

# Controlling Information Flow in Object Oriented Systems

Allaoua Maamir, Abdelaziz Fellah and Lina A. Salem

University of Sharjah, Department of Computer Science,  
P.O. Box 27272, Sharjah, United Arab Emirates  
{maamir, fellah, lina}@sharjah.ac.ae

**Abstract:** Information flow control is intended to ensure privacy and secrecy. Most of information flow control models are too restrictive. In this paper, we describe an access control model for object oriented systems. In the model access rights are applied to object attributes and methods which allow a greater flexibility. We present a filter based algorithm which intercepts every message exchanged among objects and enforces the defined access rights to ensure information privacy and secrecy.

**Keywords:** Information flow control, access control, access rights, security policy

## 1. Introduction

Applications requiring secrecy and confidentiality are growing in numbers. A few examples are electronic commerce, mobile computing, intranets and large network systems such as commercial multiuser database systems. Secrecy ensures that users access only information that they are allowed to see. Confidentiality ensures the protection of private information, such as payroll data, employee and customer records, as well as sensitive corporate data, such as internal memos and competitive strategy documents. There has been a general consensus that security is the key to the success of these applications. Therefore, we need effective mechanisms and policies to prevent accidental destruction and malicious attacks, and to control the disclosure and propagation of the information to users who should not access to the information. Information flow controls and access control models to design and implement secure systems have been widely researched. Various kinds of access control models have been studied in literature see for example [1], [5], [6], [15], [16], [18]. Information flow models are intended to address secrecy and privacy problems, however, most of them are too restrictive to be used. Decentralized label models [11]-[13], have been introduced to improve traditional models in several ways, making them more flexible by attaching flow policies to pieces of data. Another offensive method to loosen the strict information flow policies of the work in [1] has been proposed by the same authors in [18]. This method allows exceptions (i.e., waivers) which can be specified with reference to specific objects and users without disclosing sensitive information. The need for improving flexibility in information flow policies without compromising system security by disclosing sensitive information has been pointed out in previous and recent work, see for example [1], [11], [18], [21]. Recently, a promising new approach based on the use of programming language techniques for specifying and

enforcing information-flow control policies has been developed; see for example, [7]-[10].

An access rule is specified in the form  $\langle s, o, op \rangle$ , where a subject  $s$  is allowed to access or manipulate an object  $o$  through an operation  $op$ , such as read, write, or execute. The permission to perform a certain operation on an object is said to be an *access right*. Discretionary access control mechanisms restrict access to objects based solely on the identity of the subjects which are trying to access them. However, this basic principle of discretionary access control (DAC) contains a potentially illegal flow of information, that is, a subject which is granted access to an object can pass the information along to another subject. Data in an object may be obtained via other objects which then can be obtained by unauthorized subjects of the object. For example, if user  $u_1$  is allowed to read  $u_2$ 's data, but on the other hand  $u_2$  does not allow  $u_3$  to read it.  $U_2$  cannot control how  $u_1$  distributes the information it has read.  $U_1$  can read the information from  $u_2$  and then propagates it to  $u_3$  since a copy of the information is now owned by  $u_1$ . A *Trojan horse* is a computer program which works in a similar way, leaking information despite the discretionary access control. The main drawback of DAC mechanisms is that they do not impose any control on the flow of information. Thus, discretionary policies are vulnerable to Trojan horses, and DAC cannot deter hostile attempts to access sensitive information. A DAC mechanism allows users to grant or revoke access privileges to any of the objects under their control. Such users can be corporations or agencies which are the actual owner of system objects as well as the programs that process them.

Another access control model is the mandatory access control (MAC) which restricts access to objects based on the sensitivity of the information they contain and the authorization of subjects to access such information. Security labels (i.e., clearance) are associated with each object and subject that reflect the subject's trust level and ensure that sensitive information is not disclosed to subjects who are not cleared to see it. Every entity, i.e., subject and object, is classified to a security class. These mandatory policies are not particularly well suited to the requirements of organizations that process unclassified but sensitive information.

More recently, a family of reference models for role based access control (RBAC) has been proposed and investigated in research, see, for example, [14], [19], [20]. RBAC methods can be viewed as an alternative to traditional

discretionary DAC and MAC policies that are particularly attractive for commercial applications. The RBAC model has extended the framework access model to include role hierarchies. The translation of a mandatory access control model into a role hierarchy has been presented in [4]. In RBAC, access decisions are based on the roles and responsibilities of each user in the organization's structure. A *role* can be defined as a collection of access rights, which represent a set of job functions in the organization. Each user is assigned one or more roles, and each role is assigned one or more privileges. For example, within a hospital system, access rights and decisions are based on the roles that medical personnel can play in the organization. The potential role of a doctor can include prescribing medications, recommending treatments, and interpreting the results of an imaging test. The role of a nurse can include providing care for patients, measuring vital signs, and monitoring drug administration. The role of a medical assistant may include taking health histories, and performing laboratory tests. Roles can be hierarchical, mutually exclusive, collaborative, or overlapping. For example, in a hospital some roles are hierarchical. The doctor role may include all privileges granted to the nurse role, which in turn includes all privileges granted to the medical assistant role. Role hierarchies are a natural generalization of organizing roles for granting responsibilities and privileges within an organization. RBAC is used particularly for commercial applications, because it reduces the cost of security administration and the complexity of managing large networked systems. For example, RBAC has been implemented on the Web servers and particularly to an intranet computing environment in [20]. A variation of the RBAC model called object oriented role-based access model (ORBAC) has been proposed by [3]. In an ORBAC based system, object technology has been used to model application-level user access control. However, the confinement problem may occur in the ORBAC based system and objects can be accessed by unauthorized users. In order to deal with this problem, a role set assignment method based on the principles of MAC security policy has been proposed in [2].

Although the ability for discretion to specify accesses is not lost in the model of [1], the overall flexibility is reduced by the application of very tight and strict policy where the access rights are applied at the object level. A subject either has the right to read/write an entire object (read/write each attribute of the object) or none. Rights to execute methods of an object are not considered at all. The work in [24], [25] is based on RBAC. It considers flow control which requires program analysis. We believe program analysis requires rigorous and sophisticated tools. Furthermore, access rights are based on class relationships. Class relationships can help to assign access rights to subjects however assignment should be based on object instances. For example, a person  $p_1$  (from class  $p$ ) could have two friends (same kind of relationship),  $f_1$  and  $f_2$  (from the same class  $F$ ), but  $p_1$  may allow some information to pass to  $f_1$  but not to  $f_2$ . This is very common scenario in practice.

In this paper, we propose an approach based on access rights applied to object attributes and methods. Such

approach makes access control improve flexibility without increasing the potential for information leaks and disclosure. The remainder of the paper is organized as follows.

Section 2 describes the object-oriented model and introduces the basic terminology used throughout the paper. Section 3 describes the authorization model. Section 4 describes the information flow policies and the message filter is presented in Section 5. Section 6 states and proves the correctness of the filter algorithm. Some experiments results are presented in Section 7 and finally Section 8 concludes the paper.

## 2. Object-Oriented Model

Object-oriented systems are composed of objects. Objects can be defined as an encapsulation of data *state*, and *methods* for manipulating that data. *Classes* are prototypes of objects. An object is a physical implementation of a class, or an *instance*, of a class. A class is defined to be a set of *attributes* and *methods* and may have many instance objects. Objects interact and communicate by passing messages. A method of an object is invoked by sending a message to the object. Access to attributes of an object is also based on the message-passing paradigm. Messages are the means by which objects communicate, and for each message, a corresponding method is executed. If an object wants to access an attribute of another object, it sends a message requiring the execution of a method that reads that attribute and returns it to the sender. New classes can be created by reusing (*i.e.*, inheriting) attributes and methods from other existing classes. However, inheritance is beyond the scope of this paper. Further research considerations with respect to the propagation of access rights through inheritance hierarchies is being investigated.

We assume a finite set of domains  $D_1, D_2, D_n$ . Let  $D = D_1 \cup D_2 \cup \dots \cup D_n \cup \{nil\}$ , where *nil* is a special element. Every element of  $D$  is referred to as a primitive object (an integer, a string...). Let  $A$  be a set of attribute names,  $I$  a set of system object identifiers. Users of the system are considered as objects and each user has a unique user identifier.  $U$  the set of all user identifiers and  $O = I \cup U$ . Each element of  $O$  is referred to as an object identifier (*oid*).

**Definition 1:** A non primitive object,  $o$ , is a tuple  $\langle i, A, V, M \rangle$  where  $I \in I$ ,  $A$  a set of attribute names,  $V$  a set of attribute values where each value is in  $D \cup I$ , and  $M$  a set of method names.

In the above definition, a non primitive object has an oid, an ordered set of attributes, an ordered set of attribute values, and a set of methods.

**Definition 2:** A message sent from an object to another, that has a method called  $m$ , is a tuple  $\langle m, par_1, par_2, \dots, par_k \rangle$  where  $par_i \in D \cup I \cup A$ . The parameters are the arguments values to be passed to the method  $m$ .

Definition 2 states that a message is made up of the name of the method to be invoked and a set of parameters. Each parameter could be a value (a primitive object), an oid, or an attribute.

**Definition 3:** A reply to a message is either success, failure, NIL (an empty reply), or a tuple of return values  $\langle rp_1, \dots, rp_n \rangle$  where  $rp_i \in D \cup I$ ,  $I = 1, \dots, n$ .

Each object has a built-in read method and a built-in write method for each attribute. Similarly, each object has a built-in create method. Access to an attribute is carried through having the object to send a primitive message to itself. A primitive message causes the invocation of a built-in read/write. The same applies for the case of creating of an object instance. The built-in methods are said to be primitive because they do not cause invocation of other methods.

### 3. The Authorization Model

In order for a subject (a user or a system object) to access an attribute of an object or to create an object instance it must have the appropriate access right. Similarly, for an object to execute a method of an object it must have the permission to do so. We associate with each attribute,  $att$ , a read access list (RACL), and a write access list (WACL) containing the objects which have the right to read and write the attribute respectively. With each method,  $m$ , a permission list (PERL), contains the objects which can invoke the method, is associated. Each object,  $o$ , has a create access list (CACL) containing the objects which have the right to create instances of the object.

**RACL**( $att$ ) = {oid's of all object which can read  $att$ }

**WACL**( $att$ ) = {oid's of all objects which can write  $att$ }

**PERL**( $m$ ) = {oid's of all objects which have the right to invoke  $m$ }

**CACL**( $o$ ) = {oid's of all objects which have the right to create an instance of  $o$ }

By default each of the above lists contains the oid of the owner of the object. The owner of an object is the creator of the object. The above lists are determined for each object and assigned at creation time. We assume during execution those lists remain unchanged. Those lists should be determined from the business rules of the application. From now on we assume their existence.

When a user desires to start an activity, the user sends a message to an object executing a method of that object. The set of all method invocations (direct and indirect) to carry out the desired activity form what is called the user's transaction. In a transaction, if method  $m_1$  executing on object  $o_1$  invokes method  $m_2$  of  $o_2$  then the access authorizations of object  $o_1$  are checked. For example, if  $m_2$  is a read for attribute  $att$  then  $o_1$  must be in RACL( $att$ ) for the access to be granted otherwise the access is denied.

We refer to an execution of a method  $m_i$  of object  $o_i$  as  $e_i$ . In a transaction, if an execution  $e_i$  of method  $m_i$  executing on object  $o_i$  invokes the execution  $e_j$  of method  $m_j$  on  $o_j$  then the execution  $e_i$  is suspended until  $e_j$  returns. This is what is termed as synchronous interaction mode in [1]. This is the only interaction mode we assume in this paper, our model could be augmented with the other interaction modes defined in [1]. Having that set, the following defines the execution order of methods.

**Definition 4:** If an execution  $e_i$  invokes the execution  $e_j$  and the execution  $e_k$ , we say that the execution  $e_j$  precedes the execution  $e_k$  if  $e_j$  was invoked before  $e_k$ .

### 4. Information Flow

When an object,  $o_1$ , sends a message to object  $o_2$ , we say that the information flows from  $o_1$  to  $o_2$  and it is termed as forward flow. Similarly, when  $o_2$  replies to  $o_1$  we say that there is a flow of information from  $o_2$  to  $o_1$  and it is called backward flow. This corresponds to what is called forward and backward information transmission in [1]. Our assumption is identical to that in [8]. We make similar assumption to that of [1], during execution methods are not allowed to change their own code or the code of other methods. This is to ensure that no information can be hidden in method codes. Further, we assume that only the information written to object attributes is the only information that remains after the execution of a method is finished. For example static local variables like in C++ are not permitted. In the rest of the paper we would use information flow and information transmission interchangeably. In a transaction, if a transmission of information from object  $o_1$  to object  $o_2$ , and information is transmitted from  $o_2$ , to  $o_3$ , we say there is an indirect flow from  $o_1$ , to  $o_3$ . Note that it is not necessary for the information transmitted from  $o_2$  to  $o_3$  be the same as the information transmitted from  $o_1$  to  $o_2$ , it could be derived from it or not related to it at all. Hence, the information flows that we are considering are potential rather than actual. Limiting the work to only actual flows requires rigorous program code analysis [16], which is outside the scope of this paper. Certainly, considering only actual flows for control would have a great impact on overall system performance.

When the execution  $e_i$  of method  $m_i$  of object  $o_i$  invokes the execution  $e_j$  of method  $m_j$  of object  $o_j$ ,  $o_i$  sends a message  $\langle m_j, par_1, par_2, \dots, par_k \rangle$  to  $o_j$ . For the message to be allowed  $o_i$  must have permission to invoke the execution of  $m_j$  and  $o_j$  must have the right to access (to read) each of the parameters. Similarly, for the reply of the message from  $o_j$  to  $o_i$  to be allowed,  $o_i$  must have the right to access each of the reply parameters. Further, the message/reply should not enact an unsafe flow, this would be explained later. To characterize the access rights of a message parameter we define the read access list of a parameter as follows:

**Definition 5:** Let  $e_j$  be an execution of  $m_j$  of  $o_j$  invoked by an execution  $e_i$  of  $o_i$ . The message sent by  $o_i$  to  $o_j$  to invoke  $e_j$  is  $\langle m_j, par_1, par_2, \dots, par_k \rangle$ . The RACL of a parameter is defined as follows:

- (a) If  $par_i \in A$ , the parameter is an attribute  $att$ , then  $RACL(par_i) = RACL(att)$ .
- (b) If  $par_i \in I$ , the parameter is an oid, then  $RACL(par_i) = O$ .
- (c) If  $par_i \in D$ , the parameter is a primitive object (i.e., a computed value), then  $RACL(par_i) = VACL(e_i)$  where  $VACL(e_i)$  is the set of objects that can access a computed value by  $e_i$  as defined in Definition 6.

Cases (a) and (b) above are self explanatory. Case (c),  $par_i$  is a computed value by  $e_i$ . While computing  $par_i$  any of the following could be used:

- (i) Some parameters of  $e_i$ .
- (ii) Some attributes  $e_i$  has read.

- (iii) Some returned values by a method invoked by  $e_i$ .
- (iv) A computed value from any the above.

Hence,  $par_i$  should be at least as protected as any of the above elements that were used in deriving it. To avoid program analysis, which is beyond the scope of this paper, to determine how the value of  $par_i$  is derived we have considered  $VACL(e_i)$  which considers all parameters of  $e_i$  (see Definition 6). Once again we are considering potential flows in this work rather than actual flows.

**Definition 6:**  $VACL(e_i)$  is the set of objects that can access a computed value by  $e_i$ .  $VACL(e_i)$  is constructed incrementally as the execution  $e_i$  proceeds as follows:

Let  $par_1, par_2, \dots, par_n$  be the parameters of the message invoking  $e_i$ , (the  $RACL$  of each parameter is defined by Definition 5).

- (i) At the start of  $e_i$ , If  $e_i$  is invoked by a user or it has no parameters set  $VACL(e_i) := O$  otherwise set  $VACL(e_i) = RACL(par_1) \cap RACL(par_2) \cap \dots \cap RACL(par_n)$
- (ii) Each time  $e_i$  reads an attribute  $a$ , set  $VACL(e_i) = VACL(e_i) \cap RACL(a)$ .
- (iii) Each time  $e_i$  receives a reply from an execution  $e_k$  of some method  $m_k$ , set  $VACL(e_i) = VACL(e_i) \cap RACL(e_k)$  where  $RACL(e_k)$  is the set of objects that are allowed to read the reply of  $e_k$  and it is defined below in Def 7. Note case (ii) is covered by (iii), it is written for paper readability.

In (i) if  $e_i$  has no parameters (no information transmitted to it by the message) then a computed value by  $e_i$  should be accessible to all objects (no other methods are invoked so far by  $e_i$ ). if  $e_i$  is invoked by a user  $VACL(e_i)$  is set to  $O$ . That is, we only consider the information read during transaction execution not the information introduced by the user. This is the same as in [1]. If  $e_i$  is not invoked by a user and receives information from its invoker then a computed value by  $e_i$  should be at least protected as the information received and  $VACL(e_i)$  is set to  $RACL(par_1) \cap RACL(par_2) \cap \dots \cap RACL(par_n)$ . In (ii) and (iii), each time  $e_i$  receives a reply from an execution  $e_k$  it invoked then a computed value by  $e_i$  should be at least as protected as the reply and  $VACL(e_i)$  is set to  $VACL(e_i) \cap RACL(e_k)$ .

To decide whether a reply should be returned to the invoker or be blocked, we define the read access list associated with an execution.

**Definition 7:** Given an execution  $e_i$  of a method of object  $o_i$ , the read access list associated with it,  $RACL(e_i)$ , is the set of objects that are authorized to access the information in the reply of  $e_i$ .  $RACL(e_i)$  is constructed incrementally while  $e_i$  is executing as follows:

- (a) If  $e_i$  is a read of an attribute  $att$ , set  $RACL(e_i) = RACL(att)$ .
- (b) If  $e_i$  is a write or a create, set  $RACL(e_i) = O$ .
- (c) If  $e_i$  is not a read, a write, nor a create:
  - (i) At the start of  $e_i$  set  $RACL(e_i) = O$ .
  - (ii) Each time  $e_i$  reads an attribute  $att$ , set  $RACL(e_i) = RACL(e_i) \cap RACL(att)$ .
  - (iii) Each time  $e_i$  receives a reply from an execution  $e_k$  of some method  $m_k$ , set  $RACL(e_i) = RACL(e_i) \cap$

$RACL(e_k)$ . Note case (ii) is covered by (iii), it is included for paper readability.

(a) is self explanatory. (b) if  $e_i$  is a write or a create no information about the state of  $o_i$  is returned and hence  $RACL(e_i)$  is set to  $O$ . (i) is self explanatory, (iii) each time  $e_i$  receives a reply from an execution  $e_k$  it invoked then the reply of  $e_i$  should be at least as protected as the reply of  $e_k$ , hence  $RACL(e_i)$  is set to  $RACL(e_i) \cap RACL(e_k)$ . Note: the concept of  $RACL(e_i)$  is adopted from [1].

## 5. Message filtering

The message filter [23] is a trusted system component which has the ability to intercept messages exchanged among objects to control the flow of information. In this section, we elaborate on the information flow control policies using message filtering. The filter blocks a message from  $o_i$  to  $o_j$  if  $o_i$  does not have the right to access the information (parameters) in the message or the message could enact an unsafe flow as explained below. The filter would block a message/reply in the following cases:

Let the execution  $e_i$  of method  $m_i$  of object  $o_i$  invokes the execution  $e_j$  of method  $m_j$  of object  $o_j$ ,  $o_i$  sends a message  $\langle m_j, par_1, par_2, \dots, par_k \rangle$  to  $o_j$ .

- (a)  $o_i$  does not have the permission to invoke the required method by the message., i.e.  $o_i \notin PERL(m_j)$ .
- (b) If  $o_j$  does not have the right to read the information in the message. That is,  $o_j$  is not in the  $RACL$  of each parameter.
- (c) If  $o_j$  saves the information received from  $o_i$  (case of a write or a create), and it may pass it (or a derived value from it) to unauthorized objects directly or indirectly through an authorized object. In this case it said that the message enacted an unsafe (forward) flow.
  - (c1) In case of a write  $\langle WRITE, (att, val) \rangle$  For the flow of information from  $o_i$  to  $o_j$  to be safe,  $att$  should be at least as protected as  $val$ . That is,  $RACL(att) \subset RACL(val)$  must be satisfied.
  - (c2) In case of a create  $\langle CREATE, val_1, val_2, \dots, val_l \rangle$ .  $o_j$  must be a class. The create creates an object  $o$  with attributes  $att_1, att_2, \dots, att_l$  with values  $val_1, val_2, \dots, val_l$  and methods  $m_1, m_2, \dots, m_n$ . For the flow to be safe,  $RACL(att_k)$  should be set to  $\{o_i\}$  for every attribute  $att_k$  of  $o$ ,  $WACL(att_k)$  should be set to  $\{o_i\}$ , for every attribute  $att_k$  of  $o$ , and  $PERL(m_k)$  should be set to  $\{o_i\}$  for every method  $m_k$  of  $o$ .
- (d) If  $o_i$  receives information through the reply of  $o_j$ , it may pass it to unauthorized objects (directly or indirectly). In this case it said that the message enacted an unsafe (backward) flow. This information passing may happen during the execution of  $e_i$  or even after it finishes execution if the information is saved in the attributes of  $o_i$ . Hence, for the flow to be safe each attributes of  $o_i$  should be at least as protected as the reply and any value computed by  $e_i$  should be at least as protected as the reply. That is,  $\{VACL(e_i) \cup \{ \cup_k RACL(att_k) \text{ where } att_k \text{ is an attribute of } o_i \} \} \subset RACL(e_j)$  must be satisfied.

The code for message filter is given below.

**Input:** Message ( $msg$ ) sent by execution  $e_i$ , running on object  $o_i$ .  
The message requires the execution  $e_j$  on  $o_j$ .

**Output:** return reply of  $e_j$  which might be *success*, *failure*, *NIL*, or a tuple of return values.

```

1  begin
2   $RACL(e_j) = O$ .
3  if  $o_i \in U$  //  $e_j$  is invoked by a user
4  then  $VACL(e_j) := O$ 
5    invoke  $e_j$ 
6     $RACL(e_j) := o_i$ 
7     $reply := \text{reply from } e_j$ 
8    return  $reply$  to  $e_i$ 
9  else case  $msg$  of
10 (1)  $\langle \text{READ}, att \rangle$  do
11   //  $att$  is an attribute of  $o_j$ .  $e_j$  is a read.
12   if  $o_i \in RACL(att)$ 
13   then invoke  $e_j$  // let message pass
14     //  $reply$  will be set to the value of  $att$ 
15      $RACL(e_j) := RACL(att)$ 
16      $RACL(e_i) := RACL(e_i) \cap RACL(e_j)$ 
17      $VACL(e_i) := VACL(e_i) \cap RACL(e_j)$ 
18   else do not invoke  $e_j$  // block message
19      $reply := failure$ 
20   return  $reply$  to  $e_i$ 
21 (2)  $\langle \text{WRITE}, (att, val) \rangle$  do
22   //  $e_j$  is a write,  $val$  value of attribute  $att$ 
23   if  $o_i \in WACL(att)$ 
24   then if  $RACL(att) \subset RACL(val)$ 
25     //  $val$  is param. with  $RACL$ .
26     then invoke  $e_j$  //  $att$  will be set to  $val$ 
27        $RACL(e_j) := O$ 
28        $RACL(e_i) := RACL(e_i) \cap RACL(e_j)$ 
29        $VACL(e_i) := VACL(e_i) \cap RACL(e_j)$ 
30        $reply := success$ 
31     else  $reply := failure$ 
32   else  $reply := failure$ 
33   return  $reply$  to  $e_i$ 
34 (3)  $\langle \text{CREATE}, val_1, val_2, \dots, val_l \rangle$  do
35   //  $val_1, val_2, \dots, val_l$  are attribute values
36   if  $o_i \in CACL(o_j)$ 
37   then invoke  $e_j$  // create object  $o$ 
38     // with attributes  $att_1, att_2, \dots, att_l$ 
39     // with values  $val_1, val_2, \dots, val_l$ 
40     // and methods  $m_1, m_2, \dots, m_n$ 
41      $RACL(att_k) := \{o_i\}$ , for every attribute  $att_k$  of  $o$ 
42      $WACL(att_k) := \{o_i\}$ , for every attribute  $att_k$  of  $o$ 
43      $PERL(m_k) := \{o_i\}$  for every method  $m_k$  of  $o$ 
44      $RACL(e_j) := O$ 
45      $RACL(e_i) := RACL(e_i) \cap RACL(e_j)$ 
46      $VACL(e_i) := VACL(e_i) \cap RACL(e_j)$ 
47      $reply := oid$  of  $o$ 
48   else  $reply := failure$ 
49   return  $reply$  to  $e_i$ 
50 (4)  $\langle m, par_1, par_2, \dots, par_l \rangle$ 
51   // Where  $m$  is not READ, WRITE, nor CREATE.
52   //  $e_j$  is the execution of method  $m$ 
53   if  $o_i \in PERL(m)$  AND  $o_i \in RACL(par_k)$  for
54      $k = 1, \dots, l$ 
55   then if no parameters then set  $VACL(e_j) := O$ 
56   else set  $VACL(e_j) := \bigcap_k par_k$ 
57   invoke  $e_j$ 
58   if  $\{VACL(e_i) \cup \{\bigcup_k RACL(att_k) \text{ where } att_k \text{ is an}$ 
59     attribute of  $o_i\}\} \subset RACL(e_j)$ 
60   then  $reply := \text{reply from } e_j$ 
61      $RACL(e_i) := RACL(e_i) \cap RACL(e_j)$ 
62
63

```

```

64            $VACL(e_i) := VACL(e_i) \cap RACL(e_j)$ 
65   else  $reply := NIL$ 
66   else do not invoke  $e_j$ 
67      $reply := failure$ 
68   return  $reply$  to  $e_i$ 
69

```

## 6. Proof of Correctness

In this section we prove that the filter is correct. First, we show that the filter correctly builds the RACL's and the VACL's of executions. Second, we proof that the filter blocks every reply that should not be returned. Finally, we proof that the filter blocks only and only writes that may enact an unsafe flow. The reader may notice that the proofs that follow are very similar in flavor to those produced in [1]. That is due to the fact that we used a formalism based on the framework developed in [1].

**Lemma 1:** *The message filter enforces the access rights of the attributes and the permissions of methods.*

*Proof:* the control is applied before an execution starts. The lemma is trivially satisfied.

□

In the remaining proofs we assume that the access rights and permissions are satisfied. That is, a message/reply is blocked only if it may enact unsafe flow.

**Lemma 2:** *The message filter constructs the RACL's of executions correctly in accordance with Definition 7.*

*Proof:* As an execution  $e_i$  proceeds the message filter builds its RACL as follows:

If  $e_i$  is a read, write, or create then the filter sets its RACL to the RACL of the attribute read, to  $O$ , or to  $O$ , respectively (refer to lines 14, 27, 45 in the filter code). The filter never changes the RACL of  $e_i$  subsequently. This setting is in agreement with Def. 7.

If  $e_i$  is not a primitive method execution the filter updates its RACL to  $RACL(e_i) \cap RACL(e_j)$  every time it receives a non *failure* nor *NIL* reply from an execution  $e_j$  it invoked (refer to line 63 in the filter code) and this is in agreement with Definition 7.

□

**Lemma 3:** *The filter builds the VACL's of executions correctly.*

*Proof:* we prove this lemma by induction. Given a transaction  $T = e_1, e_2, e_3, \dots, e_k$  where  $e_i$  for  $i=1, \dots, k$  are the executions invoked during  $T$  and that  $e_1$  is invoked by the transaction's user,  $e_1$  is invoked before  $e_2$ ,  $e_2$  is invoked before  $e_3$ , and so on.

The base of the induction is the first execution  $e_1$ .  $VACL(e_1)$  is set to  $O$  (line 4 in the filter code) then each time it receives a reply from an execution  $e_j$  ( $e_j$  invoked it) that is not *failure* nor *NIL*,  $VACL(e_i)$  is updated to  $VACL(e_i) \cap RACL(e_j)$  (refer to line 63 in the filter code). This is in agreement with Def 6.

Assume that  $e_j$  is invoked by  $e_i$  ( $1 \leq i$  and  $j \leq k$ ) and that the lemma is true for  $e_i$ . We show the lemma is true for  $e_j$ . If  $e_j$  has no parameters (the message to invoke  $e_j$  has no parameters)  $VACL(e_j)$  is set to  $O$  otherwise it is set to

$RACL(p_1) \cap RACL(p_2) \cap \dots \cap RACL(p_n)$  where  $p_1, p_2, \dots, p_n$  are the parameters of the message (refer to line 57 in the filter code). Each time  $e_j$  receives a non *failure* nor a *NIL* reply from an execution (it invoked)  $VACL(e_j)$  set to  $VACL(e_i) \cap RACL(e_k)$  (refer to line 63 in the filter code). This is in agreement with Def 6.

□

**Theorem 1:** *The filter blocks every reply from an execution that should not be read by the invoker.*

*Proof:* Assume  $e_i$  executes on  $o_i$  invokes  $e_j$  which executes on  $o_j$ . The filter checks if  $\{VACL(e_i) \cup \{\cup_k RACL(att_i)\}\} \subset RACL(e_j)$  is not satisfied a *NIL* reply is returned (line 65 in the filter code). Hence, the theorem is satisfied.

□

**Theorem 2:** *A write is blocked if and only if it may enact an unsafe flow.*

*Proof:* Assume  $e_i$  invokes  $e_j$  and  $e_j$  is a write that writes the attribute *att* with the value *val*.

Reply of  $e_j$  is set to *failure* iff  $RACL(att) \not\subset RACL(val)$  (*val* is a parameter and it has its own RACL, refer to Def 5).

We prove that reply of  $e_j$  is set to *failure*  $\Rightarrow RACL(att) \not\subset RACL(val)$  by proving its contra positive.  $RACL(att) \subset RACL(val) \Rightarrow$  the reply of  $e_j$  is set to *success* if  $RACL(att) \not\subset RACL(val)$  the filter sets reply of  $e_j$  to *success* (refer to line 32 in the filter code). That proves  $RACL(att) \subset RACL(val) \Rightarrow$  the reply of  $e_j$  is set to *success*.

We prove that  $RACL(att) \not\subset RACL(val) \Rightarrow$  reply of  $e_j$  is set to *failure* by contradiction. Assume that  $RACL(att) \subset RACL(val)$  and that reply of  $e_j$  is set to *success*. If  $RACL(att) \not\subset RACL(val)$  the filter would set reply of  $e_j$  to *failure* (refer to line 33 in the filter code) which contradicts the assumption.

□

**Theorem 3:** *Every allowed create does not enact unsafe flow.*

*Proof:* Assume  $e_i$  executes on  $o_i$  invokes  $e_j$  which executes on  $o_j$  and  $e_j$  is a create that creates an object  $o$  with attributes  $att_1, att_2, \dots, att_l$  and attribute values  $val_1, val_2, \dots, val_l$ . In line 42, the filter sets  $RACL(att_k) = \{o_i\}$  for every attribute,  $o_i$  is the owner of  $o$ . By Definition 5,  $o_i$  is in the RACL of every parameter  $val_k$  and hence  $RACL(att_k) \subset RACL(val_k)$ .

□

## 7. Experiments

We have run some experiments in which the objects, attributes, methods, access rights, and transactions were generated randomly. The results show the model we proposed is much more flexible than that of [1] in the sense the number of legal flows that are permitted by our model is much higher. The results also show that our model had blocked every illegal flow. The experiments are described in [22].

## 8. Conclusion

We have proposed a flexible and nonrestrictive information flow model for object-oriented systems without

compromising system security or increasing the potential for information leakage and disclosure.

By considering the authorizations of object attributes and methods, we can get rid of all unnecessary messages blocking that the message filter in [1] strictly enforces. Defining authorizations for attributes and methods dovetail with access rights semantics in the real world and fits very well with the object oriented paradigm. This work leaves some issues not addressed and opens further research. One issue to be investigated is the case where methods do store information between executions (for example the case of local static variables in C++). A second issue is how to incorporate inheritance.

## References

- [1] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia, "Information Flow Control in Object-Oriented Systems", IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 4, pp. 524-538, July/August 1997.
- [2] C. N. Zhang and C. Yang, "Information Flow Analysis on Role-based Access Control Model", Information Management and Control Security, Vol. 10, No. 5, pp. 225-236, 2002.
- [3] C. N. Zhang and C. Yang, "An Object-Oriented RBAC Model for Distributed Systems", Working IEEE/IFIP Conference on Software Architecture, (WICSA), Amsterdam, The Netherlands, pp. 24-32, 2001.
- [4] S. Osborn, R. Sandhu, and Munawer, "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies", ACM Transactions on Information and System Security, Vol. 3, No. 2, pp. 85-106, 2000.
- [5] F. Pottier and S. Conchon, "Information Flow in Inference for Free", in Proc. ACM International Conference on Principles of Functional Programming, New York, NY, pp. 46-57, Sept. 2000.
- [6] G. Smith, "A New Type System for Secure Information Flow", in Proc. of 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, pp. 115-25, 2001.
- [7] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security", IEEE Journal on Selected Areas in Communications, Vol. 21, No. 1, pp. 5-19, January, 2003.
- [8] S. Zdancewic and A. C. Myers, "Secure Information Flow and CPS", in Proc. European Symposium on Programming, Vol. 2028 of LNCS, pp. 46-61, April 2001.
- [9] A. Banerjee and D. A. Naumann, "Secure Information Flow and Pointer Confinement in a Java-Like Language", in Proc. IEEE Computer Security Foundations Workshop, pp. 253-267, June 2002.
- [10] F. Pottier and V. Simonet, "Information Flow Inference for ML", in Proc. ACM Symp. on Principles of Programming Languages, pp. 319-330, Jan. 2002.
- [11] A. C. Myers and B. Liskov, "Protecting Privacy Using the Decentralized Label Model", ACM TOSEM, Vol. 9, No. 4, pp. 410-442, Oct. 2000.

- [12] A. C. Myers and B. Liskov, "Complete, Safe Information Flow with Decentralized Labels", Proceedings of IEEE S&P, Oakland, California, 1998.
- [13] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control", in Proc. ACM Symp. on Operating System Principles (SOSP), Saint-Malo, France, pp. 129-142, 1997.
- [14] R. Sandhu, "Role Activation Hierarchies", in Proc. of 3rd ACM Workshops on Role-Based Access Control, Fairfax, Virginia, pp. 22-23, 1998.
- [15] R. Sandhu and P. Samarati, "Authentication, Access Control, and Audit", ACM Computing Surveys, Vol. 28, No. 1, March 1996.
- [16] D. E. Denning, "A Lattice Model of Secure Information Flow", Comm. of the ACM, Vol. 19, No. 5, pp. 236-243, 1976.
- [17] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow", Comm. of the ACM, Vol. 20, No. 7, pp. 504-513, July 1977.
- [18] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia, "Providing Flexibility in Information Flow Control for Object-Oriented Systems", in Proc. IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 130-140, May 1997.
- [19] D. F. Ferraiolo, J. A. Gugini, and D. R. Kuhn, "Role-Based Access Control: Features and Motivations", in Proc. 11th Annual Computer Security Applications Conference, New Orleans, LA, Dec. 1995.
- [20] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A Role Based Access Control Model and Reference Implementation within a Corporate Intranet", ACM Transactions on Information and Systems Security, Vol. 2, No. 1, Feb. 1999.
- [21] A. Maamir, A. Fellah, "Adding Flexibility in Information Flow Control for Object-Oriented Systems Using Versions", International Journal of Software Engineering and Knowledge Engineering, Vol. 13, No. 3, June 2003, pp. 313-325.
- [22] A. Maamir, A. Fellah, and L. A. Salem, "Fine Granularity Access Rights for Information Flow Control in Object Oriented Systems", in Proc. The 2<sup>nd</sup> Int. Conf. on Information Security and Assurance. Busan, Korea, pp. 122-128, April 2008.
- [23] S. Jajodia and B. Kogan, "Integrating an Object-Oriented Data Model with Multilevel Security", in Proc. IEEE Symp. on Security and Privacy, Oakland, CA, USA, pp. 76-85, 1990.
- [24] S. Chou, W. Lo, and C. Lai, "Information Flow Control in Multithread Applications Based on Access Control Lists", Information & Software Technology, Vol. 48, No. 8, 2006, pp. 717-725.
- [25] S. Chou, "Embedding Role-Based Access Control Model in Object-Oriented Systems to Protect Privacy", Journal of Systems and Software, Vol. 71, No. 1-2, 2004, pp. 143-161.

### Author Biographies

**Allaoua Maamir:** Dr. Allaoua Maamir received a B.Sc. from Université des Sciences et de la Technologie, Algiers, Algeria, in 1981, M.Sc. from Georgia Institute of technology, Atlanta, Georgia, USA, in 1984, and Ph.D. from Wayne State University, Detroit, Michigan, USA, in 1994. All degrees are in Computer Science. Currently, he is an Assistant Professor of computer science at the University of Sharjah, UAE. His research interests include databases, information flow control in software systems, and data cleaning.

**Abdelaziz Fellah:** Dr. Abdelaziz Fellah received a B.Sc. from Université de Constantine, Constantine, Algeria, in 1981, M.Sc. from Case Western Reserve University, Cleveland, Ohio, USA, in 1985, and Ph.D. from Kent State University, Kent, Ohio, USA, in 1992. All degrees are in computer science. His research interests include formal languages and automata, theoretical computer science, parallelism, and information flow control in software systems.

**Lina A. Salem:** Mrs. Lina A. Salem received a B.Sc. in computer science from Ajman University, UAE, in 1999, M.S.c in computer science from the University of Sharjah, UAE. Her research interests include databases and information flow control in software systems.