

An Information Flow Control Model for both Object-Oriented and Non-Object-Oriented Systems

SHIH-CHIEN CHOU AND CHIA-WEI LAI

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

Preventing information leakage during system execution is essential for a system that manages sensitive information. The prevention can be achieved through *information flow control*. Many information flow control models have been developed, in which most are for object-oriented systems. In our opinion, the procedural C language is still in used heavily. Therefore, an information flow control model for procedural languages is helpful. We developed a model that can be used in both object-oriented and non-object-oriented systems. This paper proposes the model, which offers the following features: (1) controlling both read and write access, (2) preventing indirect information leakage, (3) detailing the granularity of access control to variables, (4) controlling module call through argument sensitivity, (5) allowing information declassification, (6) controlling information flows among cooperating systems, (7) adapting to dynamic object state change, (8) allowing purpose-oriented method invocation, and (9) avoiding improper function call for non-OO systems.

Keywords: information flow, information flow control, access control list, security, prevent information leakage, object relationship

1. INTRODUCTION

Users play roles in an executing system. A user may obtain information that can be accessed by the roles he/she plays. For example, suppose John plays the role *president* in a company's information system and a president can access all information managed by the system. Then, John can read any information by outputting the information. In this regard, preventing information leakage during the execution of a system is essential. *Information leakage* refers to leaking high security level information to low security level users. For example, if a patient in a hospital obtains another patient's case history that can only be read by the patient's doctors, information leakage occurs. To prevent information leakage, *information flow control models* can be used. Generally, an information flow occurs when information is assigned to a variable. Information flow control prevents high security level information from being assigned to low security level variables.

Many information flow control models have been developed, in which most are for object-oriented (OO) languages. In our opinion, the procedural C language is still in used heavily. Therefore, an information flow control model for the C language is helpful. In the past years, we developed models for OO languages [1, 2] and developed a model for the C language [3]. We think that the models for OO languages and that for the C lan-

Received February 23, 2005; revised November 15, 2005 & November 2, 2006; accepted November 23, 2006.
Communicated by Chu-Sing Yang and H. Y. Mark Liao.

guage can be combined, which allows the same model to be used in both OO and non-OO systems. We extracted the concepts of our models for OO languages and those for the C language to develop a new model that can be used in both OO and non-OO systems. It is named ICMR/ACL (information flow control model based on module relationships and access control lists). Since the components of an executing OO program (which are objects) and those for a C program (which are C functions) are different, we name an executing system's components as *modules* in this paper. Nevertheless, if a feature is available for only objects or C functions, we will use the term *objects* or *functions* instead of *modules*.

Primary feature offered by ICMR/ACL is *controlling both read and write access*. The necessity of read access control is obvious. As to write access control, most models merely obey the "no write down" rule [4]. In our opinion, write access should be controlled precisely to prevent information corruption. We propose that only the data sources trusted by a variable can write the variable. In addition to controlling both read and write access, ICMR/ACL offers other features listed below.

- (1) Adapting to dynamic object state change. Since this feature is only available for OO systems, we take OO aspect to describe it. An object state is *a snapshot of objects and object relationships at a time point*. We use an example to describe the need for the adaptation. Suppose a doctor can read the personal information of a patient assigned to him but cannot read that information of a patient not assigned to him. Assume initially the patient *p1* is assigned to the doctor *d1*. In this object state, *d1* is allowed to read the personal information of *p1*. Suppose after a period of time, *p1* is re-assigned to *d2*. In this object state, *d1* is no longer allowed to read the personal information of *p1*. The example reveals that *changing object state results in changing access rights*, which in turn changes secure information flows.
- (2) Preventing indirect information leakage. For example, if the module *m1* is not allowed to read the information of *m2*, ICMR/ACL prevents the information of *m2* from being leaked to *m1* via other module(s).
- (3) Detailing the granularity of access control to variables. A system's information is generally stored in the system's variables and different variables may be of different sensitivity [5]. Therefore, variables should be protected independently (*i.e.*, the granularity of control should detail to variables).
- (4) Controlling module invocation through argument sensitivity. For example, suppose a doctor can change a patient's case history using the attribute *doctor.new_case_history* as an argument. Then, using other attributes in the invocation should be denied because different variables carry different information for different purposes.
- (5) Allowing declassification of information. This feature refers to downgrading the security levels of information [6, 7]. It is useful when users in low security levels must be allowed to access information in high security levels.
- (6) Allowing purpose-oriented method invocation. This feature was proposed by the research in [8-10]. With the feature, invoking a method may be allowed for some methods but disallowed for others. The consideration is correct because different methods may be of different sensitivity [5] and therefore should be controlled independently. Since the concept of methods does not exist in non-OO systems, this feature is only available for OO systems.

- (7) Controlling information flows *among* systems. Controlling information flows among systems is necessary because systems may communicate. In the following discussion, we use *intra-system information flows* to represent information flows *within* a system and *inter-system information flows* to represent information flows *among* systems.
- (8) Avoiding improper function call for non-OO systems. The feature is necessary because different functions may be in different security levels and therefore should be controlled independently. Since OO systems are not built by functions, this feature is only available for non-OO systems.

According to our survey, no model controls information flows in both OO and non-OO systems. Moreover, no model offers all the features listed above (see Table 1 in section 2). In the rest of the paper, section 2 discusses related work. Section 3 presents the proposed model ICMR/ACL. Section 4 proves that ICMR/ACL offers the features mentioned above. Section 5 depicts the use of ICMR/ACL. Section 6 presents the implementation of the model. Finally, section 7 is the conclusion.

2. RELATED WORK

Traditional access control is achieved by access control matrix (ACM) [11]. A subject can access an object if the required access right appears in the matrix. ACM allows only static access control. DACM (dynamic access control matrix) [12] is similar to ACM. Nevertheless, it allows dynamically granting access rights to subjects. A subject may be granted different access rights under different situations. This allows dynamic allocation of access rights, which is necessary to prevent indirect leakage.

MAC (mandatory access control) is useful in access control. An important MAC model is that according to Bell&LaPadula [4], which categorizes the security levels of objects and subjects. Access control in the model follows the “no read up” and “no write down” rules [4, 5]. Bell&LaPadula’s model was generalized into the lattice model [13-15]. In the lattice model proposed by Denning [13, 14], a lattice is constructed using security classes, “can flow” relationships, and the join operator. The “can flow” relationship controls information flows and the join operator prevents indirect information leakage.

The model in [16] is based on DAC (discretionary access control). It uses ACLs of objects to compute ACLs of executions and then obtains a secure information flow condition. A message filter filters out possibly non-secure information flows. Since the computation of an execution’s ACL takes information propagation into consideration, indirect information leakage is avoided. Flexibility is added to the model by allowing exceptions during or after method execution [17, 18]. More flexibility is added using versions [19].

The purpose-oriented model [8-10] controls information flows in OO systems. It proposes that invoking a method may be allowed for some methods but disallowed for others.

The decentralized label approach [6] marks the security levels of variables using labels. A label is composed of one or more policies, which should be simultaneously

obeyed. Join operation is used to prevent indirect information leakage. Declassification is allowed. Write access is controlled. The model in [7] also applied the label approach. Every file, device, pipe, and process in UNIX is attached with a label to prevent information leakage. Join operation is used to prevent indirect information leakage. Declassification is allowed.

RBAC (role-based access control) [20-28] were also used in information flow control [20, 21]. A RBAC model defines the roles a user can play. A role is a collection of permissions [29]. When a user plays a role in a session, he possesses the permissions of the role. A user can play multiple roles and can change role during a session. Changing role facilitates enforcing the need-to-know principle. Inheritance and other relationships such as "is_junior" [23] can be established among roles to structure them. The researches in [24, 25, 27, 28] proved that RBAC is a super set of DAC and MAC. Since DAC and MAC are useful in information flow control, RBAC should be more useful in that control. Nevertheless, since the original design of RBAC is not for information flow control, most features mentioned in section 1 are not offered. RBAC can be adapted to control information flows. For example, the model in [20] adapted RBAC to control information flows in OO systems. It classifies object methods and derives a flow graph from method invocations. Non-secure information flows can then be identified from the graph.

As a summary, Table 1 concludes our survey.

Table 1. Summary of the survey.

feature model	1	2	3	4	5	6	7	8	9
i	y	y	y	y	y	y	y	y	y
ii	y		y	y					
iii	y								
iv	y			y					
v	y		y			y	y		
vi	y					y			
vii	y								
viii	y			y					
ix									
x	y								
xi	y								

Features:

1. Prevent indirect information leakage
2. Adapt to dynamic object state change
3. Detail the control granularity to variables
4. Allow purpose-oriented method invocation
5. Control module invocation through argument sensitivity
6. Allow declassification of information
7. Control both read and write access
8. Control inter-application information flows
9. Avoid improper function call for non-OO systems

Models:

- i. ICMR/ACL
- ii. MAC
- iii. The model in [5]
- iv. The purpose-oriented model
- v. The decentralized label model
- vi. The model in [15]
- vii. General RBAC models
- viii. The RBAC model in [9]
- ix. ACM
- x. DACM
- xi. The model in [26]

3. ICMR/ACL

When developing an information flow control model, an essential problem is *defining roles* played by users. In OO systems, an object can be regarded as a role because allowing only one user to access an object is easy to achieve in OO systems. Allowing only one user to access an object is necessary because controlling the access rights of a

role corresponds to controlling the access rights of the user that plays the role. On the other hand, defining roles in a non-OO system is not easy because a function in a non-OO system is not as cohesive as an object. Generally, more than one type of users may be allowed to access a function and the users may be in different security levels. If a function is regarded as a role, users in different security levels can access information managed by the function, which may result in information leakage. For example, suppose the function *getInfo* gets a user's information, and both patients and doctors can access the function. Then, the following two cases of information leakage may happen (suppose a patient is allowed to retrieve his own information only). First, a patient can use the function to retrieve a doctor's information. Second, a patient can use the function to retrieve another patient's information. Although information may be leaked when regarding functions as roles, ICMR/ACL still regards functions as roles. Nevertheless, the following requirements should be fulfilled for a C program:

RleReq 1: Every function in a C program is allowed to access by only one type of users. This solves the problem resulted by the first case mentioned above.

RleReq 2: Constraints should be established for users to access information within a function. For example, accounts and passwords should be given to patients that will access information through the function *getInfo*. This solves the problem resulted by the second case mentioned above.

3.1 Definitions

The primary information flow control mechanism in ICMR/ACL is *module relationship* as defined below.

Definition 1 A module relationship exists among modules if information may *directly* flow among the modules. Each module relationship is associated with an information flow control policy for modules to obey. If multiple information flow control policies must be obeyed, more than one module relationship should be defined among the modules in which a relationship enforces a policy.

In an OO system, a module relationship corresponds to a class relationship, which can be instantiated to link objects. Objects linked by an object relationship coexist in an *object relationship group (ORG) according to the relationship*. Direct information flows among objects are allowed only when they coexist in an ORG. For example, if messages may directly pass among two instances *m1* and *w1* of the class *employee* who are respectively assigned the roles *manager* and *worker*, and *w1* is assigned to *m1* for monitoring purposes, then a reflexive class relationship *assigned* should be established for the class *employee*. On the other hand, a module relationship in a non-OO system corresponds to invocation relationships among functions. They are generally fixed (*i.e.*, they cannot be dynamically instantiated). Functions linked by a function relationship coexist in a *function relationship group (FRG)*. In the rest of the paper, we collectively call ORG and FRG as *MRG (module relationship group)*. Nevertheless, we will use the term ORG or FRG if a feature is only available for OO or non-OO systems.

Direct information flows among modules are allowed only when they coexist in a MRG. Nevertheless, information may *indirectly* flow among modules even when no MRG exists among them. For example, suppose that the modules $m1$ and $m2$ coexist in a MRG and $m2$ and $m3$ coexist in another one. Then, information may indirectly flow from $m1$ to $m3$ via $m2$. An information flow control model should also prevent indirect information leakage.

ICMR/ACL should be embedded in a system to control information flows when the system is being executed. Since we discuss both intra- and inter-system information flow control, we give definition for ICMR/ACL from the perspective of *a set of cooperating systems*. When cooperating systems are being executed, modules are established to exchange information and MRGs are established to control direct information flows. Since ICMR/ACL controls information flows when systems are being executed, we describe ICMR/ACL from the perspective of *executing systems*. A set of executing cooperating systems embedded with ICMR/ACL is defined below (we call the set of systems $e_systems$):

Definition 2 $e_systems = (MODULE, MODULEI, XINFO, MRG, FNI)$, in which

- (a) *MODULE* is a set of modules not for system communication. In an OO system, a module (object) is composed of attributes, private variables, and methods. Moreover, a method may return a value. In a non-OO system, a module (function) is composed of variables, code, and may return a value. To simplify the presentation of the model, we collectively call attributes, variables, and return values as *variables*. To ensure secure access of variables, every variable is associated with an access control list (*ACL*), a data source (*DSOURCE*), and declassification information (*DECLINF*). 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. In an OO system, an *RACL* and a *WACL* of a variable are composed of object methods that can read and write the variable, respectively. In a non-OO system, an *RACL* and a *WACL* of a variable are composed of functions that can read and write the variable, respectively.

The *DSOURCE* of a variable records the sources of the variable's data. For example, suppose the variable var is derived from the variable $var1$ and $var2$, and $var1$ and $var2$ are respectively written by the module mx and my . Then, the *DSOURCE* of var is the set “ $\{mx, my\}$ ”. *DSOURCE* ensures that the modules writing a variable are trusted by the variable. A *DECLINF* is composed of declassification information for read access (*DECLR*) and that for write access (*DECLW*). In an OO system, a *DECLR* and *DECLW* associated with a variable are respectively composed of object methods that can read and write the variable even the methods do not possess rights to read and write the variable. In a non-OO system, a *DECLR* and *DECLW* associated with a variable are respectively composed of functions that can read and write the variable even the functions do not possess rights to read and write the variable.

To manage ORGs and roles, ICMR/ACL offers statements as follows.

1. *addORG*, which instantiates an ORG from a class relationship. Note that we use the term ORG instead of MRG in the statement because FRGs are fixed. No instantiation will happen among function relationships in a non-OO system.

2. *removeORG*, which removes an existing *ORG*.
3. *setRole*, which assigns roles to modules.

- (b) *MODULEI* is a set of modules for system communication.
- (c) *XINFO* is a set of information passed among modules.
- (d) *MRGS* is a set of MRGs as described in Definition 1.
- (e) *FNI* is a set of legal invocations among functions. If “(*fn1*, *fn2*)” exists in *FNI*, *fn1* is allowed to invoke *fn2*. This component is only available for non-OO systems to control legal function call.

3.2 Intra-System Information Flow Control in ICMR/ACL

To ensure information flow security, both direct and indirect information flows should be secure. Direct information flows include *those among modules* and *those within modules*. Those among modules are induced by module invocation.

When the module *m1* invokes *m2*, *m1* and *m2* should coexist in a MRG (according to Definition 1). Otherwise, the invocation is non-secure. Suppose *m1* and *m2* coexist in a MRG and *m1* passes the argument list “(*arg_i* | *arg_i* is the *i*'th argument)” to the parameter list “(*par_i* | *par_i* is the *i*'th parameter)” of *m2*. Then, the ACL, DSOURCE, and DECLINF of the *i*'th argument are copied to those of the *i*'th parameter. The copying is necessary because a parameter receiving an argument inherits the security level of the argument. After the copying, the invoked module is executed. When a module *m1* executes a statement that assigns the data derived from variables in the set *SRC* (which is “{*var_i* | *var_i* is a variable and *i* is between 1 and *n*}”) to the variable *d_var*, rules should be obeyed for the information flow induced by the assignment to be secure. In defining the rules, we let the RACL, WACL, DSOURCE, DECLR, and DECLW of the variable *d_var* be respectively *RACL_{d_var}*, *WACL_{d_var}*, *DSOURCE_{d_var}*, *DECLR_{d_var}*, and *DECLW_{d_var}*. Those of *var_i* be respectively *RACL_{var_i}*, *WACL_{var_i}*, *DSOURCE_{var_i}*, *DECLR_{var_i}*, and *DECLW_{var_i}*. Next, we let *SRCR* be the set “*SRC* – *DECVARR*” and *SRCW* be the set “*SRC* – *DECLVARW*”, in which *DECVARR* consists of variables declassified for the modules in *DECLR_{d_var}* to read and *DECLVARW* consists of variables declassified for modules in *DECLW_{d_var}* to write. Suppose *SRCR* is composed of *k* element namely *varr₁* through *varr_k* and *SRCW* is composed of *m* elements namely *varw₁* through *varw_m*. Then, the module *m1* executing a statement that assigns the data derived from variables in the set *SRC* to the variable *d_var* is secure only when the following two secure flow rules are obeyed:

First secure flow rule $(RACL_{d_var} \cup \{m1\}) \subseteq \bigcap_{i=1}^k RACL_{var\ r_i}$.

Second secure flow rule $WACL_{d_var} \supseteq (\bigcup_{i=1}^m DSOURCE_{var\ w_i} \cup \{m1\})$.

The first secure flow rule controls read access. It requires the modules that can read the variable *d_var* can also read the variables in the set *SRCR*. This causes the variable *d_var* to be the same restricted as or more restricted than those of the variables in the set. The rule also requires that *m1* can read the variables in the set because *m1* executes the statement. The rule uses RACLs of the variables in the set *SRCR* instead of those in the

set SRC because variables may be declassified for the modules in $DECLR_{d_var}$ to read. The second secure flow rule controls write access. It requires the modules that can write the variables in the set $SRCW$ can also write the variable d_var . This means that the modules trusted by the variables in the set should also be trusted by d_var . The condition also requires that the module $m1$ be within $WACL_{d_var}$ because $m1$ performs the write operation. Similar to the first secure flow rule, WACLs of the variables in the set $SRCW$ instead of those in the set SRC are used because of write access declassification.

After the variable d_var receives the derived data, the access rights of d_var should be changed to prevent indirect information leakage. ICMR/ACL uses the join operation [6] for the prevention and uses the symbol “ \oplus ” for the operation. The join operation changes the ACL of d_var to “ $\oplus_{i=1}^n ACL_{var_i}$ ” as defined below:

Definition 3 $\oplus_{i=1}^n ACL_{var_i} = (\cap_{i=1}^n RACL_{var_i}, \cup_{i=1}^n WACL_{var_i})$.

The join operation trusts less or the same readers. Therefore, join will not lower security level. On the other hand, it 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, DSOURCE should be adjusted to “ $DSOURCE_{d_var} = \cup_{i=1}^n DSOURCE_{var_i} \cup \{m1\}$ ”. The union of $DSOURCE_{var_i}$ is obvious because all data sources deriving the computation result should be considered data sources of the result. The module $m1$ is also a data source because $m1$ writes the derived result to d_var .

3.3 Inter-System Information Flow Control in ICMR/ACL

Cooperating systems communicate through cross-system modules ($MODULEI$ in Definition 2). The communication can be achieved through remote procedure call (RPC) or JAVA remote method invocation (RMI) [30]. Since a system does not know the contents of other systems, the mechanism for intra-system information flow control cannot be used to control inter-system information flows. We use another mechanism for inter-system information flow control. In designing the mechanism, we made the following assumptions:

- (1) The systems are well-known by programmers because programmers must determine the security level of information in every system.
- (2) A system receiving other systems' information does not possess the right to declassify the security level of the information.
- (3) Every module is allowed to invoke RPCs/RMIs. That is, no MRG restriction (see Definition 1) is applied to inter-system information flow control.

Since cooperating systems communicate through parameter exchanging, information leakage can be prevented if the following rules are obeyed: (a) the arguments passed from a system to another system are not leaked in the latter system, and (b) the value returned from a system to another system is not leaked in the latter system. To obey the rules, ACLs and DSOURCES should be associated with RPC/RMI parameters, arguments, and method return values. With the ACLs and DSOURCES, the security mecha-

nism for inter- system information flow control is described below. Lemma 5 proves the correctness of the security mechanism.

- (1) When a system initiates an RPC/RMI, the two secure flow rules should be obeyed when comparing the ACL and DSOURCE of every argument with those of the parameter receiving the argument.
- (2) After receiving arguments, the invoked RPC/RMI operates within a system. The information flow control model embedded in the system should control information flows within the system.
- (3) When the invoked RPC/RMI returns a value to the invoker, the two secure flow rules should be obeyed when comparing the ACL and DSOURCE of the return value with those of the variable receiving the return value.

4. FEATURES

This section proves that ICMR/ACL offers the features mentioned in section 1. Some of the features are obvious. First, ICMR/ACL details the access control to variables because every variable is associated with an ACL, a DSOURCE, and a DECLINF. Second, declassification information (DECLINF) associated with variables achieves declassification. Third, the two secure flow rules control both read and write access. Below we prove that ICMR/ACL offers the features not mentioned above.

Lemma 1 ICMR/ACL adapts to dynamic object state change.

Proof: Since this feature is only available for OO systems, we take the aspect of OO systems for the proof. To prove that ICMR/ACL adapts to dynamic object state change, we must prove that ICMR/ACL changes secure information flows of a system when object state changes. As described in section 1, an object state is a snapshot of objects and object relationships at a time point. Therefore, an object state is composed of an object set and an ORG set. Suppose $os_{t1} = (OBJ_{t1}, ORG_{t1})$ is the object state at the time point $t1$ and $os_{t2} = (OBJ_{t2}, ORG_{t2})$ is the object state at the time point $t2$, in which $OBJ_{t1}, OBJ_{t2}, ORG_{t1}, ORG_{t2}$ are the object sets and ORG sets at the time points $t1$ and $t2$, respectively. Suppose $os_{t1} \neq os_{t2}$ and object state changes from os_{t1} to os_{t2} . Then, the following three cases may happen: (1) $OBJ_{t1} \neq OBJ_{t2}$ but $ORG_{t1} = ORG_{t2}$, (2) $OBJ_{t1} = OBJ_{t2}$ but $ORG_{t1} \neq ORG_{t2}$, and (3) $OBJ_{t1} \neq OBJ_{t2}$ and $ORG_{t1} \neq ORG_{t2}$.

Case 1: Without loss of generality, we let $OBJ_{t2} = OBJ_{t1} \cup \{obj\}$, in which obj is an object and $obj \notin OBJ_{t1}$. In this case, $SIFL_{t1} \neq SIFL_{t2}$, in which $SIFL_{t1}$ and $SIFL_{t2}$ are respectively the set of secure information flows at the time point $t1$ and that at $t2$. $SIFL_{t1} \neq SIFL_{t2}$ because $SIFL_{additional} \subseteq SIFL_{t2}$ but $SIFL_{additional} \not\subseteq SIFL_{t1}$, in which $SIFL_{additional} = \{ifl \mid ifl \text{ is an information flow induced by the communication between } obj \text{ and the objects in } OBJ_{t1}\}$. $SIFL_{t2}$ contains $SIFL_{additional}$ because the object obj can communicate with more or less objects in OBJ_{t1} . Otherwise, obj is considered redundant.

Case 2: Without loss of generality, we let $ORG_{t2} = ORG_{t1} \cup \{org\}$, in which org is an

ORG and $org \notin ORG_{t1}$. In this case, $SIFL_{t1} \neq SIFL_{t2}$ because $SIFL_{additional} \subseteq SIFL_{t2}$ but $SIFL_{additional} \not\subseteq SIFL_{t1}$, in which $SIFL_{additional} = \{ifl \mid ifl \text{ is an information flow between } obj1 \text{ and } obj2, \text{ in which } obj1 \text{ and } obj2 \text{ coexist in } org\}$. $SIFL_{t2}$ contains $SIFL_{additional}$ because an ORG allows more or less information flows among the objects in the ORG.

Case 3: Without loss of generality, we let $OBJ_{t2} = OBJ_{t1} \cup \{obj\}$, in which $obj \notin OBJ_{t1}$ and let $ORG_{t2} = ORG_{t1} \cup \{org\}$, in which $org \notin ORG_{t2}$. According to the proofs in the above cases, $SIFL_{t1} \neq SIFL_{t2}$. \square

Lemma 2 ICMR/ACL prevents indirect information leakage.

Proof: Indirect information leakage results when a module $m2$ leaks the information retrieved from $m1$ to $m3$ if $m2$ is allowed to read the information of $m1$ but $m3$ not. To prove that indirect information leakage is prevented, we let $var1$ be a variable in $m1$ with the RACL $RACL_{var1}$. Here, $m2$ is within $RACL_{var1}$ but $m3$ not. Moreover, we let $var2$ be a variable in $m2$ whose value is derived from $var1$. After the derivation, $var2$ gets the RACL $RACL_{var2}$. Suppose that indirect information leakage exists among $m1$, $m2$, and $m3$. Without loss of generality, we assume that $m3$ can read $var2$. If this assumption is true, $m3$ is within $RACL_{var2}$. However, according to the join operation in Definition 3, $RACL_{var2}$ is the intersection of $RACL_{var1}$ and other RACLs because $var2$ is derived from $var1$. Since $m3$ is not in $RACL_{var1}$, $m3$ is not in $RACL_{var2}$. \square

Lemma 3 ICMR/ACL controls module invocation through argument sensitivity.

Proof: Suppose ICMR/ACL does not offer this feature. Then, if a module $m1$ can invoke $m2$, every variable of $m1$ can be an argument in the invocation. Without loss of generality, we let arg be an argument passed to the parameter par of the invoked module $m2$. Suppose par is within the set SRC in a statement that sets the data derived from the variables in SRC to the variable des . Moreover, suppose the ACL of des is “ $(RACL_{des}, WACL_{des})$ ”, the ACL of par is “ $(RACL_{par}, WACL_{par})$ ”, and the DSOURCE of par is $DSOURCE_{par}$. The statement is considered secure only when both the two secure flow rules are obeyed. Nevertheless, it is possible that the ACL or DSOURCE of par cause one or more of the secure flow rules to be disobeyed. For example, when the condition “ $RACL_{des} \subseteq RACL_{par}$ ” is false, the first secure flow rule is not obeyed. As another example, when the condition “ $WACL_{des} \supseteq DSOURCE_{par}$ ” is false, the second secure flow rule is not obeyed. In either case, the statement mentioned above is non-secure, which means that passing arg as an argument in the invocation from $m1$ to $m2$ causes the invocation non-secure. \square

Lemma 4 ICMR/ACL allows purpose-oriented method invocation.

Proof: Since this feature is only available for OO systems, we take the aspect of OO systems for the proof. Suppose ICMR/ACL does not offer this feature. Then, if a method $obj1.md1$ can invoke $obj2.md2$, every method in $obj1$ can invoke every method in $obj2$. Without loss of generality, we suppose that there is an invocation from $obj1.md3$ to $obj2.md4$. If $obj2.md4$ requires arguments, disallowing the invocation from $obj1.md3$ to

obj2.md4 can be achieved by properly assigning ACLs to the arguments passed to *obj2.md4* because method invocation can be controlled by argument sensitivity (see Lemma 3). If *obj2.md4* does not require an argument but returns a value, disallowing the invocation can be achieved by properly assigning ACLs to the value returned by *obj2.md4* and the variable receiving the return value. Therefore, disallowing *obj1.md3* to invoke *obj2.md4* can be achieved through proper ACL assignments. \square

Lemma 5 ICMR/ACL control inter-system information flows.

Proof: Without loss of generality, we use a two-system case in the proof and call them *sys1* and *sys2*. When a module in *sys1* invokes a module in *sys2*, the security mechanism checks the ACL and DSOURCE of every argument against those of the parameter receiving the argument. If the checking obeys the two secure flow rules, 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 the module in *sys2* will not be leaked by the parameters. Moreover, the information flow control model embedded in *sys2* controls information flows within the system, and declassification of the argument's values is not allowed within *sys2* (see assumption *b* in section 3.3). This ensures that *sys2* will not leak the arguments passed from *sys1* to *sys2*.

When *sys2* returns a value to the invoker in *sys1*, the security mechanism checks the ACL and DSOURCE of the return value against those of the variable receiving the return value. If the checking fulfills the two secure flow rules, the security level of the value returned by *sys2* is the same as or lower than that of the variable receiving the return value. This ensures that information returned by *sys2* will not be leaked by the variable receiving the return value. Moreover, the information flow control model embedded in *sys1* controls information flows within the system, and declassification of the return value is not allowed within *sys1*. This ensures that *sys1* will not leak the value returned by *sys2*. \square

Lemma 6 ICMR/ACL avoids improper function call for non-OO systems.

Proof: Since this feature is only available for non-OO systems, we take the aspect of non-OO systems for the proof. An improper function call from the function *fn1* to *fn2* may be a consequence of one or more of the following conditions: (a) *fn1* is not allowed to invoke *fn2*, (b) *fn1* passes improper arguments to *fn2*, or (c) *fn1* uses improper variable to receive the return value of *fn2*. If condition *a* is true, the valid function call *FNI* defined in Definition 2 will block the function call. If condition *b* is true, one or more statements within *fn2* will be non-secure, which blocks the function call (see Lemma 3). If condition *c* is true, one or more of the secure flow rules will be false when *fn2* returns a value to *fn1*. This blocks the function call. \square

5. EXAMPLE

We use the manager/worker system and the report generation system to depict the use of ICMR/ACL. The example does not exhibit all the features of ICMR/ACL because

they have been proved. We want to offer a concrete sense and present inter-system information flow control of ICMR/ACL. The example systems are described below.

The manager/worker system manages employees' personal information and salaries. In the system, an employee is either a manager or a worker. A worker is assigned to a manager. A manager can read the personal information and salaries of the workers assigned to him, and can change the salaries of the workers. On the other hand, a manager can only read the salaries of the workers not assigned to him. The system offers remote methods for the report generation system to retrieve employee salaries through RMI. The report generation system retrieves employee salaries from the manager/worker management system to produce reports. The retrieval is through RMI.

The systems are respectively implemented in Appendices 1 and 2 using JAVA embedded with ICMR/ACL. Appendix 1 is the manager/worker management system. It consists of classes such as *employee* (lines 7 through 8), and consists of two reflexive class relationships *assigned* (lines 1 through 2) and *not_assigned* (lines 3 through 4) for the class *employee*. In either class relationship, an employee is assigned the role *manager* or *worker* (lines 1.2). An ACL in a class relationship is composed of a RACL followed by a WACL separated by a semicolon (see line 1.5.1 for an example). The system offers a remote method *manager_worker_service.get_employee_salary* for the report generation system to invoke through RMI, which is defined within the JAVA interface *manager_worker_interface* and the JAVA class *manager_worker_service*. In the method *main* of the class *example*, the object *MWOBJ* offering the remote method is defined and registered (lines 9 through 13.3.2). In the class *example* (line 13), ICMR/ACL statements create ORGs (line 13.3.6), assign roles (line 13.3.4), and even change roles (line 13.3.10). To control inter-system information flow, the class relationship *RemoteMethodInvocation1* is defined (lines 5 through 6).

Appendix 2 is the report generation system. It consists of only one class. A class relationship *RemoteMethodInvocation2* is defined for RMI (lines 1 through 2 in Appendix 2). The system retrieves employee salaries from the manager/worker management system through RMI (lines 3.3.1 through 3.3.4). The RMI is achieved by invoking the remote method offered by the object *MWOBJ* (see the method *report_generation_service.main* in lines 3 through 4).

6. IMPLEMENTATION

We embedded ICMR/ACL in JAVA to produce the language ICMR/ACL-JV and developed a prototype environment for the language to evaluate ICMR/ACL in OO systems. We used two cooperating systems to evaluate the prototype. They were a simplified software development system and a simplified IV&V system. When evaluating the prototype, we required students to: (1) injected statements that violated the intra-system secure flow rules and (2) injected non-secure RMIs (*i.e.*, statements that violate the inter-security mechanism mentioned near the end of section 3.3). We also required students to collect the following metrics data: (1) non-secure intra-system statements, (2) non-secure inter-application statements, and (3) runtime overhead (comparing with systems not embedded with ICMR/ACL). The experiment showed that every injected non-secure statements and non-secure RMIS was identified. This means that ICMR/ACL did control

both intra- and inter-system information flows. Moreover, the averaged runtime of a system embedded with ICMR/ACL is more than 3 times the runtime of the same system without ICMR/ACL embedded. The overhead is unavoidable because information flows are dynamically checked during runtime. Although some research proposed static checking [31, 32], we find that dynamic checking is necessary because of dynamic features such as dynamic adapting to object state change. Although the overhead is unavoidable, it is large. Reducing the overhead is thus an important future work for us.

In addition to embedding ICMR/ACL into JAVA, we also embedded it into C to produce the language ICMR/ACL-C and developed a prototype system for the language. In the prototype system, we did not implement the inter-system communication. Therefore, we only used the simplified software development system mentioned above to evaluate the identification of non-secure intra-system statements from the non-OO perspective of ICMR/ACL. The experiment result showed that the injected non-secure statements were all identified. Moreover, the averaged runtime overhead was about 2.9.

7. CONCLUSION

This paper proposes an information flow control model that prevents information leakage for both object-oriented (OO) and non-OO systems. It uses module relationship and access control list (ACL) for the control. It is called ICMR/ACL (informational flow control model based on module relationships and access control lists), which offers the features below:

- (1) Controlling both read and write access. ICMR/ACL use the first and second secure flow rules to control read and write access, respectively.
- (2) Preventing indirect information leakage. ICMR/ACL uses join operation to prevent indirect information leakage.
- (3) Detailing the granularity of access control to variables. ICMR/ACL achieves the granularity by attaching ACLs to variables.
- (4) Controlling module invocation through argument sensitivity. ICMR/ACL accomplishes the feature through proper ACL assignment.
- (5) Allowing declassification of information. ICMR/ACL associates declassification information with variables to achieve declassification.
- (6) Controlling inter-system information flows. ICMR/ACL controls inter-system information flows by limiting the security levels of parameters and return values.
- (7) Adapting to dynamic object state change. ICMR/ACL uses object relationships for the adaptation. It is only available for OO systems.
- (8) Allowing purpose-oriented method invocation. ICMR/ACL accomplishes purpose-oriented method invocation through proper ACL assignment. It is only available for OO systems.
- (9) Avoiding improper function call. ICMR/ACL uses the definitions of legal function call to avoid improper function call. It is only available for non-OO systems.

REFERENCES

1. 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.
2. S. C. Chou, "Providing flexible access control to an information flow control model," *Journal of Systems and Software*, Vol. 73, 2004, pp. 425-439.
3. S. C. Chou and C. Y. Chang, "An information flow control model for C applications based on access control lists," *Journal of Systems and Software*, Vol. 78, 2005, pp. 84-100.
4. D. E. Bell and L. J. LaPadula, "Secure computer systems: unified exposition and multics interpretation," Technique Report No. MTR-2997, MITRE Corporation, 1976, <http://csrc.nist.gov/publications/history/bell76.pdf>.
5. V. Varadharajan and S. Black, "A multilevel security model for a distributed object-oriented system," in *Proceedings of the 6th IEEE Symposium on Security and Privacy*, 1990, pp. 68-78.
6. A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering Methodology*, Vol. 9, 2000, pp. 410-442.
7. M. D. McIlroy and J. A. Reeds, "Multilevel security in the UNIX tradition," *Software - Practice and Experience*, Vol. 22, 1992, pp. 673-694.
8. M. Yasuda, T. Tachikawa, and M. Takizawa, "Information flow in a purpose-oriented access control model," in *Proceedings of the International Conference on Parallel and Distributed Systems*, 1997, pp. 244-249.
9. M. Yasuda, T. Tachikawa, and M. Takizawa, "A purpose-oriented access control model," in *Proceedings of the 12th International Conference on Information Networking*, 1998, pp. 168-173.
10. T. Tachikawa, M. Yasuda, and M. Takizawa, "A purposed-oriented access control model in object-based systems," *Transactions on Information Processing Society of Japan*, Vol. 38, 1997, pp. 2362-2369.
11. M. H. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, Vol. 19, 1979, pp. 461-471.
12. M. S. Olivier, R. P. van de Riet, and E. Gudes, "Specifying application-level security in workflow systems," in *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, 1998, pp. 346-351.
13. D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, Vol. 19, 1976, pp. 236-243.
14. D. E. Denning and P. J. Denning, "Certification of program for secure information flow," *Communications of the ACM*, Vol. 20, 1977, pp. 504-513.
15. D. F. C. Brewer and M. J. Nash, "The Chinese wall access control policy," in *Proceedings of the 5th IEEE Symposium on Security and Privacy*, 1989, pp. 206-214.
16. 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.
17. 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.
18. E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia, "Providing flexibility in informa-

- tion flow control for object-oriented systems,” in *Proceedings of the 13th IEEE Symposium on Security and Privacy*, 1997, pp. 130-140.
19. A. Maamir and A. Fellah, “Adding flexibility in information flow control for object-oriented systems using versions,” *International Journal of Software Engineering and Knowledge Engineering*, Vol. 13, 2003, pp. 313-326.
 20. K. Izaki, K. Tanaka, and M. Takizawa, “Information flow control in role-based model for distributed objects,” in *Proceedings of the 8th International Conference on Parallel and Distributed Systems*, 2001, pp. 363-370.
 21. Z. Tari and S. W. Chan, “A role-based access control for intranet security,” *IEEE Internet Computing*, Vol. 1, 1997, pp. 24-34.
 22. D. J. Thomsen, “Role-based application design and enforcement,” *Database Security IV: Status and Prospects*, 1991, pp. 151-168.
 23. M. Nyanchama and S. Osborn, “Access rights in role-based security systems,” *Database Security VIII: Status and Prospects*, 1994, pp. 37-56.
 24. M. Nyanchama and S. Osborn, “Modeling mandatory access control in role-based security systems,” *Database Security IX: Status and Prospects*, 1995, pp. 129-144.
 25. R. Sandhu, “Role hierarchies and constraints for lattice-based access controls,” in *Proceedings of the 4th European Symposium on Research in Computer Security*, 1996, pp. 65-79.
 26. R. Sandhu, V. Bhamidipati, and Q. Munawer, “The ARBAC97 model for role-based administration of roles,” *ACM Transactions on Information and System Security*, Vol. 2, 1999, pp. 105-135.
 27. S. Osborn, “Mandatory access control and role-based access control revisited,” in *Proceedings of the 2nd ACM Workshop on Role-Based Access Control*, 1997, pp. 31-40.
 28. S. Osborn, R. Sandhu, and Q. Munawer, “Configuring role-based access control to enforce mandatory and discretionary access control policies,” *ACM Transactions on Information and System Security*, Vol. 3, 2000, pp. 85-106.
 29. 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.
 30. P. Niemeyer and J. B. Knudsen, *Learning JAVA*, O’Reilly, 2000.
 31. R. Focardi and R. Gorrieri, “The compositional security checker: a tool for the verification of information flow security properties,” *IEEE Transactions on Software Engineering*, Vol. 23, 1997, pp. 550-571.
 32. F. B. Schueider, “Enforceable access control policy,” *ACM Transactions on Information System Security*, Vol. 3, 2000, pp. 30-50.

APPENDIX 1

The manager/worker management system in section 5.

- ```

1. classRelationship assigned {
1.1. classes {employee} /* only one class is associated, therefore, the class relationship is reflexive */
1.2. roles {employee: manager, employee: worker} /* an employee is assigned the role manager or worker */

```

```

1.3. cardinality {manager: 1, worker: *} /* a manager can have multiple worker assigned */
1.4. modality {manager: M, worker: M} /* both manager and worker are mandatory in an
ORG according to the class relationship “assigned” */
1.5. attributeACLs { // ACLs of attributes
1.5.1. worker.personal_info {worker.get_self_personal_info, manager.get_others_personal_info;
example.main};
1.5.2. // Other attribute ACLs are omitted to simplify the appendix
1.6. } /* end of attributeACLs */
1.7. methodRetACLs { // ACLs of methodReturn values
1.7.1. worker.get_self_personal_info {manager.get_others_personal_info; NONE};
1.7.2. worker.get_self_salary {manager.get_others_salary; NONE};
1.8. } // end of methodRetACLs
2. } /* end of classRelationship */
3. classRelationship not_assigned {
3.1. // Contents omitted to simplify the appendix
4. }
5. classRelationship RemoteMethodInvocation1 { /* this class relationship is for inter-applica-
tion information flow control. The class “report_generation_service” is defined in the report
generation system. This cause ACLs to be across application, which is necessary for in-
ter-application information flow control */
5.1. classes {employee, manager_worker_service, report_generation_service}
5.2. roles {employee: employee, manager_worker_service: manager_worker_service, report_
generation_service: report_generation_service}
5.3. attributeACLs { // ACLs of attributes
5.3.1. employee.salary {employee.get_self_salary, manager_worker_service.get_employee_salary,
report_generation_service.main; NONE};
5.4. } /* end of attributeACLs */
5.5. methodRetACLs { // ACLs of methodReturn values
5.5.1. manager_worker_service.get_employee_salary {report_generation_service.main; NONE};
5.6. } // end of methodRetACLs
6. } /* end of classRelationship */

/***** JAVA program below *****/
7. class employee {
7.1. public String personal_info;
7.2. public float salary;
7.3. public String worker_personal_info; /* for a manager to retrieve a worker’s personal
information */
7.4. public float worker_salary, new_salary; /* for a manager to manage a worker’s salary
*/
7.5. public String get_self_personal_info() { /* for a manager to invoke when retrieving the
information of a worker */
7.5.1. return personal_info;
7.6. }
7.7. public void get_others_personal_info(employee e1) { /* for a manager to get the information
of a worker */
7.7.1. worker_personal_info = e1.get_self_personal_info();
7.8. }
7.9. // Other methods are omitted to simplify the appendix
8. }

```

```

/* The interface “manager_worker_interface” and the class “manager_worker_service” are
for RMI */
9. public interface manager_worker_interface extends java.rmi.Remote {
9.1. public float get_employee_salary(String employee_name);
10. }
11. public class manager_worker_service extends UnicastRemoteObject implements man-
ager_worker_interface {
11.1. public employee e1;
11.2. public float employee_salary;
11.3. public float get_employee_salary(String employee_name){
11.3.1. // identify the employee object with the name “employee_name” and set the object to “e1”
11.3.2. employee_salary = e1.get_self_salary();
11.3.3. return employee_salary;
11.4. }
12. } // end of class
13. class example{
13.1. public employee e1, e2, e3, e4;
13.2. public manager_worker_service MWOBJ; // The object MWOBJ offers RMI
13.3. public void main(){
13.3.1. // instantiate the object MWOBJ and register the object for RMI service
13.3.2. // The registration is achieved by invoking the method “Naming.rebind”
13.3.3. // instantiate employees
/* Define the role of an object in an ORG */
13.3.4. setRole(classRelationship assigned, role manager, employee e2);
13.3.5. // . . .
/* Create ORGs and assign roles into them */
13.3.6. addORG(classRelationship assigned, employee e1, employee e2);
13.3.7. addORG(classRelationship not_assigned, employee e1, employee e4);
/* some secure and non secure statements are as follows */
13.3.8. e2.get_others_personal_info(e1); /* Secure: “e1” is a worker assigned to “e2” */
13.3.9. e4.get_others_personal_info(e1); /* Non-secure: “e1” is a worker not assigned to “e4” */
/* role change below */
13.3.10. setRole(classRelationship assigned, role manager, employee e1);
13.3.11. setRole(classRelationship not_assigned, role manager, employee e1);
/* The flowing statement, which is originally secure, becomes non-secure after role change. */
13.3.12. e2.get_others_personal_info(e1);
13.4. } // end of “main”
14. } // end of class “example”

```

## APPENDIX 2

The report generation system in section 5.

1. classRelationship RemoteMethodInvocation2 { /\* this class relationship is for inter-system information flow control. The class “manager\_worker\_service” and “employee” are defined in the manager/worker management system. This cause ACLs to be across systems, which is necessary for inter-system information flow control \*/
  - 1.1. classes {employee, manager\_worker\_service, report\_generation\_service}
  - 1.2. roles {employee: employee, manager\_worker\_service: manager\_worker\_service, report\_

```

generation_service: report_generation_service}
1.3. cardinality {employee: *, manager_worker_service: 1, report_generation_service:1}
1.4. modality {employee: M, manager_worker_service: M, report_generation_service:M}
1.5. attributeACLs { // ACLs of attributes
1.5.1. employee.salary {employee.get_self_salary, manager_worker_service.get_employee_salary,
report_generation_service.main; NONE};
1.6. } /* end of attributeACLs */
1.7. methodRetACLs { // ACLs of methodReturn values
1.7.1. manager_worker_service.get_employee_salary {report_generation_service.main; NONE};
1.8. } // end of methodRetACLs
2. } /* end of classRelationship */

/***** JAVA program below *****/
3. class report_generation_service {
3.1. public float employee_salary;
3.2. public String employee_name;
3.3. public void main() {
/* the following operations invoke remote methods */
3.3.1. // Find the object "MWOBJ" offering the RMI service and set the object to "rmiObj"
3.3.2. // The object can be found using the method "Naming.lookup"
3.3.3. employee_salary = rmiObj.get_self_salary(employee_name);
3.3.4. // retrieve employee salary using the above statement until the end of employee
3.3.5. // produce a report according to the salaries
3.4. }
4. }

```



**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 a professor in the Dept. of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. His research interests include software engineering, software reuse, and information flow control.



**Chia-Wei Lai (賴家偉)** received a Master degree from the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. He is currently an engineer in the SerComm Corporation, Taipei, Taiwan.