INVITED PAPER Special Section on Autonomous Decentralized Systems Technologies and Applications for Next-Generation Social Infrastructure

Autonomous Decentralized Authorization and Authentication Management for Hierarchical Multi-Tenancy*

Qiong ZUO††, Meiyi XIE†, and Wei-Tek TSAI‡‡, Nonmembers

SUMMARY Hierarchical multi-tenancy, which enables tenants to be divided into subtenants, is a flexible and scalable architecture for representing subsets of users and application resources in the real world. However, the resource isolation and sharing relations for tenants with hierarchies are more complicated than those between tenants in the flat Multi-Tenancy Architecture. In this paper, a hierarchical tenant-based access control model based on Administrative Role-Based Access Control in Software-as-a-Service is proposed. Autonomous Areas and AA-tree are used to describe the autonomy and hierarchy of tenants, including their isolation and sharing relationships. AA is also used as an autonomous unit to create and deploy the access permissions for tenants. Autonomous decentralized authorization and authentication schemes for hierarchical multi-tenancy are given out to help different level tenants to customize efficient authority and authorization in large-scale SaaS systems.

key words: Multi-Tenancy Architecture, hierarchical multi-tenancy, Role-based Access Control Model, tenant-based access control model

1. Introduction

The Multi-Tenancy Architecture (MTA) is often used in Software-as-a-Service (SaaS) where multiple tenants can use the same code base stored in the SaaS to develop applications, with assurance of each tenant’s privacy [1]. A tenant can be a single user or an organizational entity. Today in many hierarchical organizations, a corporation may have multiple subsidiary companies, and these different subsidiaries may share significant requirements. Traditional multi-tenancy models are mostly flat: tenants have no relations to one another. Each subsidiary is built either as an independent tenant which makes customized resource sharing and re-customization among relative tenants difficult to implement, or as a user group of a tenant where the real world hierarchical relations are implied by complicated role hierarchy and sharing constraints. Recently more and more researches [1]–[4] are taking an interest in hierarchical multi-tenancy, which enables tenants to be divided into subtenants. It is a flexible and scalable architecture for representing sub-

<table>
<thead>
<tr>
<th>Access Control Requirements of Hierarchical Multi-tenancy</th>
<th>MTA with Subtenant-as-Tenant</th>
<th>MTA with Subtenant-As-User-Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tenant-level isolation</td>
<td>⟨⟩</td>
<td>⟨⟩ Tenant isolation</td>
</tr>
<tr>
<td>Isolation between parent-child tenants</td>
<td>⟨⟩ Using tenant border with no explicit parent child relationship</td>
<td>⟨⟩ Subtenant isolation</td>
</tr>
<tr>
<td>Autonomous management of each tenant</td>
<td>⟨⟩ Only system admin can create / revoke tenants and subtenants</td>
<td>⟨⟩ Using role hierarchy with constraints, but parent-to-child access is possible</td>
</tr>
<tr>
<td>Isolation between subtenants</td>
<td>⟨⟩ Using tenant border with no explicit sibling relationship</td>
<td>⟨⟩ A tenant create / drop subtenants as user groups by itself</td>
</tr>
<tr>
<td>Private resources isolation</td>
<td>⟨⟩ Using role exclusive assignment for each tenant</td>
<td>⟨⟩ Using role exclusive assignment between user groups</td>
</tr>
<tr>
<td>Private resources sharing between relative tenants</td>
<td>⟨⟩ Not allowed</td>
<td>⟨⟩ Possible data leakage if privacy is shared by its owner</td>
</tr>
<tr>
<td>Resource sharing between parent-child tenants</td>
<td>+ Trust / Federal</td>
<td>⟨⟩ Using implicit sharing with role hierarchy</td>
</tr>
<tr>
<td>Resource sharing between sibling tenants</td>
<td>+ Trust / Federal</td>
<td>⟨⟩ Using role sharing assignment</td>
</tr>
<tr>
<td>Federal sharing among tenants</td>
<td>+ Trust / Federal</td>
<td>⟨⟩ Only supported by MTA with additional Trust or Federal access control functions</td>
</tr>
</tbody>
</table>

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers
• **Autonomous Tenants**: Each tenant acts like an autonomous agent to manage its own subtenants, users, and resources. A system administrator cannot interfere with tenants’ internal affairs. Also, a tenant cannot interfere with its subtenants’ internal affairs though a subtenant may inherit resources from its parent-tenant. Furthermore, due to privacy isolation, role privilege inheritance no longer exists in the system scope. What access privilege can/cannot be inherited and what resources can/cannot be cross-level controlled are different from those in traditional MTA systems, and need to be redefined.

• **Sharing relationships among tenants**: The sharing may be between two sibling tenants or parent-child tenants. Private resources can be granted to other tenants with their owners’ permissions. “Sharing directions” may be from a parent-tenant to its subtenants, or from a subtenant to its parent, or from a tenant to its sibling.

To address the above issues, a flexible access control model for different-level autonomous tenants to share existing resources, customize their own application and keep their secrets should be supported.

The contributions of this paper are: 1) A tenant-based access control model (TMS-ARBAC) based on ARBAC for hierarchical multi-tenancy is proposed; 2) Autonomous decentralized authorization and authentication schemes are given out to help different-level tenants to customize efficient authority and authorization in large SaaS systems.

This paper is organized as follows: Sect. 2 reviews related MTA infrastructure and security mechanisms; Sect. 3 defines the TMS-ARBAC model; Sect. 4 presents the authorization and authentication schemes of hierarchical multi-tenancy; Sect. 5 analyses our model and Sect. 6 concludes this paper.

2. Related Work

2.1 MTA in SaaS

Currently there are lots of ways to implement MTA [1]. But [7] points out that “multi-tenancy” lacks of a clear definition. Most existing MTA technologies, such as data center, virtualization and middleware sharing, or PaaS-based MTA, are not real MTA, but multi-instance or multiple application deployment solutions lacking in resources sharing and in efficiency.

In MTA, most existing systems treat tenants as individual entities, fully isolated from other tenants. But SaaS promotes sharing. The relationships between tenants need to be further discussed, but only a few works related are published. [2] introduces a hierarchical tenant-tree model and its physical storage. But it only focuses on the efficiency and scalability for the authorization data access. Keystone provides identity API of hierarchical multi-tenancy for OpenStack, using domains and projects to build hierarchy of user groups and resources subsets [3]. A hierarchical multi-tenant pattern is introduced to represent the hierarchical relationships among tenants [4]. But none of them further discusses the resource sharing or application customization between different-level tenants.

From the reusing of shared resources and easy customization view, STA [1] is proposed to allow tenants to offer services for subtenant developers to customize their applications. Various STA models are defined with different customization models.

2.2 Access Control in MTA

Security is an important topic in SaaS as all tenants share the same computing resources. This paper focuses on authentication and authorization of hierarchical multi-tenancy.

To apply RBAC, one needs to identify *subjects, objects, and permissions*. In a hierarchical multi-tenancy SaaS system, a *subject* can be 1) a tenant; 2) a subtenant; 3) a user of either tenant applications or subtenant applications; An *object* can be any application or data services; and the *permissions* are between subjects and objects within the system. The following approaches have been used to apply RBAC to SaaS systems.

• Database schema and RBAC model

  Traditional IT manufacturers [8], [9] develop data-centric PaaS-based SaaS. Multiple tenants are strictly isolated by database schema and RBAC model, sharing a data center. Security strategies are mainly focused on data.

• RBAC-alike model for MTA isolation in SaaS

  After identifying the problems of applying RBAC to SaaS systems, Li [10], [11] propose a H-RBAC model, in which access control is divided into tenant-level and system-level with role delegation and time-constraints.

  T-Arbac [12] is proposed by adding “Tenant” into ARBAC model. System-level and tenant-level administrations are separated. Isolated sub-resource pool is assigned to each tenant.

  For organizations with subsidiaries, N-RBAC [13] uses hierarchical namespace structure to arrange users and roles for autonomous distributed role administration.

  MHARBAC [14] is presented for multi-hierarchies decentralized administration applications, using role-tree to support top-down authorization. Administrative scope is subdivided into user, role and permission scope.

  All these access control models focus on the separation of multiple tenants, adding access borders and constraints for resources isolation. None of them mentions the security of cross-tenant or inner-tenant collaboration. Collaboration between tenants and/or subtenants offers users to create, distribute, and access sensitive resources, causing the possibility of externalizing data.

• RBAC-alike model for multi-tenant sharing

  Extensions of RBAC model have been proposed for collaborative authorization in clouds. Except those with centralized authority that are not suitable for the cloud, there are three ways to build collaborative authorization [15]: 1) delegation in RBAC on basis of individual user decisions; 2)
federated identity and authorization services; 3) trust management into access control mechanisms.

For cross-tenant collaboration on outsourcing resources, a MT-RBAC model family [16] is proposed to provide fine-grained authorization in collaborative cloud by building trust relations among tenants.

In SaaS ecosystem, service federation is established to allow customers to use the subscribed services across multiple organizations [17].

These models are designed for cross-tenant sharing, but they are too complicated for close parent-child tenants or sibling tenants sharing.

In Salesforce [18], [19], role hierarchy, groups and territories with sharing rules are used to maintain the security of tenants, groups and users. Parent-child sharing are defined by both implicit sharing rules and explicitly grant access privileges. Delegated authentication and federated authentication are provided for cross-tenant sharing. Subtenants are treated as groups under the supervision of their parent tenants. Simple changes to groups can sometimes involve the substantial re-calculation of users’ access right.

To take advantage of the strengths of ARBAC97 and support autonomous decentralized different-level tenants to keep secret, do customization and share their resources, this research proposes the autonomous tenant access control model for hierarchical multi-tenancy in SaaS.

3. Autonomous Tenant Access Control Model for Hierarchical Multi-Tenancy in SaaS

This section introduces a new access control model for autonomous decentralized tenants called TMS-ARBAC, adding five new entity components: Autonomous Area (AA), Chief Administrative Roles (CAR), Chief Administrative Permissions (CAP), Federals (F) and Outer-roles (O_R) on ARBAC, for resource isolation and sharing of hierarchical tenants in SaaS. AA-tree is used to represent the tenants’ hierarchies.

3.1 Autonomous Area Tree Management

System administrators, tenants and their subtenants, roles and users all live in one SaaS platform. Each entity has its own work domain. Tenants and subtenants are autonomous entities with boundaries. Each tenant establishes its policies. It owns the resources rented from the system, and customizes its own application. It can create subtenants (ST) and users, authorizing them the resources the tenant owned or controlled. Every subtenant must have a parent-tenant, but a parent-tenant may have many subtenants. A subtenant gets resources from its parent-tenant, can do customization and create its own users. In multi-level STA [1], a subtenant can create its own subtenants.

Once a tenant is created, the authorization and authentication inside the tenant scope cannot be interfered by system administrators or other tenants. For tenant isolation, roles cannot be granted to users of other tenants. So, both roles and users should be narrowed down to a tenant’s territory.

To meet the above requirements, “Autonomous Area” (AA) is proposed to describe the security control area of one tenant. An AA is a named scope to restrict a tenant’s RBAC security administration range. Each AA has a unique name. Every entity (such as a role, a user, or a set of permissions) of a tenant is unique inside the tenant’s AA with a unique entity name and its AA’s name as a prefix. Access control policies for one tenant can only be assigned to the entities inside its AA.

As a tenant may have its subtenants, an AA may have its sub-AAs. All the autonomous areas in a hierarchical multi-tenancy system form an AA-tree, representing the tenants’ hierarchies, as shown in Fig. 1. However, the implicit sharing from the child-node to its parent-node is unsupported. The objects inside an AA are invisible from any other AAs, unless there is an explicit authorization from an AA-federal (see Definition 2).

3.2 TMS-ARBAC Model Definition

The entity components and their relationships in an AA are shown in Fig. 2, in which, Users (U), Roles (R), Administrative roles (AR), Permissions (P), Administrative Permissions (AP), and Sessions (S), have the same definitions as in ARBAC, but are narrowed down to a tenant’s territory. A user is distinguished by the roles assigned, to be an administrator, a tenant manger or a regular application user in its work zone. Chief administrative roles (CAR) and Chief Administrative Permissions (CAP) are added for the administrative work management of an autonomous area. Federal (F) is created for non-parent-child tenant resource sharing through Outer Roles (O_R). The TMS-ARBAC model is composed of a set of AAs. Formal definitions of TMS-ARBAC are presented as follows.

**Definition 1:** An autonomous area has the following components:

U, R, AR, P, AP, UA, AUA, PA, APA, S, and RH ⊆ R × R, ARH ⊆ AR × AR, user: S → U, have the same definitions as in ARBAC97 model with restriction in current autonomous area;
CAR, the one and ONLY one chief administrative role created in current autonomous area;

CAP, a set of chief administrative permissions in current autonomous area;

O,R, outer roles shared by other autonomous areas to current autonomous area in the same federal;

PA2T ⊆ P × AA, many-to-many permission to autonomous area assignment relation;

CAPA ⊆ (AP ∪ CAP) × CAR, many-to-one permission to chief administrative role assignment relation;

FUA ⊆ (U × O,R), many-to-many user to outer-role (O,R) assignment relation;

CAUA ⊆ U × CAR, many-to-one user to chief administrative role assignment relation;

roles: S → 2°((R ∪ O,R ∪ AR ∪ CAR), maps each session sₖ to a set of roles roles(sₖ) ⊆ {r | (∃r′ ≥ r) [(user(sₖ), r′) ∈ UA ∪ AUA ∪ FUA ∪ CAUA]} (which can change with time) and session sₖ has the permissions ⋃r∈roles(sₖ) {p | (r ∈ (R ∪ AR) ∧ (∃r′ ≥ r) [(p, r′) ∈ PA ∪ APA]) ∨ (r ∈ CAR ∪ (p, r) ∈ CAPA) ∨ (r ∈ O,R ∧ (p, r) ∈ area(r,PA)].

There is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Definition 2: The TMS-ARBAC model is denoted as a tuple (AA, TR, FR):

- AA = {a₁, a₂, ..., aₙ}, where a_i is an autonomous area, n is the cardinality of AA;
- TR denotes the sharing set between parent-child tenants. If n ≤ 1, TR is an empty set; else TR is a set of relationships denoted as {a_i, a_j, p, where a_i is the parent autonomous area of a_j, p is the permission set that a_ij inherited from a_i, or a_ij shares with its parent a_i;
- FR denotes a set of federals. A federal f is defined as a five-tuple: f = (F_{id}, C_a, F_m, F_o, F_c), representing the resource sharing relationships between non-parent-child tenants. In which,
  - F_{id} is the unique ID of the federal f.
  - C_a is an AA, the chairman of f.
  - F_m = {a₁, a₂, ..., a_k}, (k ≥ 1), is a subset of AA joined in f.
  - F_o = {(a_i, a_j, a_k, r)|a_i ∈ F_m ∧ a_k ∈ F_m ∧ i ≠ k ∧ a_i, r ∈ a_i.R}, which means a_i shares a role a_i,r with a_k in f.
  - F_c is a group of constraints to f and to the assignments from O,R to users.

According to definition 2, there may be two types of relationship between two autonomous areas: parent-child relationship and AA-federal relationship. The parent-child relationships always exist, constituting an AA-tree.

The federal relationship allows several AAs to unite a federal, and share roles between each other within the federal. The O,R of an AA a_i is a set of roles given by other autonomous areas in the federal f. f.F_{id} is added to O,R to distinguish outer roles assigned from the same AA of different federals.

\[ a_i.a_R = \bigcup_{f \in FR} \{a_i,r | F_{id}(a_i.r, f.F_{id}) (\exists(a_i, a_j, r) \in f.F_{id} [i ≠ j])\} \]

To avoid implicit privilege inheritance in role hierarchy, the partial order on outer roles is NOT allowed. To avoid privacy leakage among different federals, NO user can simultaneously belongs to different federals in one session.

The difference between TR and FR sharing is that: TR takes the granularity of permission, so that the grantee AA can flexibly combine the permissions into different roles to meet the access control requirement of tenants; FR takes the granularity of role, a local user can only access outer resources through specified roles to achieve more reliable security.

4. Autonomous Decentralized Authorization and Authentication Schemes on Autonomous Areas

4.1 Autonomous Access Control Unit

In a multi-tenancy SaaS environment, each tenant has its own users, administrators and/or subtenants, and owns elastic on-demand resources. As the number of tenants, subtenants, users and resources grows, the number of authorization roles and rules increases rapidly, a scalable architecture for authorization and authentication is needed. We use the “autonomous area” (AA) as the Autonomous Access Control Unit. Every unit has its unique Chief Security Officer (CSO), who has the highest authority in the AA, no matter this AA belongs to a tenant or a subtenant. So there is not one global centralized CSO but many CSOs distributed throughout the whole system. The administrative
work includes creating users, regular roles (RR) and administrative roles (AR), assigning permissions and creating sub-AAs, which is continued dynamically in the life cycle of the SaaS system.

In TMS-RBAC, we still use the RBAC mechanism to authorize CSOs. Besides the original AR defined in ARBAC97, we introduce Chief Administrative Role (CAR), with highest authority in an AA. All the security administrative work of CSOs can be performed through CAR. Each AA has one CAR, in charge of developing and maintaining RBAC policies in the AA.

The difference between CAR and AR is that the former can be used to manage not only RR but also AR. This difference is achieved through a special set of permissions called Chief Administrative Permissions (CAP). A CAP includes permissions to add/delete users or AR, create/modify/revoke AA, create/drop federals, and share/revoke outer-roles.

When an autonomous area is created, its CAR is created and granted the entire CAP automatically. Meanwhile, a default CSO user is created and assigned the CAR automatically. Then the CSO can be in charge of this AA. No user could delete or modify the default CSO or the CAR. They die with the deletion of AA.

4.2 Autonomous Authorization Work on an AA

In a hierarchical SaaS platform, the resources held by an AA may come from different sources and change dynamically. According to the source type, the resources in an AA can be classified into 4 categories: resources with permissions inherited from the parent-AA, resources with permissions shared by the parent-AA or the child-AA, resources with outer-roles assigned from federals and self-created resources (including customized services and private data), as shown in Fig. 3. The shading indicates those private resources are forbidden to be re-granted to others. Such resources are either secrets of current AA or private permissions shared by other AAs to current AA.

Thus the resources permissions of an AA can be divided into 3 subsets according to their security status, they are: PUBLIC (denotes public resource set that can be granted to others), PRIVATE (denotes private resource set that can only be granted to certain users of related AAs) and SECRET (denotes private resource set that can only be used in current AA).

The permissions to be granted in an AA are restricted to a subset of the AA's resource permission set. The authorization work scheme on an AA can be divided into 4 parts according to its work zone, as show in Fig. 4. Their definitions are given as follows.

1) SubAAAuthorizationWork(), which grants, updates or revokes inherited permissions to a sub-AA of current AA, including 3 kinds of services:
   - InitSubAAPermissions(), when a sub-AA is created. It creates CAP, CAR, CSO, and initializes inherited permissions for the sub-AA, which is implemented automatically by the SaaS platform;
   - ModifySubAAPermissions(), when new permissions are added or given permissions are revoked dynamically by its parent-AA;
   - RevokeSubAAPermissions(), when a sub-AA is revoked. A tenant may determine to revoke permissions from its subtenant, and then all related permission-to-role assignments should be revoked in turn. Furthermore, if the subtenant has re-granted the permissions to others, more complicated process is required to ensure an exhaustive revocation.

2) ParentAAAuthorizationWork(), which grants or revokes private resources dynamically from current AA to its parent-AA.

3) FederalAAAuthorizationWork(), which is responsible for federal sharing authorization, including 3 subsets of services:
   - FederalCreation() and FederalDrop();
   - AAJoinFederal() and AAQuitFederal();
   - OuterRoleCreation(), OuterRoleShare() and OuterRoleRovoke().

4) LocalAAAuthorizationWork(), which works as traditional ARBAC97 authorization jobs restricted in an AA, with only one exception: outer role specified to certain users with constraints.
In view of security status, roles and users are also divided into different access levels (TOP, MIDDLE, NORMAL) for resource access.

Sibling-AAs with similar organization structure or duty separation may have similar authorization requirement. Authorization workflow and template of demo sub-AAs can be recorded and recommended for sibling sub-AAs’ initialization and authorization. Since each tenant is autonomous, authority work on different AAs can be executed concurrently. Authority servers and separate databases are allocated to each subtenant for efficiency.

4.3 Decentralized Authentication Work on an AA

The authentication work scheme on an AA can be divided into two parts: user authentication and resource authentication.

(1) Decentralized user authentication

When a user login the platform, user authentication is triggered. In the SaaS system, there may be thousands or millions of tenants; each tenant may have multiple users and even subtenants; and a subtenant may also have users and even sub-subtenants. And a user may be a CSO, an administrator, a tenant manager or an application user. Thousands of users may stay in the system synchronously, applying or using different resources, causing multiple read and write operations happened synchronously on the access control tables. The user authentication work is very busy.

We use the ID of AA (named as AA_ID) to narrow down the search within a single AA scope. Read or write operations on different AAs can be simultaneously performed. As shown in Fig. 5, user_Table and role_Table are both partially indexed by AA_ID. If such tables continue to scale up, AA_ID can also be used to split the huge tables into smaller tables distributed to different database servers. An AA-catalog of AA_IDs to their distributions is recorded and synchronized in master servers. An AA-tree-index is used for AA-catalog lookup.

(2) Resource authentication tracing

In a hierarchical multi-tenancy system, a resource creator is not the only granter. The current resource holder AA with the “grant” option can grant the resource to other AAs, including its parent-AA, its sub-AA, and/or its federal members. To do resource authentication, each user and each granter of such resource should be audited.

Because privacy are forbidden to be spread one by one, only public resources permissions can be transmitted through different AAs. It seems this transmission will not cause security problems.

But if the resource is a composite one, every pieces of such resource should be traced. During the tracing chain, constraints on the grant permissions should also be noticed. If some constraint is broken, or the deadline comes, revocation will happen in such chain, recursive revocation may be triggered.

Resource authentication happens when a resource privilege is to be assigned to other, or the resource status is changing, or the resource is deleted from the system. Fails of the resource authentication may cause re-calculation of the access rights of all its users.

5. Model Analysis

5.1 Security Analysis

Considering an autonomous area as a distributed autonomous organization in the cloud, the sharing security among autonomous areas turns to be a secure interoperation problem in a hierarchical multi-tenancy SaaS environment. There are two kinds of security interoperations in our system: TR and FR.

Since TR only happens between the parent-child AAs with permissions, and the grant operations are implemented by system automatically with no secret re-grant constraints. We assert that TR will cause no security interoperation problem.

FR is built on federal member trust. The outer role sharing and user-role assignments are executed separately in their respective AAs. So no security interoperation happens in one federal. However, users belonging to two federals can steal resources from one federal to another. We use separation of duty and federal constraints to reduce such violation.

5.2 Property Analysis

The typical characteristic of TMS-ARBAC is to reconcile the demand of tenant isolation and resource sharing in hierarchical multi-tenancy SaaS.

1) The isolation and autonomy of every tenant.
- Each tenant has its isolated work area AA. Resources, roles, users and permissions are all restricted in such AA. So, an autonomous area builds a strict isolation wall from other tenants.
- System-level and tenant-level administrations are clearly separated. The SaaS system is also an AA.
Each AA has its CAR and CSO, dominating its own job functions.

- Each tenant establishes its access control policies with application requirements in its AA. Only the resource allocation and revocation obey the AA hierarchies.

2) Resource sharing for tenants

- Federal sharing: subscribed resources are shared between unrelated tenants or sibling tenants in one federal. Security is ensured by federal trust and outer-role constraints.
- Inheritance sharing: system resources are easily inherited from parent to child tenants.
- Privacy sharing between tenants: private resources are privately shared between related tenants without data leakage, which is a kind of private resource reuse.

5.3 Scalability Analysis

In our model, thousands of subtenants of the same parent tenant can co-exist. Each subtenant may also have multiple sub-subtenants and millions of users.

Compared with Salesforce who treats subtenants as groups, the sub-AA initialization authorization work seems more complicated. But after initialization of the sub-AA territory, local authorization services for roles and users assignment and management on different sub-AAs can be concurrently executed. This process is much faster than that in Salesforce, since all groups are under supervision of their tenant, and all users under groups also wait for the authorization of the tenant.

As mentioned in 4.3, the two-step-user-authentication (including the AA-tree-index catalog search and the username-in-AA search) works efficiently even when the number of users increases. But in Salesforce, groups cannot be used to narrow down the username searching scope. When the number of users increases, user authentication will be less effective.

For large-scale resource authentication, if a resource is only used in one AA, the process will be effective. But if not, cross-tenant resource authentication will reduce efficiency. That is the cost of resource reusing.

6. Conclusions

This paper analyzed the access control requirements of hierarchical multi-tenancy on Software-as-a-Service platform, emphasizing the privacy and sharing issues raised between autonomous decentralized tenants, and then gave the definition of a new TMS-ARBAC model. Autonomous Areas are used to isolate each autonomous tenant. An AA-tree with parent-child relationship and federal relationship is used to describe the tenants’ hierarchy and various sharing relationships. Authorization and authentication schemes on each autonomous area were proposed. Decentralizing authorization by autonomous area helps to improve efficiency in large hierarchical multi-tenancy SaaS. Further research work will focus on the composite services authorization and authentication work in hierarchical multi-tenancy PaaS.

References

Qiong Zuo received her M.S. and Ph.D. degrees in Computer Science from Huazhong University of Science and Technology in 1999 and 2010, respectively. She is now a lecturer of the School of Computer Science and Technology, Huazhong University of Science and Technology. Her research interests are in big data management and cloud computing.

Meiyi Xie received her M.S. and Ph.D. degrees in Computer Science from Huazhong University of Science and Technology in 1999 and 2010, respectively. She is a lecturer of the School of Computer Science and Technology, Huazhong University of Science and Technology. Her research interests are in information security.

Wei-Tek Tsai received his Ph.D. in Computer Science from University of California at Berkeley in 1986. He is now Professor in the School of Computer Science and Engineering at Beihang University, Beijing, China, and Emeritus Professor at Arizona State University.