MANAGING THE INTERTWINING AMONG USERS, ROLES, PERMISSIONS, AND USER RELATIONSHIPS IN AN INFORMATION FLOW CONTROL MODEL

Shih-Chien Chou

Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, TAIWAN
scchou@mail.ndhu.edu.tw

ABSTRACT

Information flow control prevents information leakage during the execution of an application. Many information flow control models are available and they offer useful features. In the past years, we identified that managing the intertwining among users, roles, permissions, and user relationships is essential. Since we cannot identify a model that manages the intertwining, we developed a new model to offer the feature. This paper gives definitions to the intertwining and proposes the model.

KEYWORDS

Security, information flow control, role-based access control (RBAC), prevent information leakage, user relationship

1. INTRODUCTION

Information flow control can ensure secure database interfaces [1] and secure information flows within/among operating systems [2-4]. It can also prevent information leakage for executing programs (applications) and web services [5-8]. This paper discusses only the models of preventing information leakage during application execution and excludes others.

When an application is being executed, users play roles in the application. A user playing a role may access the information that can be accessed by the role. For example, when a variable value is output and the variable can be accessed by the role "r1", then the user playing the role "r1" can obtain the output value. If sensitive information is managed by an application, information that can be accessed by a role should be controlled to prevent information leakage. The prevention can be achieved through information flow control. Generally, an information flow occurs when a value derived from a set of variables is assigned to a variable, an output device, or another system. Controlling information flows refers to ensuring that the flows will not leak sensitive information. In this regard, when a value derived from a set of variables is assigned to a variable, the variable receiving the value should be more restricted than or the same restricted as the variables in the variable set. When a value is output, the user that can access the output device should possess rights to read the value. And, when a value is passed as an argument to another system, the parameter receiving the argument should be more restricted than or the same restricted as the argument. This paper discusses the first type of assignment. The other two types of assignments are not within the scope of this paper.

Many information flow control models were developed. They offer useful features. Our research reveals that an information flow control model should offer the feature of managing the intertwining among users, roles, permissions, and user relationships. Managing the intertwining is essential but is generally ignored by existing models. We thus developed a new model. The model is developed based on RBAC (role-based access control) because RBAC is a super set of DAC (discretionary access control) and MAC (mandatory access control), which are frequently used in controlling information flows.

In RBAC, users play roles and a role possesses permissions. When a user plays a role during a session, the user possesses the permissions of the role. When a user changes role, his/her permissions change. We call the permission change caused by role change as *the intertwining among users, roles, and permissions.* Since RBAC assigns roles to users and assigns permissions to roles, changing user's role results in changing user's permissions. It thus seems that RBAC manages the intertwining mentioned above. Nevertheless, RBAC requires persons such as those in a company's monitoring team to change user's role. This may cause trouble. For example, suppose a company promotes a general customer to a VIP when the total consumption amount of the customer exceeds a threshold (the promotion corresponds to changing user's role). Also suppose that the company possesses thousands of customers. Then, handling the promotion becomes a heavy workload for the monitoring team. Moreover, handing the promotion by human beings may be error-prone. Therefore, changing user's role should be automatically handled by computer program. In this regard, we do not think that RBAC manages the intertwining mentioned above.

Managing the intertwining among users, roles, permissions, *and user relationships* is also an essential feature. Since RBAC does not manage user relationships, we cannot use RBAC to explain the intertwining. Generally, relationships such as friendship may exist among users. Changing user relationships might result in changing user's permissions. For example, suppose a company allows friends of a manager to obtain a special discount. Then, if the manager John and the customer Mary are not friends, John cannot read the discount rate to give Mary the discount. When John and Mary make friends (this results in a user relationship change), John can read the discount rate. In addition to the problem mentioned in the example, changing user relationships may result in an even serious problem. Since the problem is related to preventing indirect information leakage, we first describe the leakage.

Indirect information leakage occurs when information is leaked via the third one(s). In general, indirect information leakage can be prevented by the *join operation*. For example, suppose the information "inf1" can be read by one set of users and the information "inf2" can be read by another set. Then, the join operation allows the information derived from "inf1" and "inf2" to be read by users in the intersection of the two sets.

As mentioned above, a serious problem related to preventing indirect information leakage might occur when user relationships change. We continue the example in the previous paragraph to explain the problem. If John and Mary are not friends, John cannot read the discount rate for friends to give Mary the discount. That is, John possesses no permission to read the discount rate. Suppose at this time, the information "inf" is derived from the rate and other information that can be read by John. Then, according to the join operation, John is not allowed to read "inf". Suppose John and Mary make friends a certain time later (this results in changing user relationship). Then, John can read the discount rate this time because he can give Mary the discount now. A serious problem will result after the change, which is "should John be allowed to read the information 'inf' produced before?" The answer should be *Yes* because: (a) John can read the discount rate after the user relationship change and (b) "inf" is derived from the discount rate and other information that can be read by John. Since John can read all the information deriving "inf" after user relationship change, John should be allowed to read "inf" after the change. Allowing John to read "inf" invalidates the previous join operation because the operation disallowed John to read "inf". The invalidation requires one or more previous join operations to be redone. We call the join redoing caused by user relationship change as the *intertwining among users, roles, permissions, and user relationships*. When redoing the join operations, the current user relationships should be used as a reference. For example, the newly added friendship between John and Mary should be used to redo the join operation for the derived information "inf". The friendship will allow John to read "inf".

The above paragraphs describe the importance of managing: (a) the intertwining among users, roles, and permissions and (b) the intertwining among users, roles, permissions, and user relationships. According to our survey, no existing information flow control model manages the intertwining. This paper proposes an information flow control model InwRBAC (RBAC-based information flow control model that manages the intertwining among users, roles, permissions, and user relationships) that can manage the intertwining.

2. RELATED WORK

Mandatory access control (MAC) is useful in information flow control. The lattice model [9] is a MAC. A lattice is defined as (SC, $\rightarrow$, $\oplus$), in which "SC" is the set of security classes, "$\rightarrow$" is the "can flow" relationship, and "$\oplus$" is the join operator. The "can flow" relationship controls information flows and the join operator prevent indirect information leakage [10].

The model in [11] is based on discretionary access control (DAC), which controls information flows within object-oriented systems. ACLs (access control lists) of objects are used to compute ACLs of executions which are composed of object method(s). Possibly non-secure information flows are filtered out by a message filter. Interactions among executions are categorized into five modes to apply different security policies. Flexibility is added by allowing exceptions during or after method execution [12].

The decentralized label model [13] attaches labels to variables. The security levels of variables are shown in the labels. A label is composed of one or more policies that should be simultaneously obeyed. In general, a policy is composed of an owner and zero or more readers that can read the data. Both owners and readers are principals, which may be users, group of users, and so on.

RBAC [14] can also be used in access control. It is composed of users, roles, sessions, permissions, role hierarchies, user-role assignments (URA), role-permission assignments (RPA), and constraints. A role is composed of a set of permissions [14], which is a consequence of RPA. Roles are structured using the "$\geq$" relationship. If a relationship "x $\geq$ y" exists, "x" possesses all the permissions of "y". The "$\geq$" relationship can thus be used to construct role hierarchies. Roles are assigned to users, which result in URA. Users can establish sessions, within which a user possesses the permissions of the role assigned to him.

The model in [15] applies RBAC to control information flows in object-oriented systems. It classifies methods and derives a flow graph from which non-secure information flows can be identified. We also developed RBAC-based model [16]. It offers a read access rule to prevent information leakage and a write access rule to prevent information corruption. The model in [17] applies RBAC to control information flows within object-oriented systems. It classifies object methods and derives a flow graph from method invocations. From the graph, non-secure information flows can be identified.
Flume [2] is a decentralized information flow control (DIFC) model for operating systems. It tracks information flows in a system using tags and labels. The control granularity is detailed to processes (i.e., Flume regards the information input to and output from a process as a whole). The

secrecy tags prevent information leakage and the integrity tags prevent information corruption. The two types of controls are similar to the read and write control in our discussion. Flume also avoids information leaked to untrusted channel (e.g., sockets). The function of Laminar [3] is similar to that of Flume. Nevertheless, the control granularity is detailed to data structures (e.g., arrays) and system resources (e.g., sockets). Both Flume and Laminar are used in operating systems. Since our research focuses on embedding an information flow control model within a program to prevent information leakage, other models including Flume and Laminar are excluded in this paper.

As we have emphasized, no existing model handle the intertwining, we thus develop the new model InwRBAC.

3. INWRBAC

InwRBAC is developed for object-oriented systems. When a system is being executed, the InwRBAC security checker ensures secure information access and manages the intertwining mentioned in section 1.

When an object-oriented system is not executing, it is composed of classes and inheritance relationships. When an object-oriented system is being executed, objects are instantiated from classes and messages are passed among objects. Since InwRBAC prevents information leakage when a system is being executed, we define an executing object-oriented system *ScPg* embedded with InwRBAC as follows.

Definition 1. *ScPg* = (*OBJ*, *MSG*, *USR*, *RLE*, *UR*, *ACLS*, *URA*, *DSOURCE*, *DSTA*, *JH*), in which

a. *OBJ* is the set of objects.
b. *MSG* is the set of messages.
c. *USR* is the set of user. Users play roles.
d. *RLE* is the set of roles. InwRBAC lets a role be an object in an object-oriented system.
e. *UR* is the set of user relationships. A user relationship *ur* is defined as $ur \in (2^{USR} - \phi)$.
f. *ACLS* is the set of ACLs associated with variables. A variable may be an object attribute, a private variable, or a method return value. ACLs ensure secure access of variables. An ACL is composed of a read access control list (RACL) to control read access and a write access control list (WACL) to control write access. The ACL ACL_{var} associated with the variable *var* is defined as "$ACL_{var} = (RACL_{var}, WACL_{var}, UR_{var})$", in which:

   (1) $RACL_{var} \in 2^{USRxRLE}$, in which "x" represents Cartesian product. Since multiple users may play the same role, RACL has this definition to distinguish users. A user playing a role appearing in $RACL_{var}$ is allowed to read *var*.
   (2) $WACL_{var} \in 2^{USRxRLE}$. A user playing a role appearing in $WACL_{var}$ is allowed to write *var*. In general, existing models do not control write access. They generally follow the no write down rule [18]. We think that write operations may corrupt information and thus propose that only the roles trusted by a variable can write the variable.
   (3) $UR_{var} \in (2^{UR} - \phi)$. $RACL_{var}$ and $WACL_{var}$ are valid in a user relationship *ur* if $ur \in UR_{var}$ because user relationships affect user permissions as mentioned in section 1.

g. *URA* is a set of user to role assignments, which is defined as "$USR \rightarrow 2^{RLE}$".
h. *DSOURCE* is a set of data source. Each variable is associated with a DSOURCE to facilitate controlling write access. It records the roles that write data to the variable.

i. *DSTA* is a set of statements to automatically monitor the change of roles and user relationships. It is composed of the following statements:

   (1) *setRole(user, role)*. It assigns a role to a user.
   (2) *isRole(user, role)*. It checks the role of a user. It is a Boolean expression.
   (3) *setRelationship(relationship, usr_list)*. It establishes relationships among users.
   (4) *breakRelationship(relationship, usr_list)*. It breaks relationships among users.
   (5) *withinRelationship(relationship, usr_list)*. It checks whether the users are within a specific relationship. It is a Boolean expression.

j. *JH* records the histories of join operations. It facilitates redoing join operations invalidated by user relationship change.

3.1 Secure Information Access in InwRBAC

InwRBAC ensures that both direct and indirect information flows are secure. We first describe the security of direct information flows and then describe that of indirect ones.
Suppose a direct information flow is induced by assigning to the variable "d_var" the value derived from the variables in the set "$\{var_i \mid var_i$ is a variable and i is between 1 and $n\}$". Then, the information flow is considered secure if the following two secure access conditions are simultaneously true. To define the conditions, we let: (a) the ACL and DSOURCE of the variable "d_var" be respectively "($RACL_{d\_var}$, $WACL_{d\_var}$, UR_{d_var})" and "$DSOURCE_{d\_var}$", (b) the ACL and DSOURCE of "var_i" be respectively "($RACL_{var_i}$, $WACL_{var_i}$, UR_{var_i})" and " $DSOURCE_{var_i}$ ", and (c) "UR_{sub}" is a non-empty user relationship set.

First secure access condition: $\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{var_i} \cap UR_{d\_var})$ so that

$RACL_{d\_var} \subseteq \bigcap_{i=1}^{n} RACL_{var_i}$

Second secure access condition: $\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{var_i} \cap UR_{d\_var})$ so that

$WACL_{d\_var} \supseteq \bigcup_{i=1}^{n} DSOURCE_{var_i}$

The first secure access condition controls read access. The condition "$RACL_{d\_var} \subseteq \bigcap_{i=1}^{n} RACL_{var_i}$ " requires that "d_var" should be the same restricted as or more restricted than those of the variables in the variable set "$\{var_i \mid var_i$ is a variable and i is between 1 and $n\}$". Since RACLs and WACLs are only valid under certain user relationships (see item *f* in Definition 1), the ACL of "d_var" and those of the variables in the variable set mentioned above should be simultaneously valid in certain user relationship(s). This results in the requirement "$\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{var_i} \cap UR_{d\_var})$", in which "$UR_{sub}$" should not be empty. The second secure access condition controls write access. It requires that the data sources of the variables deriving the value that is assigned to "d_var" should be within "$\mathrm{WACL}_{\mathrm{d\_var}}$", because the data derived from the variables are written to "d_var".

After assigning the derived value to "d_var", the ACL of "d_var" should be changed by the join operation [13] to prevent indirect information leakage. We use the symbol " $\oplus$ " to represent the join operator. With join, "$\mathrm{ACL}_{\mathrm{d\_var}}$" will be changed to " $\bigoplus_{i=1}^{n} ACL_{var_i}$ " in which the value assigned to "d_var" is derived from variables in the set "$\{var_i \mid var_i$ is a variable and i is between 1 and $n\}$".

Definition 2. $\bigoplus_{i=1}^{n} ACL_{\text{var}_i} = (\bigcap_{i=1}^{n} RACL_{\text{var}_i}, \bigcup_{i=1}^{n} WACL_{\text{var}_i}, \bigcap_{i=1}^{n} UR_{\text{var}_i})$

The join operation trusts less or the same readers. Therefore, join will not lower down security level. On the other hand, the operation trusts more writers. This is reasonable because a writer that can write a variable should be regarded as a trusted data source for the data derived from the variable. In addition to joining ACLs, the DSOURCE of "d_var" will be adjusted as follows: $DSOURCE_{d\_\text{var}} = \bigcup_{i=1}^{n} DSOURCE_{\text{var}_i}$. The primary objective of InwRBAC is preventing information leakage. Below we prove that InwRBAC achieves the prevention.

Lemma 1. InwRBAC prevents information leakage, including direct and indirect leakage.

Proof. Direct leakage occurs when a variable is read by a role played by a user in which the role possesses no permission to read the variable. The case will not happen because of the first secure access condition.

Indirect information leakage occurs when a variable *var*$_2$ leaks the variable *var*$_1$ to the user *usr*$_1$ under the assumptions: (a) the role *rle*$_1$ played by *usr*$_1$ is initially allowed to read the variable *var*$_2$, (b) *rle*$_1$ is not allowed to read the variable *var*$_1$, and (c) *var*$_2$ is derived from *var*$_1$. To prove that indirect information leakage is prevented, we let the ACL of *var*$_1$ be "($RACL_{\text{var}_1}$, $WACL_{\text{var}_1}$, UR_{var_1})". We also let the ACL of *var*$_2$ after it is derived from *var*$_1$ be "($RACL_{\text{var}_2}$, $WACL_{\text{var}_2}$, UR_{var_2})". Since *rle*$_1$ is not allowed to read *var*$_1$, the condition "(usr_1, rle_1)$\notin RACL_{\text{var}_1}$" is true. If indirect information leakage occurs among *rle*$_1$, *var*$_1$, and *var*$_2$, then *rle*$_1$ can read *var*$_2$ after *var*$_2$ is derived from *var*$_1$. If this is the case, the condition "(usr_1, rle_1)$\in RACL_{\text{var}_2}$" should be true after *var*$_2$ is derived from *var*$_1$. According to the join operation in Definition 2 and the condition "(usr_1, rle_1)$\notin RACL_{\text{var}_1}$" mentioned above, the condition "(usr_1, rle_1)$\in RACL_{\text{var}_2}$" is false after *var*$_2$ is derived from *var*$_1$. The rationale is that $RACL_{\text{var}_2}$ is the intersection of $RACL_{\text{var}_1}$ and other RACLs because *var*$_2$ is derived from *var*$_1$. This proves that InwRBAC prevents indirect information leakage. #

3.2 Managing Intertwining among Users, Roles, and Permissions (Automatically Changing User-Role Assignments)

The statements in DSTA (see item *i* of Definition 1) manage the intertwining among users, roles, and permissions. We use examples below to explain the use of DSTA.

Example 1.

Suppose when a general customer will be automatically switched to a VIP if his/her total consumption amount exceeds a threshold. In this case, a program can use the following statement to change role automatically. Note that the statements "isRole" and "setRole" are DSTA statements.

```
    if ((!isRole("Mary", "VIP")) && (t_con_amount_Mary >= threshold)) setRole("Mary",
"VIP");
```

Example 2.

Suppose friends can get special discount from a manager. Also suppose John plays the manager role and Mary plays the customer role. Using the following statements, Mary can get the special discount from John. Note that the statement “withinRelationship” and “setRelationship” are DSTA statements.
if (!withinRelationship(“friend”, “John”, “Mary”)) setRelationship(“friend”, “John”, “Mary”);
if(withinRelationship(“friend”, “john”, “Mary”)) price = price*specialDiscount; /* get special discount */

On the other hand, the following statement will cause Mary to lose the permission of obtaining the special discount from John:
breakRelationship(“friend”, “John”, “Mary”);

3.3 Managing Intertwining among Users, Roles, Permissions, and User Relationships (Redoing Join Operations to Correct ACLs When User Relationship Change)

As described in section 1, user relationship change may invalidate join operations done previously. The primary objective of managing the intertwining among users, roles, permissions, and user relationships is to redo the invalidated join operations and regain the correct ACLs. The redoing can be facilitated by the component *JH* in Definition 1. Below we describe the redoing procedure.

Suppose that a variable *d_var* is derived from the variables in the set VAR_1. Since a variable may be derived from other variables, we suppose that the variables deriving the variables in VAR_1 constitute the set VAR_2, the variables deriving the variable in VAR_2 constitute the set VAR_3, and so on. The above derivation process results in ripple effects. The effects end when VAR_m is empty. To simplify the following description, we let *UVAR* be the set “$\cup_{i=1}^{n} VAR_i \cup d\_var$”, in which the ripple effects end at VAR_{n+1}. We also suppose that the earliest time the variable var_i being a derived variable is t_{var_i}, in which $var_i \in UVAR$. From t_{var_i} down to the current time, every join operation in which var_i is a derived variable should be redone. When redoing join operations, the current user relationships should be used as a reference because ACLs should be correct under the current user relationships (see the example in section 1). Below we define the component *JH*.

Definition 3. An element *jh* of *JH* in Definition 1 is defined below:
$jh = (t, d\_var, \{(var, ACL_{var}) \mid var$ is a variable that derives d_var and ACL_{var} is the ACL of var at the time $t\}, tag)$, in which

(1) *t* is the time that a join operation is done.

(2) *d_var* is the derived variable.

(3) $\{(var, ACL_{var})\}$ is the set of variable and their ACLs that derive *d_var*.

(4) If *tag* is set, *t* is the earliest time that *d_var* is a derived variable.

According to Definitions 3 and the description in the second paragraph of this subsection, we conclude the redoing of join operations using Algorithm 1.

Algorithm 1. Join operation redoing algorithm

```
1. Input data:

   1.1 $VAR_1 = \{var \mid var$ is a variable to derive $d\_var\}$
   1.2 $d\_var$: the variable derived from the variable in the set $VAR_1$

2. Algorithm:

   2.1 Backtrack $JH$ to identify $VAR_2$ through $VAR_n$ following the procedure described in the
       second paragraph of this subsection.
   2.2 Let $UVAR$ be the set "$\cup_{i=1}^{n} VAR_i \cup d\_var$".
   2.3 For each $var_i \in UVAR$, do
       2.3.1 Backtrack $JH$ to identify the earliest time $t_{\mathrm{var}_i}$ that $var_i$ is a derived variable. The
             tags in $JH$ (see Definition 3) facilitate the identification.
       2.3.2 From the time $t_{\mathrm{var}_i}$ down to the current time, mark the join operations in $JH$ in
             which $var_i$ is a derived variable.
   2.4 End do
   2.5 Redo the marked join operations recorded in $JH$ from the earliest time a join operation is
       marked down to the current time. When redoing the join operations, the current user
       relationships are used as a reference.
   2.6 Remove the user-role pairs from a newly produced ACL's RACL and WACL if the user
       does not appear in any user relationship of the ACL.
```

Lemma 2. Algorithm 1 is correct.

Proof. We prove the correctness by induction.

a. Suppose only one variable d_var is within $UVAR$ in line 2.2 of Algorithm 1. Then, d_var never plays the role of a derived variable (otherwise, there will be other variable in $UVAR$). In this case, d_var's ACL is unchanged during application execution. An unchanged ACL is correct. The rationale is that an ACL may be invalidated only when the following two conditions are simultaneously true: (a) one or more user relationships change and (b) the ACL is changed by join operation. If an ACL is unchanged, the second condition is false. Therefore, an unchanged ACL is always correct.

b. Suppose Algorithm 1 is correct when there are $(k\text{-}1)$ elements in the set $UVAR$, in which $UVAR = \{var_i \mid var_i$ is a variable, i is between 1 and $(k\text{-}1)$, and $(k\text{-}1) > 1\}$. The correctness of Algorithm 1 under the above assumption implies that, before the variable d_var is derived using the variables in $UVAR$, Algorithm 1 corrects ACLs associated with the variables in $UVAR$ by referring to the current user relationships. Let's add a variable var_k to the original $UVAR$ (we let $NewUVAR = UVAR \cup \{var_k\}$). We prove that Algorithm 1 is correct for a k-element $NewUVAR$ as follows. First, according to the assumption in the previous paragraph, the ACLs associated with the variables in $NewUVAR$ excluding var_k are correct after join redoing. Second, the ACL associated with var_k is correct because: (1) if var_k has not been changed, var_k is correct and (2) if var_k has been changed, lines 2.3 through 2.6 of Algorithm 1 corrects the ACL associated with var_k. #

Below we depict the use of Algorithm 1 using a simple example. Suppose that a customer can obtain a friend discount from a manager if the users playing the roles of manager and customer are friends. In this case, the manager role possesses permissions to read the friend discount rate (note that no one can change the rate because it is determined by the supervisor of the company). On the other hand, if the users playing the roles of manager and customer are not friends, the manager role possesses no permission to read the friend discount rate. We use the following scenario to depict the redoing of join operations when user relationships change. In the scenario, we use "dRate" to represent the friend discount rate.

a. Both John and Tom play a manager role in a company and Mary plays a customer role.
b. Initially, John and Mary are friends. Moreover, Tom and Mary are also friends.
c. The information "inf1" can be read by the roles played by both John and Tom.
d. The information "inf" is derived from "dRate" and "inf1".
e. The role played by John intends to read "inf".
f. John and Mary break friendship.
g. The role played by John intends to read "inf".
h. The role played by Tom intends to read "inf".

According to the assumption of the example and the description in items *a* and *b* of the above scenario, the ACL of "dRate" should be assigned as follows:

{(John, manager), (Tom, manager); ; {friend; John, Mary}, {friend; Tom, Mary}}

In the ACL, the list before the first semicolon is the RACL of "dRate", which is "*(John, manager), (Tom, manager)*". The list between the first and second semicolons is the WACL of the "dRate", which is an empty set in this example. And, the sets after the second semicolon are user relationship sets, which are "*{friend; John, Mary}, {friend; Tom, Mary}*". In a user relationship set, the term before the semicolon is the name of the user relationship and those after the semicolon are users possessing the relationship. For example, the user relationship set "*{friend; Tom, Mary}*" states that Tom and Mary are friends.

According to item *c* of the above scenario, the ACL of "inf1" should be assigned as follows:
{(John, manager), (Tom, manager); ; U}

The symbol "*U*" in the user relationship field means that the RACL and WACL are not constrained by user relationships. That is, the ACL is always valid.

After finishing the operation of item *d* in the above scenario, the ACL of "inf" will be assigned as follows according to the join operation in Definition 2:

{(John, manager), (Tom, manager); ; {friend; John, Mary}, {friend; Tom, Mary}}

In addition to assigning ACL to "inf", finishing the operation of item *d* in the above scenario causes the JH database to record a join operation as follows (see Definition 3 for the contents of a JH record). Here we suppose that the join operation was performed at the time "t1". Moreover, we set the tag to be true because "t1" is the earliest time that "inf" is a derived variable.
(t1, inf, {(dRate, {(John, manager), (Tom, manager); ; {friend; John, Mary}, {friend; Tom, Mary}}), (inf1, {(John, manager), (Tom, manager); ; U})}), tagTrue)

The operation in item *e* of the above scenario can be executed because no user relationship is changed and the ACL of "inf" allow the role played by John to read "inf". Nevertheless, whether the operation in item *g* of the above scenario can be executed should be checked, although the operation is the same as that in item *e* of the above scenario. The necessity of the checking is because user relationships change in item *f* of the above scenario. Let's trace Algorithm 1 to determine whether the operation in item *g* of the above scenario can be executed. The algorithm will identify that: (a) *UVAR* = {*inf*, *dRate*, *inf1*} because *inf* is derived from *dRate* and *inf1* and *UVAR* is the union of *inf* and the variables deriving *inf*, (b) $t_{inf} = t1$, (c) the join operation recorded in the JH database should be redone, and (d) the redoing should start from *t1* down to the current time by referring to the current user relationships.

According to the join operation in Definition 2, redoing the join operation in the JH database results in the following ACL for the "inf":

{(John, manager), (Tom, manager); ; {friend; Tom, Mary}}

In the ACL, John appears in the RACL but not in the user relationship set because he broke friendship with Mary (see item *f* in the above scenario). Since the validity of RACL and WACL are under the control of user relationships (see item *f* of Definition 1), the absence of John from the user relationship set means that John in the RACL is invalid. Therefore, the ACL of "inf" should be adjusted as follows (see line 2.6 of Algorithm 1):

{(Tom, manager); ; {friend; Tom, Mary}}

With the adjusted ACL, John is not in the RACL of "inf" and therefore the operation in item *g* of the above scenario cannot be executed. That is, John cannot read "inf" after he broke friendship with Mary. On the other hand, the above redoing process retains the reading permission on "inf" for "Tom". Therefore, the operation in item *h* of the above scenario can be executed.

4. EVALUATION

We evaluate InwRBAC using a simple order management system. In the system, we assumed that a customer could get special discount form his/her manager friends. We assigned the roles "manager" and "customer" to many users. We required twenty students to program the example and required them to break and establish user relationships frequently. We also required students to inject non-secure information flows into their programs. We then required students to record the percentage of non-secure information flows that were identified, in which a percentage in the figure is the value "(identified non-secure information flows) / (injected non-secure information flows)". The experiment result is shown in Figure 1. When we check the non-secure information flows identified by InwRBAC, every injected one was identified. Note that the percentages of non-secure information flows identified were generally larger than one. This is a consequence that students usually committed errors. That is, they accidentally wrote statements that cause non-secure information flows. According to the experiment result shown in Figure 1, we think that InwRBAC is useful.

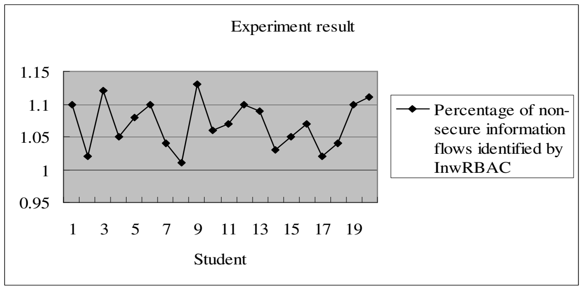

Figure 1. Experiment result

5. CONCLUSION AND FUTURE WORK

This paper discusses the importance of managing the intertwining among users, roles, permissions, and user relationships. In addition, the paper proposes a model InwRBAC for the management. Our experiment showed that the model is useful. Nevertheless, there are still problems to solve, including those listed below:

a. Problems raised by covert channels. A covert channel refers to media such as disk file that leaks information. We use an example to explain the channels. Suppose in a computer system, when an application is abnormally stopped, the operating system dumps the memory space occupied by the application to a disk file for debugging. Nevertheless, if an unauthorized user retrieves the file, he may steal the application's sensitive information by decoding the dumped file. An information flow control model cannot independently protect the dumped file. The model should cooperate with the operating system for the protection.

b. Problems raised by timing channels. Stealing information through timing channel is achieved by tracking the execution time of program segments. We use the following C program segment to explain this:

```
for(i = 0, c=0; i < 1000; i++) if (a == 1) c += 1;
```

   In the above program segment, tracking the execution time of the "for" loop can guess whether the value of "a" is 1 (i.e., a longer execution time implies that "a" is 1). To prevent this guessing, an information flow control model should identify every program segment that may leaks information through timing channel. The model should then adjust the program segment to prevent the leakage. For example, the above program segment can be adjusted by adding code as follows to prevent guessing the value of "a". The added code may be garbage code.

```
for(i = 0, c=0; i < 1000; i++) if (a == 1) c += 1;
else c += 0; /* this statement is garbage code */
```

   To do the above adjustment, the entire application should be scanned, sliced, analysed, and then adjusted. The adjustment is a difficult job.

c. Problems raised by reused objects obtained from a CORBA broker. The reused objects in this case are difficult to control. The rationale is that a reusable object can be used by multiple applications. Suppose an application uses the object "o1", information of the application may be passed to the object via arguments. Information of the arguments may be leaked when other applications use "o1" sometime later. One of my students proposed a solution to the issue as described below. After an application uses an object "o1" by invoking its method, the application invokes the method once more by passing random numbers as arguments. The purpose of the second invocation is to scramble the information passed to "o1". Nevertheless, other applications may use "o1" before the scrambling, which causes the solution to fail.

Although solving the above problems is difficult, we are finding possible solutions for them. We hope that we can develop a strong information flow control model in the future.

REFERENCES

[1] P. Li and S. Zdancewic, “Practical Information-flow Control in Web-based Information Systems”, *18’th IEEE Computer Security Foundation Workshop*, 2005.

[2] M. Krohn, A. Yip, M. Brodsky, and N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, “Information Flow Control for Standard OS Abstractions”, *SOSP’07*, 2007.

[3] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, “Laminar: Practical Fine-Grained Decentralized Information Flow Control”, *PLDI’09*, 2009.

[4] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, “Securing Distributed Systems with Information Flow Control”, *7’th Symposium on Operating System Design and Imoplementation*, 2006.

[5] S. –C. Chou and C. –H. Huang, “An Extended XACML Model to Ensure Secure Information Access for Web Services”, *Journal of Systems and Software*, vol. 83, no. 1, pp. 77-84, 2010.

[6] S. –C. Chou, “Dynamically Preventing Information Leakage for Web Services using Lattice”, to appear in *5’th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, 2010.

[7] W. She, I. –L. Yen, B. Thuraisingham, and E. Bertino, “The SCIFC Model for Information Flow Control in Web Service Composition”, *2009 IEEE International Conference on Web Services*, 2009.

[8] W. She, I. –L. Yen, B. Thuraisingham, and E. Bertino, “Effective and Efficient Implementation of an Information Flow Control Protocol for Service Composition”, *IEEE International Conference on Service-Oriented Computing and Applications*, 2009.

[9] D. E. Denning, “A Lattice Model of Secure Information Flow”, *Comm. ACM*, vol. 19, no. 5, pp. 236-243, 1976.

[10] A. Myers and B. Liskov, “Complete, Safe Information Flow with Decentralized Labels”, *14’th IEEE Symp. Security and Privacy*, pp. 186-197, 1998.

[11] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia, “Information Flow Control in Object-Oriented Systems”, *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, pp.524-538, Jul./Aug. 1997.

[12] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia, “Providing Flexibility in Information flow control for Object-Oriented Systems”, *13’th IEEE Symp. Security and Privacy*, pp. 130-140, 1997.

[13] A. Myers and B. Liskov, “Protecting Privacy using the Decentralized Label Model”, *ACM Trans. Software Eng. Methodology*, vol. 9, no. 4, pp. 410-442, 2000.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, andC. E. Youman, “Role-Based Access Control Models”, *IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.

[15] K. Izaki, K. Tanaka, and M. Takizawa, “Information Flow Control in Role-Based Model for Distributed Objects”, *8’th International Conf. Parallel and Distributed Systems*, pp. 363-370, 2001.

[16] S. -C. Chou, “Embedding Role-Based Access Control Model in Object-Oriented Systems to Protect Privacy”, *Journal of Systems and Software*, 71(1-2), 143-161, Apr. 2004.

[17] K. Izaki, K. Tanaka, and M. Takizawa, “Information Flow Control in Role-Based Model for Distributed Objects”, *Proc. 8’th International Conf. Parallel and Distributed Systems*, pp. 363-370, 2001.

[18] D. E. Bell and L. J. LaPadula, “Secure Computer Systems: Unified Exposition and Multics Interpretation”, *technique report, Mitre Corp.*, Mar. 1976. http://csrc.nist.gov/publications/history/bell76.pdf

Author

Shih-Chien Chou is currently a professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. His research interests include software engineering, process environment, software reuse, and information flow control.