Security Policy Management for Systems Employing Role Based Access Control Model

Chao Huang, Jianling Sun, Xinyu Wang and Yuanjie Si
College of Computer Science, Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang, 310027, China

Abstract: In this study, we propose the redundancy and inconsistency checking algorithms to support the policy management of systems employing role based access control model. Present method is based on the formal definition of the policy redundancy and policy inconsistency. Via constructing the role graph, we analyze the redundancy and inconsistency one by one. According to the features of each type of redundancy and inconsistency, present algorithm checks all the possible violations and generates the related policy elements to help the security administrator to amend the policy afterwards. The performance test demonstrates that the approach is efficient enough for practical usage. Present approach could guarantee the conciseness as well as consistency of the access control policy, at same time reduce the burden of access control administration significantly.

Key words: Policy inconsistency, policy redundancy, separation of duty, cardinality constraint, role hierarchy, policy checking

INTRODUCTION

The prominent progress of Internet and related technologies has promoted tremendous information and data sharing. There is a growing concern for security and privacy of data and numerous studies have shown that unauthorized access can cause great losses, especially for the financial software (Ferraiolo et al., 2003; Schaaf et al., 2001). Access control becomes more and more essential for safe and secure access to the software and hardware resources. Role Base Access Control (RBAC) model, which was proposed by Ferraiolo and Sandhu (2001) at the 90s of last century, provides a powerful way to satisfy the access control needs (Li and Tripuntara, 2006). RBAC96 is currently the most widely used access control model in enterprise area since its fine grained control over the privilege (Ferraiolo et al., 2001; Li and Mao, 2007). Furthermore, RBAC is capable of modeling a wide range of access control policies including Discretional Access Control (DAC) and Mandatory Access Control (MAC). An access control policy is a statement which specifies the rules about how to setup the process for granting or denying authorizations to the users (Coyne, 1996; Bezrakov and Deng, 1999). The access control redundancy means there are unwanted specifications about the same access rule and the access control inconsistency means there are access control rules conflicting with each other in the policy, for example rule A claims that granting role R to user Bob, however rule B claims that denying the request of assigning R to Bob, when the above access control rules coexist in one policy, the access control policy inconsistency occurs, which may conceal the security danger or even breakdown the entire access control system (Schaaf et al., 2001; Nyanchara and Osborn, 1999). Thus, it is extremely important to make sure that there is no inconsistency of an access control policy. For the access control system employing RBAC model, it is much more complicated for the policy inconsistency management, since RBAC supports more advanced constraints.

Cenonze et al. (2006) present a theoretical foundation for correlating an operation-based RBAC policy with a data-based RBAC policy via., a location consistency property. The research shows how to infer whether an operation-based RBAC policy is equivalent to any database RBAC policy and a static analysis tool called SAVES is introduced to facilitate the consistency validation. However, the inconsistencies concerning role hierarchies and cardinality constraints are not discussed. Chang and Hoh (2005) discussed the inconsistency detection in distributed component environment. Via analyzing the server-centric RBAC system, they proposed an inconsistency detection method between two access control policies and also investigated the conditions of guaranteeing consistency. However, there is no the discussion of the inconsistency in each individual distributed system, which is actually the pre-condition of the consistency in the entire distributed system. Strembeck (2004) presents the experiences with the implementation of conflict-checking methods for

Corresponding Author: Chao Huang, College of Computer Science, Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang, 310027, China
separation of duty constraints in the XORRBAC access control service and only provides a concise method to check the separation of duty conflicts, which is obviously not sufficient to make sure the consistency of system. All of the above work addressed little on the policy redundancy, which is obviously not safe and efficient enough for the policy management.

The redundancy and inconsistency need a systematic way (Li et al., 2007). In this study, pointing to the problems listed above we take all of the elements of a RBAC policy including role hierarchies, separation of duty constraints and cardinality constraints into consideration and present the formal definition of the policy redundancy and inconsistency. We also show the algorithm to check the redundancy and inconsistency of RBAC access control policy which is based on the Tarjan’s SCC algorithm (Tarjan, 1972). Compared to the methods by Chang and Hoh (2005) and Strembeck (2004), present will provide much more confidence for the consistency in RBAC policy.

The purpose of this study is to develop a comprehensive algorithm to check the policy redundancy and inconsistency to support the management of RBAC policy. The algorithm should take all the policy constraints into account and due to the frequent policy checking in daily management, the algorithm should be efficient enough for practical application.

BACKGROUND

RBAC is widely used for the specification of security requirement of most commercial applications (Yao et al., 2001; Park et al., 2001; Esmayr et al., 2004). RBAC96 model currently being used consists of the following four basic components: a set of users, a set of roles, a set of permissions and a set of sessions. A user is a human being or a process within a system. A role is a collection of permissions associated with a certain job function. Permission defines the access rights that can be exercised on a particular object in a system. A session relates a user to several roles. The whole picture of RBAC96 model is as shown in Fig. 1.

Definition 1

RBAC policy: An RBAC policy is a 7-tuple (U, R, P, RH, RP, UR, C):

- U, R, P are respectively, finite sets of users, roles and permissions, where P is defined as \( P = 2^{\text{permission} \cup \text{Object}} \)
- RH: Role hierarchy is a partial order relation defined on role sets, \( RH \subseteq (R \times R) \), if \( (r_i, r_j) \in RH \), \( r_j \) is a senior role whereas \( r_i \) is junior role and \( r_j \) inherits all the permissions of \( r_i \). The relation can also be expressed as: \( r_j \succ r_i \)

![Fig. 1: RBAC96 model](image)

- RP: Role-permission is a relation defined on role set and permission set, \( RP \subseteq (R \times P) \)
- UR: User-pole is a relation defined on the user set and role set, \( UR \subseteq (U \times R) \)
- C: The constraint set defined on the above entity and relation sets, including the separation of duty and cardinality constraint

RBAC global functions:

- senior: \( R \rightarrow 2^R \), mapping each role to a set of roles, which are senior to the role
- role_u: \( R \rightarrow 2^U \), mapping each role to a set of users who own the role, including the ones who own the role via, role hierarchy
- role_p: \( R \rightarrow 2^P \), mapping each role to a set of permissions which are assigned to the role, including the ones which are inherited from senior roles
- user_p: \( U \rightarrow 2^P \), mapping each user to a set of permissions which are owned by the user
- user_r: \( U \rightarrow 2^R \), mapping each user to a set of roles which are assigned to the user
- pms_u: \( U \rightarrow 2^R \), mapping each permission to a set of users who owns the permission
- pms_r: \( P \rightarrow 2^R \), mapping each permission to a set of roles, which have been assigned the permissions

Separation of Duty (SOD) is a fundamental technique to prevent fraud and errors, known and practiced long before even the existence of computers. SOD reduces the possibility of fraud or significant errors which may cause damage to an organization. By partitioning the tasks and the associated privileges, the accomplishment of a sensitive task requires the cooperation of multiple users imperatively. SOD constraint is not only the most important constraint in RBAC policy but also the most complicated part which causes the management of RBAC policy more difficult. There are three kinds of separation of duty in RBAC, permission separation SOD-P, role separation SOD-R and user separation SOD-U.
Definition 2
Separation of duty:

- **SOD-P**: SOD-P is the finest grained control of separation of duty, which specifies the exclusive set of permission

  \[ \forall n \geq 2, \forall (p, s) \in \text{SOD-P}, \forall p' \subseteq p | p \cap n = \cap_{\text{perm}} \text{perm}_{-}(p') = \emptyset \]

  If \( n = 1 \), we can use \((p, p_0) \in \text{SOD-P}\) for simplicity.

- **SOD-R**: SOD-R is the constraint over the role set, which specifies the user could not be assigned the exclusive roles in the meanwhile

  \[ \forall n \geq 2, \forall (r, s) \in \text{SOD-R}, \forall s \subseteq s | s \cap n = \cap_{\text{role}} \text{role}_{-}(s) = \emptyset \]

  The simplified expression can be written as \( (r, r_0) \in \text{SOD-R}\), when \( n = 1 \)

- **SOD-U**: SOD-U is the third form of separation of duty, which specifies that only one of the two users could be assigned the certain role. SOD-U \( \subseteq 2^{n} \times R\), which satisfies the following conditions:

  \( \forall (u_i, u_j, r) \in \text{SOD-U} \Rightarrow r \in \text{user}_{-} r(u_i) \land \text{user}_{-} r(u_j) \)

Besides the above SOD constraints, Cardinality Constraint (CC) is also one of the most important constraints in RBAC. Cardinality constraint regulates that at the given time, there is only one can be assigned as the certain role. For example in a company, at any time, there should be only one employee can be assigned as the general manager, otherwise there must be confusion of the management.

Definition 3
Cardinality Constraint (CC):

- **CC-P**: CC-P constraint specifies the restriction that the certain permission which can be assigned to role are only once. CC-P \( \subseteq P \), which satisfies the following condition:

  \[ \forall p \in \text{CC-P}, \Rightarrow \text{perm}_{-}(p) \leq 1 \]

- **CC-R**: CC-R constraints specifies the restriction that the specified role can only be assigned to one user. CC-R \( \in R \), which satisfies the following condition:

  \[ \forall r \in \text{CC-R}, \Rightarrow \text{role}_{-}(r) \leq 1 \]

POLICY MANAGEMENT

The major problems encountered in the management of RBAC policy are redundancy and inconsistency. Redundancy means there are unwanted specifications to describe the control rules and inconsistency means there are conflicting rules for the access control.

Take following policy \( pl \) for example:

\[ \text{pl} = (U, R, P, RH, UR, C) \]

Where:

- **U**: \{u1, u2\}
- **R**: \{r1, r2\}
- **P**: \{p1, p2, p3, p4, p5, p6\}
- **RH**: \{(r1, r2), (r2, r1), (r3, r1), (r3, r2), (r3, r3), (r3, r3), (r4, r3), (r4, r3)\}
- **RP**: \{(r5, p1), (r5, p2), (r5, p3)\}
- **UR**: \{(u1, r1), (u2, r2)\}
- **SOD-P**: \{\{p1, p2\}\}
- **SOD-R**: \{\{r1\}\}
- **SOD-U**: \{\{u1, u2, r1\}\}, CC-R(r1)

There are several redundancies and inconsistencies in the above RBAC policy. Depicting the policy \( pl \) in the Fig. 2, the redundancy of the role hierarchies among \( r_1, r_2, r_3 \) is obvious, since the hierarchy relation of \( r_1 \rightarrow r_2 \) can be deduced from \( r_1 \rightarrow r_3 \) and \( r_2 \rightarrow r_3 \). Redundancy in most situations has no dramatically negative effect on the access control. However, it will threaten the security as the evolution of the policy. For policy \( pl \), if the security administrator wants to remove the hierarchy relation

![Fig. 2: Example RBAC policy](image-url)

728
between \( r_i \) and \( r_j \), the intention of the administrator is also to cancel all the hierarchy relation which inherits from \( r_j \), however, since \( r_i \Rightarrow r_j \) exists as a redundant relation which cause the policy unconfomal.

Above RBAC policy also suffers from policy inconsistency. For the hierarchy relations among \( r_i, r_j \) and \( r_k \), it is obvious that there should be no cycle in the role graph as hierarchy is a kind of partial order relation.

**Boolean matrix for relations:** The above policy example demonstrates that the management of RBAC policy is fundamental and challenging. Boolean matrix is a good way to express the relations among the users, roles and permissions.

**Definition 4**

*Multiplication of Boolean matrix:* Boolean matrix multiplication between Boolean matrices \( A \in \{0, 1\}^{m \times n} \) and \( B \in \{0, 1\}^{n \times m} \) and \( C = A \circ B \), where \( C \in \{0, 1\}^{m \times n} \) and

\[
c_{i,j} = \sum_{k=1}^{n} (a_{i,k} \land b_{k,j})
\]

**Definition 5**

*MRH, MUR, MRP and MUP:* MRH, MUR, MRP and MUP are the matrices to represent the RH, UR, RP and user-permission relation in the policy. Setting \( I = |U| \), \( m = |R| \), \( n = |P| \)

- **MRH:** An \( m \times m \) Boolean matrix, satisfying:

\[
\forall i, j \in R, \text{MRH}(i, j) = \begin{cases} 1, & \text{if } (i, j) \in \text{RH} \\ 0, & \text{if } (i, j) \notin \text{RH} \end{cases}
\]

- **MUR:** The transitive closure of MRH, representing the complete hierarchy relations on role set

- **MRP:** An \( I \times m \) Boolean matrix, satisfying:

\[
\forall i \in R, u \in U, \text{MUR}(u, i) = \begin{cases} 1, & \text{if } (u, i) \in \text{UR} \\ 0, & \text{if } (u, i) \notin \text{UR} \end{cases}
\]

- **MUR:** An \( I \times m \) Boolean matrix, satisfying:

\[
\text{MUR}^* = \text{MRU} \circ \text{MRH}^*
\]

- **MRP:** An \( m \times n \) Boolean matrix, satisfying:

\[
\forall i \in R, p \in P, \text{MRP}(i, p) = \begin{cases} 1, & \text{if } (i, p) \in \text{RP} \\ 0, & \text{if } (i, p) \notin \text{RP} \end{cases}
\]

- **MRP:** An \( m \times n \) Boolean matrix, satisfying:

\[
\text{MRP}^* = \text{MRH}^* \circ \text{MRP}
\]

which indicate the whole role and permission relation

- **MUP:** \( l \times n \) Boolean matrix, representing the relations between users and permissions, satisfying:

\[
\text{MUP} = \text{MUR} \circ \text{MRP}^*
\]

**RBAC policy redundancy:** Redundancy means there are unwanted specifications to describe the control rules, following lists the types of the redundancy:

- **Redundancy between role hierarchies**

For \( r_i \Rightarrow r_j \), if \( \exists r_{i_0}, \ldots, r_{i_n} \in R \) satisfying:

\[
r_i \Rightarrow r_{i_0} \Rightarrow r_{i_1} \Rightarrow \cdots \Rightarrow r_{i_{n-1}} \Rightarrow r_{i_n} \Rightarrow r_j
\]

then \( r_i \Rightarrow r_j \) is redundant.

The proof is simple, since \( \Rightarrow \) is a transitive relation, via \( r_{i_0}, r_{i_1}, \ldots, r_{i_{n-1}}, r_{i_n}, r_j \) can be deducted. Figure 3a shows the role hierarchy redundancy.

- **Redundancy between SOD-P and SOD-R**

For sod-\( r_i, r_j \) \( \in \) SOD-R,

if \( \exists \text{sod-} p_1, p_2 \in \text{SOD-P} \), satisfying:

\[
p_1 \in \text{role}_p r_i \land p_2 \in \text{role}_p r_j
\]

then sod-\( r_i, r_j \) is redundant. Figure 3b shows this type of redundancy.

- **Redundancy between SOD-U and CC-R**

For sod-\( u_i, u_j \), \( r \in \) SOD-U, if \( \exists \text{sod-} r \), \( r \in \) CC-R, then sod-\( u_i, u_j, r \) is redundant. Figure 3c shows this type of redundancy.

Following is the algorithm to check the redundancies:

**Algorithm 1:** RBAC policy redundancy checking

*Input:* Policy \( P = (U, R, P, \text{RH}, \text{UR}, \text{CC}) \)

*Output:* Return false indicating there is no redundancy in the policy, otherwise return true indicating redundancy exists.

![Fig. 3a-c: RBAC policy redundancy](image-url)
For policy pl, construct the relation matrices, MRH, MUR, MRP and MUP and assuming the redundant set is RSet←∅.

Retrieve next unchecked rh = (r₁, r₂) from RH, set MRH(r₁, r₂)←0, then using the Warshall algorithm compute the transitive closure of MRH, which is MRH⁺.

- If MRH⁺(r₁, r₂) = 1, then (r₁, r₂) is a redundant role hierarchy relation, add (r₁, r₂) to the redundant set RSet, mark it as role hierarchy redundancy.
- If MRH⁺(r₁, r₂) = 0, then (r₁, r₂) is not redundant, reset MRH (r₁, r₂)←1 and continue step 2 until all the role hierarchy relation are checked.

Retrieve next unchecked sod-r (r₁, r₂), for each sod-p(r₁, p₂), if MRP⁺(r₁, p₂) = 1 or MRP⁺(r₂, p₂) = 1, then sod-r (r₁, r₂) is redundant, add it to the redundant set and mark it as SOD-R and SOD-P redundancy. Continue step 5 until all the sod-r constraints are checked.

- For each sod-u (u₁, u₂), if co-r (r) = 1, then sod-u (u₁, u₂, r) is redundant, add it to the redundant set.
- If redundant set is empty, return false, otherwise return true.

For redundancy checking, the redundancy between SOD-P and SOD-R and the redundancy of SOD-U and CC-R are easy to do, which can be accomplished according to the corresponding relation matrix, via., verifying whether the unwanted condition is true. The redundancy among the role hierarchy is more complicated than the former two, since it involves the transitive relation among roles, which cause the computation more time-consuming.

We utilize the Warshall algorithm (Floyd, 1962) to compute the research-ability between two role nodes in the policy graph. The time complexity is:

\[ O(m^3) \text{, where, } m = |R| \]

m is the role number in the policy.

The efficiency can be improved via., using DFS (Depth First Search) based reach-ability algorithm, which can reduce the time complexity to:

\[ O(n^2 + mn) \]

\[ m = |R| \text{ and } n = \sum_{i=1}^{m} \sum_{j=1}^{m} \text{MRH}[i,j] \]

where, n is the role hierarchy relation edge in the role policy graph. In practice, usually, n<<m², so DFS based reach-ability algorithm offers an advantage here.

**RBAC policy inconsistencies:** A security and safe access control policy should give the indicated responses of an access request. Any confusion or uncertainty may conceal the security danger or even breakdown the entire system. Inconsistency is the major cause of the uncertainty. RBAC policy is more complicated than DAC and MAC, since RBAC supports role hierarchy relation and more advanced constraints which facilitate the fine grained control. Figure 4 shows the inconsistencies in an RBAC policy:

- Inconsistency between role hierarchies:
  For \( r₁, r₂, r₃ \in R \), if \( r₁ \in \text{senior}(r₂) \land r₂ \in \text{senior}(r₃) \). Then there is inconsistency of the role hierarchies among \( r₁, r₂, r₃ \). In Fig. 4a, \( r₁ \) is senior to \( r₂ \), \( r₂ \) is senior to \( r₃ \), which makes all of the three roles involve the conflicts, since the junior should not get the permissions of the senior roles.

- Inconsistency between RH and SOD-R:
  For \( r₁, r₂, r₃ \in R \) and sod-r (r₂, r₃), if \( r₁ \in \text{senior}(r₂) \land r₁ \neq \text{senior}(r₃) \).

Then there is inconsistency between the role hierarchy and SOD-R constraint. In Fig. 4b, \( r₁ \) is senior to \( r₂ \) and \( r₃ \) which are restricted by the SOD-R constraint. This inconsistency will expose the system under the danger, since any user who is assigned the role \( r₁ \) can do the operations of \( r₂ \) and \( r₃ \) at the same time which is supposed to be prohibited.

- Inconsistency between RP and SOD-P
  For \( r_1 \in R \) and \( p_1, p_2 \in P \), sod-p \((p_1, p_2)\)
  if \( p_1 \in \text{role}_p(r_1) \) and \( p_2 \in \text{role}_p(r_1) \)
  Then there is inconsistency between the role-permission assignment and the SOD-P constraint. In Fig. 4c, role \( r_1 \) gains the mutual exclusive permissions via., the role hierarchy, which may cause the security danger similar to the 1 in 2.

- Inconsistency between UR and SOD-R
  For \( u \in U \), \( r_1, r_2 \in R \) and sod-r \((r_1, r_2)\)
  if \( r_1 \in \text{user}_r(u) \) and \( r_2 \in \text{user}_r(u) \)
  Then there is inconsistency between the user-role assignment and the SOD-R constraint. In Fig. 4d, user \( u \) is assigned with role \( r_1 \) and \( r_2 \), which are involved in the SOD-R relation via., role hierarchy from \( r_1 \) and \( r_2 \). Thus, \( u \) gains the exclusive permissions which is not allowed.

- Inconsistency between UR and CD-R
  For reR and co-r(r) if
  \( \exists u, u \in U, u \neq u \) and
  \( r \in \text{role}_u(u), r \neq \text{role}_u(u) \)
  Then there is inconsistency between user-role assignment and the cardinality constraint. In Fig. 4e, user \( u \) and \( u \) are assigned with role \( r_1 \), which has the cardinality constraint or inherits the cardinality constraint via., role hierarchy from \( r_1 \) which cause the inconsistency.

- Inconsistency between RP and CD-P
  For \( p \in P \) and co-p(p) if
  \( \exists r, r \in R, r \neq \text{role}_p(r) \)
  Then there is inconsistency between role-permission assignment and CD-P constraint. In Fig. 4f, role \( r_1 \) and \( r_2 \) are assigned with the same permission \( p \), which is restricted by the cardinality constraint. Such kind inconsistency will turn the system into complete confusion.

The above inconsistency definitions are the basic forms. Since, role hierarchy relation is transitive and other RBAC relations are transitive-kind relations, like the UP can be delivered via., UR and RP, there are more complicated inconsistencies which can be composed of the above basic forms.

Based on the analysis of RBAC policy inconsistency, the inconsistency checking algorithm is given as following:

**Algorithm 2: RBAC policy inconsistency checking**

**Input:** Policy \( p = (U, R, P, RH, UR, C) \)

**Output:** Return false indicating there is no inconsistency; otherwise return true indicating inconsistency exists.

- Construct role graph \( RG = (V, E) \), where, \( V = R \) and \( E = RH \) and set the inconsistency set \( ISet \)
- Find the Strong Connected Components (SCC) in \( RG \) via Tarjan’s SCC algorithm, if the output is empty set then there is no role hierarchy inconsistency, otherwise, there are inconsistencies among the role hierarchy and the output set contains the roles which cause the inconsistency
- Retrieve next unchecked sod-p \((p_1, p_2)\), for each reR, if \( MRH(r_i, r_j) \) and \( MRH(r_i, r_j) = 1 \), then there is inconsistency of \( r_i \), \( r_j \), \( r_2 \) and sod-r constraints, add them to the ISet. Continue step 3 until all the sod-r constraints are checked
- Retrieve next unchecked sod-p \((p_1, p_2)\) from the policy, for each reR, if \( MRP(r_i, p_j) = 1 \) and \( MRP(r_i, p_j) = 1 \), then add r and sod-p \((p_1, p_2)\) to the inconsistency set. Continue step 4 until all the sod-p constraints are checked
- Find the next unchecked sod-r \((r_i, r_j)\), for each \( u \in U \), if \( MUR(u, r_i) = MUR(u, r_j) = 1 \), then sod-r \((r_i, r_j)\) and u cause the inconsistency, add them to the ISet. Continue this step until all the sod-r constraints are checked.
- For each co-r(r), if
  \[ \sum_{u} MUR(u, r) > 1 \]
  (1 is the user number), then there is inconsistency, add the user set \( \{u \in U | MUR(u, r) = 1\} \) and co-r(r) to the ISet
- For each co-p(p), if
  \[ \sum_{r} MRP(r, p) > 1 \]
  then there is inconsistency, add the role set \( \{r \in R | MRP(r, p) = 1\} \) and co-p(p) to the ISet
- If ISet is empty, return false, else return true

After calculate the relation matrix MRH, MRH, MUR, MRP, it is easy to check the inconsistencies via., the algorithm except the role hierarchy inconsistency.

The essential reason of the role hierarchy inconsistency is that there are strongly connected roles (i.e., role cycle) in the role graph. Tarjan’s SCC algorithm is based on DFS, which can be used for the role hierarchy inconsistency checking. It begins from a starting role and the roles involved in a cycle will form a sub-tree of the whole DFS search tree and the root of the sub-tree is the root of the strongly connected roles.
The time complexity of Tarjan’s SCC algorithm is:

$$O(m+k)$$

where, $m = |R|$ and $k = |RH|$, so the complexity of inconsistency checking can be calculated as following:

$$O(m+k) + O(m^2C_I) + O(m^2C_J) + O(|C^\prime| + |C^\prime I|)$$

where in $n=|P|$, $l=|U|$, $C_I = |SOD-P| < n^2$, $C = |SOD-P| < mn$, $C_J = |CC-R| < m$ and $C^\prime = |CC-P| < n$ and since $m << l$ and $n << d$ so the time complexity is: $O(\text{nm}^2)$.

Although, we separate the steps to check the inconsistency for clarity, the steps can be combined to improve efficiency.

**RESULTS AND DISCUSSION**

The purpose of the experimental evaluation is two fold. First, we would like to validate that present algorithm works in a wide variety of environments ranging from small to large organization, with a few to a large number of roles and comprising of complex role hierarchy. Secondly, we would like to examine the performance of present approach with complex access control constraints. However, it is hard to find the real data of complex access control constraints that we would like to evaluate. Therefore, we created a test data generator that would allow us to verify the effectiveness of this algorithm under various conditions.

The test data generator performs as follows: first a set of roles are created, the number of which is set to a certain maximum number and the hierarchy relationships are constructed randomly under a certain ratio. Next for each role a random number of permissions are chosen to form the role. Next the constraints among roles and permissions are generated randomly. Finally, the redundancy and inconsistency are generated according to the specified amount the generating algorithm is shown in Fig. 5. It is obvious that the generator uses a simple randomization strategy in order to generate the test data. This allows us to test in a completely unbiased manner with several different situations.

The input parameters of the test data generator algorithm are as follows:

- $NR$ = No. of roles
- $NRH$ = No. of role hierarchies
- $NU$ = No. of users
- $NRd$ = No. of redundancies
- $NI$ = No. of inconsistency

The experimental data is collected under the following test environment:

- **CPU**: Intel Core 2 T7500 @ 2.20 GH
- **Memory**: 2.00 GB
- **OS**: Windows XP, SP4
- **Program language**: Java
- **JRE Version**: 1.6.0_02

Table 1 shows the CPU time records of the redundancy checking method based on Warshall algorithm and DFS algorithm, in the table the column head $RNum$ is row number and role hierarchy ratio is defined as:

$$\text{Ratio} = \frac{|RH|}{|R|}$$

According to the data in Table 1, we can get the conclusion that the Warshall based redundancy checking algorithm is much more sensitive to the role number. Obviously, the DFS based redundancy checking algorithm offers the significant performance advantage and DFS based method is actually the one we choose in our real application.

Table 2 shows the CPU time records of the inconsistency checking algorithm, as the role hierarchy ratio increases, the CPU time which is taken to perform inconsistency checking also increase. This makes sense since more hierarchies involved in the checking algorithm indicates that there are more edges in the role graph which makes the graph traverse more time consuming.
Input: NU, NR, NRH, NRd, NI
Output: role system with redundancy and inconsistency

/initialize
randomly generate the permission set PS
/Create roles
while NR > 0
    NR = (NR-1)
    choose permissions from PS and assign to role r randomly and add r to role set RS
/initialize the role hierarchy
while NRH > 0
    select two roles r1 and r2 randomly
    if (r1, r2) ∈ RH and (r2, r1) ∈ RH
        add (r1, r2) to role hierarchy set RH
    NIH = (NRH-1)
/initialize users
while NU > 0
    NU = (NU-1)
    choose roles from RS and assign to user u randomly and add u to user set US
/initialize the constraints
generate the constraints randomly, including SOD and cardinality constraints randomly
/generate the role redundancy and inconsistency
computer MRR and MRRH
for i = 0, j: |RS|+1
    for j = 0, j: |RS|+1
        if (MRR(i, j) = 0 && MRRH(i, j) = 1 && (NRd > 0))
            add (r1, r2) to RH
    NRd = (NRd - 1)
    if (MRRH(i, j) = 0 && (NI > 0))
        add (r1, r2) to RH
    NI = (NI-1)

Fig. 5: Test data generator

Table 2: CPU time records of inconsistency checking

<table>
<thead>
<tr>
<th>Role No. (RNum)</th>
<th>Role hierarchy ratio</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>16</td>
<td>32</td>
<td>51</td>
<td>72</td>
<td>77</td>
</tr>
<tr>
<td>300</td>
<td></td>
<td>27</td>
<td>56</td>
<td>87</td>
<td>120</td>
<td>137</td>
</tr>
<tr>
<td>400</td>
<td></td>
<td>49</td>
<td>98</td>
<td>150</td>
<td>204</td>
<td>242</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>78</td>
<td>150</td>
<td>228</td>
<td>308</td>
<td>372</td>
</tr>
<tr>
<td>600</td>
<td></td>
<td>107</td>
<td>212</td>
<td>321</td>
<td>432</td>
<td>528</td>
</tr>
<tr>
<td>700</td>
<td></td>
<td>146</td>
<td>286</td>
<td>452</td>
<td>580</td>
<td>712</td>
</tr>
<tr>
<td>800</td>
<td></td>
<td>191</td>
<td>374</td>
<td>565</td>
<td>757</td>
<td>933</td>
</tr>
<tr>
<td>900</td>
<td></td>
<td>239</td>
<td>474</td>
<td>734</td>
<td>956</td>
<td>1182</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>301</td>
<td>591</td>
<td>889</td>
<td>1190</td>
<td>1474</td>
</tr>
</tbody>
</table>

Compared to the methods by Chang and Hoh (2005) and Strembeck (2004), the presented approach is different in several ways. Firstly, the present approach takes all the RBAC constraints into consideration, including three types of separation of duty and two types of cardinality constraints. Constraints, which provide the advanced manipulation of the access control policy, are the most important feature of the RBAC model. Thus, the policy management methodology with no support for the constraints are of no practical use. Present approach is the first attempt to take all the RBAC constraints, which makes this approach the one which could provide the maximum management confidence. Secondly, the access control policy management is the daily basic work of security administrator, thus the policy checking algorithm should be efficient enough due to frequent calls. From the above experimental evaluation, we can see that for an access system with role number up to 1000, present approach still maintain a high performance. According to the present experience and the case study (Ferrari et al., 2003), the role number of enterprise system is much less than 1000, so the method is efficient enough for practical usage. Another advantage of the present approach is that we reduce the redundancy and the inconsistency to the graph problems, for which there have been a lot of existing well-known algorithms. Another point worth mentioning is the role hierarchy ratio, which is a kind of description of the complexity of the hierarchy relations. Similar to the concepts in graph theory, it could be considered as the graph complexity. Since, the role graph is a diagraph and in most circumstances a cyclic diagraph, the ratio of the number of edges and the number of vertexes will be relatively small. According to the experience, the role hierarchy ratio is usually less than 0.2, in such kind of situation, our approach demonstrates excellent performance. Although, the above work has not discussed the dynamic constraints, there is no significant difference compared to the static constraints except the checking happens at runtime, which requires the algorithm should be much more efficient and our approach can be extended easily to provide such kind of support.

CONCLUSIONS

In this study, we analyze the redundancy and inconsistency in role based access control policy. We give the formal definitions for each type of policy redundancy and inconsistency. Based on the analysis, the algorithms are given to check both redundancy and inconsistency. The redundancy checking algorithm is based on the Warshall algorithm and depth first search algorithm and the redundancy can be identified via., detecting whether there is the redundant path in the role graph. The inconsistency checking algorithm is based on Tarjan's SCC algorithm and the inconsistency can be identified via checking whether the role cycle exists. Compared to the others similar studies, present methods covers all the elements of RBAC policy and the checking algorithm is efficient enough for the practical usage, which could guarantee the conciseness and consistency of the RBAC policy and prevent the system from concealing security danger. The future study includes the study of redundancy and inconsistency resolution. And the severities of different inconsistencies and the automated method of inconsistency resolution will be the focus of the future research.
REFERENCES


