2018 42nd IEEE International Conference on Computer Software & Applications

Resource and Role Hierarchy Based Access Control for Resourceful Systems

Nidhiben Solanki, Yongtao Huang,
I-Ling Yen, Farokh Bastani
University of Texas at Dallas
{nxs121130, hxy131530, ilyen, bastani}@utdallas.edu

Yuqun Zhang
Southern University of Science & Technology
zhangyq@sustc.edu.cn

Abstract. Role based access control (RBAC) has been used extensively in practice since it naturally capturing the structure of the users in an organization. It is especially useful in multi-tenant cloud platforms. However, with the growing amount of data and growing number of devices, assigning permissions for these resources (such as data and devices) to roles become challenging. We develop a resource hierarchy based permission model and integrate it with RBAC to create the RRBAC (resource and role based access control) model to simplify the permission assignment in RBAC. However, realizing RRBAC requires careful design to ensure efficient permission assignment, validation, and revocation. Instead of using policy based solutions, such as XACML, we design a resource tree based approach to achieve high performance for various permission related operations. Preliminary experiments show that RRBAC approach can achieve more efficient permission assignment and validation.

Keywords: Cloud security, Role based access control, resource hierarchy, attribute based access control, permission assignment, permission validation.

1 Introduction

Cloud has been expanding and nowadays a large number of cloud providers at different scales are available. Also, more and more companies are shifting their services and applications to cloud platforms. Cloud computing has many benefits. It can reduce the overall operational cost for large and small companies due to sharing of computing resources [1]. Also, it facilitates application and service sharing cross organizations via SaaS (software as a service), Web services, and service computing technologies. Moreover, it promotes data sharing due to centralized hosting, which can provide semantically enhanced data management for effective data discovery [2]. One important issue in cloud computing is security. For example, in service sharing, the cloud provider needs to protect the services to ensure that only legitimate accesses are allowed. Same for data sharing, data resources should be protected against illegitimate accesses and information flows [3] [4]. In this paper, we focus on the access control aspect of security for cloud systems.

Many access control models have been developed in the past three decades and among all these models, role-based access control (RBAC) models are most widely used in enterprises and other organizations. Role-based models can greatly cut down the cost for policy specification. Also, Role hierarchy in RBAC provides a natural representation (role hierarchy) of the structure of the users in an organization. Role faithfully describes the responsibility and authority of the user in the position represented by the role.

The RBAC model focuses on building hierarchy of the subjects to reduce the overhead in access right specification and management, but does not consider the same for the objects (i.e., the resources to be accessed). Generally, data resources such as database and file systems have natural hierarchies. With the expansion of cloud computing to many application domains, more and more varieties of resources are to be considered and permissions assignment under the RBAC model becomes a major challenge.

Attribute-based access control (ABAC) is another access control model that has been widely used in recent years. In ABAC, attributes are defined for each subject and each object. Access control policies are defined based on these attributes. ABAC is a unified model of many conventional access control models since attributes can encompass any user descriptors such as the user name, the security level, the role, etc. More importantly, ABAC can define specific attributes and attribute values for the objects, and the desired policies can be defined based on them. In other words, ABAC can help avert the problem above.

ABAC is the most flexible and powerful access control model, but it comes at a price. Since there is no well-formed standard for attribute definition, the selection of attributes for specific systems becomes a burden. Also, the complexity for access right validation depends on the complexity of the attributes selected for the system. Moreover, the flexibility in ABAC makes policy verification and auditing difficult since the identification of the subjects having a certain attribute value can be time consuming [5]. The problem becomes more significant in the privilege revocation process when it is necessary to identify the exact subjects who may have accessed certain objects and the effects need to be reversed. To balance flexibility and operational cost, some research works consider mixed models to take advantage of RBAC and ABAC and eliminate their problems [6].

For the problem of high complexity for privilege assignment in RBAC, instead of using ABAC, we consider a resource hierarchy based approach. Similar to role hierarchy, data resources also have their natural hierarchy. Thus, permission assignment can be performed on resources based on the hierarchy. For example, in file systems, files are structured in the directory tree. Hierarchical structure also exists in database systems, from database to tables to rows/columns and to cells. An XML document, where

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.10280

different access rights are assigned to different branches at various levels, has its natural hierarchy. In an IoT system, the IoT resources can form a hierarchy based on, for example, their ownership role hierarchy. Within each IoT device, its data can be categorized into a hierarchy as well. Thus, we can make use of the resource hierarchy to ease permission assignment. Similar to role hierarchy, when a permission for a parent resource is granted to a role, it is highly likely that the permissions for all the child and descendant resources in the hierarchy are to be granted to that role. Based on this consideration, we define the concept of resource hierarchy to simplify permission assignment for cloud computing systems.

In this paper, we extend RBAC model by adding resource hierarchy as part of the access control mechanism and create the RRBAC (Resource and Role Hierarchy Based Access Control) model. By providing resource hierarchy as part of the access control mechanism, we can eliminate redundant access rights assignments. Once access right assignment is done at the parent data resource, we don't have to assign access rights to the dependent data resources. Resource hierarchy can be used to propagate access rights from parent to children data resources. In addition, resource hierarchy can also accommodate newly created data resources, as they are mostly created at the leaf levels. This also eliminates the need of assigning access rights for new resources. The access rights for new resources are also propagated from the parent data resource.

In RBAC, generally permissions are associated with atomic resources, so, permission validation can be performed directly when an atomic resource is accessed. If we allow the permissions to be defined on a group of resources and allow the subjects to access individual resources in the group, then it will be difficult to find the proper permissions without a proper mechanism. In a resource hierarchy, this problem for permission validation becomes more complex. A naïve solution is to first define the atomic resources based on the lowest granularity in which the resources can be accessed by the users. When there is a permission assignment at a non-atomic resource, the permission is propagated to all the children resources and all these generated permissions are inserted into the permission database. However, this will not simplify the permission management due to a large number of permissions and cannot improve the performance for permission validation (slow reasoning due to the large number of permission rules).

We design a resource tree based permission assignment and validation mechanism for RRBAC. A resource tree is maintained and the atomic resources are the leaves of the resource tree. For each resource (at any level of the tree), we maintain links in the tree to locate the permissions needed for access right validation. These links are established during permission assignments. We design algorithms to correctly and efficiently establish and traverse various tree links so that permission management can be greatly simplified and access rights validation can be handled efficiently.

The rest of the paper is organized as follow. Section 2 provides further literature survey (in addition to the general survey in Section 1), focusing on access control in multiple organizations. Section 3 introduces the resource and role hierarchy based access control concept and formally defines the RRBAC model. The detailed permission assignment, validation, and revocation algorithms are discussed in Section 4. Section 5 discusses the experimentation setup and the preliminary results. Section 6 concludes the paper and discusses future research directions.

2 Literature Survey

There have been significant advances in access control technologies over the last three decades. In the early era, basic access control schemes, such as access control matrix and capability lists, have been used. Multi-level security [7] introduces information flow control into access control to ensure that information will never flow from the higher security classes to the lower ones. From late 90s, RBAC has become the major paradigm, especially for large enterprises and organizations. Role hierarchy semantically reflects the structure of authorities and responsibilities of the personnel in an organization and, hence, the access rights can be defined accordingly. But due to the complexity of permission assignment in RBAC, ABAC has been adopted to compensate this deficiency. ABAC is flexible and powerful, but as discussed earlier, the power of ABAC also brings some problems.

Most of the traditional access control models, including RBAC and ABAC, are developed under the assumption of a unified environment. For example, the role hierarchy in RBAC and the attributes in ABAC are uniformly defined for the entire system. However, this frequently is not the case in multiple domain systems and handling cross domain accesses becomes a challenge. There are generally two approaches to secure cross-domain accesses. The first approach is to have a trusted mediator to integrate the subjects and objects of the two interacting domains (e.g., an integrated role hierarchy in the role-based model, an integrated lattice in the multi-level security model) [8]. In RBAC model, the solution is to create a federated role hierarchy. In ABAC, the mediator needs to provide a unified set of attributes and perform attribute translation during accesses. However, the mediator-based approach suffers from the scalability and fairness issues [8]. It also requires a fully trusted mediator to perform the integration. In [9], a mediator-free solution is proposed to secure cross-domain interoperation. Instead of creating a federated hierarchy of roles or security attributes, mappings of the roles or attributes from one domain to another are defined in a decentralized way. Access control in a domain is realized by mapping the external roles or attributes to the corresponding ones in the local domain during access right validation for incoming accesses.

Generally, ABAC requires a more complex mapping in multi-domain systems. In RBAC, one can simply map roles from one domain to another (the role is a fixed single attribute

and role name is the attribute value). For ABAC, it is necessary to first map the attributes, and then map the attribute values, which can be challenging.

3 Basic Concepts in RRBAC

Role based access control (RBAC) model has been investigated extensively in the literature and is most commonly used in practice. The basic idea of RBAC is to define roles according to the responsibilities within an organization. Hence, the organization structure is mapped into a role hierarchy. Each user is assigned to one or more roles and these roles are assigned permissions for accessing data resources. When a session is activated by a user, he/she can activate a subset of assigned roles to perform data accesses. Even though RBAC is the most widely used model in enterprises, RBAC models has some drawbacks. In RBAC managing permission assignments can have high complexity in systems with a large number of data resources. Also, it is not clear how to assign permissions for dynamically created data resources.

To solve the problems discussed above, we extend RBAC to resource and role hierarchy based access control (RRBAC). Generally, data resources have their natural hierarchy and we can perform permission assignment based on this resource hierarchy. For example, in file systems, files are structured in the directory tree. Hierarchical structure also exists in database systems, from database to tables to rows/columns and to cells. XML/HTML files also have tag structure which can be easily converted to the resource hierarchy structure. We can make use of the resource hierarchy to ease permission assignments. When a permission for a parent resource is granted to a role, it is highly likely that the permissions for all the child and descendant resources in the hierarchy are to be granted to that role as well. Based on this observation, we define the RRBAC (Resource and Role Hierarchies Based Access Control) model to simplify permission assignment. RRBAC eliminates the need for assigning permissions for each data resource. Also, data resource creations happen more frequently at the lower level of the resource hierarchy. If most of the permissions are assigned for resources at a higher level, then the problem for assigning permissions to newly created resources can greatly diminish.

Next, we define the RRBAC model by tuple $\langle U, UR, RH, DH, P, PR\rangle$. RH is the role hierarchy, which can be represented like a graph with vertices and directed edges, i.e., $RH = <R, ER>$, where $R = \{r_1, r_2, r_3, \dots\}$ is the set of roles and ER is the set of parent-child relations in the role hierarchy. We have $ER = \{(r_i, r_j) | r_i, r_j \in R\}$,where (r_i, r_j) is an edge in the role hierarchy and r_i is the parent of r_j. We also define the partial order $r_i > r_j$ if r_i is an ancestor of r_j.

U is the set of users and $U = \{u_1, u_2, u_3, \dots\}$. In RBAC, each user is assigned to one or more roles. UR is the set of user to role assignments, where

$$UR = \{(u_i, r_j) | u_i \in U, r_j \in R\}.$$

DH is the resource tree and similar to RH, and it can be expressed as the set of vertices and the set of edges, $<D, ED>$, where $D = \{d_1, d_2, d_3, \dots\}$ is the set of resources and ED is the set of parent-child relations in the resource hierarchy. We further define ED by $ED = \{(d_i, d_j) | d_i, d_j \in D\}$, where (d_i, d_j) implies d_i is the parent of d_j in the resource tree. We also define $d_i > d_j$ if d_i is an ancestor of d_j.

Now we consider permission and permission to role assignments. P in the RBAC tuple is the set of permissions. A permission p_k, $p_k \in P$, is associated to a resource d_i and access right a_j. Thus, we can express p_k as

$p_k = (d_i, a_j)$, where $d_i \in D, a_j \in A$.

Here, we assume that all resources have the same action set A and $A = \{a_1, a_2, a_3 \dots\}$. When permission p_k is assigned to a role r, it permits r to access d_i with access right a_j. Let PR denote the set of permission to role assignments, where

$PR = \{(p_k, r_j, pc) | p_k \in P, r_j \in R\}$,

where pc is the permission propagation constraint, which will be properly defined later.

When an access (r_k, d_i, a_j) is issued, the permission validation process checks whether there exists a $p_k = (d_i, a_j)$ such that $(p_k, r_j, pc) \in PR$. If so, then the access is granted; otherwise, it is declined.

In RBAC, a parent role inherits the permissions of its children. Formally, we have

$$(r_i, r_j) \in ER \wedge p_k \in P \wedge (p_k, r_j, pc) \in PR \wedge pc.rh = T \Rightarrow (p_k, r_i, pc) \in PR$$

In RRBAC, if a permission to access a parent resource d_x is assigned to a role, then the role has the permissions to access all the resources that are in the sub-tree of d_x in the resource hierarchy. Formally, we have

$$p_k = (d_x, a_i) \wedge (p_k, r_j, pc) \in PR \wedge pc.dh = T \Rightarrow \forall d_y, d_y \in subtree(d_x), (p_l, r_j, pc) \in PR,$$

where $p_l = (d_y, a_i)$.

Here, $subtree(d_x)$ is the set of nodes in the subtree of d_x.

To increase flexibility, we use pc to specify the permission propagation constraint. pc includes two fields, $pc.rh$, which specifies whether role hierarchy propagation is allowed, and $pc.dh$, which specifies whether resource tree propagation is allowed. $pc.rh$ and $pc.dh$ can be True (T) or False (F) and they have the default value True. The default permission propagation feature can reduce the effort in permission to role assignment. However, there are situations where such propagations are undesirable. Adding the control by pc provides additional capability in managing the access rights.

4 RRBAC Algorithms

Role based access control (RBAC) considers privilege propagation along the role hierarchy, while in RRBAC we consider privilege propagation along the role hierarchy and resource tree. Due to privilege propagations, the access right

validation mechanism needs to be done properly. One naïve way to do this is to derive all the permissions from the assigned ones and store them in a permission database. When given an access for a data resource, we need to check whether the permission is in the permission database. This approach requires maintaining a large number of permissions. Another potential approach is to record the permission to role assignments on the resource tree. When given an access to a data resource d_x by a role r_j, we need to traverse the resource tree from d_x up to find out whether a permission for accessing an ancestor of d_x has been assigned to r_j (without propagation constraint). If so, then r_j has access to d_x. If no such ancestor exists, then the traversal will go all the way up to the root of the resource tree, resulting in very inefficient access privilege validation.

In this section, we design algorithms to provide more efficient privilege assignment and validation in RRBAC. To simplify the discussion of the algorithm and related concepts, we consider a single access right a for all data resources. Thus, the parameter for access right "a_i", in permissions (P), are omitted in algorithm. Also, we assume that there is no constraint on permission propagations. Moreover, we assume that role hierarchy is much smaller than the resource tree in size. So, our performance improvement goal is to reduce resource tree traversal instead of role hierarchy traversal.

In the following subsection (Section 4.1), we introduce the data structure maintained to keep track of the access rights and the ideas about the operations. RRBAC has three major operations: permission assignment, validation, revocation. Due to space limit, we will only discuss the algorithm for permission assignment (Section 4.2).

4.1 Data Structure for RRBAC Algorithms

The basic idea in our design is to use the resource tree to maintain permission related information. In the resource tree, each node maintains information about permissions assigned explicitly. If the security officer assigns permission p_i for accessing resource d_x to role r_j, i.e., (p_i, r_j, pc), then d_x in the resource tree should keep r_j in its list of assigned roles. The attribute "roles with permission", specifically $d_x.rwp$, is used to record the list of all assigned roles with permission to access d_x.

To achieve efficient permission assignment and validation, we introduce two pointers to be maintained by each resource node d_x in the resource tree. First, we maintain an "access control parent node" pointer, denoted as $d_x.acpn$, to point to the closest ancestor of d_x that has at least one permission assigned to it. Let d_y denote the node that $d_x.acpn$ points to. Since the roles that can access d_y should be able to access d_x (recall that we assume default propagation), $d_x.acpn$ allows an access to d_x to be validated through d_y without needing d_x to duplicate $d_y.rwp$. Since there is no other node between d_x to d_y that has its access permission being assigned to some roles, validating the access right to d_x via d_y without traversing through the intermediate nodes will not cause any problem.

However, d_y may inherit some "roles with permissions (rwp)" from node $d_y.acpn$. Thus, permission validation procedure needs to traverse the "$acpn$" pointers up the resource tree. We can leave the traversal along "$acpn$" pointers to validation time, or we can propagate newly assigned roles down so that for any node d_x, validation can be done only by checking the rwp list of node $d_x.acpn$. We assume that the number of access right validations will be much more than the number of access right assignments and, hence, we choose to propagate down the rwp. We use a new attribute $d_x.erwp$ to record the "expanded roles with permissions" for d_x, including the ones explicitly assigned to d_x ($d_x.rwp$) and the ones obtained after propagation. We specifically maintain $d_x.rwp$ and $d_x.erwp$ because we need to retain the direct permission assignments $d_x.rwp$ so that we can check revocation requests against it.

The downward propagation of rwp does not need to go through nodes that do not keep any rwp because those nodes will simply check the access rights via their $acpn$ nodes. To avoid a full subtree traversal for rwp propagation, we maintain a pointer set $d_z.accn$ for each node d_z in the resource tree. $d_z.accn$ contains a set of pointers which point to the closest descendants of d_z with non-empty rwp. In other words, if $d_y = d_z.accn$, then d_y has a non-empty rwp and there is no node between d_z and d_y has a non-empty rwp. Note that $accn$ is a pointer set and $acpn$ is a single pointer. When a new permission to role assignment (p_i, r_j, pc) is issued and p_i is a permission for resource d_z, then r_j will be propagated to all d_y, $d_y \in d_z.accn$, and the propagation continues down along all the $accn$ pointers. At the same time, once r_j is propagated to d_y, d_y adds r_j into $d_y.erwp$.

From the discussions above, we formally define the attributes $acpn$ and $accn$ as follows.

$$d_y = d_x.acpn \Rightarrow (d_y > d_x) \wedge (d_y.erwp \neq \emptyset) \wedge (d_x.erwp = \emptyset) \wedge \forall d_t, d_y > d_t > d_x, d_t.erwp = \emptyset.$$

$$d_y = d_z.accn \Rightarrow (d_z > d_y) \wedge (d_z.erwp \neq \emptyset) \wedge (d_y.erwp \neq \emptyset) \wedge \forall d_t, d_z > d_t > d_y, d_t.erwp = \emptyset.$$

Here we analyze the properties of $d_x.erwp$. Note that $d_x.rwp$ simply records the original permission assignments to d_x so there is no need for further elaboration. If the permissions associated with d_x have never been assigned explicitly to any role, i.e., $d_x.rwp = \emptyset$, then $d_x.erwp = \emptyset$. Also, if

$$d_x.rwp = d_y.rwp \wedge \forall d_t, d_y > d_t > d_x, d_t.rwp = \emptyset,$$

then we do not need to maintain $d_x.erwp$, i.e., we can set $d_x.erwp = \emptyset$ to reduce space overhead. Similarly, if after access rights propagation, we have

$d_x.erwp = d_y.erwp$
$\wedge \forall d_t, d_y > d_t > d_x, d_t.erwp = \emptyset$,
we can also set $d_x.erwp = \emptyset$ to reduce space overhead.

Besides rwp propagation down the resource tree, we also need to consider the access right propagation up the role hierarchy. Again, to facilitate efficient permission validation, we push the complexity to permission assignment. When the security officer assigns permission p_i for accessing resource d_x to role r_j, we also traverse the role hierarchy to include all ancestors of r_j in $d_x.rwp$. Note that if we choose to traverse the role hierarchy at permission validation time for an access to d_x, there may be multiple roles in $d_x.rwp$, and we may need to traverse multiple paths from the roles in $d_x.rwp$.

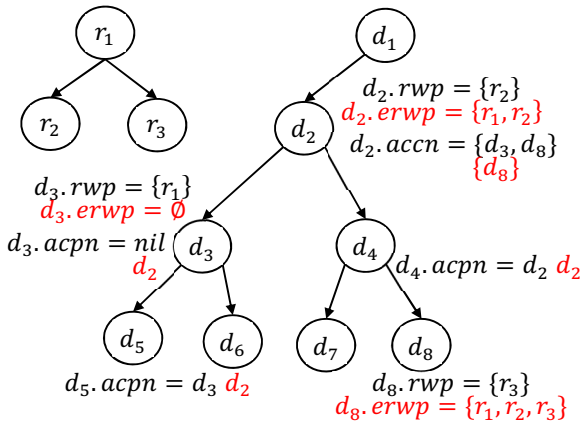

Figure 1. Example for the attributes.

We give an example (in Figure 1) to illustrate the attributes rwp, $acpn$, and $accn$. Assume that permissions $p_2 = (d_2, a)$, $p_3 = (d_3, a)$, and $p_8 = (d_8, a)$ are assigned to roles r_2, , r_1, and r_3, respectively. As can be seen, initially we have $d_2.rwp = \{r_2\}$, $d_3.rwp = \{r_1\}$, and $d_8.rwp = \{r_3\}$. After the assignments, d_2 becomes the "$acpn$" for d_3, d_4 and d_8 and d_3 becomes the "$acpn$" for d_5 and d_6. Also, d_2 needs to set its $accn$ to point to the closest descendants with nonempty rwp and we have $d_2.accn = \{d_3, d_8\}$. d_3 and d_8 do not have descendants with nonempty rwp and, hence, their $accn$ pointers are nil. The attribute values of some nodes up to this step are shown in black in the figure.

Now we consider the propagations after the example permission assignments. Following the $accn$ pointer, d_2 propagates its rwp down and we compute the expanded rwp for d_3 and d_8, where $d_3.erwp = \{r_1, r_2\}$ and $d_8.erwp = \{r_2, r_3\}$. Next we need to propagate up the role hierarchy and will get $d_2.erwp = \{r_1, r_2\}$ and $d_8.erwp = \{r_1, r_2, r_3\}$. By propagation along the role hierarchy $d_3.erwp = \{r_1, r_2\}$ is not changed. However, after we check $d_3.erwp$ against $d_3.acpn.erwp$ $(= d_2.erwp)$ and find that they are the same, we can set $d_3.erwp = \emptyset$ and $d_3.acpn = d_2$. Subsequently, we need to change the $acpn$ of those nodes originally pointed to d_3, i.e., $d_5.acpn = d_2$ and $d_6.acpn = d_2$. We also need to remove d_3 from $d_2.accn$ and now $d_2.accn = \{d_8\}$. The new values of the attributes of some nodes are shown in red in the figure.

4.2 Permission Assignment

The pseudo code for permission_assignment (d_y, r_i), which assigns permission for accessing resource d_y to role r_i, is given as follows. To make the code easier to understand, we have $d_z = d_y.acpn$ and d_x is used for nodes in the subtree of d_y.

```
permission_assignment (d_y, r_i):
   if r_i ∉ d_y.rwp then   // otherwise, do nothing
1    if d_y.rwp = ∅ then   // originally, d_y.acpn is d_z
       d_z ← d_y.acpn;
       add d_y to d_z.accn;   d_y.acpn ← nil;
2      foreach d_x in subtree(d_y)
         if d_x.rwp ≠ ∅ then
           remove d_x from d_z.accn, add d_x to d_y.accn;
           stop going further to d_x's subtree;
         else   // originally d_x.acpn = d_z
           d_x.acpn ← d_y;
         endif;
       endfor;
3    endif;
4    add r_i to d_y.rwp;
     newrwp ← {r_i};
5    foreach r_j, r_j > r_i do add r_j to newrwp; endfor;
     propagate_rwp (d_y, newrwp);
6    endif;
   endif;
```

If $r_i \in d_y.rwp$, then the permission assignment request is redundant and nothing needs to be done.

Note that rwp can be used to decide whether a node has its own "roles with permission" ($rwp \neq \emptyset$) or only has propagated rwp and uses $acpn$ to an ancestor to get its roles with permission ($rwp = \emptyset$). In the latter case, since now d_y becomes a node with its own rwp, the $acpn$ link of d_y needs to be nil and the $acpn$ links of d_y's descendants needs to point to d_y. Similarly, some original $accn$ links of d_z, if pointing to d_y's descendants, needs to become d_y's $accn$ and d_z should put d_y in its $accn$. These updates are done between code indices 1 and 3. The foreach loop at code index 2 is a subtree traversal which can be done recursively. A recursive can terminate when we reach a node, say d_x, with its own rwp, because the $accn$ and $acpn$ links in d_x's subtree nodes are all within d_x's subtree.

Besides updating $acpn$ and $accn$ pointers, we also need to update rwp and $erwp$ of some nodes (done by code between indices 4 and 6). rwp is for explicit assignments and, thus, only $d_y.rwp$ needs to be changed. For $erwp$, we first perform role hierarchy propagation to add r_i and its ancestors to $newrwp$. This is done by the code at index 5 and should be achieved by traversing the role hierarchy up from r_i to the root. The resulting $newrwp$ should be added to d_y as well as all descendants of d_y. For descendants of d_y without their

own wrp, no update is needed. The descendants of d_y with $wrp \neq \emptyset$ can be recursively traversed via the $accn$ links and it is done by function propagate_rwp, which is given as follows.

```
propagate_rwp (d, newrwp)
  d.erwp ← d.erwp ∪ newrwp;
  if d.erwp ≠ newrwp then
    foreach d_x ∈ d.accn do propagate_rwp (d_x, newrwp);
    endfor;
  endif;
```

Consider an example for the algorithm. In Figure 1, consider a new permission assignment (d_4, r_3). We perform permission propagation on the role hierarchy so $newrpw$ becomes $\{r_1, r_3\}$. Originally, $d_4.rwp$ is empty. With the new $d_4.rwp$, "$d_4.acpn$" should point to d_2 and "$d_2.accn$" should include d_4. d_7, a descendant of d_4, should now move its "$acpn$" to point to d_4. The "$accn$" link of d_2 to d_8 should be removed and moved to $d_4.accn$. Finally, $newrpw$ = $\{r_1, r_3\}$ should be added to d_4 and propagated to d_8.

5 Experimental Study

We implemented a simulation system to compare the performance of RBAC and RRBAC. A role hierarchy is generated for both cases. For resources, we first generate a resource tree for RRBAC. We then copy the resources in the resource tree to a hash table for fast references. In RBAC, each entry d_x in the hash table maintains a field $d_x.rwp$ to keep track of all the roles with permission to d_x. In RRBAC, each entry in the hash table maintains a pointer to the resource tree and the data structure of the resource tree is as discussed in Section 4.

We generate access requests to access the resources in the resource tree, including those for permission assignments and for permission validations. In the experimental study, we only consider a single operation a for all the resources.

We consider requests that assign permission(s) to a role for a single or a group of resources. In RBAC, when assigning a single-resource permission for (d_x, a) to r_i, where d_x is a leave node in the resource tree, we add r_i to $d_x.rwp$. To make the comparison fair, we also perform access rights propagation along the role hierarchy by traversing the role hierarchy to expand $d_x.rwp$. When assigning permissions to a group of resources in a subtree rooted at d_j to role r_i, we generate multiple permission assignment requests that assign permissions for (d_x, a) to r_i, for all d_x, $d_x \in subtree(d_j)$. Then, individual permission assignment requests are handled as discussed above. In RRBAC, we can directly issue a group permission assignment request. Thus, we simply implement the permission assignment algorithm discussed in Section 4.

When a user with role r_i accesses resource d_j with operation a, a resource validation request (r_i, d_j, a) will be generated. In RBAC, we simply check $d_j.rwp$ in the hash table to determine whether to grant the access. In RRBAC, we access d_j in the resource tree from the hash table and then simply follow the permission validation algorithm.

Generation of the hierarchies. We design a configurable tree generator for the generation of the resource and role hierarchies. The parameters for the generator are the expected height of the tree, L, and the average degree for a node, deg. Starting from root, for a node t, we first decide whether to expand it following a Poisson distribution with $adj_L \times (L + 1 - level(t))$ as the shape parameter, where $level(t)$ is the level (from root) of node t (with root being level 1). If a node is to be expanded, then, the degree for t is also generated following the Poisson distribution with shape parameter $adj_D \times deg$. In both cases, adj_X is used for shape sharpness.

Request generation. We consider permission assignment (PA) and permission validation (PV) requests. A PA request assigns the rights for accessing a "resource" to a "role". In a PV request, we validate a "role" for its access to a "resource". Thus, we need to select a role and a resource for each request.

In PA, it is more likely to assign permissions to roles for resources at the middle and lower levels. Going toward the higher level of the resource tree, it becomes less likely that the entire subtree has the same access rights. Thus, we select a resource from the tree by first deciding which level to select the node from using the Poisson distribution with shape parameter $adj_{SL} \times 0.6\, level(T)$, where T is the tree and $level(T)$ is the actual level of T. We use $0.6\, level(T)$ to favor the selections at the lower and middle levels. Here, adj_{SL} is the shape adjustment parameter for level selection.

We give an equal probability to nodes at the given level of the tree during node selection. We use a tree traversal algorithm to select the specific tree node at level l. For each tree node t, we maintain a counter $t.count$, which is initialized to the number of nodes in the subtree of t. The algorithm traverses the resource tree from the root. At a node t with child nodes $t.child_i, 1 \leq i \leq n$, we randomly select which child node to visit next based on a uniform distribution and the probability to visit $t.child_i$ is proportional to $t.child_i.count$. When we reach level l, the node being visited, say tl, is selected. Once a node has been selected for permission assignment, it is unlikely that it will have permission assignment again. Thus, after node selection, we reduce $t.count$ by 1, for all t, t can be tl and tl's ancestors.

For role selection in PA, access permissions for less critical resources are assigned to some lower level roles and the rights can be propagated to the higher level roles. More critical resources are assigned directly to higher level roles. Thus, we consider equal probability for all roles to be the parameter in a PA request.

In PV, accesses to resources are more likely to be at the lower levels. Thus, we use the same level selection method but with a modified shape parameter $adj_{SL} \times 0.8\, level(T)$. Generally, resource accesses follow the zipf law. Thus, we use zipf distribution for selecting a resource at a given level. The resources at the same level are randomly ranked. We first compute the total number of nodes in each level of the tree,

denoted as N_l for level l. Then, we traverse the tree and select a number in $[1,N_l]$ without duplication for each node t visited, where t is at level l. After rank assignment for t, we compute the access probability for t in level l using zipf distribution. Let $t.prob$ denote the probability of a node t at level l being accessed. Correspondingly, $t.count$ is the sum of $td.prob$ for all td, $td \in subtree(t)$. During PV request generation, we use the same tree traversal algorithm to select the specific resource at the given level.

For PV, we consider that lower level roles are more likely to access more resources. Thus, we use the resource selection algorithm for PA for role selection in PV. For level selection, we use $adj_{SL} \times 0.8\ level(T)$ to favor lower levels. Then, the same tree traversal algorithm with equal probability for all nodes is used to select the role for the PV request.

5.1 Experimental Results

We build a resource tree and role tree. The resource tree has approximately 10,000,000 nodes, with expected level of 10 and average degree 200. The role hierarchy has approximately 2000 nodes with expected tree level of 10 and average degree of 5. For both cases, $adj_L = adj_D = 1$.

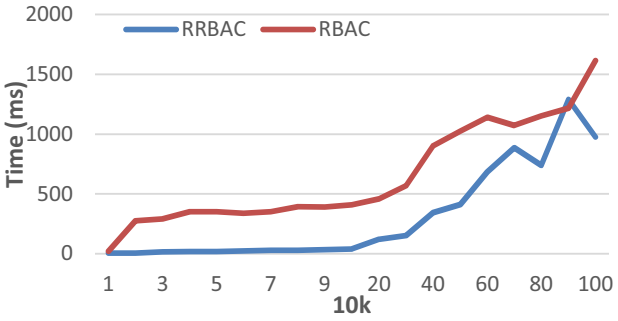

Figure 2. RRBAC vs RBAC for Permission Assignments

Once resource tree and role tree are generated, we apply RRBAC and RBAC mechanism for request generation of permission assignment (PA) and permission validation (PV).

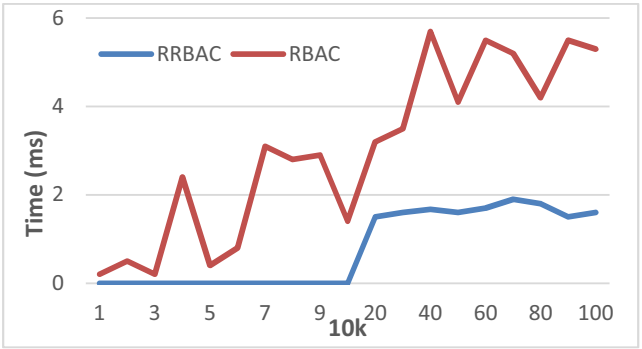

Figure 3. RRBAC vs RBAC for Permission Validation

For PA and PV, we generate 10k to 100k requests and compare performance between RRBAC and RBAC. Figure 2 clearly indicates that RRBAC has better performance for permission assignment than RBAC.

Similarly, from analysis, RRBAC has much better performance than RBAC for permission validation. This is because in RRBAC, the roles with permissions are maintained with the resource and, hence, validation decision making procedures are very efficient.

6 Conclusion

In this paper, we have extended RBAC and developed an RRBAC model to better model access control policies and to improve the convenience and efficiency for permission assignment and management in RBAC. Based on the model, we have designed a resource tree based solution to maintain the assigned permissions and developed the corresponding algorithms for permission assignment, validation, and revocation. Preliminary experimental results show that RRBAC can outperform RBAC in permission assignment.

7 Acknowledgement

This research is supported by the NSF IUCRC on Net-Centric Systems and its industrial membership, the NSF Program under Award No. IIP-1128270.

8 References

[1] J. Hamilton, "Internet Scale Service Efficiency (invited talk)," in *LADIS*, Sep. 2008.

[2] S. Zhang, I.-L. Yen and F. Bastani, "Toward semantic enhancement of monitoring data repository," in *International Conference on Semantic Computing*, Feb. 2016.

[3] N. Solanki, W. Zhu, I.-L. Yen, F. Bastani and E. Rezvani, "Multi-tenant access and information flow control for SaaS," in *ICWS*, 2016.

[4] W. She, W. Zhu, I.-L. Yen, F. Bastani and B. Thuraisingham, "Role-based integrated access control and data provenance for SOA based net-centric systems," *IEEE TSC,* vol. 9, no. 6, pp. 940-953, Nov.-Dec. 2016.

[5] C. Hu, D. Ferraiolo, D. Kuhn, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," in *NIST Special Publication 800-162*, 2014.

[6] X. Jin, R. Sandhu and R. Krishnan, "RABAC: role-centric attribute-based access control," in *Computer Network Security*, Springer, 2012.

[7] D. E. Bell and L. J. LaPadula, Secure Computer Systems: Mathematical Foundations, MITRE Corporation, 1973.

[8] P. Bonatti, M. Sapino and V. Subrahmanian, "Merging heterogeneous security orderings," in *European Symposium on Research in Computer Security*, 1996.

[9] M. Shehab, E. Bertino and A. Ghafoor, "Secure collaboration in mediator-free environments," in *ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2005.