Security-Enhanced OSGi Service Environments

Gail-Joon Ahn, *Senior Member, IEEE*, Hongxin Hu, and Jing Jin

***Abstract*—Today's home and local-area network environments consist of various types of personal equipments, network devices, and corresponding services. Since such prevalent home network environments frequently deal with private and sensitive information, it is crucial to legitimately provide access control for protecting such emerging environments. As a result, the open services gateway initiative (OSGi) attempted to address this critical issue. However, the current OSGi authorization mechanism is not rigorous enough to fulfill security requirements involved in dynamic OSGi environments. In this paper, we provide a systematic way to adopt a role-based access control (RBAC) approach in OSGi environments. We demonstrate how our authorization framework can achieve important RBAC features and enhance existing primitive access control modules in OSGi service environments. Also, we describe a proof-of-concept prototype of the proposed framework to discuss the feasibility of our approach using an open source implementation of OSGi framework known as Knopflerfish.**

***Index Terms*—Authorization, OSGi service, role-based management, security.**

I. Introduction

As we find ourselves accustomed to the pervasiveness and prevalent availability of network services, the proliferation of inexpensive digital equipments and the Internet have expedited such a rapid growth of availability to a scale scarcely imagined a few years ago. However, with the continuous connectivity to the Internet, home devices are exposed to various attacks and threats. The traditional security technologies cannot satisfy security requirements derived from a digital home environment because of the diversity of wired and wireless network, middleware modules, and restricted system resources of home appliances. Also, the users in a digital home environment mostly lack knowledge on security and network management practices [10], [13]. Just as technological innovation has enabled tremendous availability for home network environments, it is crucial to legitimately provide access control for protecting devices and digital resources in such environments. In the networked systems, access control can prescribe not only who or what process has access to a specific system resource, but also the type of access that should be granted.

Manuscript received August 24, 2008; revised January 7, 2009. First published May 15, 2009; Current version published August 19, 2009. This work was supported in part by the funds provided by the National Science Foundation under Grant NSF-IIS-0242393 and the Department of Energy Early Career Principal Investigator Award under Grant DE-FG02-03ER25565. This paper was recommended by Associate Editor N. Wu.

G.-J. Ahn and H. Hu are with the Department of Computer Science and Engineering, Ira A. Fulton School of Engineering, Arizona State University, Tempe, AZ 85281-8809 USA (e-mail: gahn@asu.edu; hxhu@asu.edu).

J. Jin is with the Department of Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC 28223 USA (e-mail: jjin@uncc.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2009.2020437

The open services gateway initiative (OSGi) provides a framework to connect diverse devices in a local area network environment such as homes, offices, and automobiles. By defining the standard execution environment and service interface, OSGi promotes the dynamic discovery and collaboration of devices and services from different sources [3]. Moreover, this framework is designed to support connectivity to the outside users allowing remote control and administrative management [23]. In OSGi service platform, the User Admin service provides user authorization functionality. However, the User Admin service's authorization architecture in OSGi is not sufficient enough to support the highly dynamic and open OSGi environments. Thus, an enhanced access control mechanism is needed for achieving interoperability, extensibility, and ease of administration and management. To overcome such issues, role-based access control (RBAC) can be considered to simplify authorization management by associating users with roles, and roles with permissions [7], [19], [24]. Because the roles within an organization typically have overlapping permissions, RBAC supports a hierarchical structure of roles to provide permission inheritance, where a senior role can inherit all permissions assigned to junior roles. As a fundamental aspect of RBAC, constraints can also allow us to lay out higher level access control policies [6], [11].

Recently, several research work [8], [9], [14] have addressed access control issues in OSGi environments using RBAC model. However, these research efforts did not consider the standardized efforts of OSGi authorization. Such generic approaches cannot accommodate all OSGi authorization requirements identified in the OSGi authorization standard [17], causing severe compatibility issues with existing OSGi applications. In addition, they failed to address important RBAC features such as constraints and role hierarchy construction related to OSGi environments. In this paper, we propose a systematic way to apply RBAC to OSGi environments. In our proposed approach, the OSGi authorization mechanism is configured and mapped to RBAC, demonstrating that OSGi authorization requirements can be fully fulfilled by RBAC. Furthermore, we incorporate important RBAC features, which were not addressed by current OSGi authorization mechanism, to enhance existing access control modules in OSGi service environments. Also, a prototype system is implemented to demonstrate the feasibility and effectiveness of our proposed solution.

The rest of this paper is organized as follows. In Section II we briefly describe OSGi service platform, OSGi authorization mechanism, RBAC, and related work. Section III gives an overview of our authorization framework. We also formalize OSGi authorization mechanism and show how OSGi authorization can be accommodated in RBAC. In addition, we discuss the reconstruction of OSGi authorization from RBAC to verify the completeness of our approach. The implementation of our

prototype is described in Section IV. Section V concludes the paper and discusses the future work.

II. Background and Related Work

A. OSGi Service Platform

OSGi specifies an open, common architecture for service providers, developers, software venders, gateway operators, and equipment vendors to develop, deploy, and manage services in a cooperative fashion. In addition, OSGi service platform is an extensible integration platform used to remotely and dynamically deploy, provide, maintain, and manage applications and services with multiple devices in networked environments [3]. We first overview the architecture of OSGi service platform. The architecture of OSGi service platform is composed of a set of bundles based on the OSGi framework that provides the basic functionalities to perform OSGi functions through downloading and executing bundles. Each bundle is a software component that contains algorithms and protocols for controlling a device. When OSGi service gateway requests a bundle, it can be read from local disk or retrieved from the repository and then executed by the service gateway. In other words, OSGi service platform allows bundles to be configured dynamically. This feature enables a home network environment to download and utilize the latest and most optimal bundles, and provides customization of gateway functions for each user taking into account individual profiles.

In OSGi service platform, the User Admin service provides the authorization functionality [17]. All bundles should use the User Admin service to find out if the users attempting to access are authorized or not. In the User Admin service authorization architecture, three components are defined as follows.

1) *User*—A user is a human being who can be identified by credentials such as a password and other identity attributes.
2) *User Group*—A user group is an aggregation of users based on common properties. For example, all family members belong to a user group named *Residents*.
3) *Action Group*—Every action that can be performed by a bundle is associated with an action group. For example, if a bundle could be used to control the alarm system, there should be an action group named *AlarmSystemControl*.

In OSGi authorization, the authorization decision is made based on the following two strategies.

1) *ANY Strategy:* A user could be allowed to carry out an action if he/she belongs to *Any* member of the action group. For example, the *AlarmSystemControl* action group contains two user groups *Administrators* and *Residents*. *Elmer*, *Pepe*, and *Bugs* belong to *Administrators* user group, and *Elmer*, *Pepe*, and *Daffy* belong to *Residents* user group as follows:

$$\begin{aligned} \textit{Administrators} &= \{\ \textit{Elmer, Pepe, Foghorn}\ \} \\ \textit{Residents} &= \{\ \textit{Elmer, Pepe, Daffy}\ \} \\ \textit{AlarmSystemControl} &= \{\ \textit{Administrators, Residents}\ \}. \end{aligned}$$

This *ANY strategy* allows any of four members, *Elmer*, *Pepe*, *Foghorn*, and *Daffy*, to activate the alarm system since all users are a member of user groups and these user groups belong to one action group.

2) *ALL Strategy:* A user is allowed to carry out an action if he/she belongs to *All* members of the action group. In the aforementioned *AlarmSystemControl* example, only *Elmer* and *Pepe* would be authorized to activate the alarm system, since *Daffy* and *Bugs* are not members of both the *Administrators* and *Residents* user groups.

TABLE I
User Groups with Basic User Members

	Elmer	Fudd	Marvin	Pepe	Daffy	Foghorn
Residents	Basic	-	-	Basic	Basic	-
Buddies	-	-	-	-	Basic	Basic
Children	-	-	Basic	Basic	-	-
Adults	Basic	Basic	-	-	-	Basic
Administrators	Basic	-	-	Basic	-	Basic

TABLE II
Action Groups with Basic and Required User Group Members

	Residents	Buddies	Children	Adults	Admins
AlarmSystemControl	Basic	-	-	-	Required
InternetAccess	Basic	-	Basic	Basic	-
TemperatureControl	Required	-	-	Required	-
WebCamAccess	Basic	Basic	-	Required	Required
PhotoAlbumView	Basic	Basic	-	-	-

The implementation of User Admin service in OSGi service platform supports a combination of both strategies by introducing two member sets, namely the *basic* member set for the *ANY strategy* and the *required* member set for *ALL strategy*. *Basic* membership defines a set of members that can get access, and *required* membership reduces this set by requiring the user to be a required member of each action group.

To accommodate this, OSGi allows to assign a user to a user group, and then, the user group is assigned to a specific action group. Tables I and II show an authorization example to demonstrate the assignment relationships, respectively.

The access decision is made based on the *basic* and *required* assignment relationships. For example, in Table II, the action group *WebCamAccess* has two *basic* members, *Residents* and *Buddies*, and two *required* members, *Adults* and *Administrators*. Thus, all users belonging to at least one of the *basic* members, *Residents* and *Buddies*, and all *required* members, *Adults* and *Administrators*, are able to carry out the webcam access action. From Table I, we can see two users *Elmer* and *Foghorn* can meet this authorization requirement.

B. Overview of RBAC Model

RBAC is an alternative policy to traditional mandatory access control (MAC) and discretionary access control (DAC) [22]. As MAC is used in the classical defense arena, the policy of access is based on the classification of objects such as security clearance [21]. The main idea of DAC is that the owner of an object has discretionary authority over the one who can access that object [12], [20]. But RBAC policy is based on the role of the subjects and can specify security policy in a way that maps to an organization's structure.

A general family of RBAC models called RBAC96 was defined by Sandhu *et al.* [19]. Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role, and a permission is an approval of a particular mode of access to one or more objects in the system or some privilege to carry out specified actions. Roles are organized in a partial order $\geq$, so that if $x \geq y$, then role x inherits the permissions of role y. Members of x are also implicitly members of y.

C. Related Work

Several related work studies the access control mechanism for home network environments where the OSGi service platform is operated. Cho *et al.* [8], [9] proposed an authorization policy management framework based on RBAC for OSGi service platform. Through the comparison of several access control models, especially DAC and RBAC, they claimed that RBAC model is flexible and more proper than DAC model for home network environments operated by the OSGi service platform. However, they did not consider the established authorization mechanism in current OSGi standard and omitted how RBAC can be effectively utilized to meet the OSGi authorization requirements. Also, their approach disregarded important RBAC features, such as role hierarchy and constraints. Our approach formalizes current OSGi authorization mechanism, and presents a systematic and comprehensive way to map OSGi authorization mechanism to RBAC configuration so that OSGi authorization requirements can be fully satisfied and enhanced by RBAC. Lim *et al.* [14] presented a mechanism to bundle authentication and authorization services using XML for the OSGi service platform. Their proposed approach uses extensible access control markup language (XACML) to specify RBAC polices for the authorization of service bundles. However, they also ignored existing OSGi authorization mechanism and mainly attempted to deploy the notion of roles in the OSGi platform directly without considering essential RBAC features. To the best of our knowledge, our approach illustrated in this paper is the first RBAC authorization solution that is compatible to the existing OSGi authorization standard, allowing the OSGi application developers and system administrators to better understand and manage security policies.

III. Our Approach

We witness that the current OSGi authorization mechanism is not intuitive for authorization administration. Especially, current OSGi mechanism is not suitable for satisfying all security requirements for defining fine-grained access control policies in a highly dynamic and open OSGi environment. Our objective is to provide an efficient and effective authorization mechanism for such network environments. To achieve this, we propose a role-based authorization framework for OSGi service environments. Our authorization framework consists of following processes.

1) *Mapping OSGi authorization mechanism to RBAC:* RBAC is a powerful mechanism for reducing the management complexity, administration cost, and potential configuration error within the organization. To achieve such features, it is inevitable to identify and derive RBAC components from OSGi authorization requirements. This identification phase allows us to discover roles, role hierarchy, assignment relations, and constraints. Those identified components from OSGi authorization requirements are used to construct a RBAC-based authorization environment that fully simulates the OSGi authorization environment, and provides more fine-grained and robust authorization services.
2) *Reconstructing OSGi authorization from RBAC:* Through mapping OSGi authorization mechanism to RBAC, the complexity of OSGi authorization management is reduced and OSGi authorization can be enhanced by adopting important RBAC features. In addition, we need to validate whether changes in the RBAC-based environment can also be reflected in OSGi authorization environment. In other words, our constructed RBAC environment should be converted back to the OSGi authorization environment, assuring those changes remain intact.

We attempt to formally define components in the current OSGi authorization mechanism, and these formal definitions are used through the rest of this paper. There are three sets of entities: users (*U*), user groups (UG), and action groups (AG). The *basic* user-to-user group assignment (BUA) is a many-to-many relation between *U* and UG. There are two kinds of many-to-many relation between UG and AG. One is *basic* action group-to-user group assignment (BAA) that reflects the *basic memberships* for each action. Another is *required* action group-to-user group assignment (RAA) that reflects the *required memberships* for each action. The following definitions formally summarize each component.

1) U is a set of users, $U = \{u_1, \ldots, u_n\}$.
2) UG is a set of user groups, $\mathrm{UG} = \{\mathrm{ug}_1, \ldots, \mathrm{ug}_n\}$.
3) $\mathrm{AG} = \mathrm{AG}_{\mathrm{Basic}} \cup \mathrm{AG}_{\mathrm{Required}}$ is a set of *basic* and *required* action groups, $\mathrm{AG} = \{\mathrm{ag}_1, \ldots, \mathrm{ag}_n\}$.
4) $\mathrm{BUA} \subseteq U \times \mathrm{UG}$, is a many-to-many *basic* user-to-user group assignment relation.
5) $\mathrm{BAA} \subseteq \mathrm{AG}_{\mathrm{Basic}} \times \mathrm{UG}$, is a many-to-many *basic* action group-to-user group assignment relation.
6) $\mathrm{RAA} \subseteq \mathrm{AG}_{\mathrm{Required}} \times \mathrm{UG}$, is a many-to-many *required* action group-to-user group assignment relation.
7) users: $\mathrm{UG} \rightarrow 2^U$ is a function mapping each user group ug_i to a set of users $\mathrm{users}(\mathrm{ug}_i) = \{u \in U | (u, \mathrm{ug}_i) \in \mathrm{BUA}\}$.
8) user_groups: $U \rightarrow 2^{\mathrm{UG}}$ is a function mapping each user u_i to a set of user groups $\mathrm{user\_groups}(u_i) = \{\mathrm{ug} \in \mathrm{UG} | (u_i, \mathrm{ug}) \in \mathrm{BUA}\}$.
9) ba_user_groups: $\mathrm{AG}_{\mathrm{Basic}} \rightarrow 2^{\mathrm{UG}}$ is a function mapping each basic action group ag_i to a set of user groups, $\mathrm{ba\_user\_groups}(\mathrm{ag}_i) = \{ug \in \mathrm{UG} | (\mathrm{ag}_i, \mathrm{ug}) \in \mathrm{BAA}\}$.
10) re_user_groups: $\mathrm{AG}_{\mathrm{Required}} \rightarrow 2^{\mathrm{UG}}$ is a function mapping each required action group ag_i to a set of user groups, $\mathrm{re\_user\_groups}(\mathrm{ag}_i) = \{\mathrm{ug} \in \mathrm{UG} | (\mathrm{ag}_i, \mathrm{ug}) \in \mathrm{RAA}\}$.
11) ba_action_groups: $\mathrm{UG} \rightarrow 2^{\mathrm{AG}_{\mathrm{Basic}}}$ is a function mapping each user group ug_i to a set of basic action groups,

$\text{ba\_action\_groups}(\text{ug}_i)] = \{\text{ag} \in \text{AG}_{\text{Basic}} | (\text{ag}, \text{ug}_i) \in \text{BAA}\}$.

12) re_action_groups: $\text{UG} \rightarrow 2^{\text{AG}_{\text{Required}}}$ is a function mapping each required user group ug_i to a set of required action groups, $\text{re\_action\_groups}(\text{ug}_i) = \{\text{ag} \in \text{AG}_{\text{Required}} | (\text{ag}, \text{ug}_i) \in \text{RAA}\}$.

A. Construction of OSGi-Compliant RBAC

In this section, we demonstrate how RBAC components and features can be utilized to support components of OSGi authorization. We describe how we can identify basic RBAC component, role hierarchy, and constraints to enforce OSGi authorization requirements.

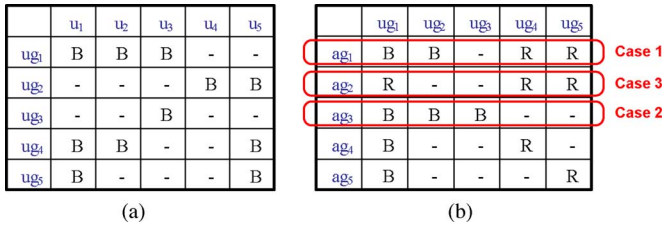
1) Basic RBAC Components Construction: The essence of RBAC is the notion of *roles* [24] to abstract users and permissions. Permissions are grouped through roles, and users obtain permissions by being assigned to roles. Users, roles, permissions, and the corresponding assignment relations are core components in RBAC.

We noted that the user set (*U*) and the action group set (AG) can essentially be treated as *users* and *permissions* in the RBAC model, respectively. In addition, the user group set (UG) in OSGi can be represented as *roles* such that actions are abstracted through user groups, and these actions can be exercised by a member of the specific user group. However, normally OSGi uses two types of assignments, namely, the basic assignment (BAA) and the required assignment (RAA) that can allow us to achieve the abstraction of actions through user groups. However, RBAC cannot accommodate this characteristic directly, so it leads us to propose several properties to bridge this gap in the course of OSGi-compliant RBAC construction.

First, we construct the role (*R*) by introducing a concept of *Private Membership*. In particular, we treat the user groups in OSGi as the private members of each role in role construction, which can be further characterized as *basic* members and *required* members depending on the property of the OSGi BAA relation and RAA relation. The detailed role construction process is explained through an OSGi authorization example, as shown in Fig. 1. In this example, Table (a) reflects the assignment relation between *U* and UG, and Table (b) shows the assignment relation between AG and UG. In Table (b), there are two types of assignment relations, *Basic* and *Required*, to reflect BAA and RAA relations, respectively. For a particular action group, we identified that there exist three possible combinations on *Basic* and *Required* assignments given as follows.

1) *Case 1*: Contains both *basic* assignments and *required* assignments, as shown in the row of ag_1, $\text{ba\_user\_groups}(\text{ag}_1) = \{\text{ug}_1, \text{ug}_2\}$ and $\text{re\_user\_groups}(\text{ag}_1) = \{\text{ug}_4, \text{ug}_5\}$.
2) *Case 2 :* Contains *basic* assignments only, as shown in the row of ag_3, $\text{ba\_user\_groups}(\text{ag}_3) = \{\text{ug}_1, \text{ug}_2, \text{ug}_3\}$.
3) *Case 3*: Contains *required* assignments only as shown in the row of ag_2, $\text{re\_user\_groups}(ag_2) = \{\text{ug}_1, \text{ug}_4, \text{ug}_5\}$.

As specified in the OSGi authorization rule, a user must be assigned to *ALL required* user groups and *ANY basic* user groups before he/she can exercise a particular action. Also, while accommodating the identified *basic* and *required* assignment patterns, we define the following mapping rules on action group basis.



	u_1	u_2	u_3	u_4	u_5
ug_1	B	B	B	-	-
ug_2	-	-	-	B	B
ug_3	-	-	B	-	-
ug_4	B	B	-	-	B
ug_5	B	-	-	-	B

(a)

	ug_1	ug_2	ug_3	ug_4	ug_5	
ag_1	B	B	-	R	R	Case 1
ag_2	R	-	-	R	R	Case 3
ag_3	B	B	B	-	-	Case 2
ag_4	B	-	-	R	-	
ag_5	B	-	-	-	R	

(b)

Fig. 1. OSGi authorization example. (a) USER-UG assignment. (b) AG-UG assignment.

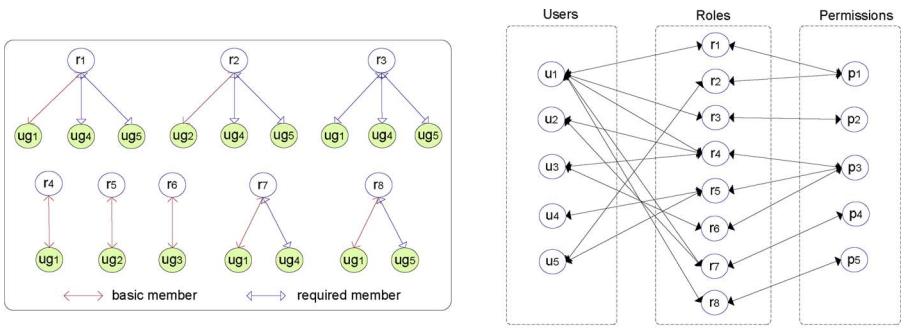
Rule 1: Introduce one role and one permission-to-role assignment (PA) for each OSGi basic action group-to-user group assignment in BAA. The role contains only one basic private member of the basic user group and ALL required members of the required user groups. The particular action group in BAA is constructed as one permission and the permission is assigned to the role.

Rule 2: In case of no basic assignment, introduce one role that contains ALL required members of the required user groups and one PA by assigning the permission to the role.

Rule 3: In terms of user-to-role assignment, a user can be assigned to a role when the corresponding user in OSGi authorization is assigned to all the private members of that role.

Now, we discuss how to use these three rules to construct the RBAC for each case that we have identified. First, we map all users and action groups in OSGi authorization to users and permissions directly as RBAC components. As identified in Case 1, the action group ag_1 has both basic member and required member. Following Rule 1, we construct roles such as two roles r_1 and r_2 for the action group ag_1 in our example. Role r_1 contains a basic member ug_1, and two required members ug_4 and ug_5. Role r_2 is constructed by a basic member ug_2, and all required members ug_4 and ug_5. Formally, $\text{members}(r_1) = \{\text{ug}_1, \text{ug}_4, \text{ug}_5\}$ and $\text{members}(r_2) = \{\text{ug}_2, \text{ug}_4, \text{ug}_5\}$. Using the same rule, the permission p_1 corresponding to the action group ag_1 is assigned to roles r_1 and r_2. Following Rule 3, a user is assigned to a role only if the user has been assigned to all private members of this role. In the example, [see Fig. 1(a)], u_1 is assigned to ug_1, ug_4, and ug_5. Since ug_1, ug_4, and ug_5 are private members of r_1, u_1 should be assigned to r_1 in RBAC. For the same reason, u_5 is assigned to r_2. From Fig. 2, u_1 and u_5 can hold the same action group ag_1, and u_1 and u_5 own the same permission p_1 via r_1 and r_5, respectively. Hence, the same authorization requirements are fulfilled through the RBAC configuration process.

In Case 2, the action group has only three basic members ug_1, ug_2, and ug_3. Using Rule 1, three roles r_4, r_5, and r_6 are constructed by these three basic members of ag_3, respectively, as shown in Fig. 2. Formally, $\text{members}(r_4) = \{\text{ug}_1\}$, $\text{members}(r_5) = \{\text{ug}_2\}$, and $\text{members}(r_6) = \{\text{ug}_3\}$. Corresponding assignment relations based on Rules 1 and 3 are shown as well.



(a) (b)

Fig. 2. Overall process of constructing RBAC from an OSGi authorization. (a) Role construction. (b) Mapping to RBAC.

TABLE III
FUNCTIONS/PREDICATES USED IN THE PAPER

Function/predicate	Description
newRole(bm, RM)	Returns a new role constructed by a basic member *bm* and/or all required members from *RM*.
addPA(p, r)	Returns True if permission p is assigned role r.
deletePA(p, r)	Returns True if assignment form permission p to role r is deleted.
addUA(u, r)	Returns True if user u is assigned role r.
isUA(u, r)	Returns True if user u has been assigned role r.
deleteUA(u, r)	Returns True if assignment form user u to role r is deleted.
isSenior(r_i, r_j)	Returns True if role r_i is senior to role r_j.
isJunior(r_i, r_j)	Returns True if role r_i is junior to role r_j.
allSenior(r)	Returns the set of all senior of role r
allJunior(r)	Returns the set of all junior of role r
parents(r)	Returns the set of all immediate senior of role r
children(r)	Returns the set of all immediate junior of role r
addParent(r_i, r_j)	Returns True if role r_i is added to role hierarchy as an immediate senior of role r_j
addChild(r_i, r_j)	Returns True if role r_i is added to role hierarchy as an immediate junior of role r_j
deleteParent(r_i, r_j)	Returns True if role r_i is deleted from role hierarchy as an immediate senior of role r_j
deleteChild(r_i, r_j)	Returns True if role r_i is deleted from role hierarchy as an immediate junior of role r_j
members(r)	Returns the set of all private members of role r
ba_member(r)	Returns the set of the basic private member of role r
re_members(r)	Returns the set of all required private members of role r
addBUA(u, ug)	Returns True if user u is assigned user group ug as a basic member.
deleteBUA(u, ug)	Returns True if basic assignment form user u to user group ug is deleted.
addBAA(ag, ug)	Returns True if action group ag is assigned user group ug as a basic member.
deleteBAA(ag, ug)	Returns True if basic assignment form action group ag to user group ug is deleted.
addRAA(ag, ug)	Returns True if action group ag is assigned user group ug as a required member.
deleteRAA(ag, ug)	Returns True if required assignment form action group ag to user group ug is deleted.

In Case 3, the action group has only required members. The second row of the OSGi authorization example [see Fig. 1 (b)] shows an action group ag_2 that has three required members ug_1, ug_4, and ug_5. Using Rule 2, one role r_3 is constructed that contains all three required members of ag_2. Formally, $\text{members}(r_3) = \{\text{ug}_1, \text{ug}_4, \text{ug}_5\}$. Fig. 2 depicts the role construction and assignment relationships.

In addition to the elements and functions in both our formal representation of OSGi authorization and the RBAC model, Table III lists additional functions and predicates which are used in the subsequent sections. To realize the basic RBAC construction from OSGi authorization, we develop a mapping algorithm, which maps OSGi authorization to basic RBAC specification. For the mapping algorithm in Fig. 3, let n_U, n_{AG}, and n_{BAA} denote the number of users, action groups, and basic assignment relationships (between action groups and user groups), respectively. The computational complexity of this mapping algorithm is $O(n_{AG} n_{BAA} n_U)$.

2) Building Role Hierarchies: We construct role hierarchy using a NTree [18] structure to define an inheritance relationship, reducing the cost of administration. For example, all managers in the same organization may have a certain set of core "management privileges," even though they work in different departments. This commonality can be exploited through a role hierarchy that enables each department manager role to inherit a generic "management" role. Role hierarchy allows the policy designer to write generic access polices once and to simplify the complexity of access control policies.

In our mapping approach, roles are constructed based on *basic* members and *required* members of action groups of OSGi authorization mechanism. Considering the private member sets of every role, we discover inclusion relations between them. For example, in Fig. 4(a), r_1 is constructed by a basic private member ug_1, and two required private members ug_4 and ug_5. Formally, $\text{ba\_member}(r_1) = \{\text{ug}_1\}$ and $\text{re\_members}(r_1) = \{\text{ug}_4, \text{ug}_5\}$; while r_7 is constructed by a basic private member ug_1 and a required private members ug_4, i.e., $\text{ba\_member}(r_7) = \{\text{ug}_1\}$ and $\text{re\_members}(r_7) = \{\text{ug}_4\}$. Hence, the following condition is *true*

$$(\text{ba\_member}(r_7) = \text{ba\_member}(r_1)) \wedge$$
$$(\text{re\_members}(r_7) \subset \text{re\_members}(r_1)).$$

Algorithm [Mapping]
Input: OSGi Authorization Specification
Output: RBAC Authorization Specification
Method:
(1) $P \leftarrow AG$;
(2) $U \leftarrow USER$;
(3) **FOR** *each* $p \in P$ **DO**
(4) $BM = ba\_user\_groups(p)$;
(5) $RM = re\_user\_groups(p)$;
(6) **IF** $(BM \neq \emptyset) \wedge (RM \neq \emptyset)$ **THEN**
/* **Case 1:** *Have both Basic Member and Required Member* */
(7) **FOR** each $bm_i \in BM$ **DO**
(8) $r = newRole(bm_i, RM)$;
(9) **IF** $(r \notin R)$ **THEN**
(10) $R.add(r)$;
(11) $addPA(p, r)$;
(12) **FOR** each $u \in U$ **DO**
(13) **IF** $(bm_i \in user\_groups(u)) \wedge (RM \subset user\_groups(u))$ **THEN**
(14) $addUA(u, r)$;
(15) **ELSE**
(16) **IF** $(BM \neq \emptyset) \wedge (RM = \emptyset)$ **THEN**
/* **Case 2:** *Only have Basic Member* */
(17) **FOR** each $br_i \in BM$ **DO**
(18) $r = newRole(bm_i)$;
(19) **IF** $(r \notin R)$ **THEN**
(20) $R.add(r)$;
(21) $addPA(p, r)$;
(22) **FOR** each $u \in U$ **DO**
(23) **IF** $br_i \in user\_groups(u)$ **THEN**
(24) $addUA(u, r)$;
(25) **ELSE**
(26) **IF** $(BM = \emptyset) \wedge (RM \neq \emptyset)$ **THEN**
/* **Case 3:** *Only have Required Member* */
(27) $r = newRole(RM)$;
(28) **IF** $(r \notin R)$ **THEN**
(29) $R.add(r)$;
(30) $addPA(p, r)$;
(31) **FOR** each $u \in U$ **DO**
(32) **IF** $RM \subset user\_groups(u)$ **THEN**
(33) $addUA(u, r)$;

Fig. 3. Mapping algorithm from OSGi authorization to RBAC authorization.

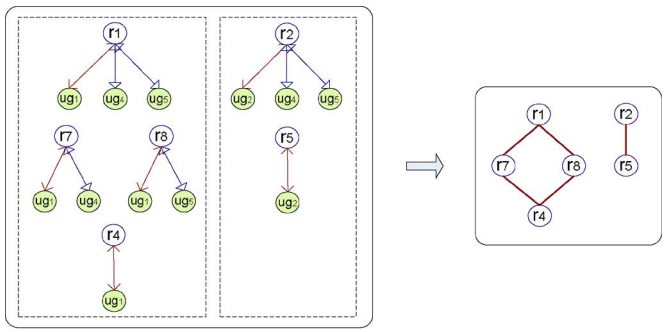


Fig. 4. Identifying role hierarchy relationships from role construction. (a) Role construction. (b) Role hierarchies.

Since the private members of r_1 include all private members of r_7, a user who is assigned to r_1 should be assigned to r_7 as well according to Rule 3. Thus, a user assigned to r_1 should have all permissions of r_7. In other words, r_1 should possess all permissions of r_7, and r_1 is more powerful than r_7. Therefore, a role hierarchy relation can be build between r_1 and r_7. As a

Algorithm [Building RH]
Input: The result of mapping OSGi authorization to RBAC
Output: RBAC authorization specification including role hierarchy
Method:
(1) **FOR** *each* $r_i \in R$ **DO**
(2) $R' \leftarrow \{r_1, \ldots, r_{i-1}\}$
(3) **FOR** *each* $r_j \in R'$ **DO**
(4) **IF** $isSenior(r_i, r_j) \wedge (r_i \notin allSenior(r_j))$ **THEN**
/* *Insert a senior role to role hierarchy* */
(5) $insertSenior(r_i, r_j)$;
(6) **IF** $isJunior(r_i, r_j) \wedge (r_i \notin allJunior(r_j))$ **THEN**
/* *Insert a junior role to role hierarchy* */
(7) $insertJunior(r_i, r_j)$;

Procedure insertSenior(r_i, r_j)
Input: r_i, r_j are two roles in RBAC authorization specification
Output: role hierarchy in which r_i is a senior of r_j
Method:
(1) $flag = 1$;
(2) **FOR** *each* $r \in parents(r_j)$ **DO**
(3) **IF** $isJunior(r_i, r)$ **THEN**
/* *Insert* r_i *between* r *and* r_j *in role hierarchy* */
(4) **IF** *each* $r_i \notin parents(r_j)$ **THEN**
(5) $addParent(r_i, r_j)$;
(6) $addChild(r_i, r)$;
(7) $deleteParent(r, r_j)$;
(8) $flag = 0$;
(9) **IF** $isSenior(r_i, r)$ **THEN**
/* *Insert* r_i *into a sub-role hierarchy* */
(10) $insertSenior(r_i, r)$;
(11) $flag = 0$;
(12) **IF** $flag = 1$ **THEN**
/* *Insert* r_i *as a immediate senior of* r_j *in role hierarchy* */
(13) $addParent(r_i, r_j)$;

Fig. 5. Algorithm for building role hierarchy.

senior role, r_1 inherits all permissions of r_7. Commonly, if any of the following two conditions is *true*, r_i is senior to r_j.

1) $(\text{ba\_member}(r_j) \neq \phi) \wedge (\text{ba\_member}(r_i) \neq \phi)$
$\wedge (\text{re\_member}(r_i) \neq \phi)$
$\wedge (\text{ba\_member}(r_j) = \text{ba\_member}(r_i))$
$\wedge (\text{re\_members}(r_j) \subset \text{re\_members}(r_i))$
2) $(\text{ba\_member}(r_j) = \phi) \wedge (\text{re\_member}(r_i) \neq \phi)$
$\wedge (\text{re\_member}(r_j) \neq \phi)$
$\wedge (\text{re\_members}(r_j) \subset \text{re\_members}(r_i))$.

In the first condition, r_i has both a basic member and required members, and r_j has at least a basic member. When r_i and r_j have same basic member, and the required member set of r_j (including empty set) is a subset of the required member set of r_i, r_i is senior to r_j. In another condition, if r_j contains only a required member set, which is a subset of the required member set of r_i, r_j is a junior role of r_i. With this methodology, two role hierarchies can be identified, as shown in Fig. 4(b).

We now introduce an algorithm for building role hierarchies described in Fig. 5. The algorithm builds role hierarchies through a series of continuous processes of adding a role to existing role hierarchies. The continuous processes are controlled by two loops that are called to compare each role $r_i \in R$ with every role $r_j \in \{r_1, \ldots, r_{i-1}\}$ in the existing role hierarchies.

If r_i is senior to r_j and r_i is not in role hierarchies, we can insert r_i to role hierarchies as a senior of r_j by calling `insertSenior` procedure. Another possible case is performing `insertJunior` procedure when r_i is junior to r_j. The algorithm is illustrated in Fig. 5.

Supposing there exits a role hierarchy and r_j is in the hierarchy, then we need to insert r_i to the hierarchy as a senior role of r_j. There are three possible cases for the inserting process that are given as follows.

1) If r_i is junior to a role $r \in parents(r_j)$, r_i is inserted between r and r_j.
2) If r_i is senior to a role $r \in parents(r_j)$, r_i should be inserted into a subrole hierarchy and a recursive process should be carried out.
3) If r_i has no relations with all parent roles of r_j, r_i is added into the role hierarchy as an immediate senior of r_j.

There are two kinds of assignment relations in RBAC: explicit assignment and implicit assignment. These assignments can result in redundant role memberships. To detect and eliminate the redundancies, the following step is also taken into account: if a user *u* has been assigned to a superior role *r* in role hierarchy, all user-to-role assignments between *u* to junior roles of *r* are examined.

3) Applying Well-Known Role-Based Constraints: RBAC is a policy-oriented approach, and allows to specify well-known security principles, such as separation of duty and least privilege. These security principles can be defined as constraints in RBAC. Also, we have proved that OSGi authorization requirements can be satisfied by RBAC in the previous discussion.

B. Reconstructing OSGi Authorization From RBAC

In the previous section, we show that OSGi authorization mechanism can be mapped to RBAC and it could be enhanced by important RBAC features. In this section, we discuss how we can reconstruct OSGi authorization mechanism from RBAC-based authorization when we change PA and user-to-role assignment (UA).[1] Note that in our approach, core RBAC components are constructed from OSGi authorization components and some OSGi authorization features are maintained in the constructed RBAC components bound to the private memberships. Thus, changes in RBAC components are constrained to maintain the consistency with OSGi authorization components. For this reason, we name RBAC, used in this paper, as an OSGi-compliant RBAC.

1) PA Change: In RBAC, an administrator can change the PA by adding or deleting an assignment relation. We first consider the ADD operation. Suppose an administrator attempts to assign a permission *p* to a role *r* in the OSGi-compliant RBAC. If the permission *p* has not been assigned to any role, the ADD operation is allowed. To reflect this change in OSGi authorization, a new *basic* assignment (*B* in the AG–UG table) should be created between the action group ag corresponding to *p* and the user group ug as the basic private member of *r*. In addition, *required* assignments (*R* in the AG–UG table) between the action group ag and all user groups as required private members of *r* should be added as well.

If *p* already has been assigned to other roles, we note that there are three potential cases from the previous construction processes.

1) The roles assigned to *p* have both *basic* and *required* private members. And the *required* private member set of these roles should be the same. If *p* is assigned to another role *r*, *r* should also have both *basic* and *required* private members, and the *required* private members must be the same as the roles that *p* has already been assigned to. Otherwise, the assignment operation is not allowed. In reflecting this new permission-role assignment, a *basic* assignment relation between the action group ag corresponding to *p* and the user group as the *basic* private member of *r* should be created in OSGi authorization.
2) All roles of *p* have only a *basic* private member. If the administrator assigns *p* to another role *r*, this role *r* should also have only a *basic* private member. The corresponding basic assignment is created in OSGi authorization.
3) If *p* has been assigned to a role and the role has only *required* private members, *p* cannot be further assigned to other roles in RBAC.

Next, we consider the *DELETE* operation of an assignment relation between a permission *p* and a role *r* in RBAC. There are no constraints for revoking permission-role assignments in RBAC. When we reconstruct OSGi authorization, if *p* has been assigned to roles other than *r*, we delete the basic assignment relation (*B*) between the action group ag corresponding to *p* and the user group ug as the basic member of *r*. For other case, if *p* has been assigned only to *r*, then all assignment relations between the action group ag and the private members of *r* in OSGi authorization are deleted. A detailed PA change algorithm is given in Fig. 6. This algorithm is used to reconstruct OSGi authorization when a PA is changed in RBAC. The complexity of PA change algorithm is $O(n_{\mathrm{UG}})$, where n_{UG} represents the number of user groups in OSGi authorization.

2) UA Change: Similar to the PA change, UA change includes adding and/or deleting a user-to-role assignment under certain conditions in RBAC. When a user *u* is assigned to a role *r* in RBAC, the corresponding user in OSGi authorization should be assigned to all private members of the role in OSGi authorization. Thus, the assignment relation between *u* and a private member $\mathrm{ug}_i \in \mathrm{members}(r)$ should be added if the user *u* has not been assigned to ug_i in OSGi authorization.

Since a user-to-role assignment in RBAC is build based on the relations between the users and all private members of this role, deleting any of such relations may cause the revocation of the user-to-role assignment in RBAC. Thus, when an administrator wants to delete a user-to-role assignment in RBAC, we cannot just delete all relations between a particular user and all private members of this role directly in OSGi, and this might result in other UA changes to other roles. To reduce the complexity, our UA change algorithm requires the administrator to select the exact private members of the role that should be revoked.

[1]Some RBAC features, such as RBAC constraints, could not be recognized during reconstruction, since the original OSGi authorization mechanism cannot support these features. However, we attempt to apply such a crucial feature to our approach.

```
Algorithm [Changing PA]
Procedure ChangePA(p, r)
Input: p, r where p is a permission and r is a role in RBAC authorization
 specification
Output: changed OSGi authorization specification
Method:
(1)  IF addPA(p, r) THEN      /* Add a permission-to-role assignment */
(2)     IF role(p) ≠ Ø THEN
(3)        IF (ba_member(r) ≠ Ø) ∧ (re_members(r).size() =
 re_user_groups(p).size()) ∧ (re_members(r) ⊆ re_user_groups(p)) THEN
(4)           addBAA(p, ba_member(r));
(5)     IF role(p) = Ø THEN
(6)        IF ba_member(r) ≠ Ø THEN
(7)           addBAA(p, ba_member(r));
(8)        IF re_members(r) ≠ Ø THEN
(9)           FOR each ug_i ∈ re_members(r) DO
(10)             addRAA(p, ug_i);
(11) IF deletePA(p, r) THEN  /* Delete a permission-to-role assignment */
(12)    IF role(p).size() = 1 THEN
(13)       IF ba_member(r) ≠ Ø THEN
(14)          deleteBAA(p, ba_member(r));
(15)       IF re_members(r) ≠ Ø THEN
(16)          FOR each ug_i ∈ re_members(r) DO
(17)             deleteRAA(p, ug_i);
(18)       deleteRole(r);
(19)    IF role(p).size() > 1 THEN
(20)       deleteBAA(p, ba_member(r));
```

Fig. 6. Algorithm for handling PA change.

```
Algorithm [Changing UA]
Procedure ChangeUA(u, r)
Input: u, r where u is a user and r is a role in RBAC authorization specification
Output: changed OSGi authorization specification
Method:
(1)  IF addUA(u, r) THEN      /* Add a user-to-role assignment */
(2)     FOR each ug_i ∈ members(r) DO
(3)        IF u ∉ users(ug_i) THEN
(4)           addBUA(u, ug_i);
(5)     FOR each r_i ∈ R DO  /* Rebuild other user-to- role assignments */
(6)        IF members(r_i) ⊆ user_groups(u) THEN
(7)           addUA(u, r_i)
(8)  IF deleteUA(u, r) THEN      /* Delete a user-to-role assignment */
(9)     IF select all THEN
(10)       FOR each ug_i ∈ members(r) DO
(11)          deleteBUA(u, ug_i);
(12)    IF select ug_i ∈ members(r) THEN
(13)       deleteBUA(u, ug_i);
(14)    FOR each r_i ∈ role(u) DO
    /* Rebuild other user-to- role assignments */
(15)       IF members(r_i) ⊈ user_groups(u) THEN
(16)          deleteUA(u, r_i)
```

Fig. 7. Algorithm for handling UA change.

The revocations are directly reflected in the user-to-user group assignments in OSGi. Nevertheless, the changes of user-to-user group assignments in OSGi authorization may also affect other user-to-role assignments in RBAC. The user-to-role assignment relations in RBAC should be rebuild to be consistent with OSGi authorization changes. The rebuilding process can be performed through checking all roles that the user is assigned to. The UA change algorithm is shown in Fig. 7. Since the UA change algorithm involves two assignment processes, the complexity of the algorithm is $O(n_{\mathrm{UG}} + n_R)$, where n_{UG} and n_R represent the number of user groups and roles in the authorization systems, respectively.

IV. Implementation and Performance Evaluation

To prove the feasibility of our approach, we design a prototype system, which is based on an open source OSGi framework implementation called Knopflerfish [2]. A security-enhanced OSGi authorization architecture is depicted in Fig. 8. In our implementation, a Web-based OSGi authorization management tool is designed to support our policy management framework. The tool is composed of two major GUIs: 1) OSGi authorization management GUI communicates with OSGi authorization engine to manage standard OSGi authorization policy[2] and 2) RBAC authorization management GUI is used to manage RBAC authorization through the RBAC authorization engine. Transformation handler is responsible for mapping OSGi authorization to RBAC, building role hierarchies, and reconstructing OSGi authorization by performing corresponding algorithms. In addition, the policy enforcement is based on the pull mode, in which a policy enforcement point (PEP) collects the related information of subjects and queries a policy decision point (PDP) for policy decision. By integrating the Sun's XACML library [4] into OSGi service platform, the authorization service as the PDP module interprets XACML polices and makes access decisions. The User Admin service as the PEP module queries the authorization service and enforces the relevant operations.

Finally, we give a scenario to illustrate the usage of our prototype system. Supposing OSGi service platform is installed in a home network, which has been integrated with the implementation of our authorization framework. *Bob* is an administrator of the home network gateway. He defines an action group *AccessWebCam*, and two user groups, *Residents,* and *Adults*, as a basic member and a required member of the action group *AccessWebCam*, respectively. Then, when he uses our tool to map the OSGi authorization to RBAC, a role *Residents_Adults* is constructed and the permission corresponding to the action group *AccessWebCam* is assigned to the role. In addition, *Bob* defines several constraints for the role *Residents_Adults*. For example, a constraint can be established to specify "the user assigned to *Residents_Adults* role can perform the *AccessWebCam* action when he/she is in his/her office between 9 am and 5 pm." Now, *Alice* wants to access the webcam installed in the home remotely. She logs into the home service gateway and retrieves *AccessWebCam* bundle. If *AccessWebCam* bundle does not exist in the service gateway, she requests the service gateway to download *AccessWebCam* bundle from the bundle repository and install it in the gateway. However, when *Alice* tries to execute the *AccessWebCam* bundle, the system denies her access request and indicates that she needs permission for starting the bundle. There are two possible cases that cause such access denials. One case would be that *Bob* has not assigned the user *Alice* to the two user groups, *Residents* and *Adults*, in OSGi authorization management or has not assigned *Alice* to *Residents_Adults* role in RBAC authorization management.

[2]The current OSGi authorization mechanism can coexist with the OSGi-compliant RBAC authorization mechanism.

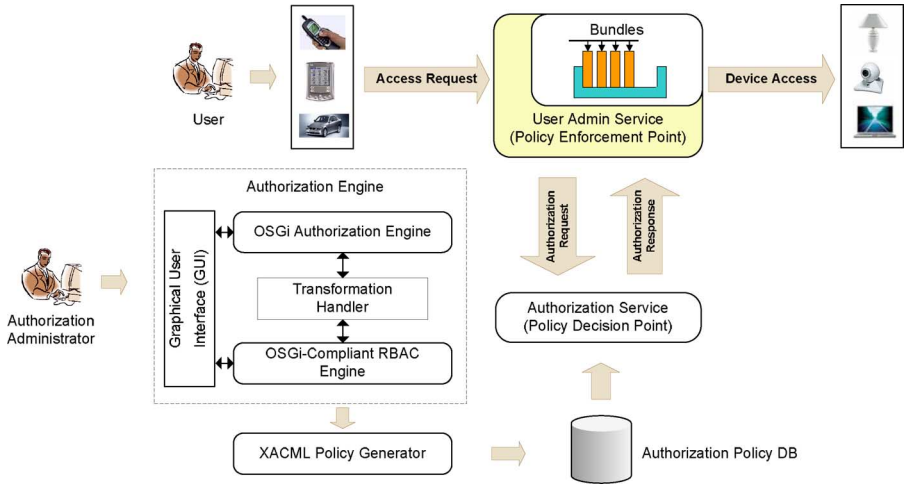


Fig. 8. Security-enhanced OSGi authorization architecture.

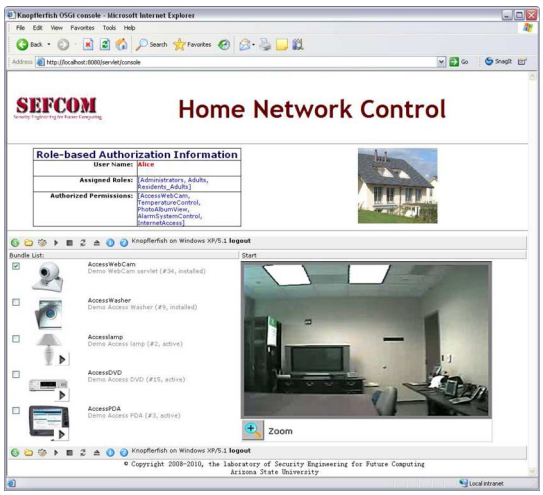


Fig. 9. Prototype system.

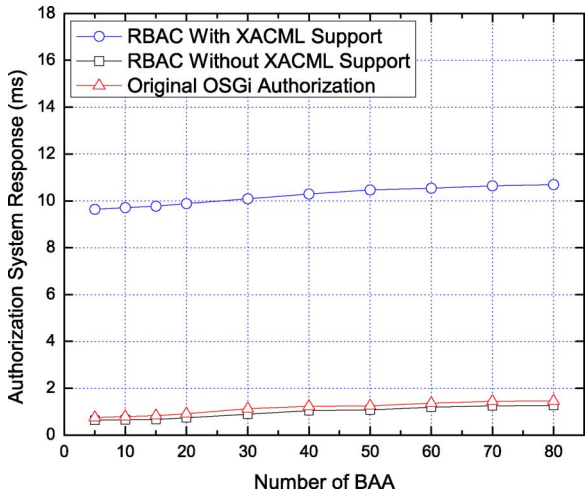


Fig. 10. Performance evaluation of the system implementation.

Another possible case is that the constraints are violated by *Alice*'s current conditions. Hence, for the former case, *Alice* can ask *Bob* to set up the corresponding assignment relations, while *Alice* can start *AccessWebCam* bundle when her situation satisfies the location and time constraints for the latter case. When all conditions are satisfied, *Alice*'s remote access request to webcam is allowed, as shown in Fig. 9.

According to the rules for constructing OSGi-compliant RBAC in our solution, the number of generated roles and the number of PAs are determined by the number of basic assignment relationships (BAAs) between action groups and user groups. Obviously, BAA assignments in OSGi authorization policies affect the response time for our RBAC authorization system. Therefore, the number of BAA is chosen as the evaluation metrics for our experiments. By randomly triggering OSGi bundle access, we measure and compare the response time of three authorization mechanisms, as shown in Fig. 10. The experiments were performed on Intel Core 2 Duo CPU 3.00 GHz with 3.25 GB RAM running on Windows XP and Apache Tomcat. Each result was measured in milliseconds and computed based on the average over simulated runs. Compared to the original OSGi authorization mechanism, RBAC authorization without XACML shows better performance, since RBAC directly maps users to permissions through roles. In other words, our experiments prove that the proposed OSGi strategies are correctly designed. When RBAC policies are specified in XACML, it introduced about 9 ms overhead to RBAC authorization due to XACML request generation and policy evaluation. However, considering the potential benefits of XACML standard such as extensibility and interoperability, we believe this overhead is acceptable.

V. Conclusion and Future Work

In this paper, we have formulated current OSGi authorization mechanism and identified its inherent limitations. In addition, we have demonstrated how OSGi authorization mechanism could be mapped to RBAC and how OSGi authorization requirements can be fulfilled to leverage important RBAC features in the OSGi environment. Furthermore, we discussed possible enhancement of OSGi authorization with role-based constraints.

A proof-of-concept prototype has been implemented to show the feasibility of our approach with possible consideration of interoperability and extensibility issues. Our future work would attempt to specify and verify complicated access control policies in OSGi environments, especially context-aware constraints including tools for policy specification and analysis. Also, we would study how our approach can be applied to universal plug and play (UPnP) [5] and digital living network alliance (DLNA) [1], which have similar security requirements as OSGi in home network environments.

REFERENCES

[1] Digital Living Network Alliance. (2003). [Online]. Available: http://www.dlna.org/home

[2] Knopflerfish. *Knopflerfish Open Source OSGi*. (2006). [Online]. Available: http://www.knopflerfish.org/index.html

[3] OSGi Alliance. *OSGi Initiative*. (1999). [Online]. Available: http://www.osgi.org.

[4] Sun Microsystems, Inc. *Sun's XACML implementation.* (2004). [Online]. Available: http://sunxacml.sourceforge.net/

[5] UPnP Forum. *UPnP Standards for Device Security and Security Console*. (2003). [Online]. Available: http://www.upnp.org/standardizeddcps/security.asp

[6] G.-J. Ahn and R. S. Sandhu, "Role-based authorization constraints specification," *ACM Trans. Inf. Syst. Security*, vol. 3, no. 4, pp. 207–226, Nov. 2000.

[7] E. Bertino, L. Khan, R. Sandhu, and B. Thuraisingham, "Secure knowledge management: Confidentiality, trust, and privacy," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 36, no. 3, pp. 429–438, May 2006.

[8] E.-A. Cho, C.-J. Moon, D.-H. Park, and D.-K. Baik, "Access control policy management framework based on RBAC in OSGi service platform," in *Proc. 6th IEEE Int. Conf. Comput. Inf. Technol.* (CIT 2006), IEEE Computer Society, Washington, DC, 2006, pp. 161–166.

[9] E.-A. Cho, C.-J. Moon, D.-H. Park, and D.-K. Baik, "An effective policy management framework using RBAC model for service platform based on components," in *Proc. 4th Int. Conf. Softw. Eng. Res., Manage. Appl. (SERA 2006)*, IEEE Computer Society, Washington, DC, 2006, pp. 281–288.

[10] C. M. Ellison, "Home network security," *Intel Technol. J.*, vol. 6, no. 4, pp. 37–48, 2002.

[11] T. Jaeger, "On the increasing importance of constraints," in *Proc. 4th ACM Workshop Role-based Access Contr.*, 1999, pp. 33–42.

[12] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino, "A unified framework for enforcing multiple access control policies," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1997, pp. 474–485.

[13] C. Lee, D. Nordstedt, and S. Helal, "Enabling smart spaces with OSGi," *IEEE Pervasive Comput.*, vol. 2, no. 3, pp. 94–98, Jul.–Sep. 2003.

[14] H.-Y. Lim, Y.-G. Kim, C.-J. Moon, and D.-K. Baik, "Bundle authentication and authorization using XML security in the OSGi service platform," in *Proc. 4th Annu. ACIS Int. Conf. Comput. Inf. Sci. (ICIS 2005)*, IEEE Computer Society, Washington, DC, 2005, pp. 502–507.

[15] M. Lorch, D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah, "The prima system for privilege management, authorization and enforcement in grid environments," presented at the 4th Int. Workshop Grid Comput., Phoenix, AR, 2003.

[16] OASIS, *XACML 2.0 Core: eXtensible Access Control Markup Language (XACML) Version 2.0*. (2005). [Online]. Available: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[17] OSGi, *User Admin Service Specification Version 1.1*. (2006). [Online]. Available: http://www.osgi.org

[18] R. Sandhu, "The ntree: A two dimension partial order for protection groups," *ACM Trans. Comput. Syst.*, vol. 6, no. 2, pp. 197–222, May 1988.

[19] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control model," *IEEE Comput.*, vol. 2, no. 29, pp. 38–47, Feb. 1996.

[20] R. Sandhu and Q. Munawer, "How to do discretionary access control using roles," in *Proc. 3rd ACM workshop Role-Based Access Contr.*, Aug. 1999, pp. 47–54.

[21] R. S. Sandhu, "Lattice-based access control models," *IEEE Comput.*, vol. 26, no. 11, pp. 9–19, Nov. 1993.

[22] R. S. Sandhu and P. Samarati, "Access control: Principles and practice," *IEEE Commun. Mag.*, vol. 32, no. 9, pp. 40–48, Sep. 1994.

[23] C. Wu, C. Liao, and L. Fu, "Service-oriented smart-home architecture based on OSGi and mobile-agent technology," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 37, no. 2, pp. 193–205, Mar. 2007.

[24] H. Zhu and M. Zhou, "Roles in information systems: A survey," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 38, no. 3, pp. 377–396, May 2008.

Gail-Joon Ahn (SM'07) received the Ph.D. degree in information technology from George Mason University, Fairfax, Virgina, 2000.

He was an Associate Professor in the College of Computing and Informatics, and the Founding Director of the Center for Digital Identity and Cyber Defense Research, and Laboratory of Information Integration, Security, and Privacy (LIISP), University of North Carolina at Charlotte, Charlotte. He is currently an Associate Professor in the Department of Computer Science and Engineering and the Director of Security Engineering for Future Computing (SEFCOM) Laboratory at Arizona State University (ASU), Tempe. His current research interests include information and systems security, vulnerability and risk management, access control, and security architecture for distributed systems. His research has been supported by the U.S. National Science Foundation, National Security Agency (NSA), U.S. Department of Defense (DoD), U.S. Department of Energy (DoE), Bank of America, Hewlett Packard, Microsoft, and Robert Wood Johnson Foundation.

Dr. Ahn is a recipient of the U.S. Department of Energy CAREER Award and the Educator of the Year Award from the Federal Information Systems Security Educators Association (FISSEA).

Hongxin Hu is currently working toward the Ph.D. degree at the Department of Computer Science and Engineering, Arizona State University, Tempe.

He is a member of the Security Engineering for Future Computing (SEFCOM) Laboratory, Arizona State University. His current research interest includes access control models and mechanisms, network and distributed system security, secure software engineering, and security in future home network.

Jing Jin is currently working toward the Ph.D. degree at the College of Computing and Informatics, University of North Carolina at Charlotte, Charlotte.

She is a member of the Laboratory of Information Integration, Security, and Privacy (LIISP), University of North Carolina at Charlotte. Her current research interests include access control and trust management, identity and privacy management, network and distributed system security, and security in health informatics.