Fine Granularity Access Rights for Information Flow Control in Object Oriented Systems

Allaoua Maamir Abdelaziz Fellah 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

One of the main features of information flow control is to ensure the enforcement of privacy and regulated accessibility. However, most information flow control models that have been proposed do not provide substantial assurance to enforce end-to-end confidentiality policies or they are too restrictive, overprotected, and inflexible. We present a model for discretionary access controls that is in harmony with the object oriented paradigm. The model uses access rights applied to object attributes and methods, thus allowing considerable flexibility without compromising system security by leaking sensitive information. Models based on message filtering intercept every message exchanged among objects to control the flow of information. We present an algorithm which enforces message filtering based on the defined access rights.

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 the information. Information flow control 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 [4, 8, 15, 18, 20, 21]. 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, 12, 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 [18] has been proposed by the same authors. 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 [8, 10, 13, 18]. 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 [1, 16, 17, 22].

An access rule is specified in the form $< s, o, op >$, 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, [6, 7, 19]. 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 [14]. 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 privilege. 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 [7]. A variation of the RBAC model called object oriented role-based access model (ORBAC) has been proposed by [23]. 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 [24].

Although the ability for discretion to specify accesses is not lost in the model of [18], 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 [3, 2] 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 presents the experiment results and finally Section 6 concludes the paper.

2. Object-Oriented Model

Object-oriented systems are composed of objects. Objects can be defined as an encapsulation of *data state*, *methods* for manipulating that data. Classes are prototypes of objects. An object is a physical implementation 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 $D_1, D_2, \ldots, D_n$. Let $D = D_1 \cup D_2 \cup \cdots \cup D_n \cup \{nil\}$ where *nil* is a special element. Every element of D is referred to as a primitive object (an integer, a string,$\cdots$). 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. Denote by U the set of all user identifiers and by $O = I \cup U$. Each element of O is referred to as an object identifier (oid).

Definition 1. *A non primitive object, o, is a tuple $<i, A, V, M>$ 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 $<m, par_1, par_2, \ldots, par_k>$ 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 $<rp_1, \ldots, rp_n>$ where $rp_i \in D \cup I$, $i = 1, \ldots, n$.*

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 abject 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 objects 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 [18]. This is the only interaction mode we assume in this paper, our model could be augmented with the other interaction modes defined in [18]. 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 [18]. Our assumption is identical to that in [22]. We make similar assumption to that of [18], 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 [4], 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 $< m_j, par_1, par_2, \ldots, par_k >$ 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 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 $< m_j, par_1, par_2, \ldots, par_k >$. 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:

(*a*) Some parameters of e_i.

(*b*) Some attributes e_i has read.

(*c*) Some returned values by a method invoked by e_i.

(*d*) 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 $par_1, par_2, \ldots, par_n$ 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) $\cap \cdots \cap$ RACL(par_n).*

(*ii*) *Each time e_i reads an attribute a, set RACL(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 Definition 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 [18]. 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) $\cap \cdots \cap$ 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.

Case (*a*) is self explanatory. (*b*) indicates 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 [18].

5. Message Filtering

The message filter [9] 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 $<m_j, par_1, par_2, \ldots, par_k>$ 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.

(c_1) In case of a write <WRITE(att, val)> 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.

(c_2) In case of a create <CREATE($val_1, val_2, \ldots, val_l$)> o_j must be a class. The operate create an object o with attributes $att_1, att_2, \ldots, att_l$ with values $val_1, val_2, \ldots, val_l$ and methods $m_1, m_2, \ldots, m_n$. For the flow to be safe RACL(att_k) should be set to $\{o_i\} \cup (\cap val_i,\ i = 1, \ldots, l)$ 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) $\cap$ $\{$RACL(att_k) where att_k is an attribute of o_i $\}\}$ $\subset$ RACL(e_j) must be satisfied.

The code for the 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.

```
begin
RACL($e_j$) = $O$.
if $o_i \in U$ // $e_j$ is invoked by a user
then VACL($e_j$) := $O$
    invoke $e_j$
    RACL($e_j$) := $o_i$
    reply := reply from $e_j$
    return reply to $e_i$
else case msg of
(1) <READ, att> do
        // att is an attribute of $o_j$.$e_j$ is a read
    if $o_i \in$ RACL(att)
    then invoke $e_j$ // let message pass
        // reply will be set to the value of att
        RACL($e_j$) := RACL(att)
        RACL($e_i$) := RACL($e_i$) $\cap$ RACL($e_j$)
        VACL($e_i$) := VACL($e_i$) $\cap$ RACL($e_j$)
        // update VACL of $e_i$
         reply := failure
    return reply to $e_i$

(2) <WRITE, (att, val)> do
        // $e_j$ is a write, val value of attribute att
    if $o_i \in$ WACL(att)
    then if RACL(att) $\subset$ RACL(val)
        // val is a parameter and it has an RACL
         then invoke $e_j$ // att will be set to val
             RACL($e_j$) := $O$
             RACL($e_i$) := RACL($e_i$) $\cap$ RACL($e_j$)
             VACL($e_i$) := VACL($e_i$) $\cap$ RACL($e_j$
              reply := success
         else reply := failure
    else   reply := failure
    return reply to $e_i$

(3) <CREATE, $val_1, val_2, \ldots, val_l$> do
        // $val_1, val_2, \ldots, val_l$ are attribute values
    if $o_i \in$ CACL($o_j$)
    then invoke $e_j$ // creates object $o$
        // with attributes $att_1, att_2, \ldots, att_l$
        // with values $val_1, val_2, \ldots, val_l$
        // and methods $m_1, m_2, \ldots, m_n$
         RACL($att_k$) := $\{o_i\} \cup (\cap val_i, i = 1, \ldots, l)$,
            WACL($att_k$) := $o_i$ for every
            attribute $att_k$ of $o$
            PERL($m_k$) := $o_i$ for every method $m_k$ of $o$
            RACL($e_j$) := $O$
            RACL($e_i$) := RACL($e_i$) $\cap$ RACL($e_j$)
            VACL($e_i$) := VACL($e_i$) $\cap$ RACL($e_j$)
            *reply* := $oid$ of $o$
        **else** *reply* := *failure*
        **return** *reply* to $e_i$
  (4) $<m, par_1, par_2, \ldots, par_l>$
      // Where $m$ is not *READ, WRITE,* nor *CREATE*.
      // $e_j$ is the execution of method $m$
        **if** $o_i \in$ PERL($m$) AND $o_j \in$ RACL($par_k$), for
        $k = 1, \ldots, l$
        **then if** no parameter **then** set VACL($e_j$) := $O$
            **else** set VACL($e_j$) := $\cap_k\, par_k$
            invoke $e_j$
            **if** $\{$VACL($e_i$) $\cup \{\cup_k$RACL($att_k$) where $att_k$
            is an attribute of $o_i\}\} \subset$ RACL($e_j$)
            **then** *reply* := *reply* from $e_j$
                RACL($e_i$) := RACL($e_i$) $\cap$ RACL($e_j$)
                VACL($e_i$) := VACL($e_i$) $\cap$ RACL($e_j$)
            **else** *reply* := *NIL*
        **else** do not invoke $e_j$
            *reply* := *failure*
        **return** *reply* to $e_i$
```

We have showed that the algorithm disallows every unsafe flow and that if a flow is allowed then it is safe. Proofs are in a technical report and interested readers can request them directly from the authors.

6. Experiments

We have run some experiments in which objects, access rights (ACL's and permissions), and transactions were randomly generated. First, we used 3 classes C_1, C_2, and C_3. C_1 has 4 attributes and 4 methods. C_2 has 3 attributes and 2 methods. C_3 has 5 attributes and 2 methods. The maximum total number of randomly generated objects is 30 objects. Each run of the experiment considered 30 randomly generated transactions. Table 1 shows for each run the transactions that were not allowed because they might have enacted unsafe flows. In each case we checked each transaction manually to see whether the transaction should have been allowed or blocked, that is we considered actual flows. We ran the same experiments with the filter in [18]. The results show that our filter has allowed a much larger number ”legal” transactions (those that were checked manually not to enact unsafe flows) that then filter of [18] as we have expected. Access rights were mapped from our model to correspond to the model of [18].

Table 1. First Experiment. NT indicates the number of transactions.

Total number of objects	NT allowed by our filter	NT allowed by filter of [18]	NT should be allowed
9	12	4	15
15	12	1	14
18	12	2	13
21	13	1	15
24	7	1	8

We ran a second set of experiments in which we considered classes with more attributes and methods. C_1 has 14 attributes and 10 methods. C_2 has 2 attributes and 8 methods. C_3 has 5 attributes and 3 methods. The results are recorded in Table 2. As expected our filter had much better results than that of [18].

Table 2. Second Experiment. NT indicates the number of transactions.

Total number of objects	NT allowed by our filter	NT allowed by filter of [18]	NT should be allowed
9	13	0	14
15	10	1	10
18	9	0	10
21	8	2	9
24	8	0	9

7. 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 [18] 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] A. Banerjee and D. A. Naumann. Secure information fbw and pointer confi nement in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

[2] S. Chou. Embedding role-based access control model in object-oriented systems to protect privacy. *Journal of Systems and Software*, 71(1-2):143–161, 2004.

[3] S. Chou and *et* al. Information fbw control in multithread applications based on access control lists. *Information & Software Technology*, 48(8):717–725, 2006.

[4] D. E. Denning. A lattice model of secure information fbw. *Comm. of the ACM*, 19(5):236–243, 1976.

[5] D. E. Denning and P. J. Denning. Certifi cation of programs for secure information fbw. *Communication of the ACM*, 20(7):504–513, July 1997.

[6] D. F. Ferraiolo and *et* al. Role-based access control: Features and motivations. In *Proc. 11th Annual Computer Security Applications Conference*, pages 241–248, Dec. 1995.

[7] D. F. Ferraiolo and *et* al. A role based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and Systems Security*, 2(1):34–64, Feb. 1999.

[8] E. Ferrari and *et* al. Providing fexibility in information fbw control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, May 1997.

[9] S. Jajodia and B. Kogan. Data model with multilevel security. In *Proc. IEEE Symp. on Security and Privacy*, pages 76–85, 1990.

[10] A. Maamir and A. Fellah. Adding fexibility in information fbw control for object-oriented systems using versions. *International Journal of Software Engineering and Knowledge*, 23(3):313–325, June 2003.

[11] A. C. Myers and B. Liskov. A decentralized model for information fbw control. In *Proc. ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997.

[12] A. C. Myers and B. Liskov. Complete safe information fbw with decentralized labels. In *Proc. of IEEE S&P,*, 1998.

[13] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–422, Oct. 2000.

[14] S. Osborn and *et* al. Confi guring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[15] F. Pottier and S. Conchon. Information fbw in inference for free. In *Proc. ACM International Conference on Principles of Functional Programming*, pages 46–57, September 2000.

[16] F. Pottier and V. Simonet. Information fbw inference for ml. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[17] A. Sabelfeld and A. C. Myers. Language-based information-fbw security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[18] P. Samarati and *et* al. Information fbw control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July/August 1997.

[19] R. Sandhu. Role activation hierarchies. In *Proc. of 3rd ACM Workshops on Role-Based Access Control*, pages 22–23, 1998.

[20] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1):241–243, March 1996.

[21] G. Smith. A new type system for secure information fbw. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 115–25, 2001.

[22] S. Zdancewic and A. C. Myers. Secure information fbw and cps. In *Proc. European Symposium on Programming*, volume 2028 of *LNCS*, pages 46–61, April 2001.

[23] C. N. Zhang and C. Yang. An object-oriented rbac model for distributed systems. In *Working IEEE/IFIP Conference on Software Architecture, (WICSA)*, pages 24–32, 2001.

[24] C. N. Zhang and C. Yang. Information fbw analysis on role-based access control model. *Information Management and Control Security*, 10(5):225–236, 2002.