Information Flow Control in Object-Based Systems

Shih-Chien Chou
Department of Computer Science and Information Engineering
National Dong Hwa University
1, Section 2, Da Hsueh Road, Shoufeng, Hualien 974, Taiwan
E-Mail: scchou@mail.ndhu.edu.tw

ABSTRACT

This paper proposes a model for information flow control in object-based systems. The model details information flow control to the method level, with which attributes, arguments, and return values can be independently assigned security levels. This improves information flow control flexibility. Moreover, the model does not use classes to represent objects. Instead, it incorporates multiple labels to control information flows among objects, which may be dynamically instantiated during program execution. We have used an example to show the necessity of information flow control among objects. Another feature provided by the model is that it allows purpose-oriented method invocation and, in the same time, avoids information leakage within an object.

Keyword: Information flow, information flow control, purpose-oriented method invocation, security

1. INTRODUCTION

Information processed by a program may be sensitive. To prevent information leakage, information flow control within an application is necessary [1-22]. Generally, information flow control refers to defining the security levels of information, and preventing information flows from high security levels to low security levels [6]. The focus of information flow control is different from that of cryptography [23] in spite that security is their common objective. Cryptography aims at encoding/decoding information sent to/received from the network, which creates a secure communication environment and therefore prevents information from being leaked to *non-users* of an application. On the other hand, information flow control prevents information from being leaked to unauthorized users *within* an application. In fact, an information flow control model should be executed in an environment with secure communication.

The simplest model for information flow control is the discretionary access control (DAC) approach, such as attaching an access control list (ACL) to every file. Since DAC fails to avoid Trojan horses [10, 15], mandatory access control (MAC) models have been proposed [6-9]. MAC generally categorizes information into security classes that possess different security levels. Information flows in MAC obey the "no read up" and "no write down" rules [6]. With this, information cannot flow to security classes with lower security levels. Although MAC avoids Trojan horses, it was identified as too restricted [5]. Some researches thus tried to add flexibility to MAC [10, 15, 19], hoping to release restriction while keeping security. Other researches use a form other than MAC and DAC, such as using labels [1-5, 14], to control information flows.

Information flow control in object-oriented systems becomes important according to the maturity of object-oriented technique. Quite a few researches developed models for that control [10-13, 15, 19, 20-21]. Most of them treat an object as a whole (i.e., objects are used as basic units for information flow control). On the contrary, some researches proposed that the attributes, methods, and return values of an object may be of different sensitivity, and hence should be managed independently

[21]. According to this consideration, invoking a method may be considered secure using a message but considered non-secure using another message. This condition holds even when the invoker is the same method. For example, under the assumption that a person can withdraw money for housekeeping but not for drinking, a bank's method "withdraw" can be invoked by a person's methods using the attribute "housekeeping_money" as an argument but cannot be invoked using "drinking_money" as an argument. This example exhibits that whether a method can be invoked should be determined by the sensitivity of arguments. This is not taken into consideration by the models we surveyed.

To control information flows in object-based systems, Takizawa identified the needs for purpose-oriented method invocation [12-13, 20]. That is, invoking a method may be allowed by some methods but disallowed by others, in which the invokers belong to the same object. For example, a person is allowed to withdraw money for housekeeping but disallowed for drinking [12]. That is, the method "bank.withdraw" can be invoked by the method "person.housekeeping" but not "person.drinking", in which both the invokers belong to the object "person". In considering purpose orientation, preventing information leakage within an object is important. For example, it is necessary to prevent a person from drinking using the money withdrawn for housekeeping. This prevention seems not provided by the model proposed by Takizawa.

Another problem associated with information flow control models for object-oriented (or object-based) systems is that they use classes to represent objects. That is, they control information flows among classes but not among objects. As known, an object is dynamically instantiated from a class during program execution. In this regard, controlling information flow among objects is difficult because the objects that may be instantiated are unknown. Controlling information flows among objects, however, is necessary. We use an example to show this. Suppose a man and a woman may be married or they may be just friends. If they are married, they can access each other's personal information. On the other hand, friends can only access

one another's general information. In this example, if a man class is used to represent all man instances (i.e., objects) and a woman class for all woman instances, information flow control between two friends and that between husband and wife cannot be differentiated. We thus believe that using classes to represent objects is insufficient in controlling information flows of object-oriented systems.

According to the description above, the following features are identified as essential to control information flows in object-oriented systems.

1. Avoid Trojan horses, which is the basic requirement.
2. Do not treat objects as a whole. Attributes, arguments, and return values of methods should be allowed to assign different security levels. With this, controlling method invocation according to the sensitivity of arguments can be achieved.
3. Allow purpose-oriented method invocation and, at the same time, prevent information leakage within an object.
4. Control information flows among objects instead of using classes to represent objects.

We designed a model that offers the features. This paper describes the model.

2. RELATED WORK

The simplest approach for information flow control is DAC. While DAC does control information access, it fails to avoid Trojan horses. Multilevel security models were thus proposed to avoid Trojan horses. For example, the research according to Bell&LaPadula [6] classifies the security levels of objects and subjects, in which subjects access objects. Information flows in the model follow the "no read up" and "no write down" rules. Bell&LaPadula's model has been generalized into the lattice model [7-9] (see [17] for a survey of lattice models). Both the Bell&LaPadula's model and the lattice model are classified as the mandatory access control (MAC) approach. In the typical lattice model proposed by Denning [8-9], a lattice (SC, $\rightarrow$, $\oplus$) is

constructed using “SC”, which is the set of security class, “→”, which is the “can flow” relationship, and “⊕”, which is the join operator. Information can flow by following the “can flow” relationship. The join operator avoids Trojan horses. MAC, however, has been criticized as too restricted [5]. Moreover, a lattice approach is difficult to apply to a system that possesses much incomparable information. For example, in an object-oriented system, attributes and methods of different objects are difficult to compare. This causes difficulty in constructing a lattice.

To loosen the restriction of MAC, quite a few models have been proposed [10, 15, 19]. Some models even use a form other than DAC and MAC, such as using labels [1-5, 14], to control information flows. We survey some researches as described below.

The model proposed in [15] controls information flows in object-oriented systems. It uses ACL of objects to compute the ACL of executions (which may consist of one or more methods). 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, no Trojan horses will result. Moreover, interactions among executions are categorized into modes including synchronized unrestricted, synchronous restricted, asynchronous, deferred reply unrestricted, and deferred reply restricted. Since different modes result in different control policies, the model loosens the restriction of MAC. More flexibility is added to the model by allowing exceptions during or after method execution [10]. These exceptions are added according to the observation that return values may be non-sensitive and therefore are allowed to flow to a module with lower security level.

The purpose-oriented model [12-13, 20] proposes that invoking a method may be allowed by some methods but disallowed by others, in which the invokers belong to the same object. For example, a person is allowed to withdraw money for housekeeping but disallowed for drinking. That is, the method “bank.withdraw” can be invoked by the method “person.housekeeping” but not “person.drinking”. Although both the latter two methods belong to the object “person”, not both of them

are allowed to invoke the former method. This consideration is correct, because the security levels of methods in an object may be different [21] and therefore can access information of different security level. The problem associated with the purpose-oriented approach in [12-13, 20] is that it does not provide solution to prevent information leakage within an object. For example, the approach cannot prevent a person from drinking using the money withdrawn for housekeeping.

The decentralized label approach [1-4] marks the security levels of variables using labels. A label is composed of one or more policies, which should be simultaneously obeyed. A policy in a label is composed of an owner and one or more readers that are allowed to read the data. Both owners and readers are principals, which may be users, group of users, and so on. Principals are grouped into hierarchies using the act-for relationships. When computation are applied to data, join operator is used to compute the label of the resulted data. This avoids Trojan horses. Currently, the model has been used to develop a programming language Jflow [2], which is based on JAVA.

The approach in [14] proposed a labeling system in the UNIX system. Every file, device, pipe, and process is attached with a label. Join operation is used to avoid Trojan horses. Moreover, the approach provides ceilings, which disallows processes to get into too sensitive locations. This avoids possible information leakage by the processes. This approach controls information flows among files, devices, and pipes. As to those among program variables, it did not control. Therefore, the approach is considered insufficient in information flow control within an application.

Role-based access control models [11, 16, 22] define the roles a subject can play. A role is a collection of permissions (i.e., access rights) [16]. When a subject plays a role, it possesses the rights belonging to the role. A subject can play multiple roles and even change role during a session [16]. Inheritance and other relationships can be established among roles [16, 22] to structure them. Moreover, constraints, such as two specific roles should be mutually exclusive, can be attached to roles. The advantage of role-based access control is that subjects can change roles dynamically, which

facilitates obeying the “need-to-know” principle. That is, a subject can switch to a role with minimum necessary rights. Note that this advantage is also offered by the labeling models (including ours). In fact, the join operation in the labeling models results in changing access rights, which can be regarded as changing roles. It seems that the role-based models operate well in an application that protects not too many resources, because roles should be defined for information flow control. In case that many resources should be protected (such as every variable in a program should be protected), defining roles becomes difficult and the access control may become imprecise.

Subjects playing a role may be a human being, a process, an object, or even a method [11]. If a subject corresponds to a method, information flow control can be detailed to the method level just as our model does. However, the access rights of a role are fixed. Accordingly, if a method can invoke another one, the invocation can occur independent of the sensitivity of the arguments passed. This contradicts our assumption that whether a method can be invoked is decided by argument sensitivity.

According to our survey, we identify that all the models, except DAC, offers the first feature described in the previous section (i.e., avoiding Trojan horses). As to the other features, none of the models we surveyed offer them.

3. EXAMPLE

This section describes an example used throughout the paper. The example uses the relationships among men and women to describe information flow control in an object-based system. Suppose a person possesses personal information and general information. Moreover, a man and a woman may be friends, husband and wife, or without relationship. If they are friends, they can read each other’s general information. If a man and a woman are married, a marriage certificate should exist. In this case, the man and the woman can read each other’s personal and general information, and change each other’s general information. Moreover, the couple can read, but not change, the information of their marriage certificate. In addition, the

marriage certificate of a couple cannot be accessed by persons other than the couple.

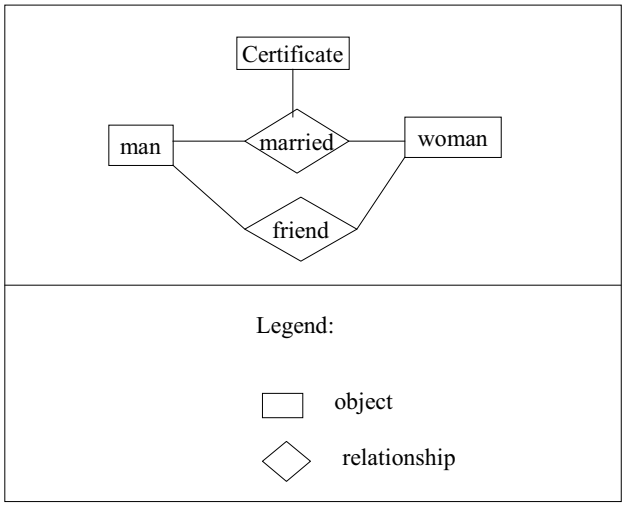

Figure 1. ERD for the example

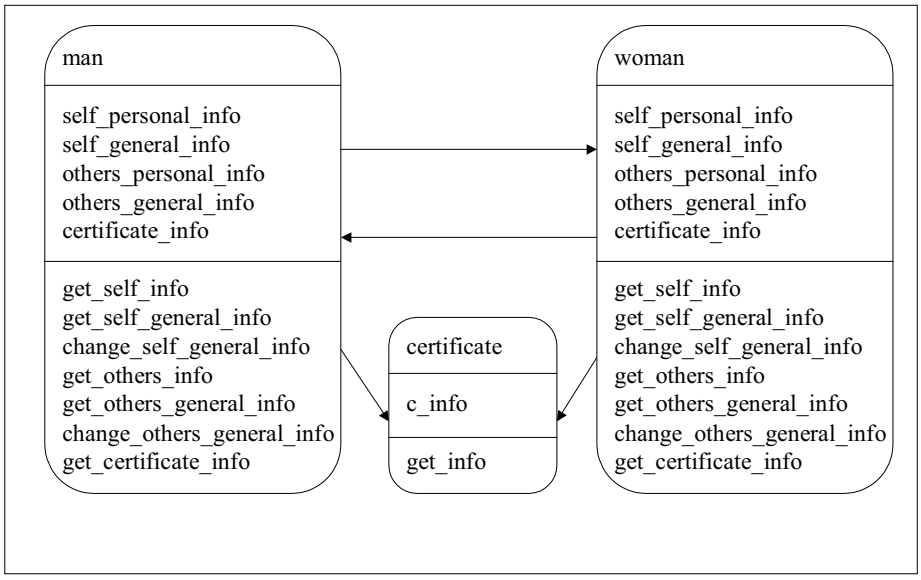

Figure 2. Object model for the example

We use the ERD (entity relationship diagram) in Figure 1 and the object model in Figure 2 to model the example. In Figure 2, a rounded-rectangle represents an object, which consists of three fields. The first field is the object's name, the second field contains the object's attributes, and the third contains the object's methods. Moreover, the arrows represent invocation relationships among objects. If an arrow

links two objects, method(s) of the object next to the arrow's tail may invoke method(s) of the object next to the arrow.

In this example, a man can read his wife's information (including personal and general information) through his method "get_others_info", which invokes her method "get_self_info" using his attributes "others_personal_info" and "others_general_info" as arguments. Since "self_personal_info" and "self_general_info" of a man are respectively used to store his personal and general information, they cannot be used as arguments to invoke his wife's method "get_self_info". This exhibits that a method can be invoked using a set of arguments but cannot be invoked using another set. A man can also change his wife's general information through his method "change_others_general_info", which invokes her method "change_self_info" using his attribute "others_general_info" as an argument. Note that a woman can access his husband's information in a similar way. Moreover, if a man and a woman were married, the man/woman can read the information of their marriage certificate by invoking the certificate's method "get_info" using his/her "certificate_info" as an argument.

If a man and a woman are just friends, they can read each other's general information through the method "get_others_general_info", which invokes the other's method "get_self_general_info" using "others_general_info" as an argument. Moreover, if a man and a woman are neither friends nor husband and wife, no information flow is allowed between them. According to our survey, no currently available model can model this example.

4. OBJECT ASSOCIATION

Before introducing our model, we first introduce object association, which is an important component in controlling information flows among objects.

As we have stated in section 1, using classes to represent objects is insufficient for information flow control. This insufficiency can also be identified in the example of section 3. Our model thus controls information flows among objects, instead of

using classes to represent objects. Nevertheless, since objects may be dynamically instantiated, controlling the flows among objects is difficult. Our model uses the concept of association [24] to solve this problem.

In an object-oriented system, relationships exist among objects. Figure 1 uses ERD to show objects and relationships. We call a relationship in the figure an *association*. Objects belonging to the same association can be grouped together. A group according to an association is called an *association group*. For example, if "man1" and "woman1" were married and "certificate1" records their marriage, then "man1", "woman1", and "certificate1" belong to an association group according to the association "married". As another example, if "man1" and "woman2" are friends, they belong to an association group according to the association "friend". After the grouping, information flow control policies can be defined, in which each group obeys a policy. In fact, each association obeys a policy. Using the example above, the group "{man1, woman1, certificate1}" obeys the policy related to husband and wife, such as that described in the example of section 3. On the other hand, the group "{man1, woman2}" obeys the policy related to friends. Furthermore, if a man and a woman do not belong to any group, no information flow can exist between them. This is an implicit policy.

Using association groups, information flow among objects can be controlled. To determine whether an information flow (from an object to another object) is secure, the association group of the objects should first be identified. Then, the policy associated with the group is used to check whether the flow is secure.

5. MODEL

Our model controls information flows within an object and those among objects. To accomplish this purpose, the model does not treat an object as a whole. Instead, it details information flow control to the method level. The model attaches labels to variables (such as object attributes and private variables within methods) for information flow control purposes. In the following text, we first give definitions and

then describe information flow control in the model.

5.1 Object-based system

An object-based system is composed of objects and messages, in which messages correspond to invocation relationships among methods. An object consists of attributes and methods. When a method invokes another method, it passes arguments to the method. Here an argument may pass value to or receive value from the method invoked. Sometimes, a method may return a value using commands like "return" in C++. Some formal definitions are given below.

Definition 1: An object-based system S is defined below:

S = (O, MS) where "O" is the set of objects and "MS" is the set of messages.

Definition 2: An object "o_i" is defined below:

o_i = (at, md), in which "at" is a set of attributes and "md" is a set of methods.

Definition 3: An attribute "at_i" is defined below:

at_i = (atn, att, atls), in which "atn", "att", and "atls" are respectively the name, type, and labels of "at_i". The definition of a label will be described later. An attribute may have multiple labels. Each label enforces the policy of an association described in section 4.

Definition 4: A method "md_i" is defined below:

md_i = (mdn, par, rtt), in which "mdn" is the name of the method, "par" is a set of parameters, and "rtt" is the type of the return value. Each parameter is specified by a name. Moreover, if the method does not return a value using the "return" statement, "rtt" can be omitted.

Definition 5: A message "ms_i" is defined below:

ms_i = (msn, arg), in which "msn" is the name of the message and "arg" is a set of arguments passed to the method invoked. An argument may be a variable or a literal (e.g., a number, a string, and a character). In an object-based system, the name of a message is actually the name of the

method invoked.

Note that Definitions 4 and 5 differentiate parameters used in a method from arguments passed to a method (see [25] to find the definitions of parameters and arguments).

5.2 Label

Labels are attached to variables for information flow control. A label is composed of an ASSOCIATION components and a RACL (read access control list) component. ASSOCIATION is the association whose security policy is enforced by the label. Moreover, RACL is used for read access control. Since read and write are dual operations (i.e., a read implies a write), RACL controls both read and write accesses.

Definition 6: A label of a variable "var1", namely "lbl_{var1}", is defined below:

lbl_{var1} = ($ASSOCIATION_{var1}$, $RACL_{var1}$), in which

$ASSOCIATION_{var1}$ is the association whose policy is enforced by "lbl_{var1}", and

$RACL_{var1}$ = {md | "md" is a method that is allowed to read "var1"}.

The reserved word "WORD" can be used in the RACL component. It represents the set containing all methods. Moreover, the reserved word "DEFAULT" should be used in the ASSOCIATION component if only one association exists among classes.

We use Example 1 to explain label, which declares the classes "man", "woman", and "certificate" described in section 3. In the example, a label is quoted by a pair of braces (i.e., "{" and "}"), and components in a label are separated by semicolon. The following string typed attribute declaration shows an example label:

String self_personal_info {married; man.get_self_info, woman.get_others_info}, {friend; man.get_self_info};

The attribute possesses two labels, which respectively enforce the policies of the

associations “married” and “friend”. The labels state that the attribute can be read by “man.get_self_info” and “woman.get_others_info” if a man and a woman are in a group according to the association “married”. On the other hand, the attribute can only be read by “man.get_self_info” if a man and a woman are in a group according to the association “friend”.

Example 1. The objects “man”, “woman”, and “certificate”

```
class man {
  attributes {
    String self_personal_info {married; man.get_self_info, woman.get_others_info},
        {friend; man.get_self_info};
    String self_general_info {married: man.get_self_info, man.get_self_general_info,
        man.change_self_general_info, woman.get_others_info,
        woman.get_others_general_info, woman.change_others_general_info},
        {friend: man.get_self_info, man.get_self_general_info,
        man.change_self_general_info, woman.get_others_info,
        woman.get_others_general_info};
    String others_personal_info {married; man.get_others_info, woman.get_self_info},
        {friend; man.get_others_info};
    String others_general_info {married: man.get_others_info,
        man.get_others_general_info, man.change_others_general_info,
        woman.get_self_info, woman.get_self_general_info,
        woman.change_self_general_info},
        {friend: man.get_others_info, man.get_others_general_info,
        man.change_others_general_info, woman.get_self_info,
        woman.get_self_general_info};
    String certificate_info {married; man.get_certificate_info, certificate.get_info},
        {friend; man.get_certificate_info };
  } /* end of attributes */

  methods {
    get_self_info(p_info, g_info) {
      p_info := self_personal_info;
      g_info := self_general_info;
    }
    get_self_general_info(g_info) {
      g_info := self_general_info;
    }
    change_self_general_info(g_info) {
      self_general_info := g_info;
    }
    get_others_info(woman_a) {
      woman_a.get_self_info(others_personal_info, others_general_info);
    }
    get_others_general_info(woman_a) {
```

```
        woman_a.get_self_info(others_general_info);
      }
      change_others_general_info(woman_a) {
        /* set a value to “others_general_info” */
        woman_a.change_self_info(others_general_info);
      }
      get_certificate_info(certificate_a){
        certificate_a.get_info(certificate_info);
      }
    } /* end of methods */
} /* end of class “man” */

class woman {
    attributes {
      String self_personal_info {married; woman.get_self_info, man.get_others_info},
            {friend; woman.get_self_info};
      String self_general_info {married: woman.get_self_info, woman.get_self_general_info,
            woman.change_self_general_info, man.get_others_info,
            man.get_others_general_info, man.change_others_general_info},
            {friend: woman.get_self_info, woman.get_self_general_info,
            woman.change_self_general_info, man.get_others_info,
            man.get_others_general_info};
      String others_personal_info {married; woman.get_others_info, man.get_self_info},
            {friend; woman.get_others_info};
      String others_general_info {married: woman.get_others_info,
            woman.get_others_general_info, woman.change_others_general_info,
            man.get_self_info, man.get_self_general_info,
            man.change_self_general_info},
            {friend: woman.get_others_info, woman.get_others_general_info,
            woman.change_others_general_info, man.get_self_info,
            man.get_self_general_info};
      String certificate_info {married; woman.get_certificate_info, certificate.get_info},
            {friend; woman.get_certificate_info };
    } /* end of attributes */

    methods {
      get_self_info(p_info, g_info) {
        p_info := self_personal_info;
        g_info := self_general_info;
      }
      get_self_general_info(g_info) {
        g_info := self_general_info;
      }
      change_self_general_info(g_info) {
        self_general_info := g_info;
      }
      get_others_info(man_a) {
        man_a.get_self_info(others_personal_info, others_general_info);
      }
```

```
      get_others_general_info(man_a) {
         man_a.get_self_info(others_general_info);
      }
      change_others_general_info(man_a) {
         /* set a value to “others_general_info” */
         man_a.change_self_info(others_general_info);
      }
      get_certificate_info(certificate_a){
         certificate_a.get_info(certificate_info);
      }
   } /* end of methods */
} /* end of class “woman” */

class certificate {
   attributes {
      String c_info {married; certificate.get_info, man.get_certificate_info,
           woman.get_certificate_info };
   }

   methods {
      get_info(cc_info){
         cc_info := c_info;
      }
   }
}
```

As we have described, labels are attached to variables. Then, what about literals? In practice, literals should also be labeled. Generally, a literal can be read by every one. Moreover, the association will not affect a literal's label. Therefore, a literal is implicitly labeled "{XCARE; WORLD}", in which the reserved word "XCARE" means "don't care".

5.3 Join

The join operator "⊔" records the derivation history of a variable's data. Therefore, it prevents Trojan horses. If the data of the variable "var3" is derived from the variables "var1" and "var2", then "var3" should be attached with a label derived from joining the label of "var1" and that of "var2". That is, assuming "lbl1", "lbl2", and "lbl3" are respectively the labels of "var1", "var2", and "var3", then "lbl3" is set "lbl1 ⊔ lb2" after the data of "var3" is derived.

Definition 7: The join of the labels "lbl1" and "lbl2", which are respectively "{ASSOCIATION$_{lbl1}$; RACL$_{lbl1}$}" and "{ASSOCIATION$_{lbl2}$; RACL$_{lbl2}$}", is defined below:

$$lbl1 \sqcup lbl2 = \{ASSOCIATION_{lbl3}; RACL_{lbl1} \cap RACL_{lbl2}\}$$

Since the join result is used to label "var3", the association of the result should be set the association of "var3". Moreover, the join operation will not lower down security level because the operation trusts less readers (or at most the same set of readers). In the following text, we prove that the join operation avoids Trojan horses.

Lemma 1: The join operation avoids Trojan horses.

Proof: A Trojan horse results when a method "md2" leaks the information retrieved from "md1" to "md3". Here we suppose that "md2" is allowed to read the information of "md1" whereas "md3" is not. To prove that Trojan horses are avoided, we let "var1" be a variable in "md1" with the label "{ASSOCIATION$_{var1}$; RACL$_{var1}$}". Note that "var1" can be read by "md2" but not "md3". We also let "var2" be a variable in "md2" and the value of "var2" is derived from "var1" and other variables. After the derivation, "var2" is labeled "{ASSOCIATION$_{var2}$; RACL$_{var2}$}".

Suppose that a Trojan horse exists among "md1", "md2", and "md3". Without loss of generality, we assume that "md3" can read "var2", which is derived from "var1" (this results in a Trojan horse). If this assumption is true, "md3" is contained by "RACL$_{var2}$". However, according to the join operation in Definition 7, "RACL$_{var2}$" is the intersection of "RACL$_{var1}$" and other RACLs because "var2" is derived from "var1" and other variables. Since "md3" is not in "RACL$_{var1}$" (i.e., "md3" is not allowed to read "var1"), "md3" is not in "RACL$_{var2}$". This contradicts the assumption. #

5.4 Secure information flow

In an object-based system, information may flow within an object or among objects. Both kinds of flows should be secure.

To make sure the information flows within an object are secure, the value derived from the variables "var1", "var2", . . ., "varn" can flow to the variable "derived_var" only when the following *secure flow condition* is true.

Secure flow condition: $(RACL_{derived\_var} \subseteq RACL_{var1}) \wedge (RACL_{derived\_var} \subseteq RACL_{var2}) \wedge ... \wedge (RACL_{derived\_var} \subseteq RACL_{varn}) \wedge (\{mdx, mdy\} \subseteq RACL_{var1}) \wedge (\{mdx, mdy\} \subseteq RACL_{var2}) \wedge ... \wedge (\{mdx, mdy\} \subseteq RACL_{varn})$

Here we suppose the label of the variable "derived_var" is "$\{ASSOCIATION_{derived\_var}; RACL_{derived\_var}\}$" and the label of the i^{th} variable to derive "derived_var" is "$\{ASSOCIATION_{vari}; RACL_{vari}\}$". Moreover, "mdx" is the method deriving "derived_var" and "mdx" is invoked by "mdy". The conditions "$(RACL_{derived\_var} \subseteq RACL_{var1})$", "$(RACL_{derived\_var} \subseteq RACL_{var2})$", and "$(RACL_{derived\_var} \subseteq RACL_{varn})$" require that "derived_var" must be more restricted then "var1", "var2", and "varn". The conditions "$\{mdx, mdy\} \subseteq RACL_{var1}$", "$\{mdx, mdy\} \subseteq RACL_{var2}$", "$\{mdx, mdy\} \subseteq RACL_{varn}$", and so on are also necessary because the variables to derive "derived_var" are directly read by "mdx" and indirectly read by "mdy".

After the value of the variable "derived_var" is derived, its label becomes "lbl1 $\sqcup$ lbl2 $\sqcup$ . . . $\sqcup$ lbln", in which "lbl1", "lbl2", and "lbln" are the labels of "var1", "var2", and "varn", respectively.

To make sure information securely flows among objects, when the message "(md2, arg1, arg2, . . ., argn)" is passed from the method "md1" to the method "md2", the following operations should be taken in sequence. Here we suppose that "md2" provides the parameters "par1, par2, . . ., parn".

1. Label “pari” as “{$ASSOCIATION_{argi}$; $RACL_{argi}$}”, in which “argi” is the i^{th} argument passed to “md2”, “pari” is the i^{th} parameter of “md2”, and the label of “argi” is “{$ASSOCIATION_{argi}$; $RACL_{argi}$}”. This operation copies labels of arguments to parameters. This copying is safe because information flows will be controlled within the method.
2. When the method is executed, the *secure flow condition* should be ensured.
3. If the invoked method uses a “return” statement to return a value, the label of the variable being returned should be checked against the variable that receives the return value. Here the *secure flow condition* should be true. Otherwise, the return information flow is considered non-secure.

We use Example 1 to explain the above operations. Suppose a man’s method “get_others_info” invokes his wife’s “get_self_info” using his “others_personal_info” and “others_general_info” as arguments. Then, the parameters “p_info” and “g_info” of the woman’s method “get_self_info” will respectively labeled “{married; man.get_others_info, woman.get_self_info}” and “{married: man.get_others_info, man.get_others_general_info, man.change_others_general_info, woman.get_self_info, woman.get_self_general_info, woman.change_self_general_info}”. The association “married” is used here because the man and the woman are husband and wife.

When the woman’s method “get_self_info” is executed, the label of her attribute “self_personal_info”, which is “{married; woman.get_self_info, man.get_others_info}”, is compared with the label of “p_info” described above. The comparison result fulfills the *secure flow condition*. Therefore, the statement “p_info := self_personal_info;” within the woman’s method “get_self_method” can be securely executed. According to the similar comparison, the statement “g_info := self_general_info;” can also be securely executed.

5.5 Purpose-oriented invocation without information leakage

To show that our model allows purpose-oriented method invocation and prevents

information leakage within an object, we use another example described below.

> "The information of a patient includes the personal information and case history. When a doctor heals a patient, the doctor is allowed to read both the patient's personal information and case history. On the other hand, when a doctor browses a patient's information for non-healing purposes, the doctor is not allowed to read the patient's personal information."

The example is modeled as Example 2. The ASSOCIATION component of the labels are set "DEFAULT" because there is only one association between doctors and patients.

Example 2. The classes "doctor" and "patient"

```
class patient {
   attributes {
      String personal_info {DEFAULT; patient.get_info, doctor.heal};
      String case_history {DEFAULT; patient.get_info, patient_get_case_history,
            doctor.heal, doctor.browse};
   } /* end of attributes */

   methods {
      get_info(p_info, c_history) {
         p_info := personal_info;
         c_history := case_history;
      }
      get_case_history(c_history) {
         c_history := case_history;
      }
   } /* end of methods */

} /* end of object "patient" */

class doctor {
   attributes {
      String patient_personal_info {DEFAULT; doctor.heal, patient.get_info};
      String patient_case_history {DEFAULT; doctor.heal, doctor.browse,
            patient.get_info, patient.get_case_history};
   }

   methods:
      heal(patient_a){
         . . .
         patient_a.get_info(patient_personal_info, patient_case_history);
         /* heal the patient */
         patient_a.change_case_history(patient_case_history);
         . . .
      }

      browse(patient_a){
```

```
        . . .
        patient_a.get_case_history(patient_case_history);
        /* browse the case history */
        . . .
      }
   } /* end of methods */
} /* end of object “doctor” */
```

In Example 2, the patient’s attribute “personal_info” is labeled “{DEFAULT; patient.get_info, doctor.heal}”, which allows the doctor’s method “heal” to invoke whereas disallows the doctor’s other methods, such as “browse” to invoke. This accomplishes purpose-oriented method invocation. In addition, the doctor’s attribute “patient_personal_info” is labeled “{DEFAULT; doctor.heal, patient.get_info}”, which disallows the doctor’s method “browse” to read the attribute and therefore prevents “browse” from reading a patient’s personal information retrieved by the method “heal”. This avoids information leakage within an object.

6. CASE STUDY

The example described in section 3 is revisited here. We use the proposed model to label the application. The labeling result is shown in Example 1. Note that we simplify the example by hiding the possibly complicated structure of an attribute. For example, the attributes “self_personal_info” and “self_general_info” of a man or woman may be as complicated as a C structure. Below we trace Example 1 to prove the requirements described in section 3 are fulfilled.

Requirement 1: If a man and a woman are friends but not husband and wife, they can only read each other’s general information.

When a man and a woman are just friends, they will belong to a group according to the association “friend”. The man can read the woman’s general information through his method “get_others_general_info”, which invokes her “get_self_general_info” using his “others_general_info” as an argument. Since the label of the man’s attribute “others_general_info” and that of the woman’s

attribute “self_general_info” fulfills the *secure flow condition*, the statement “g_info := self_general_info” in the woman’s method “get_self_general_info” can be securely executed. The woman can read the man’s general information in a similar way. *Note that labels with the association “friend” are used here.*

On the other hand, if the man tries to read the woman’s personal information by invoking her method “get_self_info” (through his method “get_others_info”) using his attributes “others_personal_info” and “others_general_info” as arguments, the invocation will be blocked. The rationale is that the label of the woman’s attribute “self_personal_info” disallows any access from a friend. Moreover, if the man tries to change the woman’s general information by invoking her method “change_self_general_info” (through his method “change_others_general_info”) using his attribute “others_general_info” as an argument, the invocation will also be blocked. The rationale is that the woman’s attribute “self_general_info” does not allow a friend’s “change_others_general_info” to access.

Requirement 2: If a man and a woman are married, they can read each other’s personal and general information, and change each other’s general information.

If a man and a woman are husband and wife, *the label used to check security is that with the association “married”*. With the labels in Example 1, all those required in this requirement can be done. Let’s use the “change each other’s general information” as an example to explain this.

When a man wants to change his wife’s general information, his method “change_others_general_info” invokes her method “change_self_general_info” using his attribute “others_general_info” as an argument. Since the label of his attribute “others_general_info” and that of his wife’s attribute “self_general_info” fulfill the *secure flow condition*, the statement “self_general_info := g_info;” in her method “change_self_general_info” can be securely executed.

Requirement 3: A men and his wife can read, but not change, the information of their

marriage certificate. And, the marriage certificate of a couple cannot be accessed by persons other than the couple.

If a man, a woman, and a certificate are in the same group according to the association "married", the man/woman can read the certificate's information through his/her "get_certificate_info", which invokes the certificate's method "get_info" using his/her attribute "certificate_info". Since the label of his/her attribute "certificate_info" and that of the certificate's attribute "c_info" fulfill the *secure flow condition*, the statement "cc_info := c_info;" within the certificate's method "get_info" can be securely executed. Note that *labels with the association "married" are used* in this case. As to changing the information of the certificate, the certificate does not provide an operation to do this.

If a man or a woman who is not in the same group with a certificate according to the association "married", reading the certificate's information will be prohibited. The rationale is that no label is provided by "certificate" for the association other than "married".

The above description discusses the information flows among objects within an association group. As to objects that are not coexisting in a group, no information flow among them is allowed. For example, if a man and a woman are strangers to each other, information flows between them are prohibited.

As we have proposed, our model offers four features mentioned near the end of section 1. The first feature, which is avoiding Trojan horses, is proved in section 5.3. The third one, which is allowing purpose-oriented method invocation and preventing information leakage within an object, is discussed in section 5.5. The fourth one, which is controlling information flows among objects, is discussed in this section. As to the second one, which is controlling method invocation according to the sensitivity of arguments, we use an example below to show it.

As described in Requirement 2 mentioned in this section, a man can change his wife's general information through his method "change_others_general_info", which

invokes his wife's method "change_self_general_info" using his attribute "others_general_info" as an argument. If we use the man's attribute "self_general_info" as an argument to invoke his wife's method "change_self_general_info", the invocation will be blocked. The rationale is that the label of the man's attribute "self_general_info", which is "{married: man.get_self_info, man.get_self_general_info, man.change_self_general_info, woman.get_others_info, woman.get_others_general_info, woman.change_others_general_info}", indicates that his wife's method "change_self_general_info" cannot read that attribute. This blocks the statement "self_general_info := g_info;" within the woman's method "change_self_general_info".

7. CONCLUSIONS

This paper proposes a model to control information flows in object-based systems. The model does not use classes to represent objects. Instead, it incorporates multiple labels to control information flows among objects. Examples have been used to prove the needs for that control. Moreover, information flow control in the model is detailed to the method level, with which attributes, arguments, and return values can be independently labeled. This improves information flow control flexibility. In addition, the model supports purpose-oriented method invocation and prevents information leakage within an object. The model offers the following features:

1. It details the information flow control level to methods. Detailing the control to the method level allows return values, arguments, and attributes to possess different security levels. This improves the flexibility of information flow control. For example, it is possible that a method can be invoked using a set of arguments, but cannot be invoked using another set of arguments. Treating an object as a whole cannot achieve this.
2. The model uses multiple labels to associate with a variable, in which each label

enforces a security policy. Here, a security policy is required by an object association. With multiple labels, when a method of an object invokes a method of another object, the association group is first identified. Then, labels that should be attached to arguments are identified according to the association group. The labels are then used to check security within the method invoked. Attaching multiple labels to a variable facilitates controlling information flows among objects. This feature is not offered by the models we surveyed.

3. The model allows purpose-oriented method invocation and, at the same time, avoids information leakage within an object.
4. The model avoids Trojan horses through label join. Although this feature is offered by every model except DAC, we still list it as a feature of our model because of its importance.

REFERENCES

1. A. C. Myers and B. Liskov, “A Decentralized Model for Information Flow Control”, *Proc. 17’th ACM Symp. Operating Systems Principles*, pp. 129-142, 1997.
2. A. C. Myers, “JFlow: Practical Mostly-Static Information Flow Control”, *Proc. 26’th ACM Symp. Principles of Programming Language*, 1999.
3. A. Myers and B. Liskov, “Complete, Safe Information Flow with Decentralized Labels”, *Proc. 14’th IEEE Symp. Security and Privacy*, pp. 186-197, 1998.
4. A. Myers and B. Liskov, “Protecting Privacy using the Decentralized Label Model”, *ACM Trans. Software Eng. Methodology*, vol. 9, no. 4, pp. 410-442, 2000.
5. C. J. McCollum, J. R. Messing, and L. Notargiacomo, “Beyond the Pale of MAC and DAC - Defining New Forms of Access Control”, *Proc. 6’th IEEE Symp. Security and Privacy*, pp. 190-200, 1990.
6. D. E. Bell and L. J. LaPadula, “Secure Computer Systems: Unified Exposition and Multics Interpretation”, *technique report, Mitre Corp.*, Mar. 1976. http://csrc.nist.gov/publications/history/bell76.pdf
7. D. F. C. Brewer, and M. J. Nash, “The Chinese Wall Security Policy”, *Proc. 5’th IEEE Symp. Security and Privacy*, pp. 206-214, 1989.
8. D. E. Denning, “A Lattice Model of Secure Information Flow”, *Comm. ACM*, vol. 19, no. 5, pp. 236-243, 1976.
9. D. E. Denning and P. J. Denning, “Certification of Program for Secure Information Flow”, *Comm. ACM*, vol. 20, no. 7, pp. 504-513, 1977.
10. E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia, “Providing Flexibility in Information flow control for Object-Oriented Systems”, *Proc. 13’th IEEE Symp. Security and Privacy*, pp. 130-140, 1997.
11. K. Izaki, K. Tanaka, and M. Takizawa, “Information Flow Control in Role-Based Model for Distributed Objects”, *Proc. 8’th International Conf. Parallel and Distributed Systems*, pp. 363-370, 2001.
12. M. Yasuda, T. Tachikawa, and M. Takizawa, “Information Flow in a Purpose-Oriented Access Control Model”, *Proc. 1997 International Conf. Parallel and Distributed Systems*, pp. 244-249, 1997.
13. M. Yasuda, T. Tachikawa, and M. Takizawa, “A Purpose-Oriented Access Control

Model", *Proc. 12'th International Conf. Information Networking*, pp. 168-173, 1998.

14. M. D. McIlroy and J. A. Reeds, "Multilevel Security in the UNIX Tradition", *Software - Practice and Experience*, vol. 22, no. 8, pp. 673-694, 1992.
15. P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia, "Information Flow Control in Object-Oriented Systems", *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, pp.524-538, Jul./Aug. 1997.
16. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, andC. E. Youman, "Role-Based Access Control Models", *IEEE Computer*, vol. 29, no. 2, pp. 38-47, 1996.
17. R. S. Sandhu, "Lattice-Based Access Control Models", *IEEE Computer*, vol. 26, no. 11, pp. 9-19, Nov. 1993.
18. R. Focardi and R. Gorrieri, "The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties", *IEEE Trans. Software Eng.*, vol. 23, no. 9, pp. 550-571, 1997.
19. S. Jajodia and B. Kogan, "Integrating an Object-Oriented Data Model with Multilevel Security", *Proc. 6'th IEEE Symp. Security and Privacy*, pp. 76-85, 1990.
20. T. Tachikawa, M. Yasuda, and M. Takizawa, "A Purposed-Oriented Access Control Model in Object-Based Systems", *Trans. Information Processing Society of Japan*, vol. 38, no. 11, pp. 2362-2369, 1997.
21. V. Varadharajan and S. Black, "A Multilevel Security Model for a Distributed Object-Oriented System", *Proc. 6'th IEEE Symp. Security and Privacy*, pp. 68-78, 1990.
22. Z. Tari and S.-W. Chan, "A Role-Based Access Control for Intranet Security", *IEEE Internet Computing*, vol. 1, no. 5, pp. 24-34, 1997.
23. W. Ford and M. S. Baum, *Secure Electronic Commerce, second edition*, Prantice-Hall, 2001.
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
25. H. M. Deitel and and P. J. Deitel, *C: How to Program*, Prentice-Hall, 2001.