Attributes Enhanced Role-Based Access Control Model

Qasim Mahmood Rajpoot[1(✉)], Christian Damsgaard Jensen[1], and Ram Krishnan[2]

[1] Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
{qara,cdje}@dtu.dk
[2] Department of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio, USA
ram.krishnan@utsa.edu

Abstract. Attribute-based access control (ABAC) and role-based access control (RBAC) are currently the two most popular access control models. Yet, they both have known limitations and offer features complimentary to each other. Due to this fact, integration of RBAC and ABAC has recently emerged as an important area of research. In this paper, we propose an access control model that combines the two models in a novel way in order to unify their benefits. Our approach provides a fine-grained access control mechanism that not only takes contextual information into account while making the access control decisions but is also suitable for applications where access to resources is controlled by exploiting contents of the resources in the policy.

Keywords: Context-aware access control · RBAC · Attributes · Content-Based access control · Role-permission explosion · Role-explosion

1 Introduction

RBAC [9] is the current standard access control model and has been a focus of research since last two decades. The RBAC paradigm encapsulates privileges into roles, and users are assigned to roles to acquire privileges, which makes it simple and facilitates reviewing permissions assigned to a user. It also makes the task of policy administration less cumbersome, as every change in a role is immediately reflected on the permissions available to users assigned to that role. A study [19] indicates that adoption of RBAC in commercial organizations is continuously increasing.

Due to the advent of pervasive systems, authorization control has become complex as access decisions may depend on the context in which access requests are made. The contextual information represents a measurable contextual primitive and may entail such information being associated with a user, object and environment [6]. For example, an access control policy may depend on the user's

© Springer International Publishing Switzerland 2015
S. Fischer-Hübner et al. (Eds.): TrustBus 2015, LNCS 9264, pp. 3–17, 2015.
DOI: 10.1007/978-3-319-22906-5_1

current location, the object being currently in a *specific state*, and the *time of day* when the access is requested. It has been recognized that RBAC is not adequate for situations where contextual attributes are required parameters in granting access to a user [16]. Another limitation of RBAC is that the permissions are specified in terms of object identifiers, referring to individual objects. This is not adequate in situations where a large number of objects in hundreds of thousands exist and leads to role-permission explosion problem. Moreover, in many applications, access to data is more naturally described in terms of its semantic contents [2], for example, in a rating system of movies, violent movies are restricted to audiences above a certain age, based on the movie contents.

A relatively new access control paradigm, ABAC [13,23] has been identified to overcome these limitations of RBAC [7]. ABAC is considered more flexible as compared to RBAC, since it can easily accommodate contextual attributes as access control parameters [16]. However, ABAC is typically much more complex than RBAC in terms of policy review, hence analyzing the policy and reviewing or changing user permissions are quite cumbersome tasks.

On one hand, both RBAC and ABAC have their particular advantages and disadvantages. On the other hand, both have features complimentary to each other, and thus integrating RBAC and ABAC has become an important research topic [7,12,14]. Also, NIST has announced an initiative [16] to integrate RBAC and its various extensions with ABAC in order to combine the advantages offered by both RBAC and ABAC. In this context, we proposed earlier the concept of an integrated RBAC and ABAC access control model [20]. In this paper, we extend it further by presenting the formal model for our Attribute Enhanced Role-Based Access Control model. We also present algorithms for two different ways in which access requests may be evaluated. Moreover, we analyze the properties of our model with the help of a scenario.

The model that we propose in this paper retains the flexibility offered by ABAC, yet it maintains RBAC's advantages of easier administration, policy analysis and review of permissions. In addition, our solution has the following key features: *a*) it allows to make context-aware access control decisions by associating conditions with permissions that are used to verify whether the required contextual information holds when a decision is made, *b*) it offers a content-based authorization system while keeping the approach role-oriented, in order to retain the advantages offered by RBAC. We achieve this by allowing to specify permissions using attributes of the objects rather than using only their identifiers.

The rest of the paper is organized as follows: Sect. 2 summarizes related work and compares our approach to prior work. In Sect. 3, we present the components of the proposed access control model while Sect. 4 presents a formal model and different possibilities in which a request may be evaluated. Section 5 discusses potential benefits offered by the proposed approach. We conclude the paper and identify future directions in Sect. 6.

2 Related Work

Kuhn et al. [16] announced a NIST initiative to incorporate attributes into roles in order to merge features of RBAC and ABAC. In response to this initiative,

Jin et al. [14] present first formal access control model called RABAC. They extend RBAC with user and object attributes and add a component called permission filtering policy (PFP). The PFP requires specification of filtering functions in the form of Boolean expression consisting of user and object attributes. Their solution is useful to address the role-explosion problem and as a result facilitates user role assignment. However, the approach does not incorporate environment attributes and is not suitable for systems involving frequently changing attributes, e.g., location and time. Also, our approach is significantly different in the sense that we make a fundamental modification in RBAC by using attributes of the objects in the permissions, addressing the issue of role-permission explosion, faced while using RABAC. Huang et al. [12] present a framework to integrate RBAC with attributes. The approach consists of two levels: underground and aboveground. The underground level makes use of attribute-based policies to automate the processes of user-role and role-permission assignment. The aboveground level is the RBAC model, with addition of environment attributes, constructed using attribute-based policies. Their work is different than ours in that it focuses on automated construction of RBAC. Xu and Stoller [22] focus on migration of RBAC-based systems to ABAC in order to avoid limitations of RBAC. They present a solution to mine attribute-based policies from an already configured RBAC model.

Several efforts have been reported which extend RBAC to include the context of access. Some of the key works in this area include environment roles [4], spatio-temporal RBAC [21] and context-aware RBAC [17]. However these approaches typically require creation of a large number of closely related roles, causing the role-explosion problem. Ge et al. [11], and Giuri et al. [10] focus on resolving the issue of role explosion by providing the mechanism of parametrized privileges and parametrized roles. However, the permissions in these solutions refer to objects using their identifiers. Few approaches propose a variant of RBAC categorizing the objects into groups or types in an attempt to resolve the role-permission explosion issue [5,15,18]. Grouping the objects allows to associate a single attribute with each object. The permissions are then specified using the group attribute – referred to as views in [15] and object classes in [5] – where each permission refers to a set of objects in that group. Moreover, as the number of object attributes grow, the number of groups increase exponentially. This makes task of policy administration cumbersome since for every new object to be added in the system it has to be associated with all those groups to which it belongs. Another area of research relevant to ours is content-based access control, where access to a resource is dependent on the information contained within the resource. Prior literature mainly uses attribute-based approaches to handle this requirement [1,2]. However, these approaches suffer from the ABAC limitations, discussed earlier. Using a combination of roles and attributes may help in simplifying the management and policy modification, as discussed in Sect. 5.

3 Overview of the Proposed Model

This section presents an overview of the proposed Attributes Enhanced Role-Based Access Control model (AERBAC). Figure 1 depicts our access control

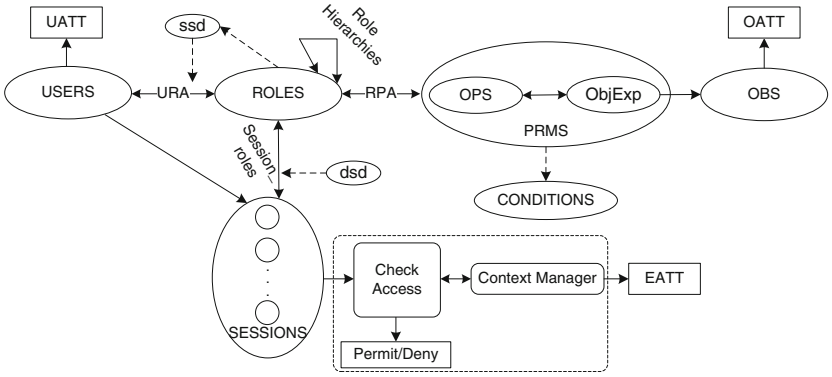

Fig. 1. Attributes enhanced role-based access control (AERBAC) model

model and its components. The entities users, roles, objects and operations have the same semantics as in RBAC. Users and objects in our model are associated with attributes too. We also incorporate the environment attribute to fully capture the situation in which access needs to be authorized. The dotted-box in Fig. 1 represents the modules of the architectural design to enforce this model. Below, we first describe the attributes and then discuss semantics of different components involved in AERBAC, including permissions, conditions, sessions and request evaluation.

Attributes: Attributes capture the properties of specific entities (e.g. user). We define an attribute function for each attribute that returns the value of that attribute. Each attribute is represented by a range of finite sets of atomic values. For example, the range of branch attribute is a set of names of branches semantically relevant for the application domain. *User attributes* capture the properties of the user who initiates an access request. Examples of user attributes are title, specialization, location, security clearance etc. *Object attributes* are used to define the properties of the resources protected by the access control policy. Examples of object attributes include type, status, location, time of object creation etc. *Environment attributes* capture external factors of the situation in which the access takes place. Temperature, occurrence of an incident, system mode or other information which not only pertains to a specific object or user, but may hold for multiple entities, are typically modeled as environment attributes.

An attribute may be either *static* or *dynamic*. The values of *static* attributes rarely change e.g. designation, department, type etc. On the other hand, *dynamic* attribute values may change frequently and unpredictably, so they may well change during the lifetime of a session. Examples of such attributes include officer in command, location, occurrence of an incident etc. They are also referred to as contextual attributes in the literature [6].

Permissions and Conditions: In contrast to the traditional approaches in RBAC, the permissions in AERBAC refer to objects indirectly, using their attributes. A permission refers to a set of objects sharing common attributes,

e.g. type or branch, using a single permission, in contrast to separate permissions for each unique object. This is particularly relevant in those domains where several objects share common attribute values. This helps in significantly reducing the number of permissions associated with a role, while increasing the expressiveness and granularity of access control in a role-centric fashion.

In our proposed model, a permission consists of an object expression and an authorized operation on the object set denoted by the expression. Object expressions are formed using the attributes of objects. Each permission is associated with one or more conditions, which must be evaluated to be true in order for the user to exercise that permission. A condition associated with a permission may contain attributes of all entities including users, objects and environment. In some applications, it is required to compare user and object attributes – for example, in a bank, a manager of a branch is allowed to access only those accounts belonging to his own branch. The proposed model allows to perform such comparisons using conditions.

An example of a permission is: p= *((oType(o) = secret* $\wedge$ *oStatus(o) = active), read)* which states that a role having this permission can perform read operation on the objects denoted by the given object expression. Here *oType* and *oStatus* are object attribute functions that return the values of respective attributes for a given object. Suppose that the permission p is constrained by a condition c= *(uMember(u) = premium* $\wedge$ *time_of_day()* $\leq$ *uDutyExpire(u))* where *uMember* and *uDutyExpire* are user attribute functions that return the attribute values of a given user, whereas *time_of_day()* is an environment attribute function. This condition implies that, in order to be granted the permission p, the user must be a premium user and time of access must be before the end of user's duty timing.

The Context Manager is responsible for propagating the updated values of dynamic attributes of the users, objects and environment. Depending on the application, some of these attribute values may also be provided by the user while placing an access request, however the application must ensure the authenticity of such information before using it in access decisions.

Session: A session contains a list of permissions associated with the roles activated by the user. As described earlier, the permissions are different from standard RBAC permissions in terms of referring to the objects using their attributes and being tied with the conditions that are evaluated every time a permission is to be exercised. Hence, the CheckAccess function needs to be re-defined.

Access request: An important consideration, in environments motivating the proposed approach, is that the user's request may also be based on the attributes of the objects. For instance, in a medical imaging application, a user might want to view all images containing specified characteristics e.g., objects with *type = tumor* and *domain = hospital-nw*. For a user request to be granted, there must exist an object expression in the user's session that denotes the requested objects, and the condition tied to that object expression must be evaluated to be true. There are different possibilities in which such a request may be evaluated and we discuss them later in the paper (cf. Sect. 4.1).

Table 1. Sets and Functions used in AERBAC

- USERS, ROLES, OBS, and OPS (users, roles, objects and operations respectively)
- URA $\subseteq$ USERS $\times$ ROLES, a many-to-many mapping of user-to-role assignment;
- SESSIONS, the set of sessions;
- user_sessions(u: USERS) $\rightarrow 2^{\text{SESSIONS}}$, the mapping of user u onto a set of sessions;
- session_roles(s: SESSIONS) $\rightarrow 2^{\text{ROLES}}$, the mapping of session s onto a set of roles. Formally: session_roles(s_i) $\subseteq$ { r $\in$ ROLES | (session_user(s_i), r) $\in$ URA};
- avail_session_perms(s: SESSIONS) $\rightarrow 2^{\text{PRMS}}$, the permissions available to a user in a session.

- UATT, OATT and EATT represent finite sets of user, object and environment attribute functions respectively.
- For each *att* in UATT $\cup$ OATT $\cup$ EATT, Range(*att*) represents the attribute's range, a finite set of *atomic* values.
- attType: UATT $\cup$ OATT $\cup$ EATT $\rightarrow$ {setType, atomicType}, specifies attributes as set or atomic valued.
- OBJ_EXP = Set of all object expressions formed using the language given in Table 2.
- COND = Set of all conditions formed using the language given in Table 2.
- PRMS = $2^{(\text{OPS} \times \text{OBJ\_EXP})}$, the set of permissions.
- RPA $\subseteq$ ROLES $\times$ PRMS $\times$ COND
- Each attribute function in UATT, OATT and EATT returns either atomic or set values.

$$\forall ua \in \text{UATT}.\, ua : \text{USERS} \rightarrow \begin{cases} \text{Range(ua)} & \text{if attType}(ua) = \text{atomicType} \\ 2^{\text{Range(ua)}} & \text{if attType}(ua) = \text{setType} \end{cases}$$

$$\forall oa \in \text{OATT}.\, oa : \text{OBS} \rightarrow \begin{cases} \text{Range(oa)} & \text{if attType}(oa) = \text{atomicType} \\ 2^{\text{Range(oa)}} & \text{if attType}(oa) = \text{setType} \end{cases}$$

$$\forall ea \in \text{EATT}.\, ea \rightarrow \begin{cases} \text{Range(ea)} & \text{if attType}(ea) = \text{atomicType} \\ 2^{\text{Range(ea)}} & \text{if attType}(ea) = \text{setType} \end{cases}$$

4 Formal AERBAC Model

In this section, we propose the formal model that incorporates the attributes of the user, object and environment into RBAC in a role-oriented fashion. We define the sets and functions used in AERBAC in Table 1. The upper part of the table shows the sets and functions defined in NIST RBAC which are also applicable to AERBAC. We provide further sets and functions needed for AERBAC in the lower part of the table. UATT, OATT and EATT represent sets of attribute functions for users, objects and environment, respectively. The notion we used for attribute representation is adapted from [13]. We use first order logic to make formal descriptions, and follow the convention that all unbound variables are universally quantified given as Range(att). Each attribute function returns

Table 2. Language to form object expressions and conditions

$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid (\varphi) \mid$ set setcompare set | atomic $\in$ set | atomic atomiccompare atomic setcompare ::= $\subset$ | $\subseteq$ | $\nsubseteq$ atomiccompare ::= $<$| $=$ | $\leq$ | $\neq$
To define an object expression, set and atomic are as follows: – set::= setoa(o:OBS) | ConsSet – atomic::= atomicoa(o:OBS) | ConsAtomic – setoa $\in$ {oa | oa $\in$ OATT $\wedge$ attType(oa) = setType} – atomicoa $\in$ {oa | oa $\in$ OATT $\wedge$ attType(oa) = atomicType}
For condition specification, set and atomic are as follows: – set::= setua (session_user(se)) | setoa(o:OBS) | setea() | ConsSet – atomic::= atomicua (session_user(se)) | atomicoa(o:OBS) | atomicea() | ConsAtomic – setua $\in$ {ua | ua $\in$ UATT $\wedge$ attType(ua) = setType } – atomicua $\in$ {ua | ua $\in$ UATT $\wedge$ attType(ua)= atomicType } – setoa $\in$ {oa | oa $\in$ OATT $\wedge$ attType(oa) = setType} – atomicoa $\in$ {oa | oa $\in$ OATT $\wedge$ attType(oa) = atomicType} – setea $\in$ {ea | ea $\in$ EATT $\wedge$ attType(ea) = setType} – atomicea $\in$ {ea | ea $\in$ EATT $\wedge$ attType(ea) = atomicType}

either a set or an atomic value, determined based on the type of the attribute (i.e. attType). Attribute functions in UATT and OATT take as an argument a user and an object, respectively. Each attribute functions in EATT may or may not require an argument, depending on the attribute and the target system. For instance, in a banking system with multiple branches, an environment attribute function would require the branch name to return the value of an environment attribute, e.g., current-system-load, in that branch.

The role-permission assignment (RPA) relation captures permissions that are assigned to a role when a given set of conditions are fulfilled. Clearly, the permission set may change for a role if the conditions vary between requests. Permissions in AERBAC are specified using object expressions. The language to define an object expression and a condition is given in the first part of Table 2. The second part of the table specifies how instances of *set* and *atomic* may be formed to define an object expression and a condition. *ConsSet* and *ConsAtomic* are constant sets and atomic values. The object expressions may be specified using only attributes of the objects. While for specifying a condition, attributes of user, object and environment may be used. The function sesseion_user(se) is defined in NIST RBAC [9] that returns the user to whom a given session *se* belongs to.

4.1 Access Decisions

The main role of the access control mechanism is to verify whether a user u, requesting access to object o, using an operation op, is authorized to do so. As mentioned above, a user request can either explicitly specify an object, by listing its identifier, or can implicitly denote a set of objects using the attributes of the objects. If the user request is not for a specific object but rather a set of objects, the system must consider the given criteria to return the requested objects. Once a user submits an access request, the request is to be evaluated against the policy. The function checkAccess in RBAC needs to be modified such that it takes the user request as input, processes the request as per the format of a given request, and returns the result. In the following, we elaborate on evaluation of both identifier-based and attribute-based requests.

(a) Identifier-based request: In identifier-based request, the user specifies the identifier of the object to be accessed. The evaluation of such type of request is straight-forward. In this case, the input of the function checkAccess consists of a session se, an operation m, and an object obj. Recall that a permission consists of an object expression and an operation and is constrained by a condition. The checkAccess function returns true if and only if (i) there exists a permission p, in the $avail\_session\_perms$ of session se, that contains an object expression which evaluates to true for obj, (ii) m matches op, and iii) the corresponding condition c evaluates to true.

(b) Attribute-based request: Using the second form of request, user may specify the attributes of the object in his/her request, rather than a unique identifier of the object. Specifying the object attributes in the request implies that the user wishes to access all those objects which have the specified attribute values. Below we discuss two possibilities to formulate and process such requests.

(b.1) Resource query: In this approach, user request contains an expression similar to the object expressions. An example user request could be: $Req = < se, (otype = secret \wedge odept = admin \wedge ostatus = inactive), write >$ which states that the owner of the session se wishes to exercise the write operation on the objects denoted by the given object expression. The checkAccess function receives as input the access request Req and returns the authorized objects to the user, if request is granted, otherwise the request is denied. The given expression is converted to a query and the resulting objects are retrieved from the resource database. Next step is to find the applicable object expressions by matching the user's requested operation with the ones mentioned in the permission set existing in user's session. Once the object expressions are shortlisted, they are evaluated one-by-one for each object returned by the query. If an object expression and its corresponding condition evaluate to true for an object, the object is added into the list of authorized objects to be granted to the user. Finally, user is granted access to all those objects for which an object expression and its corresponding condition return true. Figure 2 presents algorithm for this approach. Since the object expressions are to be evaluated for each returned object, this approach may prove to be expensive in cases where several objects are returned by the query formed based on user's request.

Algorithm 1

Input: An access request: Req = <se, re, m >consisting of session identifier *se*, request expression *re*, and operation *m*.
Output: 1) Accept and return authorized objects, 2) Reject otherwise
Begin:

```
relevant_expressions = Φ;
object_set = Φ;
authorized_objects = Φ;
object_set = search_objects*(re);
if object_set ≠ Φ then
    for all perm<object_exp, op>∈ avail_session_perms do
        if m = op then
            relevant_expressions ← relevant_expressions ∪ object_exp;
        end if
    end for
    for all object ∈ object_set do
        for all object_exp ∈ relevant_expressions do
            if evaluate†(object_exp, object) then
                if eval_cond‡(condition, object, session_user(se)) then
                    authorized_objects ← authorized_objects ∪ object;
                    break;
                end if
            end if
        end for
    end for
end if
if authorized_object ≠ Φ then
    return authorized_objects;
end if
return Reject;
```

End

* search_objects(re) returns a set of objects existing in the resource database that are denoted by the constraints specified in expression *re*, in the request.
† evaluate(object_exp, object) returns TRUE if *object_exp* evaluates to true for the given *object*, else returns FALSE.
‡ eval_cond(condition, object, session_user(se)) returns TRUE if given *condition* evaluates to true for the given *object* attributes and the attributes of the user and the environment.

Fig. 2. Algorithm for access request evaluation using resource query

(b.2) Attribute values: An alternative strategy is to evaluate the user's request against the object expressions before retrieving the actual objects from the resource database. In this approach, rather than providing an expression, user specifies his/her access request by specifying the object attribute values of the desired objects. The checkAccess function receives as input the user request *Req*

and returns the objects denoted by object attribute values given in *Req*, if request is granted, otherwise the request is denied. To process user request, all

Algorithm 2

Input: An access request: Req = < se, obj_att_values, m> consisting of session identifier *se*, object attribute values *obj_att_values*, and operation *m*.
Output: 1) Accept and return authorized objects, 2) Reject otherwise
Begin:

```
relevant_expressions = Φ;
authorized_objects = Φ;
for all perm < object_exp, op > ∈ avail_session_perms do
    if m = op ∧ check_relevancy*(obj_exp, obj_att_values) then
        if evaluate† (object_exp, obj_att_values) then
            if eval_cond‡(condition, obj_att_values, session_user(se) then
                authorized_objects = get_objects††(obj_att_values);
            end if
        end if
    end if
end for
if authorized_object ≠ Φ then
    return (Accept, authorized_objects)
end if
return (Reject)
```

End

* check_relevancy(object_exp, obj_att_values) returns TRUE if the given *object_exp* uses only those object attribute functions referred in *obj_att_values*
† evaluate(object_exp, obj_att_values) returns TRUE if the given *object_exp* evaluates to true when the object attribute functions are replaced with *obj_att_values*
‡ eval_cond(condition, obj_att_values, session_user(se)) returns TRUE if the given *condition* evaluates to true for the given object attributes and the attributes of the user and environment
†† get_objects(obj_att_values) returns a set of objects existing in the resource database that satisfy *obj_att_values*

Fig. 3. Algorithm for access request evaluation using attribute values

those object expressions existing in user's session are identified which use the attributes mentioned in the user's request and the operation specified in that permission matches with requested operation. Object expressions that include an attribute not specified by the user request are not relevant. Next, for each shortlisted object expression, the attribute functions in the object expression are given the user provided attribute values. For instance, if a user specifies the following object attribute in his/her request: *(otype = classified; odept = pg; ostatus = active)* and suppose we find an object expression as follows: *(otype(o) = classified ∧ odept(o) ⊆ {pg, ug, admin})*. Upon picking the values of the object attribute functions *otype* and *odept* from user given attribute values we get: *(classified = classified ∧ pg ⊆ {pg, ug, admin})* which would evaluate to

true. As soon as an object expression and its corresponding condition return true, the user's request is granted and rest of the object expressions are ignored. When an expression returns true we form a query based on the object attribute values specified in the user request and the user is granted access to all those objects returned by the query. Algorithm for this approach is given in Fig. 3.

Note that we never evaluate an object expression which uses an object attribute not given in the user's request. This is because we replace the object attribute functions with the user given attribute values, hence any object expression involving those object attributes not given by the user cannot be evaluated. The query to get the authorized objects is formed using the object attributes mentioned in the user's request. Once an object expression returns true, this query may restrict the list of returned objects based on any additional attributes mentioned in the user's request. In the example above, the returned result is restricted based on additional object attributes *ostatus* which are mentioned in the user's request but does not exist in the expression which enables the request.

This approach is superior to resource query in terms of making an access decision by evaluating only the object expressions, without having to retrieve objects from the resource database. This is important, since many requests can be denied at this point without the overhead of object retrieval and condition evaluation. An obvious assumption made in this form of user request is that the multiple object attributes mentioned in the user request are always combined using logical conjunction operator.

5 Discussion

To illustrate the features of the proposed access control model, we present an example below, inspired from the online entertainment store example presented in [23]. Suppose an online entertainment store streams movies to subscribed users. Suppose, there are two different types of users; Adult and Juvenile. Adult users can view all movies while Juvenile can view only G-rated movies. Using the standard RBAC approach, clearly we need two roles to represent *Juvenile* and *Adult* users. In each role the permissions have to be specified using identifiers of the objects individual movies. Considering that there may exist thousands of movies in the database, referring each with its identifier would lead to role-permission explosion problem. To address this issue, AERBAC integrates roles and attributes in a novel way and uses the attributes of the objects in the permissions rather than identifiers of individual objects. Table 3 provides an example where permissions make use of object attributes. In this example, the role *Adult* is inherited by *Juvenile* role and hence inherits permissions assigned to *Juvenile* role.

In order to model multiple characteristics associated with user, object or environment, the number of roles in RBAC increase exponentially. Suppose we want to ensure that only premium users may view newly released movies and regular users may view newly released movies only during promotional periods. To represent these conditions in standard RBAC, we would need to create at least six roles: Adult_premium, Adult_promo, Adult_regular, Juvenile_premium, Juvenile_promo and Juvenile_regular, where Adult_promo

Table 3. Permissions in AERBAC

Role	Permissions
Adult	(view, (rating(m) = R))
Juvenile	(view, (rating(m) = G))

Table 4. Example configuration using AERBAC

Role	Permissions	Conditions
Adult	(view, (rating(m) = R ∧ release(m) = new))	(userType(u) = premium ∨ today ∈ PromoDates)
	(view, (rating(m) = R ∧ release(m) = old))	None
Juvenile	(view, (rating(m) = G ∧ release(m) = new))	(userType(u) = premium ∨ today ∈ PromoDates)
	(view, (rating(m) = G ∧ release(m) = old))	None

and Juvenile_promo roles would be available to users only during promotional periods. Configuring this using AERBAC, we need only two roles: Adult and Juvenile as we use attributes of objects in the permissions and other attributes in the condition corresponding to each permission. Table 4 provides the configuration of this scenario using the proposed approach.

Our motivation to integrate RBAC with attributes is to obtain advantages associated with both RBAC and ABAC, while addressing the limitations of RBAC and ABAC. Using a pure ABAC approach, in configuring situation such as above requires writing policy rules. When a user request needs to be evaluated, the relevant rules are identified using the attributes associated with requesting user, requested object and current environment. These shortlisted rules are then evaluated one-by-one unless we find a rule which allows the request. In contrast, our approach requires evaluation of only those object expressions which are associated with the roles activated by a user in his/her session. Note that this may significantly reduce the number of rules to be evaluated. Moreover, the user or environment attributes used in the conditions are evaluated only if an object expression evaluates to true for a given request. This is particularly useful in cases where user or environment attributes are dynamic and their current values are reported at the time of request evaluation. In our approach, such values would only need to be obtained if an object expression in the user's session returns true. This indicates that many user requests may be denied, just by evaluating object expressions, without obtaining the current values for user and environment attributes.

5.1 Merits of the Proposed Model

As discussed above, the object expressions and conditions that are to be evaluated against a user request are determined by the roles a user activates in a session. Imagine a user assigned to a senior executive role in an organization

which has several privileges. For a user in this role, we might allow to access specific resources without giving any consideration to the time of request and location of user, for instance. This implies that there may be some attributes which are not relevant for a given role and hence the number of conditions and object expressions to be evaluated for that role may be reduced.

Compared to ABAC, our approach provides a systematic mechanism to evaluate a subset of policy rules which are determined based on the user's roles, yet retaining the advantages offered by RBAC including quick assignment and revocation of roles to users, reviewing of permissions assigned to a user or role, and reduced complexity of administration in large organizations. Moreover, we believe several limitations of the RBAC and ABAC approaches may be overcome using the approach we proposed. Below, we enlist some of these limitations and discuss how our approach overcomes these problems.

1- Fine-grained Access Control: RBAC provides a coarse-grained access control model where as many applications require a much finer-degree of granularity [8]. In order to satisfy the requirements posed by such applications, a large number of roles have to be created when pure RBAC is used. Using the proposed approach, we may provide a finer-grained access control mechanism without creating a large number of roles. As discussed in the example, we achieve this by associating conditions at permission level to check further attributes associated with a user and environment rather than granting a permission merely based on being a member of a role.

2- Context-aware Access: RBAC cannot easily handle dynamically changing attributes [7]. It typically does not support making contextual decisions unless many similar roles are created causing role-explosion problem. We provide a mechanism to incorporate these dynamically changing attributes in a role-centric manner yet without requiring to create a large number of roles. An important feature of our approach is checking the values of such attributes at the time of granting access rather than checking them at the time of session creation as done typically in RBAC.

3- Easy Auditing: When ABAC is used in a considerably large organization having a large number of policy rules, it may not be practically feasible to audit what permissions have been granted to a user. In ABAC, any combination of attributes may essentially grant an access and hence it requires to analyze all policy rules with an exhaustive enumeration of attributes used in each policy rule [7]. Our approach makes it simpler to audit what permissions may be granted to a user because of being role-centric while adding the flexibility and fine-grained access features offered by ABAC. When auditing for a particular position or employee, we need to consider only the policy rules given in the roles assigned to that position or employee.

4- Policy Modification Visualization: One of the issues in the ABAC approach is that the consequences of a newly added or removed policy rule are not easy to visualize [3]. It is not clear what set of users will be effected by a change in the policy. A change in policy essentially may affect those users who

we wish to remain authorized to access a particular resource but they are no more authorized since a policy rule is removed. In our approach, it is relatively easy to visualize what is the impact of adding or removing a policy since policy specification is at the level of role. Therefore, a change in policy can effect only those users who are assigned to a role being modified.

6 Conclusion

In this paper, we proposed an access control model that integrates RBAC and ABAC bringing together the features offered by both models. In our model, the attributes may be associated with users, objects and environment allowing the request context to be considered in making access control decisions. Unlike traditional RBAC approaches, permissions in our model consist of operations and object expressions enabling content-based access control. We presented different request evaluation mechanisms that may be used by various applications depending on their requirements. We demonstrated the merits of the proposed model in the discussion section using a scenario. In the future, we plan to work on formally analyzing the properties offered by the proposed model as compared to existing access control model including ABAC and RBAC, and to develop an XACML profile of the proposed model. Further directions for future work include use of cache mechanisms to further expedite the access control decision process, to extend the model with continuous enforcement to deactivate a role or revoke a permission when context conditions fail to hold, and to include negative authorizations in the model.

Acknowledgments. The work of first two authors is supported by a grant from the Danish National Advanced Technology Foundation. The work of the third author is supported by a US National Science Foundation grant CNS-1423481.

References

1. Adam, N.R., Atluri, V., Bertino, E., Ferrari, E.: A content-based authorization model for digital libraries. IEEE Trans. Knowl. Data Eng. **14**(2), 296–315 (2002)
2. Bertino, E., Moustafa A.H., Walid A.G., Elmagarmid, A.K.: An access control model for video database systems. In: International Conference on Information and Knowledge Management, pp. 336–343. ACM (2000)
3. Best Practices in Enterprise Authorization: The RBAC/ABAC Hybrid Approach (EmpowerID). http://blog.empowerid.com/Portals/174819/docs/EmpowerID-WhitePaper-RBAC-ABAC-Hybrid-Model.pdf
4. Covington, M.J., Long, W., Srinivasan, S., Dev, A.K., Ahamad, M., Abowd, G.D.: Securing context-aware applications using environment roles. In: Symposium on Access Control Models and Technologies, pp. 10–20. ACM (2001)
5. Chae, J.H., Shiri, N.: Formalization of RBAC policy with object class hierarchy. In: Dawson, E., Wong, D.S. (eds.) ISPEC 2007. LNCS, vol. 4464, pp. 162–176. Springer, Heidelberg (2007)

6. Covington, M.J., Sastry, M.R.: A contextual attribute-based access control model. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2006 Workshops. LNCS, vol. 4278, pp. 1996–2006. Springer, Heidelberg (2006)
7. Coyne, E., Weil, T.R.: ABAC and RBAC: scalable, flexible, and auditable access management. IT Prof. **15**(3), 14–16 (2013)
8. Fischer, J., Marino, D., Majumdar, R., Millstein, T.: Fine-grained access control with object-sensitive roles. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 173–194. Springer, Heidelberg (2009)
9. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. (TISSEC) **4**(3), 224–274 (2001)
10. Giuri, L., Iglio, P.: Role templates for content-based access control. In: Workshop on Role-Based Access Control, pp. 153–159. ACM (1997)
11. Ge, M., Osborn, S.L.: A design for parameterized roles. In: Farkas, C., Samarati, P. (eds.) Data, Application Security and Privacy Conference. IFIP, vol. 144, pp. 251–264. Springer, Heidelberg (2004)
12. Huang, J., Nicol, D.M., Bobba, R., Huh, J.H.: A framework integrating attribute-based policies into RBAC. In: Symposium on Access Control Models and Technologies, pp. 187–196. ACM (2012)
13. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 41–55. Springer, Heidelberg (2012)
14. Jin, X., Sandhu, R., Krishnan, R.: RABAC: role-centric attribute-based access control. In: Kotenko, I., Skormin, V. (eds.) MMM-ACNS 2012. LNCS, vol. 7531, pp. 84–96. Springer, Heidelberg (2012)
15. Kalam, A.A.E., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miege, A., Saurel, C., Trouessin, G.: Organization based access control. In: 4th International Workshop on Policies for Distributed Systems and Networks. IEEE (2003)
16. Kuhn, D.R., Coyne, E.J., Weil, T.R.: Adding attributes to role-based access control. IEEE Comput. **43**, 79–81 (2010)
17. Kulkarni, D., Tripathi, A.: Context-aware role-based access control in pervasive computing systems. In: Symposium on Access Control Models and Technologies, pp. 113–122. ACM (2008)
18. Moyer, M.J., Abamad, M.: Generalized role-based access control. In: International Conference on Distributed Computing Systems, pp. 391–398. IEEE (2001)
19. O'Connor, A.C., Loomis, R.J.: Economic Analysis of Role-Based Access Control. NIST Report (2010)
20. Rajpoot, Q.M., Jensen, C.D., Krishnan, R.: Integrating attributes into role-based access control. In: Samarati, P. (ed.) DBSec 2015. LNCS, vol. 9149, pp. 242–249. Springer, Heidelberg (2015)
21. Ray, I., Toahchoodee, M.: A spatio-temporal role-based access control model. In: Barker, S., Ahn, G.-J. (eds.) Data and Applications Security 2007. LNCS, vol. 4602, pp. 211–226. Springer, Heidelberg (2007)
22. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies from RBAC policies. In: 10th International Conference and Expo on Emerging Technologies for a Smarter World (CEWIT), pp. 1–6. IEEE (2013)
23. Yuan, E., Tong, J.: Attributed Based Access Control (ABAC) for Web Services. In: International Conference on Web Services. IEEE (2005)