The Formal Model of DBMS Enforcing Multiple Security Polices

Yongzhong He, Zhen Han
School of Computer, Beijing Jiaotong University, China
State Key Lab of CAS, Beijing, China, Oakland University, Michigan, USA
Email: {yzhhe, zhan}@bjtu.edu.cn

Huirong Fu, Guangzhi Qu
Department of CSE, Oakland University, Michigan, USA
Email: {fu, gqu}@oakland.edu

Abstract— **The formal security policy model and security analysis is necessary to help Database Management System (DBMS) to attain a higher assurance level. In this paper we develop a formal security model for a DBMS enforcing multiple security policies including mandatory multilevel security policy, discretionary access control policy and role based access control policy. A novel composition scheme of policies is introduced. And the security properties are comprehensively and accurately specified in terms of about 17 state invariants and state transition constraints. Furthermore, the security of the model is proved with the Z/EVES theorem prover.**

Index Terms—-**multiple security policies; formal language; security invariant; theorem proving**

I. Introduction

Formal modeling plays a key role in building trusted software systems, such as trusted OS, DBMS or other applications. Various security evaluation standards and criteria, including Common Criteria [1], TCSEC [2] of US, GB17859 [3] of China, require that formal model and security analysis be presented to be evaluated at higher assurance level.

The SeaView [4] model is one of the earliest influential security models for multilevel secure database based on Bell LaPadula model (BLP) [5]. It is verified in [6]. Recently [7] extended the object structure of the SeaView Model and, modeled the security policy in formal language and proved the security with the help of Coq proof assistant. But role based access control (RBAC) [8] is not supported in these models. A powerful and flexible authorization mechanism was proposed in [9] to enforce any different security polices in DBMS. However, due to efficiency consideration, this mechanism is never adopted in practical DBMS. A design framework of database system supporting BLP, DAC (Discretionary Access Control) and RBAC policies was proposed in [10] while the formal security model is not presented. Because many commercial DBMSs are supporting these policies, there is an emerging demand for a formal and accurate security model of DBMS supporting all the previous security policies.

Manuscript received January 1, 2009; revised June 1, 2009; accepted July 20, 2009.

In this paper, a formal security policy model (named as SEPOSTG, for Secure PostgreSQL) of DBMS supporting multiple security policies is proposed, based on which we enhanced the open source DBMS PostgreSQL [11] to enforce multiple security policies and aimed to boost its security level to TCSEC B2.

Our main contributions in this paper are threefold: 1) Multiple security policies including BLP, DAC, and RBAC are modeled. Especially, a novel composition scheme of multiple policies is proposed. 2) We present 15 state invariants and 2 state transitions in this paper, most of them are either new or improved compared to those in the literature, to model the security properties after analyzing the security policies thoroughly and comprehensively. 3) The model is expressed in formal language Z [12], and can be proved automatically with the help of the theorem prover Z/EVES.

As the security model for fine-grained entities such as tuples, attributes, elements in a tuples is well defined in other works, we do not consider those entities in this paper so as to greatly facilitate the security proof of our model. Furthermore, our model can be easily integrated with the other security models consisting of fine-grained entities because of its modular design.

The organization of this paper is as follows. In Section 2, the policy composition scheme of multiple policies is discussed. In Sections 3, 4, 5, the basic definition of entities and principals, security requirement termed as invariants and constraints, and operation rules are presented sequentially. The security analysis of the model is described in Section 6.

II. Policies Composition

In the SEPOSTG model, three typical security policies, namely BLP, DAC, and RBAC, will be enforced. Among the policies, multilevel BLP is required by the standards for higher assurance level. From management point of view, the management privileges are assigned to a few managers in RBAC such that management privileges are centralized and strictly protected, while every owner has the management privileges with regarding to his own data in DAC such that the management privileges is distributed and flexible. Although RBAC can simulate DAC or BLP model, the efficience is damaged. Therefore, our model supports both of them. The problem is the choice of policies composition scheme so as to make the model consistent, expressive, and effective. As to an access request, the access control decision made by the SEPOST model depends on the composition of decisions by BLP, DAC and RBAC respectively. We introduce a decision function to model the access control decision made by the composed policies.

Suppose that, for a complete model, in one state of the system, for any access request to the model, the access control decision would be {YES, NO, UNDEFINED}. Then, we can build 3^8=6561 decision functions for composed policies from BLP, DAC and RBAC. According to the analysis of each decision function, most of them are impractical. Apparently, BLP is a mandatory policy that every access request should respect. However, the composition of RBAC and DAC with 'OR' relation is not suitable, because it will result in unnecessarily replicated authorizations. Based on the evaluation standards and the above observation, one may choose the composition as the following function:

$$RD = (RBAC \vee DAC) \wedge BLP$$

The connotation of this function is, all access requests must meet BLP policy, and meet either RBAC or DAC. Table 1 illustrates the composition when all policies involved are complete (without UNDEFINED decision). If the decision of any component policy is UNDEFINED with respect to an access request, then the decision of the composed policy is UNDEFINED.

Table 1 Policies Composition

RBAC decision	DAC decision	BLP decision	SEPOSTG decision
YES	YES	YES	YES
YES	YES	NO	NO
YES	NO	YES	YES
YES	NO	NO	NO
NO	YES	YES	YES
NO	YES	NO	NO
NO	NO	YES	NO
NO	NO	NO	NO

However, the above straightforward composition is not satisfactory in practice. As we know, RBAC is more suitable for organizational policy, while DAC is more suitable for personal policy. Consequently, with regarding to the decision function *RD,* one may use DAC policy to corrupt the RBAC policy. The attack is as follows. The standard RBAC can not explicitly express negative authorization (which means to deny one role the privilege of access to one object) so that it is assumed by default that if there is no authorization for an access request then the request is denied. However, one subject which is assumed by the organizational policy administrator to be denied access to one object in RBAC may be granted access privilege in DAC by another subject. In this way, the effect of the RBAC policy becomes unexpected. One approach to this problem is to add negative privilege explicitly to RBAC policy, but the constraints of the approach is that it will add significant burden to the system administrator because there are huge volumes of data in DBMS generally.

We take another approach to the problem. The objects are classified into two categories based on the owner of the object. The owners of objects are: system administrator, and others. RBAC policy are applicable to the objects owned by system administrator, DAC policy are applicable to objects owned by other owners. The owner of an object can be changed by the security administrator so that we can change the policy on an object by changing its owner. When a user is removed from the system, the owner of objects owned by this user is changed to be the system administrator. Furthermore, for all privilege with no specific objects (e.g. system privileges), it is subject to RBAC policy.

Hence, the decision function of policy composition in our model SEPOSTG is formalized as:

$$RD(o) = \begin{cases} BLP(o) \wedge RBAC(o) \\ \qquad \textit{if } user\_kind(owner(o)) = \text{SYSADM} \\ \qquad \vee\, o = nil \\ BLP(o) \wedge DAC(o) \quad \textit{otherwise} \end{cases}$$

Remark: *user_kind*(), *owner*() and SYSADM will be defined in the following section.

III. Basic elements and Definitions

In this section, we define the basic elements of our model SEPOSTG, including data types, constraints and variables.

A. Data types

There are seven basic data types in this model.

Security Labels: [CLASSES, LEVELS, CATES], CLASSES is for security label type which is composed by linear ordered levels LEVELS and patiral ordered categories CATES.

Roles: [ROLES], for Role based access control.

Subjects: [USERS, SESSIONS, TRANS], SESSIONS models the process between user login and logout, TRANS is correspondent to transaction in DBMS.

User Kinds:

UKIND := SYSADM|SECADM|AUDADM|COM.

In order to limit the power of administrator, there are three kinds of administrators introduced to assume different duties. The duty of the system administrator is the routine of system management, such as user or data object creation/removal; the duty of the security administrator is the maintenane of security policy; the auditor is in charge of audit policy. It is required in this model that system administrator has no privileges to read/write access to the content of an object.

Objects: [DATABASES], [MREALTIONS, MREAL-IDS, MVIEW-IDS, MTUPLES, ELEMENTS], [AUXS, SUBORDS]. There are many kinds of objects in the database, including database, relation, view, tuple, element, auxilary object and subordinate object. MREALTIONS is composed of MREAL-IDS and MVIEW-IDS.

Operations and Privileges: Operation and privilege are tightly related but have different usages, [OPERATIONS, PRIVILEGES]

Parameter types: for element data types. [VALUES, BOOL, ATTRS]

B. Constants and Mappings

Reserved users: { **sysadmin**, **secadmin**, **audadmin** }. There are three reserved users which cannot be removed at any time: system administrator, security administrator, auditor. They are called as initial administrators.

Reserved Security labels: { **syshigh**, **syslow**, **trusted**}.

*operations:*OPERATIONS, including CREATE USER, DROP USER, ACTIVATE ROLE, DEACT ROLE, INSERT, DELETE, UPDATE, SELECT; AUDIT ON, UPGRADE, DOWNGRADE, etc. The full list of the *operations* are not presented here.

Privileges:PRIVILEGES, including initial system administrator privieleges: CREATEUSER, DROPUSER, CREATEDB, CREATETABLE, etc.and initial secucity administrator privieleges: CREATEROLE, DROPROLE, CREATELABEL, DROPLABEL, etc.; initial auditor administrator privieleges: AUDIT, AUDSET, AUDREAD, AUDPURGE, AUDBACKUP. All initial administrators have privileges of: LOGIN, LOGOUT, ACT, DEACT, UPLABEL.

A constant mapping is used to describe the max privileges which can be granted to a user or an administrator:

$$\text{adm\_perms} : \{\text{sysadmin}, \text{secadmin}, \text{audadmin}, \text{com}\}$$
$$\rightarrow \wp(\text{PRIVILEGES} \times \mathit{all})$$

Different types of objects have different owner privileges, which are defined by:

$$\text{owner\_privs} : \{\text{database}, \text{real}, \text{view},$$
$$\text{tuple}, \text{element}, \text{aux}, \text{subords}\} \mapsto \wp\,\text{PRVILEGES}$$

All privileges have one of the three privilege types. For example, Log in , Log out, etc. belong to 'read' type; create, drop etc. belong to 'write' type; and SELECT, INSERT tuples belong to multilevel operation type 'mop':

$$\text{priv\_type} : \text{PRIVILEGES} \mapsto \{\text{read}, \text{write}, \text{mop}\}.$$

All users have the minimum privies as:

public_perms={ LOGOUT, ACTROLE, BEGINTRANS, ENDTRANS }

The mapping 'op-privs' is used to model the privileges that an operation should hold. It is required that every access request must have the object privilege as well as the object data container's privilege. For example, to create a relation *r* in database *db* one must have the privileges of CREATEREAL and LOGON *db*; to select the tuple in relation *r* one must have the privileges of SELECT *r*, ACCESS *r* and LOGON *db*.

There are five attribute predicates introduced below. The first predicate *ateq* is defined as: for any tuple *t*: MTUPLES, if the number *i*:N attribute's value is equal to *v*: VALUES, then *ateq*(*i*,*v*,*t*) is TRUE. Similarly, *atge, atgr, atle, atlt* means greater or equal, greater, less or equal, less respectively.

$$\mathit{ateq} : \text{N} \times \text{VALUES} \times \text{MTUPLES} \mapsto \text{BOOLEAN}$$
$$\mathit{atge} : \text{N} \times \text{VALUES} \times \text{MTUPLES} \mapsto \text{BOOLEAN}$$
$$\mathit{atgr} : \text{N} \times \text{VALUES} \times \text{MTUPLES} \mapsto \text{BOOLEAN}$$

The '*obj*' preditcate means that when o_1:OBJECTS equals o_2:OBJECTS then *obj*(o_1, o_2) is true.

$$\mathit{obj} : \text{OBJECTS} \times \text{OBJECTS} \mapsto \text{BOOLEAN}$$
$$\mathit{all} : \text{OBJECTS} \mapsto \text{BOOLEAN}$$

C. Variables

The variables may be changed by operations, and are the indispensable part of the state of the system.

The state variables related to subject include user set, session set and transaction set.

$$\mathit{user\_exists} : \wp\,\text{USERS}$$
$$\mathit{session\_exists} : \wp\,\text{SESSIONS}$$
$$\mathit{trans\_exists} : \wp\,\text{TRANS}$$

The types of user are introduced to manage the difference between administrators and common users. Different kinds of users have different privileges.

$$\mathit{user\_kind} : \text{USERS} \mapsto \text{UKIND}$$

One session is correspondent to one user's login, and one transaction is correspondent to one session, and one session can only access to one database. So we have:

$$\mathit{session\_user} : \text{SESSIONS} \mapsto \text{USERS}$$
$$\mathit{trans\_session} : \text{TRANS} \mapsto \text{SESSIONS}$$
$$\mathit{session\_database} : \text{SESSIONS} \mapsto \text{DATABASES}$$

Object related state variables include database set, relation set, real relation set, view set, tuple set, etc.

$$\mathit{database\_exists} : \wp\,\text{DATABASES}$$
$$\mathit{relation\_exists} : \wp\,\text{MRELATIONS}$$
$$\mathit{real\_exists} : \wp\,\text{MREAL-IDS}$$

The data objects in DBMS are related in a tree structure, and parent node is called as the container of child node. Database is at the root of the tree, and is the largest container. Element is at the leaf and is not container.

$$\mathit{real\_database} : \text{MREAL-IDS} \mapsto \text{DATABASES}$$
$$\mathit{view\_database} : \text{MVIEW-IDS} \mapsto \text{DATABASES}$$
$$\mathit{view\_reals} : \text{MVIEW-IDS} \mapsto \wp\,\text{MREAL-IDS}$$

Role based access control policy is an important part of this model. The state variables for role based access control include roles set, role containing mapping (*role_co*), role supervising mapping (*role_su*), role and permission mapping, user role mapping, session and activated roles mapping, static separation of duty, dynamic separation of duty, and the preconditions of role activation. All of them are listed below respectively. Particularly, *role_co* and *role_su* are improvement of role hierarchy. *role_co* is the same as the traditional role hierarchy, but *role_su* is used to model the supervisor role which has only the management privileges of its subordinate roles.

$role\_exists : \wp \text{ROLES}$

$role\_co : \text{ROLES} \mapsto \wp \text{ROLES}$

$role\_su : \text{ROLES} \mapsto \wp \text{ROLES}$

$role\_perms : \text{ROLES} \mapsto \wp(\text{PRIVILEGES} \times \wp(\text{ATTRS} \times \wp(\text{VALUES})))$

$rbac\_user\_perms : \text{USERS} \mapsto \wp(\text{PRIVILEGES} \times \wp(\text{ATTRS} \times \wp(\text{VALUES}))))$

$user\_roles : \text{USERS} \mapsto \wp \text{ROLES}$

$session\_roles : \text{SESSIONS} \mapsto \wp \text{ROLES}$

$role\_ssd : \wp((\wp \text{ROLES}) \times \text{N})$

$role\_dsd : \wp((\wp \text{ROLES}) \times \text{N})$

$role\_pre : \wp(\text{ROLES} \times \wp \text{ROLES})$

The state variables for discretionary access control (DAC) are as follows. *dac_uer_perms* means that a user's permission set of (Privilege, object predicate, parameters list).

$\text{OBJECTS} := \text{DATABASES} \cup \text{MRELATIONS} \cup \text{MTUPLES} \cup \text{ELEMENTS} \cup \text{AUXS} \cup \text{SUBORDS}$

$object\_owner : \text{OBJECTS} \mapsto \text{USERS}$

$dac\_user\_perms : \text{USERS} \mapsto \wp(\text{PRIVILEGES} \times \wp(\text{ATTRS} \times \wp(\text{VALUES})))$

The variables for mandatory access control (MAC) include security labels set etc. Note that a security label *c*:CLASSES is composed of level and category which reflectes the *class_level_cate* mapping.

$class\_exists : \wp \text{CLASSES}$

$level\_exists : \wp \text{LEVELS}$

$cate\_exists : \wp \text{CATES}$

$class\_level\_cate : \text{CLASSE} \mapsto \text{SLEVELS} \times \text{CATES}$

In the security labels set *class_exists* there is a partial order *dom* (which is called as 'dominate' relation), and in level set *level_exists* there is a total order dom_1, and in *cate_exists* there is a partial order dom_2.

In models supporting multilevel access control (BLP), all subjects and objects should be properly marked with security labels. The security label of a session initialed by a trusted user can be dynamically updated. As changing session security label will lead to start a new service process, we require that the security label of a transaction cannot be changed during its lifetime.

The subject to security label mapping is:

$user\_class : \text{USERS} \mapsto \text{CLASSES}$

$session\_class : \text{SESSIONS} \mapsto \text{CLASSES}$

$trans\_class : \text{TRANS} \mapsto \text{CLASSES}$

The object to security label mapping is:

$database\_class : \text{DATABASES} \mapsto \text{CLASSES}$

$real\_class : \text{MREAL-IDS} \mapsto \text{CLASSES}$

$view\_class : \text{MVIEW-IDS} \mapsto \text{CLASSES}$

For easy reference in the model, we define an integrated object to security label mapping, and an object to type mapping:

$object\_class := database\_class \cup real\_class \cup view\_class \cup tuple\_class \cup element\_class \cup aux\_class \cup subord\_class$

$object\_type : \text{OBJECTS} \mapsto \{\text{database}, \text{real}, \text{view}, \text{tuple}, \text{element}, \text{aux}, \text{subords}\}$

Lastly, we define two key variables of the model: current access permission state variable and current administration permission variable. The first variable models the non-administration (not the GRANT privileges) permissions of a transaction' access request, and is computed legally based on RBAC policy, DAC policy and BLP policy. Another variable, current administration permission variable, is concerned with the permissions related to GRANT privileges.

$cur\_perms : \text{TRANS} \mapsto \text{PRIVILEGES} \times \text{OBJECTS}$

$cur\_adm\_perms : \text{TRANS} \mapsto \text{PRIVILEGES} \times \text{OBJECTS}$

IV. Invariants and Constraints

The key part of the DBMS security policy model is the definitions of security which are formalized as state invariants or/and state transition constraints. State transition constraints are enforced in every access request which will cause a state transition, while state invariants should be kept for every state which is reachable for any access requests from any initial state. Based on the evaluations standards (GB17859, TCSEC and CC), the characteristics of the component policies, and the DBMS application requirements, there are 14 state invariants and 2 state transition constraints identified in SEPOSTG, most of them are not reported before in the literature.

It is required that all reserved security labels (syshigh, syslow, trusted) are always members of *class_exists*; all security labels except trusted are dominated by syshigh and dominate syslow. Suppose that $class\_level\_cate(c_1) = (a_1,b_1)$, $class\_level\_cate\ (c_2) = (a_2,b_2)$, then $c_1\ dom\ c$ *iff* $a_1\ dom_1\ a_2$, $b_1\ dom_2\ b_2$. Let $fst\ (a,b)=a$, $snd(a,b)=b$, formally we have **Security Labeling Invariant**:

$\{\text{syshigh}, \text{syslow}, \text{trusted}\} \in class\_exists$

$\forall c : \text{CLASSES} \bullet c \in class\_exists \wedge c \neq \text{trusted} \Rightarrow \text{syshigh}\ dom\ c \wedge c\ dom\ \text{syslow}$

$$\forall c_1, c_2 : \text{CLASSES} \bullet c_1, c_2 \in class\_exists \wedge c_1\ dom\ c_2 \Rightarrow fst(class\_level\_cate(c_1))\ dom_1\ fst(class\_level\_cate(c_2)) \wedge snd(class\_level\_cate(c_1))\ dom_2\ snd(class\_level\_cate(c_2))$$

There is an invariant to model the security labeling of subject including session, transaction and user. It is required that the security label of all sessions and transactions is not ‘trusted’; and for untrusted users, the security label of session is dominated by user’s security label who opens the session, and for all users, the security label of transaction is dominated by session’s security label. Formally we have **Subject Security Label Invariant**:

$$\begin{aligned}&\forall s:\text{SESSIONS}, tr:\text{TRANS}\bullet session\_class(s)\neq \text{trusted}\\ &\quad\wedge trans\_class(tr)\neq\text{trusted}\\ &\quad\wedge session\_class(trans\_session(tr))\\ &\qquad dom\ trans\_class(tr)\\ &\quad\wedge session\_user(s)\neq\text{trusted}\\ &\qquad\Rightarrow user\_class(session\_user(s))\\ &\qquad\quad dom\ session\_class(s)\end{aligned}$$

Another invariant is to model the security labeling of objects in the hierarchy. It is required that the security label of all objects cannot be ‘trusted’, and the security label of an object dominates the security label of its parent in the hierarchy. For simplicity, we here only give the invariant for relation and database objects, invariants for other types of objects can be constructed similarly. Formally **Object Security Label Invariant** is:

$$\begin{aligned}&\forall d:\text{DATABASES}, r:\text{MREAL-IDS}\\ &\quad\bullet database\_class(d)\neq\text{trusted}\\ &\quad\wedge real\_class(r)\neq\text{trusted}\wedge real\_class(r)\\ &\qquad dom\ database\_class(real\_database(r))\end{aligned}$$

There are three reserved administrators in the system. Each administrator is able to upgrade common user to be the same kind of administrator as itself, and/or grants the self privileges to the users. Therefore formally we have **Initial Administrator Invariant**:

$$\begin{aligned}&\{\text{sysadmin}, \text{secadmin}, \text{audadmin}\}\in user\_exists\\ &\wedge user\_kind(\text{sysadmin})=\text{SYSADM}\\ &\wedge user\_kind(\text{secadmin})=\text{SECADM}\\ &\wedge user\_kind(\text{audadmin})=\text{AUDADM}\\ &\wedge rbac\_user\_perms(\text{sysadmin})=\text{adm\_perms(sysadmin)}\\ &\wedge rbac\_user\_perms(\text{secadmin})=\text{adm\_perms(secsadmin)}\\ &\wedge rbac\_user\_perms(\text{audadmin})=\text{adm\_perms(audadmin)}\end{aligned}$$

One kind of administrator cannot have the distinct management privileges of the other kind. Suppose *user_eperms* includes all permissions derived from all component policies, $\subseteq_{im}$ is slightly different from normal set operation $\subseteq$. It is defined as: if every permission $(p,f(o))$in A is in B or is derivable from B then $A\subseteq_{im}B$. Formally **Administrator Separation of Duty Invariant**:

$$\begin{aligned}&\forall u:\text{USERS}\\ &\quad\bullet user\_kind(u)=\text{SYSADM}\Rightarrow\\ &\qquad user\_eperms(u)\subseteq_{im}\text{adm\_perms(sysadmin)}\\ &\quad\wedge user\_kind(u)=\text{SECADM}\Rightarrow\\ &\qquad user\_eperms(u)\subseteq_{im}\text{adm\_perms(secadmin)}\\ &\quad\wedge user\_kind(u)=\text{AUDADM}\Rightarrow\\ &\qquad user\_eperms(u)\subseteq_{im}\text{adm\_perms(audadmin)}\\ &\quad\wedge user\_kind(u)=\text{COM}\Rightarrow\\ &\qquad user\_eperms(u)\subseteq_{im}\text{adm\_perms(com)}\end{aligned}$$

The owner of an object has all the privileges including management privileges with respect to the object. Formally **Ownership privileges Invariant**:

$$\begin{aligned}&\forall o:\text{OBJECTS}\\ &\quad\bullet\forall p:\text{owner\_privs}(object\_type(o))\\ &\qquad\bullet(p,o)\subseteq rbac\_user\_perms(object\_owner(o))\end{aligned}$$

All users by default have the common privileges which are termed as Public privileges. Formally **User Public Privileges Invariant**:

$$\begin{aligned}&\forall u:\text{USERS}\\ &\quad\bullet\text{public\_perms}\in rbac\_user\_perms(u)\end{aligned}$$

The static separation of duty SSD is defined differently from SSD in RBAC. In our model, the SSD:= $(\{r_1, r_2,\ldots r_n\},t)$ means that one user cannot have privileges (including inheritance) of *t* roles from $\{r_1, r_2,\ldots r_n\}$. Formally **Static Separation of Duty**:

$$\begin{aligned}&\forall u:\text{USERS}, (rs,t):(\wp\text{ROLES})\times\text{N}, rt:\wp\text{ROLES}\\ &\quad\bullet(rs,t)\in role\_ssd\wedge rt\subseteq rs\\ &\qquad\wedge((\bigcup_{r\in rt}role\_eperms(r))\subseteq(rbac\_user\_perms(u)\\ &\qquad\quad\cup dac\_user\_perms(u)\\ &\qquad\quad\cup\bigcup_{r\in user\_roles(u)}role\_eperms(r))\\ &\quad\Rightarrow\#rt<t\end{aligned}$$

Remarks: *role_eperms(r)* is all permissions of role *r* including derived permissions, and it can be further defined with *role_perm* and *role_co*,*role_su*.

In our model, the DSD:= $(\{r_1, r_2,\ldots r_n\},t)$ means that one session cannot have privileges (including inheritance) of *t* roles from $\{r_1, r_2,\ldots r_n\}$. Formally **Dynamic Separation of Duty**:

$$\begin{aligned}&\forall s:\text{SESSIONS}, (rs,t):(\wp\text{ROLES})\times\text{N}, rt:\wp\text{ROLES}\\ &\quad\bullet(rs,t)\in role\_dsd\wedge rt\subseteq rs\\ &\qquad\wedge((\bigcup_{r\in rt}role\_eperms(r))\subseteq\\ &(rbac\_user\_perms(session\_user(u))\\ &\qquad\quad\cup dac\_user\_perms(session\_user(u))\\ &\qquad\quad\cup\bigcup_{r\in session\_roles(s)}role\_eperms(r))\\ &\quad\Rightarrow\#rt<t\end{aligned}$$

If the owner of an object is system administrator or the object is *nil*, then there exists a privilege with respect to this object in the current access permission state variable, iff the correspondent transaction is authorized with the privilege from RBAC policy. Formally **Role Security Invariant**:

$$\begin{aligned}
&\forall p:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS}, tr:\mathrm{TRANS}\\
&\quad \bullet (p,o)\in cur\_perms(tr)\wedge(o=nil\vee\\
&user\_kind(object\_owner(o))=\mathrm{SYSADM})\\
&\qquad\Rightarrow \exists d:\wp(\mathrm{ATTRS}\times\wp(\mathrm{VALUES}))\\
&\qquad\quad \bullet(\forall(a,vs)\in d\bullet a(vs,o))\\
&\qquad\quad \wedge((p,d)\in\\
&\quad rbac\_user\_perms(session\_user(trans\_session(tr)))\\
&\qquad\qquad \vee(p,d)\in\\
&\qquad\qquad \bigcup_{r\in session\_roles(trans\_session(tr)))} role\_eperms(r))
\end{aligned}$$

If the owner type of an object (not as a *nil* object) is not system administrator, then there exists a privilege with respect to this object in the current access permission state variable, iff the correspondent user is authorized the privilege from DAC policy. Formally **Role Security Invariant**:

$$\begin{aligned}
&\forall p:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS}, tr:\mathrm{TRANS}\\
&\quad \bullet (p,o)\in cur\_perms(tr)\wedge o<>nil\\
&\qquad \wedge user\_kind(object\_owner(o))<>\mathrm{SYSADM}\\
&\qquad\Rightarrow \exists d:\wp(\mathrm{ATTRS}\times\wp(\mathrm{VALUES}))\\
&\qquad\quad \bullet(\forall(a,vs)\in d\bullet a(vs,o))\\
&\qquad\quad \wedge(p,d)\in\\
&dac\_user\_perms(session\_user(trans\_session(tr)))
\end{aligned}$$

There exists a 'read' privilege with respect to an object in the current access permission state variable, iff security label of the correspondent transaction dominates the security label of the object. Formally **Simple Security Invariant**:

$$\begin{aligned}
&\forall p:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS},\\
&d:\wp(\mathrm{ATTRS}\times\wp(\mathrm{VALUES})), tr:\mathrm{TRANS}\\
&\quad \bullet (p,o)\in cur\_perms(tr)\wedge priv\_type(p)\in\{\mathrm{read},\mathrm{mop}\}\\
&\qquad\Rightarrow trans\_class(tr)\ dom\ object\_class(o)
\end{aligned}$$

There exists a 'write' privilege with respect to an object in the current access permission state variable, iff security label of the correspondent transaction is dominated by the security label of the object. Formally ***-Security Invariant**:

$$\begin{aligned}
&\forall p:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS}, tr:\mathrm{TRANS}\\
&\quad \bullet (p,o)\in cur\_perms(tr)\wedge priv\_type(p)=\mathrm{write}\\
&\qquad\Rightarrow object\_class(o)=trans\_class(tr)
\end{aligned}$$

All administrators and object owner have the grant privileges with respect to the privileges specific to them. Formally **Management Privilege Invariant**:

$$\begin{aligned}
&\forall u:\mathrm{USERS}\bullet u\in\{\mathrm{sysadmin},\mathrm{secadmin},\mathrm{audamin}\}\\
&\quad\Rightarrow \forall pe\in rbac\_user\_perms(u)\bullet(\mathrm{PRADM},pe)\\
&\qquad \in user\_admin\_perms(u)\\
&\forall pr:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS}\\
&\quad \bullet pr\in object\_priv(object\_type(o))\\
&\qquad \Rightarrow(\mathrm{PRADM},(pr,o))\\
&\qquad\quad \in user\_admin\_perms(object\_owner(o))
\end{aligned}$$

ALL administrators and object owner have the grant privileges with respect to the privileges specific to them. Formally **Management Privilege Invariant**:

$$\begin{aligned}
&\forall u:\mathrm{USERS}\bullet u\in\{\mathrm{sysadmin},\mathrm{secadmin},\mathrm{audamin}\}\\
&\quad\Rightarrow \forall pe\in rbac\_user\_perms(u)\bullet(\mathrm{PRADM},pe)\\
&\qquad \in user\_admin\_perms(u)\\
&\qquad\quad \in user\_admin\_perms(object\_owner(o))\\
&\forall pr:\mathrm{PRIVILEGES}, o:\mathrm{OBJECTS}\\
&\quad \bullet pr\in object\_priv(object\_type(o))\\
&\qquad \Rightarrow(\mathrm{PRADM},(pr,o))
\end{aligned}$$

Lastly, we have two state transition constraints with respect to role activation and administrator role assignment.

DBMS-CONST-01 (Role Activation Constraint) A role is activated iff the corresponding user is assigned to the role and all prerequisite roles are present in the previous state. Formally:

$$\begin{aligned}
&\forall r:\mathrm{ROLES}, s:\mathrm{SESSIONS}\\
&\quad \bullet session\_roles'(s)=session\_roles(s)\cup\{r\}\\
&\qquad\Rightarrow r\in user\_roles(session\_user(s))\wedge\\
&\qquad\quad \exists rs:\wp\mathrm{ROLES}\bullet(r,rs)\in role\_pre\\
&\wedge rs\subseteq session\_user(s)
\end{aligned}$$

DBMS-CONST-02 (Administrator Constraint) When a user is upgraded to an administrator, it will not be downgraded. Formally:

$$\begin{aligned}
&\forall u:\mathrm{USERS}\bullet user\_kind(u)\in\\
&\quad \{\mathrm{SYSADM},\mathrm{SECADM},\mathrm{AUDADM}\}\\
&\qquad\quad \wedge u\in user\_exists'\\
&\qquad \Rightarrow user\_kind(u)=user\_kind'(u)
\end{aligned}$$

V. Operation Rules

The operation rules are the mechanism of the model to enforce security policies. There are six types of operations: subject creation/deletion, data object create/deletion, security policy management, audit management and authorization, and public operations. Our model is modular such that newly added operations can be proved independently without affecting the security of previous operations. In this paper, we do not list all the operation rules due to space limitation. However the model is self contained and the correctness of the presented model is not affected.

There are object predicates in authorization of this model, so that it cannot directly decide whether an object is authorized. For notation, if there is a permission (*p*,*f*) in the authorization of user *u* in *rbac_user_perms*(*u*) and *dac_user_perms*(*u*) such that *f*(*o*)=true, then the user is authorized implicitly with (*p*,*o*). Similarly, a role activated in a session may be authorized implicitly with (*p*,*o*). No matter implicitly or explicitly, the subject is called as authorized. Next, some typical and important operation rules are presented. Others may be similarly constructed.

1. CREATE USER

This operation creates a user account in the system.

Input is: requesting transaction *tr*?, and user name *u*?.

Checking if : 1) CREATEUSER is in current permission *cur_perms*(*tr*); or the user *session_user*(*trans_session*(*tr*)) who initializes the transaction *tr*? is authorized with CREATEUSER, or the activated roles in session *trans_session*(*tr*) are authorized with CREATEUSER. 2) *u*? does not belong to *user_exists*.

If all condition is checked OK, then output CREATE USER, and update state:

$\mathit{user\_exists}' = \mathit{user\_exists} \bigcup u?$,

$\mathit{user\_kind}' = \mathit{user\_kind} \bigcup \{u? \mapsto \mathrm{COM}\}$

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{CREATEUSER}, u?)\}$,

Otherwise output ERROR.

2. CREATE TABLE

This operation creates a real relation.

Input is: requesting transaction *tr*?, and table *tb*?.

The Checking conditions are similar to that of CREATE USER.

If all conditions are checked OK, then output CREATE TABLE, and update state:

$\mathit{real\_exists}' = \mathit{real\ \_exists} \bigcup tb?$

$\mathit{real\_class}' = \mathit{real\ \_class} \bigcup \{tb? \mapsto \mathit{trans\_class}(tr)\}$

$\mathit{real\_database}' = \mathit{real\ \_database} \bigcup \{tb? \mapsto \mathit{session\_dabase}(\mathit{trans\_session}\ (tr))\}$

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{CREATETABLE}\ tb?)\}$.

3. CREATE VIEW

This operation creates a view relation.

Input is: requesting transaction *tr*?, and view *v*?.

Checking if: 1) CREATEVIEW is in current permission *cur_perms*(*tr*); or the user *session_user*(*trans_session*(*tr*)) who initializes the transaction *tr*? is authorized with CREATEVIEW, or the activated roles in session *trans_session*(*tr*) are authorized with CREATEVIEW. 2) for every *r?* in *rs*?, the current perimission *cur_perms*(*tr*) has the permission of (ACCESS *r*?); or for transaction *tr* related user *session_user*(*trans_session*(*tr*)) has authorization of (ACCESS *r*?), or the activated roles in session *trans_session*(*tr*) have authorization of (ACCESS *r*?) 3) *v*? does not belong to *view_exists*.

If all conditions are checked OK, then output CREATE TABLE, and update state:

$\mathit{view\_exists}' = \mathit{view\ \_exists} \bigcup vi?$

$\mathit{view\_class}' = \mathit{view\_class} \bigcup \{vi? \mapsto \mathit{trans\_class}(tr)\}$

$\mathit{view\_database}' = \mathit{view\_database} \bigcup \{vi? \mapsto \mathit{session\_dabase}(\mathit{trans\_session}\ (tr))\}$

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{CREATEVIEW}\ vi?)\}$

4. DROP TABLE

This operation drops a real relation

Input is: requesting transaction *tr*?, and table *tb*?.

Checking if 1) the Checking condition 1) is similar to that of CREATE USER. 2) $tb? \in \mathit{real\_exists}$, $\mathit{real\_database}(tb?) = \mathit{session\_database}(\mathit{trans\_session}\ (tr))$. 3) $\mathit{trans\_class}(tr) = \mathit{real\_class}(tb?)$.

If all conditions are checked OK, then output CREATE TABLE, and update state:

$\mathit{real\_exists}' = \mathit{real\ \_exists} \setminus tb?$

$\mathit{real\_class}' = \mathit{real\ \_class} \setminus \{tb? \mapsto \mathit{trans\_class}(tr)\}$

$\mathit{real\_database}' = \mathit{real\ \_database} \setminus \{tb? \mapsto \mathit{session\_dabase}(\mathit{trans\_session}\ (tr))\}$

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{DROPTABLE}\ tb?)\}$.

And DROP all subordinated entity, view, tuples recursively.

5. SELECT

The operation reads the tuples from relations.

Input is: requesting transaction *tr*?, and table *tb*?, and condition *cf*?.

Checking if : 1) (ACCESS, *tb?*) is in current permission *cur_perms*(*tr*); or the user *session_user*(*trans_session*(*tr*)) who initialize the transaction *tr*? is authorized with (ACCESS, *tb?*), or the activated roles in session *trans_session*(*tr*) are authorized with (ACCESS, *tb?*). 2) (SELECT, *tb?*) is in current permission *cur_perms*(*tr*); or the user *session_user*(*trans_session*(*tr*)) who initializes the transaction *tr*? is authorized with (SELECT, *tb?*), or the activated roles in session *trans_session*(*tr*) are authorized with (SELECT, *tb?*). 3) $tb? \in \mathit{view\_exists} \bigcup \mathit{real\_exists}$. 4) If $\mathit{object\_type}(tb?) = \mathrm{TABLE}$, then $\mathit{real\_database}(tb?) = \mathit{session\_database}(\mathit{trans\_session}\ (tr))$, if $\mathit{object\_type}\ (tb?) = \mathrm{VIEW}$, then $\mathrm{view}\_\mathit{database}(tb?) = \mathit{session\_database}\ (\mathit{trans\_session}\ (tr))$. 5) *trans_class*(*tr*) *dom object_class* (*tb*?)

If Check is OK then outputs all tuples that meet *cf*?, authorization predicates, and *trans_class*(*tr*) *dom* . Then the new state is:

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{ACCESS}\ tb?), tr \mapsto (\mathrm{SELECT}\ tb?)\}$.

6. INSERT

The operation inserts the tuple into a real relation.

Input is: requesting transaction *tr*?, and table *tb*?, and *tu?*.

Checking conditions are similar to that of SELECT. New state is:

$\mathit{cur\_perms}' = \mathit{cur\_perms} \bigcup \{tr \mapsto (\mathrm{ACCESS}\ tb?), tr \mapsto (\mathrm{INSERT}\ tb?)\}$.

$\mathit{tuple\_exists}' = \mathit{tuple\_exists} \bigcup \{\ tu?\ \}$

$\mathit{tuple\_class}' = \mathit{tuple\_class} \bigcup \{\ tu? \mapsto \mathit{trans\_class}(tr)\}$

$\mathit{tuple\_real}' = \mathit{tuple\_real} \bigcup \{\ tu? \mapsto tb?\}$

Note: The **DELETE, UPDATE** operation rules are omitted due to space limitation. Because the security model for fine-grained data such as tuples, attributes, elements is similar to that of previous model in the literature, our model focuses only on the security policies on coarse grained entities such as database, table, and view etc. As such, the operation rules for select, delete, update, and insert in this model are not concerned with the details of tuples/attributes/elements.

VI. Security Analysis

The definitions of system, system security, security state in this paper are the same as traditional BLP model and not presented here.

Definition 7.1 (Initial state of SEPOSTG) Initially, the state variables are:

(1) Initial users related variables: including initial users set, user types, and user classifications:

$$user\_exists = \{\text{sysadmin, secadmin, audadmin}\}$$

$$user\_kind = \{\text{sysadmin} \mapsto \text{SYSADM}, \text{secadmin} \mapsto \text{SECADM}, \text{audadmin} \mapsto \text{AUDADM}\}$$

$$user\_class = \{\text{sysadmin} \mapsto \text{trusted}, \text{audadmin} \mapsto \text{trusted}, \text{secadmin} \mapsto \text{trusted}\}$$

(2) The initial security labels: syshigh, syslow, and trusted are members of *class_exists*, and hlevel, llevel are members of *level_exists,* and hcate, lcate are members of *level_*exists. Formally:

$$\{\text{syshigh}, \text{syslow}, \text{trusted}\} \in class\_exists$$

$$\{\text{hlevel}, \text{llevel}\} \in level\_exists$$

$$\{\text{hcate}, \text{lcate}\} \in cate\_exists$$

$$class\_level\_cate(\text{syshigh}) = (\text{hlevel}, \text{hcate})$$

$$class\_level\_cate(\text{syslow}) = (\text{llevel}, \text{lcate})$$

$$\text{syshigh}\ dom\ \text{syslow}$$

$$\text{hlevel}\ dom_1\ \text{llevel}$$

$$\text{hcate}\ dom_2\ \text{lcate}$$

(3) All the initial administrators are authorized with corresponding permissions which are reflected in *rbac_user_perms*, *user_adm_perms*. In concrete, system administrator has all system management permissions; security administrator has security management permissions; and auditor has audit management permissions.

(4) Other state variables are NULL initially.

Theorem 7.1 The initial state of model SEPOSTG is secure with respect to all the state invariants of SEPOSTG. The operation rules of SEPOSTG are secure with respect to all the state invariants and state transition constraints of SEPOSTG.

This theorem can be automatically proved with the help of theorem prover. Actually, we have proved the above theorem in Z/EVES [12] with regard to most of the security properties. Some security properties are proved manually because of the limited expressiveness of Z/EVES.

Acknowledgment

This work was partly supported by China 973 under Grant 2007CB307101, 863 under Grant 2007AA01Z41, and Doctoral Program Foundation of University from Department of Education of China under Grant 200800041052. Also, this paper is based upon work in part supported by the National Science Foundation under Grant 0716527, Grant 0736877, Michigan Space Grant Consortium and Faculty Research Fellowship at Oakland University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, MSGC, or Oakland University.

References

[1] Common Criteria,www.commoncriteriaportal.org

[2] Trusted Computing Security Evaluation Criteria. *csrc.nist.gov/publications/secpubs/rainbow/*

[3] GB17859. www.ga.dl.gov.cn/djbh/GB17859-1999.doc

[4] Terasa F. Lunt, Dorothy E. Denning, Roger R. Schell,Mark Heckman, and William R. Shockley, "The SeaView Security Model," IEEE Transactions on software engineering vol.16.NO.6,June 1990.

[5] D. E. Bell and L. J. LaPadula, "Secure computer system: Unified exposition and multics interpretation," MTR-2997,Revision 1, Mar.1976.

[6] R. A. Whitehurst and T. F. Lunt, "The SeaView verification,"in Proc. Second Workshop Foundations of Computer Security. Fran-conia, NH, IEEE Computer Society Press, June 1989.

[7] Hong Zhu, et al.. Formal Specification and Verification of an Extended Security Policy Model for Database Systems. Proceedings of the 2008 Third Asia-Pacific Trusted Infrastructure Technologies Conference table of contents. pp 132-141

[8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," IEEE Computer, Vol. 29, Issue 2, pp.38–47, 1996.

[9] Elisa Bertino, Sushi1 Jajodia, and Pierangela Samarati, Supporting Multiple Access Control Policies in Database Systems, Proceedings of the 1996 IEEE Symposium on Security and Privacy,1998

[10] Min-A Jeong; Jung-Ja Kim; Yonggwan Won, A flexible database security system using multiple access control policies, PDCATapos;2003, LCNS, Springer Berlin / Heidelberg.Volume , Issue , 27-29 Aug. 2003

[11] PostgreSQL, open source for DBMS, *www.postgresql.org*

[12] Z/EVES,www.cs.kent.ac.uk/people/staff/gsn2/zeves/

Yongzhong He received Ph.D degree from Institute of Software, Chinese Academy of Sciences in 2006. Then he joined Beijing Jiaotong University as an assistant professor with School of Computer. His research interests are in computer security and cryptographic protocols.

Zhen Han is a professor with School of Computer at Beijing Jiaotong University. His research interests include computer security and trusted computing.

Huirong Fu joined Oakland University as an assistant professor in 2005. Her primary research interests are in information assurance and security, networks, Internet data centers, and multimedia system and services.

Guangzhi Qu joined Oakland University as an assistant professor in 2007. His research interests include system performance optimization and evaluation, parallel and distributed systems, data mining, and the science of computing.