certain duration *d* starting at a specific time *t*. As a result, the user(s) in role r_g is authorized to activate role r_h at time *t* for duration of *d*. Also, he may use *can_delegate* relation to permit regular users to delegate their memberships in specific roles to other users. To motivate the use of negative authorization in the context of RBAC, consider the example of a military unit that has a *Commander* and four staff officers, usually known as *G1* through *G4* as depicted in Figure 3.

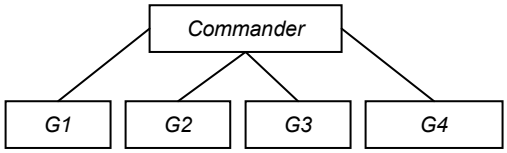

Figure 4 : RBAC Hierarchy for a Battalion

The commander can delegate his authority to any of his staff officers as long as the staff officer meets certain criteria specified by the military regulations.

Table 1

Attributes in the System: a_1: rank-type = officer a_2: Staff course = T a_3: Leadership course = T a_4: Rank ≥ Lt. Colonel a_5: Assignment Order = T
Authorization Rules: $a_1 \wedge a_2 \Rightarrow \{G_1, G_2, G_3, G_4\}$ $a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \Rightarrow Commander$ $\neg a_4 \Rightarrow \neg Commander$
can_delegate(*Commander*, G_1, *d*, *t*)

In Table 1, we show a security policy that specifies a possible real world situation. The policy uses negative authorization to prevent a *Commander* from delegating his role to a staff officer whose rank is lower than a Lt. Colonel.

4.3 Analysis of RB-RBAC-ve .

4.3.1 Conflict Due to Negative Authorization.

Introducing "¬" to the RHS may lead to conflict in the state of a single user *wrt* a single role. The conflict is due to simultaneous positive and negative authorizations. In Figure 5, the symbol "⇒" in a rule means that attribute expression ae_i *produces* the roles listed to the right of the arrow. Using the set of authorization rules shown in the figure, the following are several variations of conflict:

a. Case 1: Conflict among unrelated rules like the one between $rule_2$ and $rule_3$. If u satisfies $rule_2$ and $rule_3$ simultaneously then u should be authorized to activate r_1 (i.e. u is in P state *wrt* r_1) and denied r_1 at the same time (i.e. u is in N state *wrt* r_1). This case is represented by the following:

$$(u, ae_i) \in U\_AE \wedge (u, ae_j) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg r \in RHS(ae_j)$$

Where U_AE is defined such that U_AE = $\{(u, ae_i) | (u, ae_i) \in U \times AE \wedge u \text{ satisfies } ae_i\}$. $(u, ae_i) \in U\_AE$ means that u is authorized to $RHS(ae_i)$.

b. Case 2: Conflict among related rules: $rule_3$ and $rule_5$ are conflicting because if u satisfies $rule_3$ then he is denied r_1 (i.e. u is in N state *wrt* r_1), but at the same time, authorized to assume r_1 (i.e. u is in P state *wrt* r_1) because $rule_3 \geq rule_5$. This case is represented by the following:

$$(u, ae_i) \in U\_AE \wedge (u, ae_j) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg r \in RHS(ae_j) \wedge ((ae_i \rightarrow ae_j) \vee (ae_j \rightarrow ae_i))$$

c. Case 3: Conflict between implicit assignment i.e. via an authorization rule and explicit assignment i.e. via *can_assume* or *can_delegate*. Suppose that the SSO issued the following:

can_assume(r_4, r_3, t, d)

This allows users who are authorized to r_4 to activate r_3. If u satisfies ae_1, i.e. u is in N state *wrt* r_3, and at the same time is authorized to r_4. Nonetheless, the *can_assume* relation above authorizes u to r_3, which leads to a conflict.

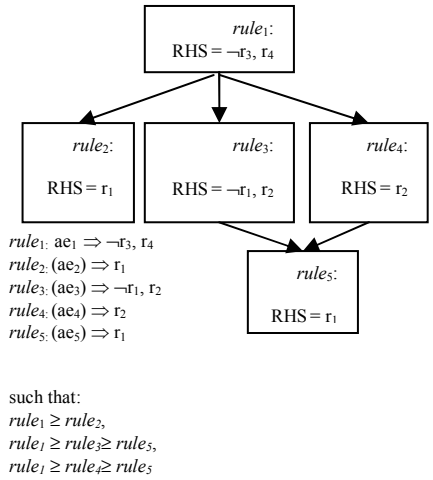

Figure 5

4.3.2 Conflict Resolution Policies. Conflict resolution policies have been discussed extensively in the literature, see for example, [13, 15 and 12]. Most notable among them are:

a. Denial Takes Precedence (DTP): Negative authorizations are always adopted when conflict exists.
b. Permission Takes Precedence (PTP): Positive authorizations are always adopted when conflict exists.

These two policies in their original form suffer the following deficiencies:

a. They are very rigid in the sense that they do not allow specification of special cases that violate the policy enforced. Suppose a hospital has a policy that has the following authorization rules:

$rule_1$: No. of years in residency ≤ 1 ⇒ intern

$rule_2$: No. of years in residency ≤ 1 ⇒ ¬ ER_doctor

Naturally, during the holiday seasons large numbers of the medical staff take their yearly vacation. However, this period of the year witnesses a surge in the number of people admitted to the emergency room. Clearly, additional medical staff is needed to handle this surge in demand of medical care. The administration may allow interns to work in the ER, and hence authorizes them to role ER-doctor.

One way to handle this is to change the hospital policy by deleting $rule_2$. This course of action is not preferred because it might lead to unseen side effects. Also, it might lead to a breach in the security policy if the SSO forgets to add it back after the holiday season is over. A better solution is to use *can_assume* relation as follows:

can_assume(intern, ER_doctor, t, d)

This authorizes interns to activate the role ER_doctor, i.e. to work in the emergency room. *can_assume* conflicts with $rule_2$. However, if DTP is enforced, the

interns will not be able to work in the ER unless *rule*$_2$ is deleted. What is needed in this situation is a relaxed version of DTP that allows the stating of this exception in the security policy.

b. DTP with negative authorization is useful in a closed policy environment. However, PTP renders negative authorization meaningless in such environments. This is so because *wrt* any role that is associated with negative authorization, there could be only one of the following possibilities:
   i. Conflict may arise: Since PTP is enforced, the negative authorization is ignored.
   ii. No conflict may arise: There is no need for the negative authorization since we are assuming a closed policy.

Based on that, we argue that there is a need for more flexible conflict resolution policies. The following section discusses newly formulated conflict resolution policies; some of them specify DTP policy with varying degrees of flexibility.

4.3.2.1 Localized DTP (LDTP). DTP policy resolves any conflict in favor of denial. This is rather restrictive since it means the more rules a user satisfies, the higher is the risk that he might be denied access to a role due to a conflict in authorization which is counter-intuitive. We propose modifying the DTP policy such that the conflict among *unrelated* rules is resolved in favor of permission. In other words, the denial is localized to conflict among comparable rules. We name the modified policy: the Localized DTP, or LDTP for short. Based on this, applying the LDTP policy on the three cases of conflict mentioned in the previous section results in the following authorizations:

Case 1: $(u,r) \in$ URAuth i.e. *u* is authorized to activate role *r*.

Case 2: $(u,r) \notin$ URAuth

Case 3: $(u,r) \notin$ URAuth

4.3.2.2 Flexible DTP (FDTP). This policy enforces DTP in cases where conflict occurs among authorization rules but it enforces PTP if conflict occurs between the implicit assignment and explicit assignment. Thus, when FDTP is enforced, in the example of the hospital discussed above, an intern can work as an ER doctor via *can_assume* relation without the need to remove *rule*$_2$ from the authorization rules set. In other words, FDTP policy authorizes *u* to role *r* if there is no conflict *wrt* role *r*, or if there is a *can_assume* relation which authorizes *u* to role *r* even if *u* receives a negative authorization *wrt* to *r*. Applying the FDTP policy on the three cases of conflict mentioned in the previous section results in the following authorizations:

Case 1: $(u,r) \notin$ URAuth

Case 2: $(u,r) \notin$ URAuth

Case 3: $(u,r) \in$ URAuth

In Table 2 we summary how the afore-discussed policies compare and contrast.

Table 2

Policy ↓	Conflicting Parties		
	Comparable Rules	Non-comparable Rules	Rules and SSO-initiated authorization (*can_assume* and *can_delegate*)
DTP	Denial	Denial	Denial
PTP	Permission	Permission	Permission
LDTP	Denial	Permission	Denial
FDTP	Denial	Denial	Permission

The entry at the intersection of the fourth row with the third column, for example, means that under LDTP if the conflicting parties are non-comparable rules, then permission prevails.

4.3.2.3 Weighted Rules. Authorization rules are assigned weights according to criteria determined by the enterprise such as:

a. The seniority of the rule, so *rule*$_3$ has higher weight than *rule*$_5$ in Figure 4 and, thus, the negative authorization is enforced.
b. When rules administration is decentralized, the SSO may authorize a junior security officer (JSO) to administer a specific group of rules. Conflict may arise among rules specified by SSO and JSO. One way to resolve this is by considering the seniority of the rule issuer. Based on this, an SSO-issued rule has higher weight than a rule issued by a junior security officer and, thus, the authorization obtained via the higher rule prevails.

4.3.2.4 Labeled Roles. This policy requires assigning label to each role. This label could be either one of the following values: DTP or PTP. If r_g and r_h are roles such that they are respectively labeled DTP and PTP, then in case of conflict *wrt* r_g, DTP is always enforced, while PTP is enforced in case of r_h. The notion and notation of role ranges [17] could be utilized in this context. An *assign_label* relation can be defined as follows:

$$assign\_label \subseteq \{\text{DTP, PTP}\} \times 2^{IR}$$

So, *assign_label*(DTP, $[r_g, r_h]$) assigns DTP label to all roles in the range $[r_g, r_h]$.

4.3.3 Users' Authorization. We have discussed several policies that can be deployed to resolve conflicts that may arise among authorization appointed to a specific user. In this section, we modify the definition of the set "URAuth" under selected policies to reflect the impact of conflict, if it exists, on user's authorization. While it is possible to do

this with respect to all the conflict resolution policies that we have discussed, for the sake of brevity, we choose to focus on PTP, DTP, LDTP and FDTP.

Definition 1

1. URAuth, *A*, and *B* are imported from Model A, and are mentioned here for convenience:
   URAuth= $\{(u,r)|\ (\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)\}$
   A= RHS of URAuth above.
   B= $(\exists rule_j)$ [$(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge$ *can_assume*$(r', r, t, d) \wedge$ *can_assume* has not expired]
2. Let $C = \neg\ (\exists rule_j)[(u,\ ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]$ which means that user *u* has a negative authorization *wrt r* via satisfying a rule $rule_j$.
3. Let $C' = \neg\ (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)]$ which means that user *u* has a negative authorization *wrt r* via satisfying another rule $rule_j$ that is comparable to a rule that positively authorizes *u* to *r*.
4. URAuth varies according to the policy enforced:
   a. PTP: URAtuh in PTP with/without *can_assume* is similar to the corresponding URAtuh in Model A.
   b. DTP : $URAuth^{DTP} = A \wedge C$, or
      $URAuth^{DTP} = \{(u,r)|\ (\exists rule_i)[(u,\ ae_i) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg\ (\exists rule_j)[(u,\ ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]\}$
   c. DTP with *can_assume*: $URAuth^{DTP \text{ with } can\_assume} = (A \vee B) \wedge C$, or
      $URAuth^{DTP \text{ with } can\_assume} = \{(u,r)|\ ((\exists rule_i)[(u,\ ae_i) \in U\_AE \wedge r \in RHS(ae_i)]$
      $\vee\ (\exists rule_j)\ [(u,\ ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge$ *can_assume*$(r', r, t, d) \wedge$ *can_assume* has not expired $]) \wedge \neg\ (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]\}$
   d. LDTP: We modify the term *C* to require the conflicting rules to be comparable. Call the modified term *C'*, thus $URAuth^{LDTP} = A \wedge C'$
      $URAuth^{LDTP} = \{(u,r)|\ ((\exists rule_i)[(u,\ ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \wedge \neg\ (\exists rule_j)[(u,\ ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)]\ )$
   e. LDTP with *can_assume*: $URAuth^{LDTP \text{ with } can\_assume} = (A \vee B) \wedge C'$
      $URAuth^{LDTP \text{ with } can\_assume} = \{(u,r)|\ ((\exists rule_i)[(u,\ ae_i) \in U\_AE \wedge r \in RHS(ae_i)]$
      $\vee\ (\exists rule_j)\ [(u,\ ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge$ *can_assume*$(r', r, t, d) \wedge$ *can_assume* has not expired $]) \wedge \neg\ (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)]\ \}$
   f. FDTP: $URAuth^{FDTP} = URAuth^{DTP}$
   g. FDTP with *can_assume*: $URAuth^{FDTP \text{ with } can\_assume} = (A \wedge C) \vee B$
      $URAuth^{FDTP \text{ with } can\_assume} = \{(u,r)|\ ((\exists rule_i)[(u,\ ae_i) \in U\_AE \wedge r \in RHS(ae_i)]$
      $\wedge \neg\ (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)\ ]\ )$
      $\vee\ (\exists rule_j)\ [(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge$ *can_assume*$(r', r, t, d) \wedge$ *can_assume* has not expired $]\}$

Table 3 summaries the definition of URAuth under different policies.

Table 3

Policy	URAuth	
	Without *can_assume*	With *can_assume*
PTP	A	$(A \vee B)$
DTP	$A \wedge C$	$(A \vee B) \wedge C$
LDTP	$A \wedge C'$	$(A \vee B) \wedge C'$
FDTP	$A \wedge C$	$(A \wedge C) \vee B$

4.3.4 Impact on Roles Hierarchies. The concept of a given role hierarchy (GRH) that represents the current business practice of the enterprise is discussed in [11] and [16]. The GRH is identical to role hierarchies defined in RBAC96, that is, it is permission-driven:

$$(r_i \geq_{GRH} r_j) \rightarrow r_{j\ permissions} \subseteq r_{j\ permissions}$$

where $\geq_{GRH}$ has the same semantics as in RBAC96. As such, inheritance of permissions flows upward in the GRH. When a GRH is present, $rule_i$ such that $ae_i \Rightarrow \neg\, r_g$ may have one of the following two possible semantics:

a. Propagation prohibited: Users who satisfy ae_i should be prohibited from assuming r_g. This is the interpretation given previously.
b. Propagation allowed: Negative authorization propagates upward in GRH such that users who satisfy ae_i should be prohibited not only from assuming r_g, but also from assuming any role r_k such that $r_k \geq_{GRH} r_g$. This ensures that the user cannot circumvent the system by assuming r_k, whose permissions are a superset of r_g's. From a functional perspective, this may not be desirable since it is usually the case that the prohibition is targeting users who merely satisfy $rule_i$, but not those who can assume roles higher in the hierarchy by virtue of satisfying rules senior to $rule_i$, which usually means that they meet higher security requirement. Allowing the negative authorization to propagate upward requires modification of the definition of URAuth. For a user to be authorized to a role *r*, not only do we require that *u* has positive authorization *wrt r* and does not have negative authorization *wrt r*, but we also require that *u* does not have negative authorization wrt any role *r'* such that $r \geq_{GRH} r'$ i.e. *r* is senior to *r'* in GRH.

Definition 2

URAuth definition is modified to take propagation of negative authorization into account. We need to modify term *C* as follows:

Term *C* becomes: $C_{modified} = \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r' \in RHS(ae_j) \wedge r \geq_{GRH} r']$

Notice that we can replace the term *C'* in Definition 8 with $C_{modified}$ since $r \geq_{GRH} r'$ implies that the rules that generate r and r' are comparable.

4.3.5 User State Diagram. Suppose that the system that implements RB-RBAC has the following set of rules only:

$rule_i$: $ae_i \Rightarrow r_g$

$rule_j$: $ae_j \Rightarrow r_h$

$rule_k$: $ae_k \Rightarrow \neg r_h$

Let's consider the following scenarios assuming DTP is in effect and using Figure 2:

Scenario 1: Assume that *u* satisfies $rule_j$ only and, thus, $(u, r_h) \in$ URAuth. In other words, *u* could be in any of the following states *wrt* r_h: P, D, or Act. A change in *u*'s attributes or in the authorization rules may cause the system that implements RB-RBAC to invoke $rule_k$ assigning negative authorization to *u wrt* r_h. Accordingly, $(u, r_h) \notin$ URAuth and *u*'s state will be changed from P to N or from D or Act to R. The arrows labeled ae/r represent this.

Scenario 2: Assume that *u* satisfies $rule_i$ only. Hence, $(u, r_g) \in$ URAuth. As a result, *u* could be in any of the following states *wrt* r_g: P, D, or Act. A change in *u*'s attributes or in the authorization rules that cause the system that implements RB-RBAC to invoke $rule_k$ assigning negative authorization to *u wrt* r_h. If $r_g \geq r_h$ and propagation is allowed, *u*'s state will be changed as in scenario 1.

A change in *u*'s attributes or in the authorization rules may make *u* no more able to satisfy $rule_k$, and thus, *u* is no more authorized to $\neg r_h$. Also, *u* could become unable to satisfy $rule_k$ either because it was modified or deleted. This results in changing his state from N back to P, or from R to D.

4.3.6 Enforcement Requirements. Enforcing the negative authorization requires that the system which implements RB-RBAC has access to all relevant attributes. This requirement affects the architectural options that can be used to enforce RB-RBAC-ve since the system must either have these attributes under its control or be granted access to them when needed. If this is not the case, then users may evade the model. Consider rules $rule_2$ and $rule_3$ in Figure 4. If these rules were in public domain or were somehow unconcealed, then users whose attributes satisfy both ae_2 and ae_3 can avoid $rule_3$ simply by not providing the attributes necessary to satisfy ae_3. Though this may not be a problem under PTP policy, it amounts to a security breach under DTP policy. If RB-RBAC has access to users' attributes, DTP policy can be enforced.

4.3.7 Monotonicity. RB-RBAC-ve permits specifying the rules such that the set of roles that a user is authorized to decreases as the number of rules he satisfies increases. Suppose that we have $(\neg r_g) \in RHS(ae_i)$ and $\{r_g, r_h\} \subseteq RHS(ae_j)$. If a user *u* satisfies $rule_j$, then he is authorized to r_g and r_h. In case of DTP, if *u* satisfies both rules, he is authorized to r_h only. The above shows that RB-RBAC-ve is non-monotonic.

5. Discussion and Future Work

We have shown how to modify RB-RBAC so that it allows negative authorization. Negative authorization in the context of RBAC is a novel concept. RB-RBAC-ve is the first RBAC model that provides detailed analysis of different aspects of negative authorization in an RBAC context. This analysis includes providing semantics for the negative authorization in this new territory, identifying cases of conflict, suggesting several new conflict resolution policies and analyzing the impact of negative authorization on IRH, GRH and any RB-RBAC enforcement architecture.

The conflict resolution policies presented requires further analysis. For example, in the Labeled Roles resolution policy, there are some subtle issues that need to be analyzed further. Suppose we have two roles r_g and r_h such that $r_g \geq r_h$. Suppose also that we assign the labels DTP and PTP to r_g and r_h respectively. If *u* satisfies authorization rules such that he has conflict in both roles, then based on the labels assigned to the roles, *u* is authorized to r_h but not to r_g. This reduces the privileges available to *u*, which is not problematic since senior roles are naturally assigned more permission and, as thus, it is wise to err on the side of denial in case of conflict. However, assume that the labels were reversed and that *u* has conflict in both roles. The resolution will be such that *u* is authorized to r_g but not to r_h, which is very problematic since r_h's permissions are a subset of r_g's permissions. To follow this policy strictly, we need to suspend this subset of permissions, which may render r_g deficient or even meaningless. We have not found any practical example in which this scenario is applicable. So, when assigning labels to roles, we require that the roles higher in the hierarchy receive labels of equal or higher level than their juniors. We assume that DTP label is higher than PTP. We believe this requirement is reasonable since senior roles are naturally assigned more permission, so they need more protection.

Another candidate for future work is introducing the concept of parameterized roles to RB-RBAC family and

analyzing its impact on different aspects of the models such as user authorization, IRH.

6. References

[1] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, "Role-Based Access Control Model*", IEEE Computer*, 29(2), Feb. 1996.

[2] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST Model for Role-Based Access Control: Towards a Unified Standard", *In Proceedings of the fifth ACM workshop on Role-based access control table of contents*, Berlin, Germany, 2000, Pages: 47 - 63.

[3] D. Ferraiolo, R. Sandhu, S. Gavrila , and R. Kuhn, "Proposed NIST Standard for role-based access control: towards a unified standard", *In ACM Transaction on Information and System Security (TISSEC), Vol. 4, Number 3*, August 2001.

[4] D. Ferraiolo, J. Barkley, and R. Kuhn, "A Role Based Access Control Model and Reference Implementation Within a Corporate Intranet", *ACM Transactions on Information and Systems Security, 2(1):34-64*, February 1999.

[5] J. Park, R. Sandhu and G. Ahn, "Role-based Access Control on the Web", *In ACM Transactions on Information and System Security, Vol. 4, No 1*, 2001.

[6] Joon S. Park, Ravi Sandhu, and SreeLatha Ghanta. *"*RBAC on the Web by Secure Cookies" *In Proceedings of the IFIP WG11.3 Workshop on Database Security,* Chapman & Hall, July, 1999.

[7] A. Kern, A. Schaad and J. Moffett, "An Administration Concept for the Enterprise Role-Based Access Control Model", *SACMAT'03*, June 1-4, Como, Italy.

[8] C. Ramaswamy and R. Sandhu, "Role-Based Access Control Features in Commercial Database Management Systems", *NISSC* 1998.

[9] A. Kern, "Advanced Features for Enterprise-Wide Role-Based Access Control", *In Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, USA, December, 2002, pages 333-342.

[10] M. Al-Kahtani and R. Sandhu, "A Model for Attribute-Based User-Role Assignment", *In Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 9-13, 2002.

[11] M. Al-Kahtani and R. Sandhu, *"*Induced Role Hierarchies with Attribute-Based RBAC*",* *In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Villa Gallia, Como, Italy, June 2-3, 2003.

[12] **S. Jajodia, P. Samarati, M. Sapino and V. Subrahmanian, "Flexible support for Multiple Access Control Policies" , *In ACM Transactions on Database Systems*, Vol. 26, No. 2, June 2001.**

[13] E. Bertino, P. Samarati, and S. Jajodia, "An Extended Authorization Model for Relational Databases", *In IEEE Transactions On Knowledge and Data Engineering, Vol. 9, No. 1*, January-February 1997.

[14] E. Bertino, P. Samarati, and S. Jajodia, "Authorizations in Relational Database Management Systems*", In Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, VA.Nov. 3–5). ACM, New York, pp. 130–139.

[15] S. Jajodia, P. Samarati and V.S. Subrahmanian, "A logical Language for Expressing Authorizations", *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[16] M. Al-Kahtani, "A Family of Models for Rule-Based User-Role assignment", *A Ph.D. dissertation submitted to George Mason University*, 2004.

[17] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 Model for Role-based Administration of Roles", *ACM Transactions on Information and System Security. Vol.2, No.1*, Feb. 1999, pages 105-135.

Appendix A: Relevant Definitions from RB-RBAC

U, R, and P, imported from RBAC96, are the sets of users, roles, and permissions respectively. In addition RB-RBAC Model A has the following components.

2. A set of attribute expressions AE. Elements of AE are denoted as $ae \in$ AE (See the language in section 3.2.5.1).
3. A set of authorization rules where each rule $rule_i$ is written as: $ae_i \Rightarrow$RHS where $\Rightarrow$ is read "generates" or "yields" and RHS $\subseteq$ R.
4. Function *RHS*(ae_i) = RHS returns the set of roles that user *u* who satisfies ae_i is authorized to activate.
5. U_AE = {(u, ae_i)| (u, ae_i) $\in$ U$\times$AE $\wedge$ u satisfies ae_i}, (u, ae_i) $\in$ U_AE means that u is authorized to *RHS*(ae_i).
6. IR is the set of roles produced by all authorization rules:
   IR = { r_g | ($\exists ae_i$) [$ae_i \in$ AE $\wedge$ $r_g \in$ *RHS*(ae_i)}
7. URAuth = {(*u*,*r*)| ($\exists rule_i$)[(*u*, ae_i) $\in$ U_AE $\wedge$ *r* $\in$ *RHS*(ae_i)}. For the sake of convenience, we will call the right hand side of this definition as "*A*". We will refer to it in future definitions to simplify the relation of different models to each other.

The concept of session and the functions *sessions* and *user* are imported from RBAC96:

8. *sessions* : U $\rightarrow 2^S$, a function mapping each user u_i to a set of sessions
9. *user* : S $\rightarrow$ U, a function mapping each session s_i to the single user *user*(s_i) (constant for the session's lifetime)
10. URA $\subseteq$ URAuth, URA = {(*u*,*r*)| (*u*, *r*) $\in$ URAuth $\wedge$ *u* is currently activate *wrt r* }
11. URD $\subseteq$ URAuth, URD = {(*u*,*r*)| (*u*, *r*) $\in$ URAuth, $\wedge$ *u* has activated *r* at least once but is not currently active *wrt r* }
12. URP $\subseteq$ URAuth, URP = {(*u*,*r*)| (*u*, *r*) $\in$ URAuth $\wedge$ *u* has never activated *r*}
    URAuth = URA $\cup$ URD $\cup$ URP
    URA $\cap$ URD = $\varnothing$
    URA $\cap$ URP = $\varnothing$
    URD $\cap$ URP = $\varnothing$
13. URN $\subseteq$ U$\times$AE, URN = {(*u*,*r*)| (*u*, *r*) $\notin$ URAuth $\wedge$ *u* has not activated *r* in the past}
14. URR $\subseteq$ U$\times$AE, URR = {(*u*,*r*)| (*u*, *r*) $\notin$ URAuth $\wedge$ *u* had activated *r* at least once in the past }
15. *User_State*(*u*, *r*) =
    Case:
    a. (*u*, *r*) $\in$ URP: *User_State*(*u*, *r*) = P.
    b. (*u*, *r*) $\in$ URA: *User_State*(*u*, *r*) = Act
    c. (*u*, *r*) $\in$ URD: *User_State*(*u*, *r*) = D.
    d. (*u*, *r*) $\in$ URR: *User_State*(*u*, *r*) = R.
    e. (*u*, *r*) $\in$ URN: *User_State*(*u*, *r*) = N.
    f. Del: *u* is deleted by SSO.
    These states are mutually exclusive. The state Del is a terminal state.
16. *roles* : S $\rightarrow 2^R$, a function mapping each session s_i to a set of roles *roles*(s_i) $\subseteq$ {*r* | (*user*(s_i), r) $\in$ URAuth} (which can change with time)
17. *can_assume* relation: Specification is provided in [16]
18. URAuth$^{\text{with } can_assume}$ = {(*u*,*r*)| ($\exists rule_i$)[(*u*, ae_i) $\in$ U_AE $\wedge$ *r* $\in$ *RHS*(ae_i)
    $\vee$ ($\exists rule_j$) [(*u*, ae_j)$\in$ U_AE $\wedge$ *r'* $\in$ *RHS*(ae_j) $\wedge$ *can_assume*(*r'*, *r*, *t*, *d*) $\wedge$ *can_assume* has not expired])}
    Let's call the second term in the right hand side *B*, and hence we say:
    URAuth$^{\text{with } can_assume}$ = $A \vee B$
19. ($rule_i \geq rule_j$) $\leftrightarrow$ ($ae_i \rightarrow ae_j$).
20. IRH $\subseteq$ IR x IR is a relation such that r_g is senior to r_h ((r_g, r_h) $\in$ IRH is also written as $r_g \geq r_h$):
    IRH= {(r_g ,r_h) | ($\forall rule_i$) [($ae_i \Rightarrow r_g$) $\rightarrow$ ($\exists rule_j$) [$rule_i \geq rule_j \Lambda$ $ae_j \Rightarrow r_h$]]}
    Intuitively, this means r_g is senior to r_h in IRH if every rule that produces r_g is senior to a rule that produces r_h.
21. IRH = {(r_g ,r_h) | (*u*, r_g) $\in$ URAuth $\rightarrow$ (*u*, r_h) $\in$ URAuth}
22. $IR_{\sim}$ is the set of equivalence classes that results from defining relation "mutually senior to one another" on IR such that:
    [r_i] = { r_j | r_i and r_j are mutually senior to one another}
23. IRH = {([r_g] ,[r_h]) | $\forall u \forall r_g \in [r_g] \forall r_h \in [r_h]$ [((*u*,r_g) $\in$ URAuth $\rightarrow$ (*u*,r_h) $\in$ URAuth)
    $\wedge$ ((*u*,r_h) $\in$ URAuth $\rightarrow$ (*u*,r_g) $\in$ URAuth)]}
24. *IRH* and *GRH* are the sets of roles in IRH and GRH respectively.

Appendix B: Syntax Diagrams of ASLA Language