High Assurance Discretionary Access Control for Object Bases

Elisa Bertino* Pierangela Samarati* Sushil Jajodia†

Abstract

Discretionary access control, based on checking access requests against users' authorizations, does not provide any way of restricting the usage of information once it has been "legally" accessed. This makes discretionary systems vulnerable to Trojan Horses maliciously leaking information. Therefore the need arises for providing additional controls limiting the indiscriminate flow of information in the system. This paper proposes a message filter complementing discretionary authorization control in object-oriented systems to limit the vulnerability of authorization systems to Trojan Horses. The encapsulation property of the object-oriented data model, which requires that access to objects be possible only through defined methods, makes information flow in such systems have a very concrete and natural embodiment in the form of messages and their replies. As a result, information information flow can be controlled by mediating the transmission of messages exchanged between objects. The message filter intercepts every message exchanged between objects to ensure that information is not leaked to objects accessible by users not allowed for it.

1 Introduction

Data protection is an important requirement for any system managing information. Two kinds of policies can be used for providing information protection: *discretionary* and *mandatory*. Discretionary policies restrict access to information on the basis of the users' identity and on authorizations stating the accesses that each user can execute on the objects of the system. Mandatory policies restrict access to information on the basis of classifications assigned to subjects and objects in the systems and relationships that must be satisfied on the flow of information in the system. The main drawback of mandatory policies is their rigidity which makes them unsuitable for many application environments. Therefore, most general purpose commercial DBMSs only provide protection through discretionary authorizations [9]. However, discretionary access controls do not provide any form of control on the usage of information once it has been "legally" accessed. This characteristic makes the discretionary control vulnerable to Trojan Horses embedded in applications. In particular, a malicious user can embed in some user's program, a Trojan Horse that, once the program is invoked, surreptitiously modifies data in the user's file or writes them in files accessible by the malicious user. If the user running the program has the necessary authorizations, this hidden and malicious function is considered as legitimate by the discretionary authorization control. Discretionary controls can therefore be easily bypassed and hence do not provide a real assurance on the satisfaction of the protection requirements stated through the authorizations.

In this paper we present an approach for dealing with Trojan Horses in object-oriented systems. The paper is organized as follows. Section 2 summarizes previous researches on complementing discretionary access control to cope with the Trojan Horse problem. Section 3 outlines our approach. Section 4 illustrates the object-oriented data model to which the control is applied. Section 5 introduces formal definitions and notations. Section 6 presents the authorization model which enforces the access control and discusses its weaknesses. Section 7 characterizes the flow of information inside the system. Section 8 discusses different control policies and presents the message filter algorithm. Finally, Section 9 presents some conclusions and outlines future work.

2 Previous work

The need for additional controls complementing the checking performed by discretionary authorization systems has been pointed out by other researchers and some work has been done on this issue. Some research efforts have been aimed at complementing access control with forms of access restriction that cannot be expressed in the discretionary authorization model [13, 8, 16]. In [13], the control is enforced by having all objects cre-

*Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy.

†Center for Secure Information Systems and Department of Information and Software Systems Engineering, George Mason University, 4400 University Drive, Fairfax, VA 22030-4444, U.S.A.
The work of S. Jajodia was partially supported by a grant from the National Science Foundation under IRI-9303416.

1st Conf.- Computer & Comm. Security '93-11/93 -VA,USA
© 1993 ACM 0-89791-629-8/93/0011...$1.50

ated during the execution of a process inherit the access control list of the objects read in the execution of the process itself prior to their creation. A similar proposal is made in [16] where each object has associated with it two protection attributes: the current access and the potential access. The current access attribute describes what can be done to whom on a particular object, and the potential access attribute describes what can be done by whom to the information in this object. Hence, potential access provides a mean of propagating possible access restrictions to the information once it has been released.

Other research efforts on complementing discretionary control have been specifically aimed at eliminating, or at least limiting, the vulnerability of such a control to Trojan Horses [18, 4, 12]. Walter et al. [18] propose the application of a strict need to know policy for limiting information flow during process execution in operating systems environment. A process is allowed to copy information from an object to another object only if the set of users allowed to read the second object is a subset of the set of users allowed to read the first object. The main drawback of this solution is the complexity and the rigidity of the control. Boebert and Ferguson [4] propose the use of a dynamic linker that compares the name of the user who invokes a program with the name of the originator of the program, and the name of the owner of the data files. If a user invokes a program owned by someone else and the program tries to tamper with any of the user's file, the dynamic linker will recognize the name mismatch and raise the appropriate alarms. This approach has two main drawbacks. Generally, no one user owns his own copy of an application program, and in a system there exists only one copy of the application program to be shared by all the users. Therefore, for the application program to be used by a user, the application program may need to be allowed to read and write the user's files. The second drawback of this approach is that it does not cope with Trojan Horses embedded in a program owned by the user running the program, which will therefore be allowed to freely exploit their effects. Note also that this solution may help in coping with Trojan Horses whose effect is to tamper the files of the user whereas it does not control the leakage of information. Another approach, proposed by Karger [12] consists of limiting the files accessible by the application programs on the basis of some knowledge on the program themselves. The control requires the specification of patterns of names describing the objects to be accessed by an application. When an application is run, a name checker compares the names of each object to be accessed against the specified pattern. If the object's name satisfies the pattern the access is granted, otherwise the user running the program is queried about the access requested. For example, a latex compiler can have as input, besides the tex library files, only files whose name ends with ".tex" and can write only files whose name ends with ".aux", ".log", or ".dvi". Note that this solution may prevent copying information in objects outside the scope of the application, but it does not guarantee complete protection. In particular, a Trojan Horse embedded in an application can leak the information to files which satisfy the name checker control but which are accessible to users not allowed to access the input files.

3 Overview of our approach

In this paper we deal with the problem of Trojan Horses in object-oriented systems. Object-oriented systems are characterized by the fact that every entity of the system is seen as an object. Every object has a set of attributes (properties) and a set of methods, i.e. procedures executable on the object. The attributes of an object can be accessed only through methods specified for the object which can be invoked by sending proper messages to the object. Thus, messages are the only means through which information flow between objects can be enacted. Due to this characteristic, information flow in object-oriented systems has a very concrete and natural embodiment in the form of messages and their replies. This makes it possible to control information flow by mediating the flow of messages exchanged between objects.

In our approach, messages are not allowed to be freely exchanged between objects. By contrast, every message and its reply are intercepted by a message filter, which decides how to handle them according to the security policy. The message filter may either let the information be transmitted unaltered, block it, or take some other actions, such as restricting the execution of the invoked methods. The task of the message filter is enforcing the strict need to know policy, therefore ensuring that information does not flow to objects accessible from users not authorized to read the objects from which the information has been read. The modularity of object-oriented system, where operations on objects are with objects as methods, makes it possible to isolate specific operations and provide flexibility in the application of the policy which may otherwise prove to be too rigid. In particular the strict need to know policy control can be executed either at the level of each single elementary operation (i.e., read or write), not allowing the operation to be executed if this may cause unsafe information flow, or at the level of the information produced by a method, therefore restricting the transmission of messages (an their replies) between objects.

In the paper, we discuss different interpretations of the strict need to know policy in object-oriented systems. We illustrate how different ways of enforcing the strict to know policy may have different effects on computation and how they all can be useful in some respects. To allow flexibility in the application of the policy, we then consider method executions that can run under different modes. Each of such modes has different effects on the type of control to be applied by the message filter.

In particular, we allow method executions to run synchronously or asynchronously. Indeed, the strictly sequential relationship between operations may unnecessarily limit the execution of some operations. The advantage of considering asynchronous executions, in addition to synchronous ones, is that it allows us to express independence between operations, therefore increasing

the computations executable without generating illegal information flow.

Moreover, we consider method executions that can run under restricted or unrestricted modes. If a method is executed under restricted mode, the strict need to know policy is enforced on the reply generated by the method. In particular the reply may be blocked if the object waiting for it is accessible by users that are not allowed to access the information contained in the reply. By contrast, if a method is executed under unrestricted mode, no constraint is enforced on the reply; therefore, information is freely transmitted between objects. In this case, the verification of the strict need to know policy will be ensured by restricting possible write operations on objects accessible from users not allowed to read the objects where the information has been taken. That is, write operations are forbidden in an object if this object receives some information from another object, and the set of users authorized to read the object to be written is not a subset of the set of users authorized to read the object in which the information has been read.

In summary, our model supports synchronous and asynchronous message exchanges, and restricted and unrestricted method execution modes. By combining these possibilities, executions can therefore run: (i) asynchronously, (ii) synchronously under restricted mode, and (iii) synchronously under unrestricted mode.[1] Those different options that we provide as part of our model decrease the intrinsic rigidity of the strict need to know policy therefore making it more flexible and adaptable to different application environments.

4 The object-oriented data model

An object-oriented system is a collection of objects communicating via messages and their replies.

To formalize the model we consider to have a finite set of domains $D_1, D_2, \ldots, D_n$. Let D be the union of all the domains together with a special element, *nil*, i.e., $D = D_1 \cup D_2, \ldots \cup D_n \cup \{nil\}$. We refer to every element of D as a *primitive object*. Moreover, let A be a set of symbols called *attribute names*, I a set of *identifiers*, M a set of finite strings called *methods*, and V a set of *values* defined as $V = D \cup I$. Then, the elements of the object oriented model can be characterized as follows [10].

Definition 1 (Object) An *object* is either a primitive object or a quadruple $o = (i, a, v, \mu)$ such that $i \in I$, $a = (a_1, \ldots, a_n), a_j \in A$ for all $j = 1, \ldots, n$, $v = (v_1, \ldots, v_k), v_j \in V$ for all $j = 1, \ldots, k$, and $\mu \subseteq M$.

Definition 1 states that an object is characterized by its identifier, which uniquely identifies the object in the system, an ordered set of attributes, an ordered set of values associated with the attributes, and a set of methods corresponding to procedures associated with the objects. In the following $i(o), a(o), v(o)$, and $\mu(o)$ denote respectively the identifier, the sets of attributes, the set of values, and the set of methods associated with object o.

Definition 2 (Message) A *message* g is a triple $g = (h, p, r)$ where h is the message *name*, $p = (p_1, \ldots, p_k)$, with $p_j \in V, j = 1, \ldots, k$, is an ordered set of values called message *parameters*, and $r = (r_1, \ldots, r_n), r_j \in V, j = 1, \ldots, n$ is the set of *return values*.

Similarly to the notation used for objects, in the following $h(g), p(g)$, and $r(g)$ denote respectively the name, the set of parameters, and the reply of message g.

The set of messages an object can respond to is called the *interface* of the object. The interface of the object determines which particular method, out of the set of methods $\mu(o)$ defined for the object, has to be executed upon reception of a given message. The interface f_o of object o is formally defined as a function $f_o : H \rightarrow \mu(o) \cup \{void\}$ where H is the set of all possible message names. Object o responds to all the messages h such that $f_o(h) \neq \{void\}$.

When an object receives a message, the corresponding method is executed. The execution of the method can imply having the object send a message to itself or to another object, reading or writing any of its attributes, and/or creating a new object. A reply is eventually returned to the object which sent the message.

Access to the internal attributes of an object and creation of objects are enforced by having the object sending special predefined messages to itself. These messages cause the execution of *built-in* methods providing the desired operations. The predefined messages an object o can send itself are as follows:

- A *read* message, denoted by $g = (\text{READ}, (a_j), r)$, returns the value of attribute a_j if $a_j \in a(o)$, *failure* otherwise.
- A *write* message, denoted by $g = (\text{WRITE}, (a_j, v_j), r)$, assigns value v_j to attribute a_j and returns *success* if $a_j \in a(o)$, does not produce any effect and returns *failure* otherwise.
- A *create* message, denoted by $g = (\text{CREATE}, (v_1, \ldots, v_n), r)$, creates a new object which inherits attributes and methods from o and whose attributes have the values passed as parameters of the message. The message returns the identifier i of the new object, if the creation succeeds, returns a *failure* otherwise.

5 Notations and definitions

In this section we introduce notations and definitions that will be used later on in the paper.

Each activity is started by having a user sending a message to an object. The execution of the corresponding method may cause the object to send further messages to itself or to a different object. The messages the object can send to itself include *read, write,* and *create.*

[1] The execution mode (either restricted or unrestricted) states how to filter the execution reply. Then, it cannot be applied to asynchronous executions whose replies are not returned to the invokers.

We refer to the set of all method executions invoked (directly or indirectly) as a consequence of the reception of a message from a user as a *transaction.*[2] Therefore a transaction consists of the execution of some methods. We refer to the user sending the message as the *owner* of the transaction.

In the following $t(o,m)$ denotes the execution of method m on object o during a transaction.[3] For sake of simplicity we will indicate with t a method execution when the method and the object involved are not of interest for the explanation. We will explicitly refer to the method being executed and the object on which the method is executed as $m(t)$ and $o(t)$ respectively, when needed. We will use r, w, and c, instead of t, to denote execution of *read, write*, and *create* methods respectively.

We consider that methods can be executed either synchronously or asynchronously. In the first case the sender of the message waits for the invoked execution to complete and the reply to return. In the latter case a *nil* reply is immediately returned to the sender which therefore proceeds independently from the execution invoked. By default, executions are synchronous. However, in some cases[4] we require executions to run asynchronously. Note that asynchronous executions may raise the problem of dealing with concurrent executions accessing the same object. We refer the reader to [17] for this.

In the following, $t_i \rightarrow_s t_j$ denotes the invocation of the synchronous execution t_j by t_i, whereas $t_i \rightarrow_a t_j$ denotes the invocation of the asynchronous execution t_j by t_i. We will use $t_i \rightarrow t_j$ to denote either one of them.

The following definitions characterize the relationships between executions belonging to the same transaction.

Definition 3 (Invocation dependency) Given two method executions t_i and t_j in a transaction T, we write $t_i \twoheadrightarrow_s t_j$ if there exist $t_1, \ldots, t_n \in T, (n \geq 0)$, such that $t_i \rightarrow_s t_1 \rightarrow_s \ldots \rightarrow_s t_n \rightarrow_s t_j$.
We write $t_i \twoheadrightarrow_a t_j$ if there exists $t_1, \ldots, t_n \in T, (n \geq 0)$, such that $t_i \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_j$, and at least one execution is asynchronous.

Definition 4 (Invocation order) Given two method executions t_i and t_j, invoked by the same execution t_k, we say that t_i is invoked before t_j, written $t_i <_i t_j$, if the message requiring the invocation of t_i is sent before the message requiring the invocation of t_j.

Note that since executions can run asynchronously, invocation order between executions does not necessarily corresponds to their execution order. In particular, if an execution is to be performed asynchronously it cannot be considered as preceding any of the execution invoked after it. This is formalized by the following definition.

Definition 5 (Execution order) Given two method executions t_i and t_j, t_i *precedes* t_j, written $t_i <_e t_j$, iff $t_i <_i t_j$ and there exists t_l such that $t_l \rightarrow_s t_i$ or $t_l \twoheadrightarrow_s t_i, t_l <_i t_j$.

Definition 5 states that execution t_i precedes execution t_j if and only if t_i is a synchronous execution and either is invoked before t_j or t_i depends, through a chain of invocations of synchronous execution, on a t_l preceding t_j.

Note that if all executions are synchronous, i.e., the sender of a message blocks waiting for the execution invoked by the message to terminate, for any two executions in a transaction, either one is dependent or precedes the other. By contrast, if executions can be asynchronous, there may exist no relationship between two executions. If two executions are neither in a dependency nor in a precedence relationship, we say that they are *independent.*

To graphically represent the relationship between method executions in a transaction, we define a *method invocation tree* as follows. The root of the tree is the method execution invoked upon reception of the message from the user. If, during execution t_i, execution t_j is invoked, t_j is inserted in the tree as a child of t_i. To respect the order between invocations, for any two executions t_h and t_k, if $t_h <_i t_k$, t_i will appear on the left of t_k in the tree. To distinguish between synchronous and asynchronous executions, invocations of asynchronous executions are represented with thick lines.

Note that, since no message can be sent as part of the execution of any of the built-in (read, write, and create) methods, built-in method executions are always leaves of the tree.

Example 1 Consider transaction T illustrated in Figure 1, the following relationships hold:

$\twoheadrightarrow_s = \langle t_1,t_2\rangle, \langle t_2,t_3\rangle, \langle t_1,t_3\rangle, \langle t_1,t_6\rangle, \langle t_6,t_8\rangle, \langle t_1,t_8\rangle, \langle t_1,t_9\rangle, \langle t_4,t_5\rangle$

$\twoheadrightarrow_a = \langle t_2,t_4\rangle, \langle t_2,t_5\rangle, \langle t_1,t_4\rangle, \langle t_1,t_5\rangle, \langle t_6,t_7\rangle, \langle t_1,t_7\rangle$

$<_e = \langle t_2,t_6\rangle, \langle t_2,t_7\rangle, \langle t_2,t_8\rangle, \langle t_2,t_9\rangle, \langle t_3,t_4\rangle, \langle t_3,t_5\rangle, \langle t_3,t_6\rangle, \langle t_3,t_7\rangle, \langle t_3,t_8\rangle, \langle t_3,t_9\rangle, \langle t_6,t_9\rangle, \langle t_8,t_9\rangle$

6 The authorization model

In this section we describe the authorization model by means of which access control is performed. The authorization model states how access authorizations are specified and how access decisions are taken on the basis of the specified authorizations. It is outside the scope of this paper to propose an authorization model for object-oriented systems. Many authorization models have been proposed and are still under study [1, 2, 3, 6, 7, 14]. In order to make our approach widely applicable,

[2] Note that it is not a transaction in the common sense of the term. The user can receive a reply and terminate a transaction even if some of the methods invoked are still executing (asynchronous computation).

[3] Note that a method can be invoked more than once on a given object during a transaction.

[4] We will elaborate on this is Section 8.

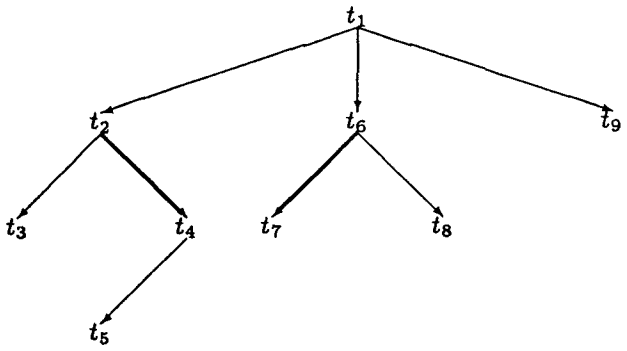

Figure 1: Example of method invocation tree

we consider a simple and general authorization model where the subjects to which authorizations can be referred are the users of the system, the objects are the objects of the object-oriented model, and the access modes executable by the subjects on the objects are the elementary access modes, i.e., *read*, *write*, and *create*.[5] Every authorization is a triple ⟨*user, access-mode, object*⟩, stating that *user* can access the *object* in the specified *access-mode*. We refer to the set of the authorizations as Authorization Base (AB for short).[6]

For every object, a Read Access Control List (RACL), a Write Access Control List (WACL), and a Create Access Control List (CACL) can be defined containing the users who can respectively read, write, and create instances of the object. Formally:

$$\mathbf{RACL(o)} = \{u \mid \exists \langle u, \text{read}, o\rangle \in \text{AB}\}$$

$$\mathbf{WACL(o)} = \{u \mid \exists \langle u, \text{write}, o\rangle \in \text{AB}\}$$

$$\mathbf{CACL(o)} = \{u \mid \exists \langle u, \text{create}, o\rangle \in \text{AB}\}.$$

Every time the execution of a built-in method is required in a transaction, access control is performed to determine whether the user who started the transaction has the necessary authorization for the access. If the user has the proper authorization, access is allowed, otherwise it is rejected. In the following we consider this control to be always applied and we will not discuss it in more details. Therefore, we consider the following property to hold.

Property 1 (Discretionary property) Read, write, and create operations are executed in a transaction only if the user who started the transaction has the authorization for them. Formally:

$$\begin{aligned} \forall T: \quad & r_i \in T \Rightarrow u \in \text{RACL}(o(r_i)) \\ & w_i \in T \Rightarrow u \in \text{WACL}(o(w_i)) \\ & c_i \in T \Rightarrow u \in \text{CACL}(o(c_i)) \end{aligned}$$

where $u = \text{owner}(T)$.

[5] The create access mode can be applied only to class objects.

[6] For our purposes, authorizations may be considered as being represented by means of access control lists associated with each object stating the users authorized for accessing the object and the access mode they are allowed to execute.

6.1 Weakness of the discretionary control

The application of discretionary control guarantees that access to objects is executed only by users authorized for that. In particular, a request by a subject to access in a given mode (i.e., read, write, or create) an object will be allowed if and only if the subject has the authorization for the access mode on the object. However, the discretionary access control does not guarantee the satisfaction of the protection requirements as stated through the authorizations. In particular, in such a model, it is possible for a user who does not have any authorization on an object to read the information in the object without violating the restrictions imposed by the discretionary control. The problem is that no control is enforced by the discretionary model on the flow of the information in the system. Hence, once a user has read some information, no restriction is imposed on the usage of the information by the user. If on one hand, this may correspond to what one would like to see in the discretionary control, where the only condition on the access is to be authorized, on the other hand, this makes the control prone to be easily bypassed and hence unreliable.

To better understand this problem, consider the following example. Consider users x and y and two objects o_1 and o_2. Suppose that user x has the read authorization on o_1 and the write authorization on o_2; whereas user y has the read and write authorization on o_2. Object o_1 has a method m_1 whose task is to read some information in the object and return it to the invoker. Suppose that a Trojan Horse is embedded in m_1's code. The Trojan Horse consists in sending a message g to object o_2 passing it the information read. When the message is received by object o_2, method m_2 is executed causing a write operation on o_2.

Suppose now that user x send a message to o_1 invoking the execution of method m_1. The read operation on object o_1 is allowed since x has the necessary authorization. Then, a message is sent to o_2 and a write operation is required. Again, since x has the write authorization on o_2 the operation is allowed. Then, the reply is sent back to x. The sending of the message to o_2, i.e., the flow of information from object o_1 to object o_2, is caused by the Trojan Horse and completely hidden to x. Then, despite x's willing and despite the discretionary control, information read in o_1, and hence not readable from y, has been written in o_2, and hence made readable to y.

The flow of information between the two objects has not even been tracked by discretionary control, which considers each operation as singularly taken and not in the context in which it is executed. In particular, every operation is considered legitimate as far as the owner of the transaction has the authorization for it.

This simple example, shows how easily discretionary control can be bypassed and hence that no assurance is offered on the satisfaction of the protection requirements by the only application of discretionary control. This justifies the need for the application of further controls complementing the discretionary authorization model by restricting the flow of information in the system. These further control should provide assurance

that if some information is not accessible by some users, these users will not indirectly be able to get it.

7 Information flow

Objects exchange information by means of messages. If the execution invoked by a message is to be performed synchronously, i.e., the sender blocks waiting for the execution invoked to terminate, the message can enforce *bidirectional* information transmission. The *forward* transmission is carried through the list of parameters contained in the message, the *backward* transmission is carried through the reply. By contrast, if the execution invoked by a message is to be executed asynchronously, i.e, a *nil* reply is immediately sent to the sender (which therefore proceeds its execution regardless of the status of the execution invoked) the message can enforce only the *forward* transmission.

Not every time a message is exchanged between two objects there is a flow of information between them. For example, an object can acquire information only by changing its internal state, i.e., by writing any of its attributes. Thus, if no such changes occur in an object, no information flow to the object is actually enacted.

In particular, we make the following assumptions:

- There can exist information flow *from* an object only if information is *read* from the object.
- There can exist information flow *to* an object only if information is *written* into the object.

Note that although these assumptions may seem trivial, they are not. For example, in [10] different assumptions are made. In particular, to have an information flow from an object it is sufficient that the object sends (or replies to) some message, regardless of whether information has been read in the object. Our approach has its justification in the fact that the information we do not want to flow is the information regarding the status of the objects, i.e., the values of its attributes which can be known only after a read operation on the object.

It may be argued that not considering the sending of the message or the reply themselves as a "source" of information may allow information embedded in the methods to freely flow in the system. Then, if information on the object's attributes has been "hidden" inside a method code, an illegal information flow can be enacted in spite of the controls. To ensure that no information about the value of the object's attributes can be hidden inside the method's specifications, we do not allow method codes to be changed during normal execution.

To represent the concepts above, we distinguish between *transmission* of information between objects, meaning a direct communication between the objects, i.e, through a message or its reply, and *flow* of information between objects enacted by a read operation on the "source" object and the subsequent write operation on the "destination" object. As we have already discussed, the transmission of information, i.e., a communication, between two objects does not imply a flow of information between the two objects. Also the reverse may not always be true, i.e, a flow of information between two objects does not necessary imply a communication between the two objects. If the flow of information is enforced by means of a transmission of information between the two objects the flow is said to be *direct*, otherwise the flow is *indirect*. In other words, there is a direct information flow when an object reads information stored in its internal attributes and sends it to another object which writes it in any of its attributes. There is an indirect information flow when the information is passed from an object to another through the mediation of one or more other objects.

Note that for an information flow to be enacted, it is not necessary that the information written be the same as the information read. In particular, there exists information flow also when the information written is derived by executing some computation over the information read, i.e., some transformation has been applied to the information read.

The following definition characterizes the information flows in a transaction.

Definition 6 (Information flow) There exists a flow from object o_i to object o_j in a transaction T iff the transaction reads information from o_i and, at a later time, writes information in o_j. Formally, there exists a flow from object o_i and object o_j, iff there exist method executions $t_i, t_j \in T, t_i <_e t_j, o(t_i) = o_i, o(t_j) = o_j, m(t_i) = read, m(t_j) = write$ or $create$.

Note that flows in Definition 6 are potential flows. Indeed, the information written in o_j may not depend on the information read in o_i. Determining whether a potential flow is an actual flow would require analyzing how the information written has been produced [5]. This analysis is outside the scope of this paper. We therefore take the pessimistic hypothesis that all potential information flows are actual flows.

For an information flow to respect the protection requirements as stated by means of the authorizations, the object in which the information is written must be protected in reading at least as the object from which the information has been read. An information flow which satisfies this condition is said to be *safe*. This is formalized by the following definition.

Definition 7 (Safe information flow) An information flow from object o_i to object o_j is *safe* iff $\mathrm{RACL}(o_j) \subseteq \mathrm{RACL}(o_i)$.

A transaction respects the protection requirements if and only if it does not enact any unsafe flow. This is formalized by the following definition.

Definition 8 (Safe transaction) A transaction T is *safe* iff all information flows in it are safe. Formally, T is safe iff:

$\forall r_i, w_j, c_z \in T$:

$r_i <_e w_j \Rightarrow \mathrm{RACL}(o(w_j)) \subseteq \mathrm{RACL}(o(r_i))$

$r_i <_e c_z \Rightarrow \mathrm{RACL}(o(c_z)) \subseteq \mathrm{RACL}(o(r_i))$.

Definition 8 states that a transaction is safe if, during its execution, no information is leaked in objects readable from users not authorized for it.

8 The message filter

We propose the use of a *message filter* to complement the discretionary access control in an object oriented-system in order to ensure the verification of the protection policies. The concept of message filter was first introduced in [10] for the application of the mandatory policy in object-oriented systems. There, a filter intercepts every message exchanged by the objects in the system and, based on the security levels of the sender and of the receiver, as well as some auxiliary information, decides how to handle the message.

Similarly, we propose the use of a message filter which intercepts every message exchanged between the objects in a transaction to guarantee that no unsafe flow takes place.

An important requirement which must be taken into account in determining the control to be applied is that the control should impact as little as possible on the availability of the system to the users. For example, a naive filtering policy could be that only objects with same ACLs can exchange messages between each other. This requirement would however be too restrictive and a system enforcing it would be of little use to the users. Hence, different ways of ensuring the satisfaction of the protection requirements, i.e., flow safety, may have different effects on the system usage by the users. Before introducing our message filter, we discuss some approaches which can be used to ensure transaction safety.

8.1 Possible approaches

As we have illustrated in Section 7, information flow from an object to another object is performed through different steps. In particular, to have information flow there must be:

1. a *read* operation on the "source" object
2. a *chain of communications* of information connecting the "source" object to the "destination" object (information transmission)
3. a *write* operation on the "destination" object (information acquisition).

Where, if the chain of communications consists of one transmission only (i.e., there is direct communication between the source and the destination), the flow is direct.

The strict need to know policy imposes that no flow be completed if the destination object is not at least as protected as the source object. This requirement can be interpreted, and therefore enforced, in different ways. In particular, it can be enforced by restricting the execution of any of the steps above, i.e., by restricting:

- the read operations
- the transmission of information (i.e., the communication between objects) or
- the write operations (i.e., the acquisition of the information).

The restrictions on the communications can be enforced on any of the transmissions connecting the two objects.

All above solutions ensure that no unsafe flow takes place. However, they have different effects on the transaction executions. Indeed, an approach like "read everything, limit write" has a different impact on the transaction execution than an approach like "limit read, write everything", although both of them are aimed to ensure that no unsafe flow takes place.

For instance, consider objects o_1, o_2, and o_3, and users x and y. Suppose that x is authorized to read and write all the objects, whereas y is only authorized to read object o_2, that is $x \in$ RACL(o_1), RACL(o_2), RACL(o_3), WACL(o_1), WACL(o_2), WACL(o_3), and $y \in$ RACL(o_2), Suppose now that user x sends a message (g_1) to object o_1. Upon reception of g_1 by o_1, method m_1 is executed, consisting of sending a message (g_2) to o_2 and of a subsequent write operation on o_1. Upon reception of g_2 by o_2, method m_2 is invoked consisting of sending a message (g_3) to o_3, of the execution of some computation, and of a subsequent write operation on o_2. Upon reception of g_3 by o_3, method m_3 is executed consisting of a read and a subsequent write operation on o_3. The transaction, whose method invocation tree is illustrated in Figure 2, has an unsafe flow due to the execution of the write operation on o_2 after the read operation on o_3, i.e., after o_2 received the reply to g_3 from o_3. The unsafe flow can be forbidden by either: (i) forbidding the read operation on o_3, (ii) blocking the reply of g_3 from o_3 to o_2, or (iii) forbidding the write operation on o_2. Any of these solutions satisfies the requirement of blocking the unsafe flow, however, they all have a different effect on the transaction execution. In particular, blocking the read operation on o_3 may cause the subsequent write operation on o_3 to be uncorrect. Blocking the reply to be returned to o_2 may cause the reply to be returned by o_2 to o_1 to be uncorrect (since it does not take into account the information read in o_3). Finally, blocking the write operation on o_2 has the effect of not producing an up-to-date copy of o_2 at the end of the transaction.

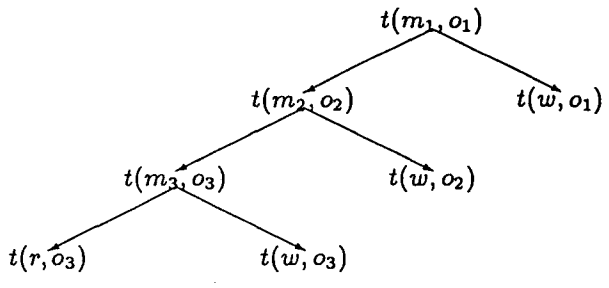

Figure 2: Example of method invocation tree

In the object-oriented model, read operations on an object can be invoked only by the object itself. Then, restricting read operations, which are necessarily required by the object inside another execution on it, may be a too restrictive approach. Indeed, the object should be able to access its own attribute and the restriction should instead be enforced at the moment the information is leaked outside the object. In the following section, we consider in more details the restriction on information transmission and acquisition.

8.2 Control policies

There are two different times at which the message filter can act to block possible unsafe flows to an object.

- When the information is to be sent to the object, by not allowing information to pass if the object is not be able to store it, i.e., blocking the message or its reply.
- When the information is to be stored in the object, therefore allowing information to be freely exchanged imposing, however, constraints on its acquisition, i.e., blocking write and create operations.

The approach of always blocking the transmission of information to an object if the object cannot store it may not always be the correct solution. Let us examine first the case of unsafe forward transmission, which arises when an object sends a message to an object less protected in reading, provided that more protected objects have been read. Blocking the messages sent from more protected objects to objects less protected is a very strong limitation. Indeed, an object can acquire information from an object less protected, and the only way it can do so is by sending the less protected object a message requiring information and obtain the information through the reply. Blocking the message would therefore imply not allowing the backward transmission of the message, transmission that is completely legitimate and desirable. For this reason, the most appropriate solution seems, in this case, to allow the message to pass and ensure that no unsafe flow takes place by blocking possible write operations the less protected object may require.

Consider now the case of unsafe backward transmission, arising from having an object replying to a message sent to it by a less protected object. Obviously, the forward transmission the message carries is safe and then there is no need to block the message. A possible solution to avoid the unsafe backward transmission to take place is to always return a nil reply to the low level object and proceeding with an asynchronous execution. This is the solution that has been proposed in [10]. The motivation of such an approach relies on the fact that the low level object would not be able anyway to store the information in any of its attributes and therefore there is no need to pass it such information. Moreover, blocking the reply of the message does not have here, as it had in the case of forward transmission, the drawback of blocking further legitimate flow between the two objects.

However some cases can be found where blocking the information to be passed back to low level objects may not be the right solution. In particular, it might be that a low level object needs some high level information in order to produce a reply to return to some high object, i.e., the low object is providing a service. Then, not returning the information to the low object would compromise the success of the transaction. Therefore, it seems more appropriate, in this case, to let the information (reply) pass and ensure that no unsafe flow takes place by blocking possible write operations on the low level object.

Notice that this solution also has some drawbacks. Indeed, passing an object information read from objects more protected than the object itself will have the effect of not allowing any subsequent write operation on the object, to guarantee no unsafe flow takes place, therefore possibly compromising the success of the transaction.

Hence, which of the two approaches has to be preferred over the other depends on the specific situation. For this reason, we do not restrict our model to the application of any of them in particular, but allow the control to be executed either on the information transmission or on its acquisition according to the specific invocation to be controlled, i.e., to the specific message intercepted by the filter. This gives our control flexibility, therefore overcoming the drawbacks that any of the approaches can have in specific situations.

In particular, we allow executions to be invoked either as *restricted* or *unrestricted.* If an execution is invoked as restricted, then no reply will be returned if the invoking object would not be able to store it, i.e, if some information has been read in a more protected object. By contrast, if an execution is invoked as unrestricted no constraint is imposed on the reply, and possible unsafe flows will be blocked at the time of the write operations. The restricted execution corresponds to applying the application of the strict need to know policy on the reply to be returned, i.e., on the transmission of information; whereas the unrestricted execution does not impose any constraint on the information transmission, therefore requiring to block its possible acquisition.

The specification of whether an execution must run as restricted or unrestricted is made by the sender at the invocation time. Indeed it is the object sending the message which knows what the information will be used for, i.e., for example, if more protected information can be returned since no write operation will be executed. We note however, that different approaches can be taken, for example, the receiver of a message could impose the invoked execution to run under restricted mode if no information transmission to less protected object is wished. However, for sake of simplicity we consider the specification to be made by the execution invoker.

As we have already discussed in Section 4, executions can run asynchronously. If an execution is invoked to be run asynchronously then a *nil* reply is immediately returned to the invoker which then proceeds independently from the execution invoked. Also the specification of whether an execution must be performed synchronously, i.e., the sender waits for it to complete, or

asynchronously, is made by the sender upon the execution invocation. It is then task of the message filter to provide for asynchronous execution, i.e., to intercept the message requiring asynchronous invocation, return a nil reply to the invoker and discard the actual reply produced by the execution.

Summarizing, executions can be asynchronous or synchronous. Moreover, synchronous executions executions can be restricted or unrestricted. The specification of whether an execution must be asynchronous, restricted or unrestricted is made by the invoker object when sending the message. To formalize this we extend the definition of message. A message is now defined as a 4-ple $g = (h, p, r, c)$ where h, p, and r have the meaning illustrated in Definition 2, and c denotes the mode under which the execution to be invoked upon reception of the message must be performed. Element c can have value:

NULL No restriction is applied on the invoked execution

RST The invoked execution runs in restricted mode, i.e., its reply will be filtered

ASYN The invoked execution has to be performed asynchronously.

8.3 The message filtering algorithm

We now present the message filtering algorithm, i.e., the controls and the actions executed by the message filter upon interception of a message.

The message filtering algorithm, illustrated in Figure 3, works as follows.

Consider a message g sent by object o_i to object o_j. Let t_i be the method execution in o_i that sent the message to o_j, and t_j the execution to be invoked on o_j upon reception of message g by o_j.

The two major cases correspond to whether g is a primitive message.

The first case deals with primitive messages, i.e., read, write, or create. If the read method is invoked, then no constraint is enforced by the filter. If the write method is invoked, the message filter allows its execution only if this does not result in any unsafe flow. Then, the write operation is allowed if and only if any read operation preceding it in the transaction either has been executed on an object less protected than o_j or has been called inside a restricted execution invoked by an object less protected than o_j. If the create method is invoked, no constraint is enforced by the message filter on the execution. The transaction owner will be given access privileges on the new object.[7]

The second case deals with non-primitive messages. We can distinguish three different sub-cases, according to the different conditions which can be put on the execution invoked. If no condition is required on the execution (c = NULL), then no control is enforced by the message filter. By contrast, if the execution is invoked to be performed under restricted mode (c = RST), the reply generated by the execution must be filtered. In particular, if the execution has not invoked any read operation on an object more protected than o_j or if so, the read operation was executed inside a restricted execution invoked by an object less or equally protected in reading than o_j, then the actual reply is returned; otherwise a nil reply is returned. If the execution has to be run asynchronously (c = ASYN), the message filter immediately returns a nil reply to the sender then discarding the eventual actual reply produced by the execution.

[7] We will elaborate on this in Section 8.4.

8.4 Administration of authorizations

Whether a flow of information between two objects is safe depends on authorizations the users have on the two objects. As a consequence, the authorizations on the objects determine the decision of the message filter on how to handle the message. Hence, the correctness of the message filter's controls strongly depends on the correctness of the specified authorizations: if the authorizations are not correct, then also the message filter decisions may be not correct. An incorrect decision of the message filter may cause safe flows to be blocked or unsafe flows to be allowed.

To ensure the correctness of our control, we impose some restrictions on the administrations (i.e., granting and revoking) of authorizations.

The first restriction is that authorizations may be granted and revoked only outside the execution of normal transactions. To understand the importance of this requirement, suppose that authorizations can be changed (i.e., granted or revoked) during normal execution. Then, a malicious user could embed in a method a Trojan Horse that, when executed by another user u, gives the malicious user access authorization to the objects of u. Since u is authorized to grant someone else access to his own objects these grants would be considered as legitimate. However, their execution was hidden and not wished by u. This simple example shows the importance of considering authorization administration as separate from the normal use of the system.

The second restriction is that the authorizations on an object cannot change when the object is being accessed by some transaction. This restriction ensures the consistency of the ACL of an object at any point in time during the transaction execution. Therefore, it avoids that an object which has been considered as not readable by some users is then made readable for them, or vice versa, thus compromising the correctness of the control. The requirement that the authorizations specified for an object cannot change during normal execution, is similar to what in the mandatory policy is known as tranquility principle which states that the security level of an active object cannot be changed. Similarly, we require the protection state (i.e., the set of authorizations associated with the object) not to be changed when the object is being accessed by a transaction.

Note that, for our purposes, i.e., avoiding information to be disclosed to users not authorized to read it, it would have been sufficient to prevent the read set of an object to be changed during normal execution.

Message Filtering Algorithm

% *Let* $g = (h, (p_1, \ldots, p_k), r, c)$ *be the message sent.*
% *Let* t_j *be the execution to be invoked on* o_j.
if $h \in$ {READ, WRITE, CREATE}
  % *g is a primitive message*
  then case
    (1) g = (READ, $(a_t), r$) : % *allow unconditionally*
      $r \leftarrow$ value of a_t
      return r **to** t_i
    (2) g = (WRITE, $(a_t, v_t), r$) :
      % *allow if it does not enforce any unsafe flow*
      if $\forall r_x <_e w_j$: RACL($o(r_x)$) $\supseteq$ RACL($o(r_j)$)
        $\vee$ $\exists t_k, t_k <_i w_j, t_k \twoheadrightarrow_s r_x$,
        $t_h \rightarrow_s t_k, t_k \in$ RST,
        RACL($o(t_h)$) $\supseteq$ RACL(o_j)
        then [$a_t \leftarrow v_t$; $r \leftarrow$ *success*]
        else $r \leftarrow$ *failure*
      return r **to** t_i
    (3) g = (CREATE, $(v_1, \ldots, v_k), r$) :
      % *create object. Give privileges to*
      [CREATE *i* **with values** $v_1, \ldots, v_k$ and
      RACL(i)= WACL(i)= CACL(i) = u]
      return r **to** t_i
  end case

  else case % *i.e., g is a non-primitive message*
    (4) c = NULL:% *let g pass, return actual reply*
      invoke t_j
      $r \leftarrow$ reply from t_j
      return r to t_i
    (5) c = RST: % *let g pass, filter reply*
      invoke t_j
      if ($\forall r_x, t_j \twoheadrightarrow_s r_x$: RACL($o(r_x)$) $\supseteq$ RACL(o_i)
        $\vee$ $\exists t_h, t_k, t_j \twoheadrightarrow_s t_k \twoheadrightarrow_s r_x$,
        $t_h \rightarrow_s t_k, t_k \in$ RST,
        RACL($o(t_h)$) $\supseteq$ RACL(o_i)
        then $r \leftarrow$ reply from t_j
        else $r \leftarrow$ NIL
      return r **to** t_i
    (6) c = ASYN:
        % *let g pass, inject NIL reply,*
        % *ignore actual reply*
      $r \leftarrow$ NIL
      return r **to** t_i
      invoke t_j
      discard reply from t_j
  end case
endif

Figure 3: Message filtering algorithm

However, we believe that it is a good principle to consider administrative operations as separate from the normal activity.

A further issue about authorization administration is the assignment of authorizations on objects created inside a transaction. Since a create operation, may be seen as a write operation on the created object, for the transaction safety to be respected it would be sufficient to require that a user can be authorized to read the new object only if he has the read authorization for all the objects read by the transaction prior to the create operation. However, according to our approach of keeping authorization administration outside the normal transaction execution, we consider that upon creation of an object, only the user who started the transaction is given the authorizations to access the object. Authorizations on the created object can be granted by the object's owner to other users once the transaction has completed.

9 Conclusions and future research

Discretionary access control alone does not provide any assurance on the satisfaction of the protection requirements stated through the authorizations. Then, the need of complementing access control by providing a mean for limiting the indiscriminate flow of information in the system. In this paper we have proposed the use of a strict need to know policy for overcoming the vulnerability of discretionary control policies in object-oriented systems. We have stressed how characteristics of object-oriented systems make information flow easily representable and therefore controllable. The strict need to know policy is enforced by a message filter intercepting every message exchanged between objects to ensure that no information is leaked to objects accessible from users not allowed for it. The model allows for different options in the application of the policy therefore making the policy more flexible and adaptable to the specific situations.

The work presented in this paper can be extended in many respects. In particular, the model can be extended to support "exceptions" to the strict need to know policy. Indeed, there may be the need for releasing sensitive information to users not allowed to directly access it. However, this release of information must be strictly controlled. Exceptions can for instance be allowed after querying human users about it or only inside certified trusted software. In particular specific trusted methods could be allowed to execute without obeying the strict need to know policy. Other extensions may concern the underlying authorization model. In particular, a more richer authorization model can help in exploiting object-oriented characteristic for providing access restriction without compromising availability. Methods could be considered as objects of the authorizations as done in [1, 2, 14]. Moreover methods could be considered as subjects of the authorizations. The interpretation of the strict need to know policy to such an extended model would result more flexible and help in overcoming the rigidity proper of such control.

References

[1] AHAD, R. ET AL., "Supporting access control in an object-oriented database language," *Proc. Third International Conference on Extending Database Technology (EDBT)*, Vienna (Austria), Springer-Verlag Lecture Notes in Computer Science, Vol. 580, 1992.

[2] BERTINO, E., "Data hiding and security in an object-oriented database system," *Proc. Eighth IEEE International Conference on Data Engineering*, Phoenix (Ariz.), February 1992.

[3] BERTINO, E. AND WEIGAND, H., "An approach to authorization modeling in object-oriented database systems," To appear in *Data and Knowledge Engineering*.

[4] BOEBERT, W.E. AND FERGUSON, C.T., "A partial solution to the discretionary Trojan Horse problem," *Proc. of the* 8^{th} *Nat. Computer Security Conf.*, Gaithersburg, MD, October 1985

[5] DENNING, D.E., "A lattice model of secure information flow," in *Communication of the ACM*, Vol. 19, No. 5, May 1976

[6] DITTRICH, K., HARTIG, M., AND PFEFFERLE, H., "Discretionary access control in structurally object-oriented database systems," in *Database Security, II: Status and Prospects*, C.E. Landwehr, ed., North-Holland, Amsterdam, 1989.

[7] FERNANDEZ, E. B., GUDES, E., AND SONG, H., "A security model for object-oriented databases," *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 1989.

[8] GRAUBART, R., "On the need for a third form of access control," *Proc. of the* 12^{th} *Nat. Computer Security Conf.*, Gaithersburg, October 1989

[9] GRIFFITHS, P.G., AND WADE, B., "An authorization mechanism for a relational database system," *ACM TODS*, Vol. 1, No. 3, September 1976

[10] JAJODIA, S. AND KOGAN B., "Integrating an object-oriented data model with multilevel security," *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 1990

[11] JAJODIA, S., KOGAN, B., AND SANDHU, R., "A multilevel-secure object-oriented data model," Technical Report, George Mason University, 1992.

[12] KARGER, P.A., "Limiting the damage potential of discretionary Trojan Horses,", *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 1987

[13] MCCOLLUM, C.J., MESSING, J.R. AND NOTARGIACOMO L., "Beyond the pale of MAC and DAC - Defining new forms of access control," *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 1990

[14] RICHARDSON, J., SCHWARZ, P., AND CABRERA, L.F., "CACL: efficient fine-grained protection for objects," *Proc. OOPSLA'92*, Vancouver, B.C. Canada, October 1992

[15] SANDHU, R., THOMAS, R., AND JAJODIA, S., "Supporting timing channel free computations in multilevel secure object-oriented databases," in *Database Security V: Status and Prospects*, C.E. Landwehr and S. Jajodia, eds., North-Holland, 1992.

[16] STOUGHTON, A., "Access flow: a protection model which integrates access control and information flow," *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 1981

[17] THOMAS, R., SANDHU, R., AND JAJODIA, S., "Implementing the message filter object-oriented security model without trusted subjects," in *Database Security V: Status and Prospects*, C.E. Landwehr and S. Jajodia, eds., North-Holland, 1992.

[18] WALTER, K.G., OGDEN, W.F., ROUNDS, W.C., BRADSHAW, F.T., AMES, S.R., AND SUMAWAY, D.G., "Primitive models for computer security," Tech. Report ESD-TR-4-117, Case Western Reserve University, Cleveland, OH, January 1974