JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 147-161 (2006)

MRBAC/AR: an Information Flow Control Model to Prevent Both Intra- and Inter-Application Information Leakage

SHIH-CHIEN CHOU
Department of Computer Science and Information Engineering
National Dong Hwa University
Hualien, 974 Taiwan
E-mail: scchou@mail.ndhu.edu.tw

Preventing information leakage during program execution is essential for modern applications. This paper proposes a model to prevent information leakage for object-oriented systems, which is based on role-based access control (RBAC). It is named MRBAC/AR (modified RBAC for both intrA- and inteR-application information flow control) because it is a modification of RBAC96. It offers the following features: (a) adapting to dynamic object state change, (b) adapting to dynamic role change, (c) avoiding Trojan horses, (d) detailing access control granularity to variables, (e) controlling method invocation through argument sensitivity, (f) allowing declassification, (g) allowing purpose-oriented method invocation, (h) precisely controlling write access, and (i) preventing both intra- and inter-application information leakage. We evaluated MRBAC/AR through experiments. The evaluation result is also shown in this paper.

Keywords: information security, access control, information flow control, prevent information leakage, indirect information leakage

1. INTRODUCTION

In the old days, software engineers primarily developed functionally correct systems. This development attitude should now be changed. A modern system should be protected against various attacks such as: (a) attack from outside a system such as unauthorized users, hackers, viruses, worms, and so on, (b) attack from dishonest programmers and system software because they may create trap doors to leak information, and (c) attack from inside a system (e.g., legal users of the system) when the system is running. Preventing the third type of attack is essential because systems may manage sensitive information. Our research focuses on preventing the third type of attack for object-oriented systems and excludes the other types of attack. That is, we assume that the network is secure and the programmers and system software are honest.

Preventing the third type of attack corresponds to *preventing information leakage within an application* when the application is *running*. It is about access control, which is a language-based security issue [1] because an access control model should be embedded in a language for implementing a secure application. Access control within a system can be achieved through *information flow control*. Many information flow control models

Received November 7, 2003; accepted May 24, 2004.
Communicated by Shiuh-Pyng Shieh.

have been developed [2-10]. We involved in the research of information flow control for years and propose that an information flow control model should offer the following three types of features: (a) dynamic features, (b) static features, and (c) controlling information flows among applications. The first two types of features are *intra-application features* and the third is an *inter-application feature*. While intra-application information flow control prevents information leakage within an application, inter-application information flow control prevents information leakage among applications. Before further discussion of the control, we give an example used throughout the paper. Suppose two systems are in a company. The first system manages employee information and the second produces employee salary reports. Within the first system, an employee may be a worker or a manager. Every worker is assigned to a manager. When a manager monitors a worker assigned to him, the manager can read the worker's personal information, general information, and salary, and can change the salary. If the manager browses the worker's information for non-monitoring purposes, he can only read the worker's general information and salary. If a worker is not assigned to a manager, the manager can only read the worker's general information and salary. For evaluation purposes, statisticians can read employees' salaries to produce a salary distribution for everyone to read. In the second system, employee salaries are obtained from the first system to produce salary reports.

If the above systems are implemented as object-oriented applications, the following features should be offered, in which the first three are dynamic features, the next five are static ones, and the last is an inter-application feature.

a) Adapt to dynamic object state change. An object state is *a snapshot of objects and object relationships at a certain time*. We use the example above to explain the need for adaptation. Assume that worker "*w*1" is assigned to manager "*m*1". With this object state, "*m*1" is allowed to read the personal information of "*w*1". Suppose after a period of time, "*w*1" is re-assigned to manager "*m*2". With this object state, "*m*1" is no longer allowed to read the personal information of "*w*1". The example reveals that *changing object state results in changing access rights*. To adapt to dynamic object state change, access rights should be allowed to change during program execution.
b) Adapt to dynamic role change. In an object-oriented system, an object or object method plays a role. Dynamic role change refers to changing object or method's role during program execution. Since role change may result in access right change (e.g., access rights of "*w*1" will change when he changes role from "worker" to "manager"), access rights should be allowed to change during program execution to adapt to dynamic role change.
c) Avoid Trojan horses. A Trojan horse occurs when information is leaked indirectly. For example, suppose object "*o*1" is allowed to read the information of "*o*2," and "*o*2" is allowed to read the information of "*o*3," but "*o*1" is not allowed to read the information of "*o*3". Then, the information of "*o*3" may be leaked to "*o*1" via "*o*2" when "*o*1" accesses "*o*2" after "*o*2" reads the information of "*o*3". To prevent the leakage, after "*o*2" reads the information of "*o*3," the access rights of "*o*2" should be changed to prevent "*o*1" from reading the information of "*o*2". This change avoids Trojan horses.
d) Detail the granularity of control to variables. Information flow control models con-

trol the access of information stored in variables. Since different variables may be of different sensitivity [8], variables should be protected independently.

e) Control method invocation through argument sensitivity. For example, a manager can change the salary of a worker assigned to him. In the change, the manager invokes the method "employee.change_salary" using the attribute "employee.new_sal- ary" as an argument. If another attribute is used as an argument in the invocation, the invocation may be invalid because different variables carry different information for different purposes.

f) Allow declassification. Declassification refers to downgrading security level of information [2]. In the example above, a statistician can read employee salaries to produce a salary distribution for every reader to read. When the statistician is computing the salary distribution, the join operation [2] will change the access rights of the salary distribution to prevent Trojan horses. According to the join operation, the salary distribution is impossible for every reader to read. To allow every reader to read the distribution, security level of the salary distribution should be declassified.

g) Allow purpose-oriented method invocation [7]. With this feature, invoking a method may be allowed for some methods but disallowed for others, even when the invokers belong to the same object. For example, when a manager is browsing the information of a worker assigned to him, the manager can read the worker's general information using the worker's method(s). On the other hand, the manager cannot read the worker's personal information using the worker's method(s).

h) Control write access. Most models merely obey the "no write down" rule [3]. Since write access is destructive, it should be carefully controlled to prevent data corruption. We propose that only the data sources trusted by a variable can write the variable.

i) Prevent information leakage among applications. This feature is necessary because applications may cooperate (communicate), during which applications may exchange information. We use the example above to explain the need of this feature. Suppose the second system obtains an employee's salary using the employee's name as an argument passed to the first system. The first system then returns the employee's salary to the second system. Also suppose that workers cannot access employee salaries. With this limitation, the return values of the first system should not be leaked to workers within the second system.

We developed information flow control models for object-oriented systems based on role-based access control (RBAC) [9, 10]. The models we developed fail to offer the features of adapting to dynamic role change, allowing declassification, and preventing inter-application information leakage. Since the missing features are essential, we enhanced the models to obtain a new one called MRBAC/AR (modified RBAC for both intrA- and inteR-application information flow control). This paper proposes MRBAC/AR.

2. RELATED WORK

The model in [4] controls information flows in object-oriented systems. It uses ACLs of objects to compute ACLs of executions. A message filter is used to filter out

possibly non-secure information flows. Since the computation of an execution's ACL takes information propagation into consideration, Trojan horses are avoided. The model uses different modes of information flow control to loosen the restrictions of MAC. Flexibility is added to the model by allowing exceptions [5].

The purpose-oriented model [7] proposes that invoking a method may be allowed for some methods but disallowed for others, even when the invokers belong to the same object. This consideration is correct, because the security levels of an object's methods may be different [8]. Different methods can thus access information at different security levels. The model uses object methods to create a flow graph, from which non-secure information flows can be identified.

The decentralized label approach [2] marks the security levels of variables using labels. A label is composed of policies, which should be simultaneously obeyed. A policy in a label is composed of an owner and zero or more readers that are allowed to access the data. Both owners and readers are principals, which may be users and group of users. Principals are grouped into hierarchies using act-for relationships. The join operation is used to avoid Trojan horses. Declassification is allowed. Write access is controlled.

RBAC [11, 12] is also used in information flow control. In RBAC, a *role* is a collection of *permissions* [12]. A role can establish one or more *sessions*. During a session, a user playing a role possesses the permissions of the role. The original design of RBAC is for database security. Since the access of a database is generally controlled by database administrators who are people, assigning permissions to roles is generally achieved by humans. Although the assignments of permissions to roles can be changed, *the change needs the involvement of one or more persons*. Therefore, RBAC cannot adapt to dynamic object state change when using it to control information flows for software systems. The rationale is that humans are not capable of making the change when a program is executing, especially when the frequency of change is high. In addition, adapting to dynamic role change cannot be achieved by the general cases of RBAC because the change needs the intervention of persons. Since the original design of RBAC was not for object-oriented system information flow control, the general cases of RBAC do not offer most of the features mentioned in section 1.

The model in [6] applies RBAC to access control within object-based systems. It classifies object methods and derives a flow graph from method invocations. From the graph, non-secure information flows can be identified.

3. MRBAC/AR

MRBAC/AR is based on RBAC96 [11]. A difficult problem to solve by MRBAC/AR is adapting to dynamic object state change. We found that class relationships [13] are useful in the adaptation. In using class relationships for the adaptation, every relationship should be associated with an *access control policy*. An access control policy is composed of access control rules. For example, *a manager is allowed to change a worker's salary when the manager is monitoring the worker* is an access control rule of the employee management system in the example of section 1.

When using class relationships in adapting to dynamic object state change, class relationships should be instantiated to link objects. Objects linked by an instance of a class

relationship obey the relationship's access control policy. When the relationship linking two or more objects changes, the access control policy must be changed. This corresponds to adapting to dynamic object state change. MRBAC/AR defines an *instance of a class relationship* as a *session*. When a class relationship is instantiated to link objects, a session is established among the objects. With this definition, changing object state corresponds to changing sessions. When objects change session, the access control policy changes. This change accomplishes the adaptation of dynamic object state change.

As described above, every class relationship is associated with an access control policy. The access control policies of all class relationships in a system constitute the access control policy of the system. In MRBAC/AR, information flows within a session is allowed whereas those among sessions are prohibited. Moreover, information flows among objects within a session should obey the policy of the class relationship from which the session is instantiated.

By now, sessions and their access control policies have been defined. What should still be defined are permissions and roles. A permission is composed of a variable and its access rights. The access right of a variable may be "*R*" (for "read"), "*W*" (for "write"), or "*RW*" (for both "read" and "write"). A role is a set of permissions. *The role is played by an object method* because methods manipulate variables. An object is defined as a *composite role* because it contains methods. Since roles in MRBAC/AR are object methods, dynamically changing a role within an object causes other roles, i.e., methods, in the same object to change. For example, when a worker becomes a manager, every method in the worker object changes role. Therefore, *changing role in MRBAC/AR corresponds to changing composite role.*

In addition to the components mentioned above, MRBAC/AR associates one more component DSOURCE (data source) with each variable to facilitate write access control. The additional component records the methods from which a variable's data are derived.

Definition 1 *MRBAC/AR* = (*RELATIONSHIP*, *SESSION*, *SESSION_OBJECT_ MAPPING*, *CONSTRAINT*, *DSOURCE*), in which

a) *RELATIONSHIP* is a set of class relationships. A relationship can be instantiated to create sessions. Definition 2 defines a class relationship.
b) *SESSION* is a set of sessions. Each session is an instance of a class relationship.
c) *SESSION_OBJECT_MAPPING* is a set of functions, each of which maps a session to the objects that are within the session.
d) *CONSTRAINT* is the set of constraints.
e) *DSOURCE* is the set of data sources.

Definition 2 The *RELATIONSHIP* component in MRBAC/AR is the set of class relationships. A class relationship rel_i is defined below:

rel_i = (*NAME*, *CLASS*, *METHOD*, *VARIABLE*, *PERMISSION*, *ROLE*, *METHOD_ROLE_MAPPING*, *COMPOSITE_ROLE*, *DECLASSIFICATION*), in which

a) *NAME* is the name of the relationship.
b) *CLASS* is the set of classes linked by the relationship. A composite role name is asso-

ciated with a class. Instances of the class play the composite role.
c) *METHOD* is the set of class methods. A method belongs to a class.
d) *VARIABLE* is the set containing attributes, method variables, and method return values. A variable belongs to a class.
e) *PERMISSION* is the set of permissions in the relationship.
f) *ROLE* is the set of roles in the relationship. *A role is played by a method* and is composed of a set of permissions.
g) *METHOH_ROLE_MAPPING* is a set of functions, each of which maps a method to a role (which means that the method plays the role).
h) *COMPOSITE_ROLE* is the set of composite roles. A composite role is composed of roles, i.e., methods. Composite roles are used in role change.
i) *DECLASSIFICATION* is a set of special variables for declassification.

4. INTRA-APPLICATION INFORMATION FLOW CONTROL IN MRBAC/AR

MRBAC/AR controls both intra- and inter-application information flows. The former is described in this section, and the latter in section 5. Intra-application information flows in a system include direct flows and indirect flows. Indirect flows refer to accessing information via the third one. For example, after the method "md1" reads the information of "*var*1" into "*var*2," a method that read "*var*2" corresponds to indirectly reading "*var*1" via "*md*1". Both direct and indirect flows should be secure.

In an object-oriented system, direct flows include the flows *among methods* and those *within a method*. Information flows among methods are induced by statements that involve messages (method invocation). Other information flows are flows within a method. When "*obj*1.*md*1" invokes "*obj*2.*md*2," "*obj*1" and "*obj*2" should be within a session. Otherwise, the invocation is not allowed. Suppose "*obj*1" and "*obj*2" are within a session and "*obj*1.*md*1" passes the arguments "(*arg*1, *arg*2, …, *argn*)" to the parameters "(*par*1, *par*2, …, *parn*)" of "*obj*2.*md*2". Then, the access rights, and DSOURCE of an argument should be copied to the corresponding parameter. This copying is necessary because a parameter receiving an argument inherits the security level of the argument. After the copying, the invoked method is executed and every information flow within the method should be secure. To ensure secure information flows within a method, the following *secure information flow conditions* should be true when the value derived from the variables "*var*1," "*var*2," …, "*varn*" is assigned to the variable "*d_var*" (supposing the derivation appears in the method "*mdx*" that plays the role "$role_{mdx}$").

First secure information flow condition: $(\{\{var1, R\}, \{var2, R\}, \ldots, \{varn, R\}\} \subseteq role_{mdx})$.

Second secure information flow condition: $(\{d\_var, W\} \in role_{mdx}) \wedge (\{d\_var, W\} \in (\cap_{i,j}\, role_{dsource\_var(i,j)}))$.

The permission "{*var*1, *R*}" means "*var*1" is allowed to be read whereas "{*d_var*, *W*}" means "*d_var*" is allowed to be written. The notation "$role_{dsource\_var(i,j)}$" is the role

played by the j^{th} method in the DSOURCE of the i^{th} variable that derives "*d_var*". The notation "$\cap_{i,j}$ $role_{dsource\_var(i,j)}$" is the intersection of the roles' permissions, in which the roles are played by the methods in the DSOURCEs of the variables deriving "*d_var*".

The first secure information flow condition controls read access. It requires that the method "*mdx*" should be allowed to read the variables deriving "*d_var*" because "*mdx*" directly reads the variables. The second secure information flow condition controls write access. It requires that the method "*mdx*" as well as every method in the DSOURCEs of the variables deriving "*d_var*" must possess a permission to write "*d_var*". Requiring "*mdx*" and DSOURCEs to be allowed to write "*d_var*" is obvious because the write operation is performed by "*mdx*" and every data source that affects "*d_var*" should be regarded as a data source of "*d_var*".

The two secure information flow conditions ensure secure *direct* information flows. As mentioned above, the security of *indirect* information flows should also be ensured. Ensuring this security corresponds to avoiding Trojan horses. We use the join operation [2] (the symbol is "⊕") to avoid Trojan horses. If the value of the variable "*var*3" is derived from the variables "*var*1" and "*var*2," the access rights of "*var*3" will be changed by the join operation. To define the join operation, let: (a) "R_{var1}" and "R_{var2}" be respectively the read sets of "*var*1" and "*var*2" (a read set of a variable contains the methods that are allowed to read the variable), (b) "W_{var1}" and "W_{var2}" be respectively the write sets of "*var*1" and "*var*2" (a write set of a variable contains the methods that are allowed to write the variable), and (c) "$DSOURCE_{var1}$" and "$DSOURCE_{var2}$" be respectively the DSOURCEs of "*var*1" and "*var*2". When "*var*3" is derived from "*var*1" and "*var*2," then "R_{var3}," "W_{var3}," and "$DSOURCE_{var3}$," will be set by the result of "*var*1 ⊕ *var*2" as defined in Definition 3. Here "R_{var1}/R_{var2}" and "W_{var1}/W_{var2}" can be extracted from the permissions containing "*var*1/*var*2". After the join, the resulting "R_{var3}" and "W_{var3}" should be used to change the permissions containing "*var*3".

Definition 3 If "*var*3" is derived from "*var*1" and "*var*2" within the method "*mdx*," then "*var*1 ⊕ *var*2" will set "R_{var3}," "W_{var3}," and "$DSOURCE_{var3}$," as follows:

$$R_{var3} = R_{var1} \cap R_{var2}$$
$$W_{var3} = W_{var1} \cup W_{var2}$$
$$DSOURCE_{var3} = DSOURCE_{var1} \cup DSOURCE_{var2} \cup \{mdx\}$$

4.1 Features

The feature of controlling write access is achieved by the second secure information flow condition. The feature of detailing the granularity of access control to variables is inherent because permissions are composed of access rights associated with variables. The feature of declassification is achieved by the *DECLASSIFICATION* component in Definition 2. Other features are proved below.

Lemma 1 MRBAC/AR adapts to dynamic object state change.

Proof: MRBAC/AR provides the statements "addSesson" and "removeSession" to create and remove sessions. Therefore, MRBAC/AR allows dynamic object state change. To

prove that MRBAC/AR adapts to dynamic object state change, we have to prove that MRBAC/AR changes the secure and/or non-secure information flows of an application when object state changes (changing the flows corresponds to changing access rights within the system).

Suppose $os_{t1} = (OBJ_{t1}, SN_{t1})$ is the object state at time $t1$ and $os_{t2} = (OBJ_{t2}, SN_{t2})$ is the object state at time $t2$, in which OBJ_{t1}, OBJ_{t2}, SN_{t1}, SN_{t2} are respectively the object sets and session sets at $t1$ and $t2$. If $os_{t1} \neq os_{t2}$, the following three cases may happen: (a) $OBJ_{t1} \neq OBJ_{t2}$ but $SN_{t1} = SN_{t2}$, (b) $OBJ_{t1} = OBJ_{t2}$ but $SN_{t1} \neq SN_{t2}$, or (c) $OBJ_{t1} \neq OBJ_{t2}$ and $SN_{t1} \neq SN_{t2}$.

Case a. Without loss of generality, let $OBJ_{t2} = OBJ_{t1} \cup \{obj\}$, in which *obj* is an object and $obj \notin OBJ_{t1}$. In this case, $NSIFL_{t1} \neq NSIFL_{t2}$, in which $NSIFL_{t1}$ and $NSIFL_{t2}$ are respectively the sets of non-secure information flows at times $t1$ and $t2$. The rationale is that $NSIFL_{additional} \subseteq NSIFL_{t2}$ but $NSIFL_{additional} \not\subset NSIFL_{t1}$, in which

$$NSIFL_{additional} = \{ifl \mid ifl \text{ is an information flow between } obj \text{ and } obj1 \text{ in which } obj1 \in OBJ_{t1}\}$$

$NSIFL_{t2}$ contains $NSIFL_{additional}$ because no session exists between the object *obj* and the objects in OBJ_{t1}, which results in more non-secure information flows.

Case b. Without loss of generality, let $SN_{t2} = SN_{t1} \cup \{SN_i\}$, in which SN_i is a session and $SN_i \notin SN_{t1}$. In this case, $SIFL_{t1} \neq SIFL_{t2}$, in which $SIFL_{t1}$ and $SIFL_{t2}$ are respectively the sets of secure information flows at times $t1$ and $t2$. The rationale is that $SIFL_{additional} \subseteq SIFL_{t2}$ but $SIFL_{additional} \not\subset SIFL_{t1}$, in which

$$SIFL_{additional} = \{ifl \mid ifl \text{ is an information flow between } obj1 \text{ and } obj2, \text{ in which } obj1 \text{ and } obj2 \text{ coexist in } SN_i \wedge ifl \text{ fulfills both secure information flow conditions}\}$$

$SIFL_{t2}$ contains $SIFL_{additional}$ because an additional session SN_i exists in SN_{t2}, which results in more secure information flows.

Case c. Without loss of generality, let $OBJ_{t2} = OBJ_{t1} \cup \{obj\}$, in which *obj* is an object and $obj \notin OBJ_{t1}$. Moreover, let $SN_{t2} = SN_{t1} \cup \{SN_i\}$, in which SN_i is a session and $SN_i \notin SN_{t2}$. According to the proofs in the above cases, $SIFL_{t1} \neq SIFL_{t2}$ and $NSIFL_{t1} \neq NSIFL_{t2}$.

The proofs of Cases a through c state that different object states result in different secure information flows and/or non-secure information flows. Therefore, MRBAC/AR adapts to dynamic object state change. ❑

Lemma 2 MRBAC/AR adapts to dynamic role change.

Proof: MRBAC/AR provides the statement "setCompositeRole" to change the object role. Therefore, MRBAC/AR allows dynamic role change. To prove that MRBAC/AR

adapts to dynamic role change, we have to prove that MRBAC/AR changes a session's access rights when the composite roles played by one or more objects in the session change.

When a session SN_i is instantiated from a class relationship rel_i, each object in SN_i plays a composite role. A composite role is composed of roles played by methods. Moreover, every role (method) is composed of a set of permissions. If the object *obj* is within SN_i and the composite role played by *obj* is changed from *cr*1 to *cr*2, then the permissions within SN_i will be change from PM_{cr1} to PM_{cr2} as described below:

$$PM_{cr1} = \{pm_i \mid pm_i \text{ is a permission within } SN_i \wedge \text{methods of } obj \text{ do not appear in } pm_i\} \cup \{pm_i \mid pm_i \text{ is a permission within } SN_i \wedge \text{methods of } obj \text{ appear in } pm_i \wedge obj \text{ plays the composite role } cr1\}$$

$$PM_{cr2} = \{pm_i \mid pm_i \text{ is a permission within } SN_i \wedge \text{methods of } obj \text{ do not appear in } pm_i\} \cup \{pm_i \mid pm_i \text{ is a permission within } SN_i \wedge \text{methods of } obj \text{ appear in } pm_i \wedge obj \text{ plays the composite role } cr2\}$$

Generally, the two sets are different. Otherwise, the composite roles *cr*1 and *cr*2 can be regarded as the same. This proves that MRBAC/AR adapts to dynamic role change. ❑

Lemma 3 MRBAC/AR avoids Trojan horses.

Proof: A Trojan horse results when a method *md*2 leaks the information retrieved from *md*1 to *md*3 in which *md*2 is allowed to read the information of *md*1 but *md*3 is not. To prove that Trojan horses are avoided, let *var*1 be a variable in *md*1 that can be read by the methods in set R_{var1}. According to the above assumption, *md*2 is in set R_{var1} but *md*3 is not. Also let *var*2 be a variable in *md*2 whose value is derived from *var*1 and other variables. After the derivation, R_{var2} is modified by the join operation.

Suppose that a Trojan horse exists among *md*1, *md*2, and *md*3. Without loss of generality, assume that *md*3 can read *var*2. If this assumption is true, *md*3 is within R_{var2}. However, according to the join operation in Definition 3, R_{var2} is the intersection of R_{var1} and other sets of methods because *var*2 is derived from *var*1 and other variables. Since *md*3 is not in R_{var1}, *md*3 is not in R_{var2}. This contradicts the assumption. ❑

Lemma 4 MRAC/AR controls method invocation through argument sensitivity.

Proof: Suppose MRBAC/AR does not offer this feature. Then, if a method *obj*1.*md*1 can invoke *obj*2.*md*2, every variable of *obj*1 can be used as an argument in the invocation.

Without loss of generality, let *arg* be an argument in the invocation. Also let *arg* be involved in an assignment statement "*des = expression_of*(*arg*, *var*1, *var*2, …, *varn*)" within method *obj*2.*md*2. Suppose the set of methods that can read and write *des* are respectively R_{des} and W_{des}, those of *arg* are respectively R_{arg} and W_{arg}, and the DSOURCE of *arg* is $DSOURCE_{arg}$. The assignment statement is considered secure only when both

secure information flow conditions are fulfilled. Nevertheless, it is possible that the permissions or DSOURCE related to *arg* cause one or more of the secure information flow conditions to be false. For example, when $R_{des} \not\subset R_{arg}$ and $R_{des} \neq R_{arg}$, the first secure information flow condition is false. As another example, when $DSOUCE_{arg} \not\subset W_{des}$ and $DSOURCE_{arg} \neq W_{des}$, the second secure information flow condition is false. In either case, the assignment statement is non-secure, which means that passing *arg* as an argument in the invocation from *obj*1.*md*1 to *obj*2.*md*2 causes the invocation non-secure. This contradicts the assumption at the beginning of this proof. ❑

Lemma 5 MRBAC/AR allows purpose-oriented method invocation.

Proof: Suppose MRBAC/AR does not offer this feature. Then, if a method *obj*1.*md*1 can invoke *obj*2.*md*2, every method in *obj*1 can invoke every method in *obj*2.

Without loss of generality, suppose that there is an invocation from *obj*1.*md*3 to *obj*2.*md*4. If *obj*2.*md*4 requires arguments, disallowing the invocation from *obj*1.*md*3 to *obj*2.*md*4 can be achieved by properly assigning access rights to the arguments passed to *obj*2.*md*4 because method invocation can be controlled by argument sensitivity (see Lemma 4). Here assigning access rights to a variable can be achieved by assigning permissions to methods. If *obj*2.*md*4 does not require an argument but returns a value, disallowing the invocation from *obj*1.*md*3 to *obj*2.*md*4 can be achieved by properly assigning access rights to the variable returned by *obj*2.*md*4 and the variable receiving the return value. According to the above description, disallowing *obj*1.*md*3 to invoke *obj*2.*md*4 can be achieved through proper access right assignment. This contradicts the assumption made at the beginning of this proof. ❑

5. INTER-APPLICATION INFORMATION FLOW CONTROL IN MRBAC/AR

We first describe the inter-application security mechanism in MRBAC/AR and then prove its correctness.

5.1 Inter-Application Security Mechanism

In designing MRBAC/AR for inter-application information control, we assume that cooperating applications communicate with one another through remote procedure call (RPC) or JAVA remote method invocation (RMI). We also make the following assumptions:

a) The cooperating applications are well-known by programmers. With this assumption, programmers can correctly determine the security level of information in every application. Note that although an application does not know the details of other applications, programmers can know them. This facilitates designing the policy of controlling inter-application information flow by the programmers.
b) Objects and methods for communication among applications are known by the appli-

cations. This assumption is easy to achieved because RPCs or RMIs offered by an application can be identified by other applications after the RPCs or RMIs were registered.

c) Within an application, information received from other applications should not be declassified.

Inter-application flow control in MRBAC/AR is based on this rule: *when information is passed from one application to another, the security level of the information being passed should be the same as or lower than the security level of the variable receiving the information*. According to the rule, the following inter-application information flow security requirements (InterFlowReq) are obtained.

InterFlowReq 1 When a RPC or RMI occurs, the security level of an argument should not be higher than that of the parameter receiving the argument's value.

InterFlowReq 2 When returning from a RPC or RMI, the security level of a variable receiving the return value should not be lower than that of the return value.

To implement these requirements, we assign access rights to every parameter of RPC/RMI. Remember that assigning access rights to a variable can be achieved by assigning permissions to methods. Defining access rights using information across applications can be achieved because programmers know the details of every application (see assumption a) above). With the access rights assigned to parameters of RPC or RMI, the security mechanism for inter-application information flow control is described below:

a) When an application initiates an RPC or RMI, the two secure information flow conditions should be true when comparing the permissions and DSOURCE related to an argument with the conditions related to the parameter receiving the argument's value.
b) After receiving arguments, the invoked RPC/RMI operates within an application. The information flow control model embedded in the application should control the information flow within the application.
c) When the invoked RPC or RMI returns a value to the invoker, the two secure information flow conditions should be true when comparing the permissions and DSOURCE related to the return value with the conditions related to the variable receiving the return value.

5.2 Proof of Correctness

Without loss of generality, we use RMI and a two-application case in the proof. We call the applications *app*1 and *app*2 and assume that the remote method being invoked is *obj*2.*md*2.

When *app*1 invokes remote method *obj*2.*md*2 through RMI, the security mechanism checks the permissions and DSOURCE related to an argument against those related to the parameter receiving the argument. If the checking fulfills the two secure information flow conditions, the security level of the parameter is the same as or higher than that of the argument. This ensures that information passed to the parameters of *obj*2.*md*2 will

not be leaked by the parameters. This fulfills *InterFlowReq* 1 in the previous section. Moreover, the information flow control model embedded in *app*2 controls information flows within the application, and declassification of the argument's values is not allowed within *app*2 (see assumption *c* in the previous section). This ensures that *app*2 will not leak the arguments passed from *app*1 to *app*2.

When *obj*2.*md*2 returns a value to the invoker in *app*1, the security mechanism checks the permissions and DSOURCE related to the return value against those related to the variable receiving the return value. If the checking fulfills the two secure information flow conditions, the value returned by *obj*2.*md*2 is the same as or lower than that of the variable receiving the return value. This ensures that information returned by *obj*2.*md*2 will not be leaked by the variable receiving the return value. This fulfills *InterFlowReq* 2. Moreover, the information flow control model embedded in *app*1 controls information flows within the application, and declassification of the return value is not allowed within *app*1 (see assumption *c* in the previous section). This ensures that *app*1 will not leak the value returned by *obj*2.*md*2.

6. EVALUATION

We embedded MRBAC/AR in JAVA to produce the language MRBACL/AR. An application written in MRBACL/AR should first be processed by the MRBACL/AR preprocessor. The output of the preprocessor is a pure JAVA program containing the original JAVA program and an *access control monitor*. During program execution, the monitor controls information flows within the JAVA program. Primary functions of MRBACL/AR preprocessor are listed below:

a) Record MRBAC/AR information described in Definitions 1 and 2.
b) Add program code to check both intra- and inter-application information flows.
c) Add program code to perform the join, and change permissions affected by the join.
d) Add program code to adapt to session change (i.e., object state change) and handle access right changes according to session change.
e) Add program code to adapt to dynamic role change and handles access right changes according to role change.

We evaluated MRBAC/AR using the example in section 1 extended by order and employee management functions. We selected twenty-five students to program the example and then execute their programs. We collected the following metrics:

a) execution time of the programs without MRBAC/AR embedded,
b) execution time of the programs with MRBAC/AR embedded, and
c) number of statements (per 100 LOC) that violate the access control policy in the programs embedded with MRBAC/AR (the statements are regarded as non-secure statements).

Results of the experiment result are shown in Fig. 1 in which the data of metrics *a* is normalized to one. The result shows that the average execution time of the programs with

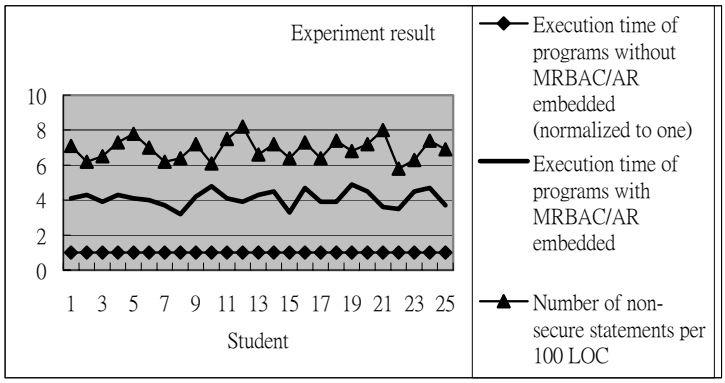

Fig. 1. Experiment result.

MRBAC/AR embedded is about four times those without MRBAC/AR embedded. This runtime overhead cannot be avoided because access control is mostly checked dynamically (because of the dynamic features). The figure also shows that about seven non-secure statements per 100 LOC were identified. Since non-secure statements can be identified, we believe that MRBAC/AR is useful.

7. CONCLUSIONS

This paper proposes a role-based information flow control model for object-oriented systems. It is a modification of RBAC96, which is named MRBAC/AR (modified RBAC for both intrA- and inteR-application information flow control). It uses secure information flow conditions to ensure information flows security. Features offered by MRBAC/AR is summarized as follows:

a) It uses sessions (instances of class relationships) and session change statements to adapt to dynamic object state change.
b) It uses composite role change statement to adapt to dynamic role change.
c) It avoids Trojan horse by using the join operation.
d) It details the control granularity to variables because access rights are associated with variables.
e) It controls method invocation through argument sensitivity through proper permission management.
f) It allows purpose-oriented method invocation.
g) It allows declassification using declassified variables.
h) It carefully controls write access. This prevents information corruption by untrusted data sources.
i) It controls inter-application information flows by controlling the security level of arguments, parameters, return values, and the variables receiving return values.

MRBAC/AR has been embedded in JAVA to produce the language MRBACL/AR.

We evaluated MRBAC/AR using examples programmed in MRBACL/AR. The evaluation shows that MRBAC/AR is useful in controlling information flows for object- oriented systems.

REFERENCES

1. A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, Vol. 21, 2003, pp. 5-19.
2. A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering Methodology*, Vol. 9, 2000, pp. 410-442
3. D. E. Bell and L. J. LaPadula, "Secure computer systems: unified exposition and multics interpretation," Technical Report, No. ESD-TR-75-306, Mitre Corporation, 1976, http://csrc.nist.gov/publications/history/bell76.pdf.
4. P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia, "Information flow control in object-oriented systems," *IEEE Transactions on Knowledge Data Engineering*, Vol. 9, 1997, pp. 524-538.
5. E. Bertino, S. de C. di Vimercati, E. Ferrari, and P. Samarati, "Exception-based information flow control in object-oriented systems," *ACM Transactions on Information System Security*, Vol. 1, 1998, pp. 26-65.
6. K. Izaki, K. Tanaka, and M. Takizawa, "Information flow control in role-based model for distributed objects," in *Proceedings of 8th International Conference on Parallel and Distributed Systems*, 2001, pp. 363-370.
7. M. Yasuda, T. Tachikawa, and M. Takizawa, "A purpose-oriented access control model," in *Proceedings of 12th International Conference on Information Networking*, 1998, pp. 168-173.
8. V. Varadharajan and S. Black, "A multilevel security model for a distributed object-oriented system," in *Proceedings of 6th IEEE Symposium on Security and Privacy*, 1990, pp. 68-78.
9. S. C. Chou, "Embedding role-based access control model in object-oriented systems to protect privacy," *Journal of Systems and Software*, Vol. 71, 2004, pp. 143-161.
10. S. C. Chou, "L^nRBAC: a multiple-leveled role-based access control model for protecting privacy in object-oriented systems," *Journal of Object Technology*, Vol. 3, 2004, pp. 91-120.
11. R. Sandhu, "Role hierarchies and constraints for lattice-based access controls," in *Proceedings of 4th European Symposium on Research in Computer Security*, 1996, pp. 65-79.
12. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, Vol. 29, 1996, pp. 38-47.
13. G. Booch, *Object-Oriented Analysis and Design with Application*, 2nd ed., The Benjamin/Cummings Publishing Company, 1994.

Shih-Chien Chou (周世杰) received a Ph.D. degree from the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. He is currently an Associate 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.