Information Flow Control in Object-Oriented Systems

Pierangela Samarati, Member, IEEE, Elisa Bertino, Member, IEEE, Alessandro Ciampichetti, and Sushil Jajodia, Senior Member, IEEE

Abstract—In this paper, we describe a high assurance discretionary access control model for object-oriented systems. The model not only ensures protection against Trojan horses leaking information, but provides the flexibility of discretionary access control at the same time. The basic idea of our approach is to check all information flows among objects in the system in order to block possible illegal flows. An illegal flow arises when information is transmitted from one object to another object in violation of the security policy. The interaction modes among objects are taken into account in determining illegal flows. We consider three different interaction modes that are standard interaction modes found in the open distributed processing models. The paper presents formal definitions and proof of correctness of our flow control algorithm.

Index Terms—Object-oriented databases and systems, security, discretionary access control, mandatory access control, Trojan horse, distributed processing.

1 INTRODUCTION

An important requirement of any multiuser data management system is to provide facilities for defining and enforcing access control policies [4]. Commercial multiuser database management systems (DBMSs), including relational DBMSs and recent object-oriented DBMSs (OODBMSs), provide access control mechanisms supporting the definition and enforcement of authorizations. An authorization specifies that a subject, an active entity of the system, is authorized to access an object, a passive entity of the system containing information, under a given access mode, e.g., read, write, or execute. In general, access control mechanisms provided by commercial DBMSs are discretionary; i.e., the grant of authorizations on an object is at the discretion of the object administrator. The main drawback of discretionary access control is that it does not provide a real assurance on the satisfaction of the protection requirements.

Although each access is controlled and allowed only if authorized, access restrictions can be easily bypassed. Indeed, discretionary policies do not impose any restriction on the usage of information by a subject once the subject has obtained it; that is, further dissemination of information is not controlled. As an example, a subject that is able to read some data can pass them without the cognizance of the data administrator to other subjects even though they are not authorized to read them. This weakness makes discretionary policies vulnerable to malicious attacks from Trojan horses embedded in programs. A Trojan horse is a computer program with an apparent or actual useful function, that contains additional hidden functions that surreptitiously exploit the legitimate authorizations of the invoking process.

To understand how a Trojan horse can leak information to unauthorized users, despite the discretionary access control, consider the following example. Suppose user Vic (the victim) creates a file mydata and writes personal data in it. Consider now user Burt (the bad guy), a subordinate of Vic, who wants to read the personal data of Vic even though he does not have the authorization. To achieve this, Burt creates another file, stolen_data, and gives Vic the authorization to write this file (Vic is not informed about this). Moreover, Burt modifies the code of an application generally used by Vic to include two hidden operations: a read operation on mydata and a write operation on stolen_data. Then, he gives the application to Vic. Suppose now that Vic executes the application. Since the application executes on behalf of Vic, every access is checked against the authorizations of Vic, and both the read and write operations are allowed. As a result, during execution, the personal information in mydata is copied to stolen_data, and can then be read by Burt. This simple example illustrates how easily the restrictions stated by the discretionary authorizations can be bypassed and, therefore, the lack of assurance with respect to the authorizations imposed by discretionary policies. For this reason discretionary policies are considered unsafe and not satisfactory for environments with stringent protection requirements.

Mandatory access control policies provide a way to protect data against illegal accesses such as those perpetrated through Trojan horses. These policies are mandatory in that the accesses to be allowed are determined on the basis of
predetermined precise rules. An example of mandatory policy is multilevel security, where each object and each subject in the system is assigned an access class consisting of two components: a security level and a set of categories. The security level is an element of a totally ordered set. In a military system, the levels generally considered are: Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U), where TS > S > C > U. The set of categories is a subset of an unordered set of elements (e.g., NATO, Nuclear, Army, etc.). Access classes are partially ordered as follows: An access class \( c_1 \) dominates (\( \geq \)) an access class \( c_2 \) iff the security level of \( c_1 \) is greater than or equal to that of \( c_2 \) and the categories of \( c_1 \) include those of \( c_2 \). Two classes \( c_1 \) and \( c_2 \) are said to be incomparable if neither \( c_1 \geq c_2 \) nor \( c_2 \geq c_1 \) holds. The security level of the access class of an object reflects the sensitivity of the information contained in the object, i.e., the potential damage that could result from unauthorized disclosure of the information stored in the object. The security level of the access class of a user, also called clearance, reflects the user’s trustworthiness not to disclose sensitive information to users not cleared to see it. Categories are used to provide finer grained security classifications of subjects and objects than classifications provided by security levels alone, and are the basis for enforcing need-to-know restrictions.

Access control in multilevel protection systems is based on the “no read-up” and “no write-down” principles formulated by Bell and LaPadula [1]. Satisfaction of these principles prevents information stored in high level objects from flow to objects at a lower or incomparable level. A generalization of the mandatory access control policies is represented by the information flow model of Denning [6].

The main drawback of mandatory policies is their rigidity, which makes them unsuitable for many application environments. In particular, in most environments there is a need for a form of access control to designate specific users who are allowed or who are forbidden access to an object with respect to specific privileges.

It thus appears that there is a need for access control mechanisms able to provide the flexibility of discretionary access control and, at the same time, the high assurance of mandatory access control. The development of such a high assurance discretionary access control mechanism poses several difficult challenges. Because of this difficulty, limited research effort has been devoted to this problem and no satisfactory solutions have been proposed so far. In this paper, we describe an approach that takes a first step towards the definition of such an access control mechanism. The research is in the framework of object-oriented systems. In such systems, all information transmissions are through message exchanges. An object interacts with another object by sending a message. Upon receipt of the message, the receiver object executes an appropriate program among those defined for it (called its methods) and sends back a reply to the sender object. As a result, information flow can be controlled by mediating the messages exchanged between objects.

### 1.1 Overview of Our Approach

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 handles them according to the security policy. The message filter may let the information be transmitted unaltered, block it, or take some other action, such as restricting the execution of the invoked methods. The message filter enforces the need-to-know policy by ensuring that information read from an object does not flow into objects accessible to users not authorized to access the object from which the information has been read. The modularity of object-oriented systems, where operations on objects are via methods, makes it possible to isolate specific operations and provide flexibility in the application of this policy. In particular the need-to-know policy control can be executed either at each single elementary read or write operation, by not allowing an operation to be executed if this may cause unsafe information flow, or at the level of the information produced by a method, by restricting the transmission of messages (and their replies) between objects.

To allow for flexibility in the application of the policy, we consider method executions that can run under different modes. Each mode has a different effect on the control applied by the message filter. In particular, our model provides for synchronous as well as for asynchronous executions since a strict sequential relationship between operations may unnecessarily forbid the execution of write operations if certain read operations have been executed before them. The advantage of considering asynchronous executions, in addition to synchronous ones, is that they permit independence between operations to be expressed, thereby allowing additional computations that do not generate illegal information flows. Moreover, we consider method executions that can be restricted or unrestricted. If a method is executed as restricted, the need-to-know policy is enforced on the reply produced by the method. In particular the reply is blocked if the object waiting for it is accessible to users not allowed for the information read in the execution. By contrast, if a method is executed as unrestricted, no constraint is enforced on its reply which is, therefore, returned unchanged to the invoker of the method. In this case, the satisfaction of the need-to-know policy is ensured by restricting possible write operations on the objects. Basically, write operations to an object are forbidden if this object has received 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.

### 1.2 Organization of the Paper

The rest of this paper is organized as follows. Section 2 discusses previous work in complementing discretionary access control to restrict the possibilities of violations. Section 3 illustrates the object-oriented discretionary data model to which the control is applied. Section 4 describes the reference discretionary authorization model which enforces the access control. Section 5 illustrates the execution of activities in object-oriented systems. Section 6 characterizes...
izes the flow of information in transaction executions. Section 7 illustrates our approach for controlling the flow of information and proves the correctness of our flow control algorithm. Section 8 discusses the administration of authorizations in our model. Section 9 addresses issues related to the implementation of our flow control algorithm. Finally, Section 10 presents some conclusions and outlines future work.

2 PREVIOUS WORK

Some research efforts have been aimed at extending the discretionary authorization model with forms of access constraints that cannot be enforced within the existing models [7], [13], [15].

McCollum et al. [13] propose a new form of access control which imposes restrictions on the flow of information in the system. Their model allows the owner of a file to retain control on the dissemination of the information contained in the file by associating with each data object an access control list. This list propagates through subject and object labels to all objects into which the content of o may flow.

A similar approach has been proposed by Stoughton [15]. In this proposal, each object has two protection attributes: the current access and the potential access. The current access attribute describes what can be done by whom to the information in the object; and the potential access attribute describes what can be done by whom to this information, even if it is copied to some other object. Hence, the potential access attribute provides a way of propagating access restrictions to the information once it has been released.

Other research efforts on complementing discretionary access control have been specifically aimed at eliminating, or at least limiting, its vulnerability to Trojan horses [2], [3], [10], [16].

Bingham [2], [17] proposes an approach that provides assurance by allowing only valid compilers, created with a privileged action, to create and modify machine-code. All resource requests, which are made as a result of compiler-generated descriptors, are mediated by a Master Control Program (MCP) and allowed only if authorized.

Walter et al. [16] propose the application of a strict need-to-know policy for limiting information flow during process execution in operating system environments. 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.

Boebert and Ferguson [3] propose to control Trojan horses by interposing a protected subsystem between the suspected program and the actual file system. All file references are forced to go through a dynamic linker which compares the name of the user who invoked the program, the name of the originator of the program, and the name of the owner of any data file accessed by the program. If a user invokes a program owned by someone else and this program contains a Trojan horse which attempts to tamper with the user’s files, the dynamic linker will recognize the name mismatch and raise the appropriate alarms.

Another approach, proposed by Karger [10], controls the effects of Trojan horses in discretionary systems by limiting the files accessible by the application programs on the basis of some knowledge on the program themselves. This approach requires that names patterns describing the objects to be accessed by each application program be specified. All accesses required by an application are mediated by a message checker. The message checker compares the name of the object to be accessed against the pattern specified for the application. If the object’s name satisfies the pattern, the access is granted, otherwise the user running the program is queried about the access requested.

3 THE OBJECT-ORIENTED DATA MODEL

In this section, we characterize the object-oriented data model and introduce definitions and notations which will be used throughout the paper. We consider an object system, based on the message-passing paradigm, which includes the basic concepts of objects, attributes, methods, and messages. Note, however, that we do not consider features like inheritance, aggregation, or versions.

In object-oriented systems, each-real-world entity is represented by an object. Each object is associated with a unique object identifier (oid), which is fixed for the whole life of the object, a set of instance attributes (also called instance variables), and a set of procedures, called methods. The value of an attribute can be an object or a set of objects. The values of the attributes of an object represent the object state. The set of methods of an object represents the object behavior. The state of an object can be accessed (and possibly modified) by invoking some method on the object. Methods are invoked by sending the object the appropriate messages. Hence, an object communicates with the other objects in the system via messages and their replies. Objects which share the same sets of attributes and methods are grouped together into a class. Each object belongs to (is an instance of) a class.

To formalize the model we assume a finite set of domains $D_1, D_2, ..., D_n$. Let $D$ be the union of all the domains together with a special element, nil; i.e., $D = D_1 \cup D_2 \cup ... \cup D_n \cup \{nil\}$. 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, $N$ a set of finite strings called message names, 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 [8].

**DEFINITION 1 (Object).** An object $o$ is either a primitive object or a four-tuple $(i, a, \nu, \mu)$ such that $i \in I$, $a = (a_1, ..., a_n)$ with $a_j \in A$ for all $j = 1, ..., n$, $\nu = (\nu_1, ..., \nu_k)$ with $\nu_j \in V$ for all $j = 1, ..., k$, and $\mu \subseteq M$.

Definition 1 states that an object $o$ is characterized by its oid, which uniquely identifies $o$ 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 $o$. In the following, $i(o)$, $a(o)$, $\nu(o)$, and $\mu(o)$ denote, respectively, the oid, the sets of attributes, the set of values, and the set of methods associated with object $o$. 

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 26, 2009 at 16:03 from IEEE Xplore. Restrictions apply.
DEFINITION 2 (Message). A message \( g \) is a pair (name, pars) where name \( \in N \) is the message name and pars = (par\(_1\), ..., par\(_k\)), with par\(_j\) \( \in V \cup A \) for all \( j = 1, ..., k \), is an ordered set of values called message parameters.

When an object receives a message, the corresponding method is executed. The execution of the method can require the object to send a message to itself or to another object, to read or write any of its attributes, or to create a new object. A reply is eventually returned to the object which sent the message. The reply to a message is an ordered set of return values defined as reply = \( (rp\(_1\), ..., rp\(_n\)) \), with rp\(_j\) \( \in V \cup \{\text{failure, success}\} \) for all \( j = 1, ..., n \).

Similar to the notation used for objects, name\(_{(g)}\), pars\(_{(g)}\), and reply\(_{(g)}\) denote, respectively, the name, the set of parameters, and the reply of message \( g \).

Access to the attributes of an object and creation of new object instances are enforced by having the object send primitive messages to itself. These messages cause the execution of built-in methods providing the desired operations. Built-in methods are elementary in the sense that they can execute without invoking other methods. The primitive messages an object \( o \) can send to itself are as follows:

- A read message, denoted by \( g = (\text{READ}, (a)) \). If \( a \in o(a) \), the message returns the value of attribute \( a \). Otherwise, it returns failure.
- A write message, denoted by \( g = (\text{WRITE}, (a, v)) \). If \( a \in o(a) \), attribute \( a \) is assigned the value \( v \) and the message returns success. Otherwise, no effect is produced, and the message returns failure.
- A create message, denoted by \( g = (\text{CREATE}, (v_1, ..., v_n)) \). A new object is created which inherits all attributes and methods from object \( o \). The attributes of the new object are assigned the values contained as parameters in the message. If the creation succeeds, the message returns the oid of the new object; otherwise, it returns failure.

Each activity in the system is started by a message sent by a user to an object. We refer to the set of all method executions invoked (directly or indirectly) as a consequence of the receipt of a message \( m \) sent by a user as a transaction. The user who sends \( m \) is the initiator of the transaction.

Throughout the discussion we use \( e_i \) to denote the execution of the \( i \)th method invoked during a transaction. Given an execution \( e_i \), notations name\(_{(e)}\) and oid\(_{(e)}\) denote the method being executed and the object on which the method is executed, respectively. We will use \( r \), \( w \), and \( c \) instead of \( e \), to denote the execution of read, write, and create methods, respectively.

4. The Reference Authorization Model
In order to make our approach widely applicable, we consider a simple and general authorization model. The subjects, to whom authorizations are granted, 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: read, write, and create (the create access mode can be applied only to class objects). Every authorization is a triple \( (u, m, o) \), stating that user \( u \) can exercise access mode \( m \) on object \( o \). We refer to the set of authorizations holding at a given time as the authorization base (AB for short). With every object, we associate a read access control list (RACL), a write access control list (WACL), and a create access control list (CACL) containing the users who can, respectively, read, write, and create instances of the object. Formally:

\[
\begin{align*}
\text{RACL}(o) &= \{ u \in U \mid \exists (u, \text{read}, o) \in \text{AB} \} \cup \text{owner}(o) \\
\text{WACL}(o) &= \{ u \in U \mid \exists (u, \text{write}, o) \in \text{AB} \} \cup \text{owner}(o) \\
\text{CACL}(o) &= \{ u \in U \mid \exists (u, \text{create}, o) \in \text{AB} \} \cup \text{owner}(o),
\end{align*}
\]

where owner\(_{(o)}\) denotes the owner of \( o \) and \( U \) denotes the set of users in the system. The owner of an object is included in all the ACLs because the owner can always grant any authorization on the object to himself.

Every time the execution of a built-in method is required, the authorizations of the transaction’s initiator are checked. If the user has the authorization for the access, the access is granted; otherwise, the access is denied. In the following, we consider this control to be always applied and will not discuss it further; i.e., we assume that the following property holds:

PROPERTY 1 (Discretionary property). A read, write, or create operation on an object is executed in a transaction \( T \) only if the user who initiated \( T \) belongs to the corresponding ACL of the object.

5. Interaction Modes
We consider a general object-oriented system which supports the following basic interaction modes among objects:

Synchronous. The sender \( e_i \) of the message requiring the execution \( e_j \) stops executing and waits for the invoked execution to terminate and the reply to return. Upon reception of the reply, the sender resumes.

Asynchronous. The sender \( e_i \) of the message requiring the execution \( e_j \) immediately receives a null reply (“nil”) from the system and continues executing independently of \( e_j \). No actual reply to the message is sent to \( e_i \) by \( e_j \).

Deferred reply. Upon sending the message, the sender \( e_i \) receives a message identifier and continues its computation up to a point where it explicitly suspends to wait for the reply. The reply produced by the execution \( e_j \) invoked is returned upon explicit request of \( e_i \).

Furthermore, to allow objects to control the information they receive from other objects, we allow these basic interaction modes to be combined with a filtering option which can be unrestricted or restricted. If an execution is invoked as restricted, a null reply, instead of the real one, is returned to the invoker if the invoker should not be allowed to store the information contained in the reply. The “restricted” option obviously does not apply to asynchronous execution.

3. In the following, we use the term ACL to refer to RACL, WACL, and CACL indiscriminately.
4. The need for this option will be clarified in Section 7.
tions (which do not return any reply by definition). If an execution is invoked as unrestricted, no further action is taken on the reply, which is therefore returned according to the specified basic interaction mode. The combination of basic interaction modes and filtering options results in five different interaction modes with which executions can be invoked, as illustrated in Table 1.

If a message is sent requiring the execution of a method in a specified interaction mode \( m_d \), we say that the method is invoked and executed under mode \( m_d \). The invoker object specifies the interaction mode when sending the message. If nothing is specified, the synchronous-unrestricted case is assumed. Interaction modes can be changed by modifying the method code and recompiling it. Note that possible modifications to interaction modes may change the semantics of the execution. However, they do not affect the correctness of the message filter, which ensures that no unsafe flow takes place (possibly by not allowing write operations) regardless of the interaction modes used in the transaction. To take the interaction mode into consideration, we extend the definition of message as follows: A message is characterized by a triple \( g = (\text{name}, \text{pars}, \text{mode}) \) where \( \text{name} \) and \( \text{pars} \) have the meaning given in Definition 2, and \( \text{mode} \) denotes the mode under which the execution to be invoked upon receipt of the message must be performed. We require all built-in primitive methods to be executed under the synchronous-unrestricted mode.

We view the message requiring the reply of an execution previously invoked under deferred reply mode as a primitive message \( \text{get} \) denoted by \( g = (\text{GET}, \text{mid}, \text{mode}) \), with mode = synchronous-unrestricted. The message returns the reply produced by the execution invoked by the message whose identifier is \( \text{mid} \). An important requirement for deferred reply is that both the invoked message and the request for the reply must be invoked within the same method execution. Moreover, the message requiring the reply must be sent after the message requiring the corresponding deferred reply execution. In the following, we do not consider the execution invoked by a get message as an execution by itself, but rather as a completion of the execution requiring the deferred reply. Given an execution \( e \) is invoked in the deferred reply mode, we use the notation \( \hat{e} \) to denote the execution requiring the reply produced by \( e \).

In the remainder of the paper, we denote invocation of execution \( e_i \) by \( e_i \) by an arrow with a subscript and a superscript indicating the basic mode and the filtering option for the invocation, respectively. Thus, \( \rightarrow^s \), \( \rightarrow^r \), \( \rightarrow_a \), \( \rightarrow_d \) denote the synchronous-unrestricted, synchronous-restricted, asynchronous, deferred reply unrestricted, and deferred reply restricted invocation, respectively. We will omit the specification of either the basic interaction mode or the filtering option, and therefore the corresponding superscript/subscript, when its value is not of interest in the explanation. For instance, \( \rightarrow^s \) indicates either synchronous-unrestricted or synchronous-restricted invocation, \( \rightarrow \) denotes invocation of method executions under any mode.

The following definitions characterize the order of method invocations and executions inside a transaction.

DEFINITION 3 (Invocation order). Given two method executions \( e_i \) and \( e_j \) invoked by the same execution \( e_k \), we say that \( e_i \) is invoked before \( e_j \) written \( e_i <_{ij} e_j \), if the message requiring the invocation of \( e_i \) is sent before the message requiring the invocation of \( e_j \).

The relationship among method executions in a transaction can be represented graphically by means of a transaction execution tree defined as follows: The root of the tree is the method execution invoked when a message is received from the user. If execution \( e_i \) is invoked by execution \( e_k \), \( e_i \) is inserted as a child of \( e_k \) in the tree. To preserve the order between invocations, for any two executions \( e_h \) and \( e_r \), if \( e_h <_{ir} e_r \), \( e_h \) appears on the left of \( e_r \) in the tree. Because no message can be sent as part of the execution of any of the built-in methods, built-in method executions always appear as leaves of the tree. Each arc is labeled with the interaction mode with which the corresponding invocation has been issued. An example of a method invocation tree is illustrated in Fig. 1. For sake of simplicity no label is associated with arcs corresponding to the default synchronous-unrestricted mode.

Since executions can run under asynchronous and deferred reply modes, the invocation order does not necessarily coincide with the order with which the executions are completed. For instance, an asynchronous execution invoked before another execution cannot be considered as being completed before the latter. The following definition characterizes the execution order.

DEFINITION 4 (Execution order). Given two method executions \( e_i \) and \( e_j \) invoked by the same execution \( e_k \), such that \( e_i <_{ir} e_j \), we say that \( e_i \) precedes \( e_j \) written \( e_i <_{e} e_j \) if one of the following conditions holds:

1) \( e_k \rightarrow_{ir} e_i \) or
2) \( e_k \rightarrow_{ir} e_i \wedge \hat{e}_i <_{ir} e_j \).

Definition 4 states that if execution \( e_i \) invokes execution \( e_j \) and, subsequently, execution \( e_j \) then \( e_i \) precedes execution \( e_j \) iff either

1) \( e_i \) is invoked synchronously, or
2) \( e_i \) is invoked under deferred reply mode and the message requiring the reply of \( e_i \) is sent before invoking \( e_j \).

<table>
<thead>
<tr>
<th>Filtering option</th>
<th>Synchronous</th>
<th>Asynchronous</th>
<th>Deferred reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unrestricted</td>
<td>Synchronous-Unrestricted (SU)</td>
<td>Asynchronous</td>
<td>Def. reply-Unrestricted (DU)</td>
</tr>
<tr>
<td>Restricted</td>
<td>Synchronous-Restricted (SR)</td>
<td>N/A</td>
<td>Def. reply-Restricted (DR)</td>
</tr>
</tbody>
</table>
However, the transaction illustrated in Fig. 1. We have:

Each message between objects enforces \( e_i \) message and, therefore, executed concurrently with the other executions subsequently invoked by \( e_j \) and \( e_k \) (since \( e_4 \) is asynchronously invoked and \( e_5 \) is asynchronously invoked). However, \( e_4 \not<_{e} e_5 \) and \( e_5 \not<_{e} e_6 \) (since \( e_6 \) does not wait for \( e_5 \)'s reply before invoking \( e_6 \)).

![Fig. 1. A transaction execution tree.](image)

For example, consider the executions invoked by \( e_1 \) in the transaction illustrated in Fig. 1. We have: \( e_2 \preceq_{e} e_4 \preceq_{e} e_5 \preceq_{e} e_6 \preceq_{e} w_{o_j} \). Moreover, \( e_2 \preceq_{e} e_4 \preceq_{e} e_5 \preceq_{e} w_{o_i} \), and \( e_6 \not<_{e} w_{o_j} \).

6 INFORMATION FLOW

Each message between objects enforces bidirectional transmission of information. The forward transmission carries information from the sender to the receiver through the list of message parameters. The backward transmission carries information from the receiver to the sender through the reply. Messages invoking asynchronous executions and messages whose reply is blocked due to the enforcement of the restricted filtering option enforce unidirectional transmission of information, from the sender to the receiver.

We distinguish between transmission of information, meaning the communication of information from one object to another object (either through a message or its reply), and flow of information, meaning the acquisition of information by an object about the state of another object. Information flow between two objects does not necessarily require a direct transmission (i.e., message exchange) between the objects (indirect flow). Conversely, transmission of information between two objects does not necessarily imply flow of information between them. For example, an object can acquire information only by changing its internal state (i.e., by writing to some of its attributes). Thus, if no changes occur in an object, no information flow to the object is enacted. We make the following assumptions:

1) There can exist information flow from an object only if information is read from the object.
2) There can exist information flow to an object only if information is written into the object.

Although these assumptions may seem trivial, they are not. For instance, in [8] different assumptions are made. In particular, an information flow from an object may simply arise if the object sends (or replies to) some message, regardless of whether information in the object has been read. Our assumption is based on the fact that the information about the values of the object’s attributes cannot be known unless a read operation is executed on the object. To ensure that no information about the value of an object’s attributes can be hidden inside method specifications, we do not allow methods to change their own code or the code of other methods. Changes to method code must be requested directly by the users (trusted path).

Summarizing, information flow from an object \( o_i \) to another object \( o_j \) requires:

1) a read operation on the “source” object \( o_i \),
2) a chain of transmission of information connecting the “source” object to the “destination” object (information communication), and
3) a write operation on the “destination” object \( o_j \) (information acquisition).

Note that for an information flow to be enacted, it is not necessary that the information written be same as the information read. Information flow exists also when the information written is derived by executing some computation on the information read.

To define information flow between objects, we first characterize the transmission of the information, acquired through a read operation, by means of message exchanges. Information acquired or transmitted is represented in terms of the objects from which it has been read. Backward transmission depends on the mode under which the execution has been required. If a message is sent under asynchronous mode or under restricted mode and the reply is blocked because of the restriction, no information is exchanged between the two executions through the message reply. To determine whether the reply of a restricted execution can be returned to the invoker, we introduce the concept of read access control list (RACL) associated with an execution. Given an execution \( e \), its RACL, denoted with RACL(\( e \)), is the list of users who are authorized to read the information carried by \( e \)'s reply.

The following definitions characterize the transmission of information in the two possible directions and the RACL of executions.
DEFINITION 5 (Backward transmission). Let \( e_i \) and \( e_j \) be two method executions such that \( e_i \rightarrow e_j \). The information transmitted from \( e_i \) to \( e_j \) through the message reply, denoted as \( e_i \Delta e_j \), is recursively defined as follows:\n
\[
e_i \Delta e_j = \begin{cases} \{ o(e_j) \} & \text{if } \text{name}(e_j) = \text{read} \\ \emptyset & \text{if } \text{name}(e_j) \in \{ \text{write}, \text{create} \} \text{ OR } e_i \rightarrow_a e_j \text{ OR } e_j \rightarrow_a e_i \text{ block the reply} \\ \bigcup \{ e_z \Delta e_j \mid e_j \rightarrow e_z \} & \text{otherwise} \end{cases}
\]

Definition 5 states that the information carried by a message reply from execution \( e_j \) to its invoker execution \( e_i \) is equal to:

- the set composed of the object read, if \( e_j \) is a read execution;
- the empty set, if either \( e_j \) is a write or create execution, \( e_j \) has been invoked in asynchronous mode, or \( e_j \) has been invoked as restricted and the reply is blocked;\n- the union of the information which \( e_j \) has received from the executions it has, in turn, invoked, otherwise.

DEFINITION 6. (RACL of an execution). Let \( e_j \) be a method execution. The read access control list of \( e_j \), denoted as \( \text{RACL}(e_j) \), is defined as follows:

\[
\text{RACL}(e_j) = \begin{cases} \text{RACL}(o(e_j)) & \text{if } \text{name}(e_j) = \text{read} \\ U & \text{if } \text{name}(e_j) \in \{ \text{write}, \text{create} \} \text{ OR } e_j \Delta e_i = \emptyset \text{ for all } e_i \text{ such that } e_j \rightarrow e_i \\ \bigcup \{ \text{RACL}(e_i) \mid e_i \rightarrow e_j, e_i \Delta e_j \neq \emptyset \} & \text{otherwise} \end{cases}
\]

Definition 6 states that the RACL of an execution \( e_j \) is equal to:

- the RACL of the object read, if \( e_j \) is a read execution;
- the set of all users, if either
  1. \( e_j \) is a write or create execution, or
  2. \( e_j \) is a non-primitive execution and all executions invoked by it have returned no information;
- the intersection of the RACLs of the executions invoked by \( e_j \) that have returned to \( e_j \) some information, otherwise.

For instance, with reference to the transaction illustrated in Fig. 1 and assuming that the RACLs of the objects are as in Fig. 2, the RACL of restricted execution \( e_6 \) is \( \text{RACL}(e_6) = \{ x \} \).

DEFINITION 7 (Forward transmission). Let \( e_i \) and \( e_j \) be two method executions such that \( e_i \rightarrow e_j \). The information transmitted from \( e_i \) to \( e_j \) through the message requiring the invocation of \( e_j \), denoted as \( e_i \triangleright e_j \), is recursively defined as follows: Let \( e_0 \) be the execution that invoked \( e_i \), then

\[
e_i \triangleright e_j = \begin{cases} \emptyset & \text{if } o(e_i) \in U \\ \bigcup \{ e_z \Delta e_j \mid e_j \rightarrow e_z \} \cup e_i \triangleright e_i & \text{otherwise} \end{cases}
\]

5. Note that in the first two cases the reply is empty due to the semantics of execution \( e_j \), whereas in the last case the reply is empty due to the restriction explicitly requested.

Fig. 2. An example of RACLs.

Definition 7 states that the information transmitted from \( e_i \) to \( e_j \) is equal to:

- the empty set, if \( e_j \) has been invoked directly by a user;
- the union of the information which \( e_j \) has received from the executions it has invoked that precede \( e_i \) and the information which \( e_j \) has received from its invoker.

It is easy to remember our notation. The first term always represents the execution which transmits the information, the second term represents the execution which receives it, and \( \triangleright \) and \( \Delta \) points to the direction in which information is transmitted.

For simplicity, we will use \( e_i \triangleright \emptyset \text{ and } \forall e_j \) to denote \( e_i \triangleright \emptyset \text{ and } e_j \triangleright e_i \) respectively, when \( e_i \) is not needed to be identified. Note that this does not introduce any ambiguity because an execution is invoked by exactly one execution (i.e., there exists exactly one such \( e_j \)).

Given the definitions above, we can now formalize the definition of information flow as follows:

DEFINITION 8 (Information flow). There exists a flow from object \( o_i \) to object \( o_j \) in a transaction \( T \), written \( o_i \triangleright o_j \), if a method execution \( e_j \) exists such that \( o(e_j) = o_j \), name\( (e_j) \in \{ \text{write, create} \} \), and \( o_i \in \forall e_j \).

Definition 8 states that there exists a flow between objects \( o_i \) and \( o_j \) in a transaction if and only if a write or create method that has received information about object \( o_i \) is executed on \( o_j \). The reason why \( \forall e_j \) is used in the definition is that the only information available to a write or a create execution is the information which has been transmitted to the execution by its invoker (by assumption, write and create executions are primitive and cannot invoke other executions).

Fig. 3 illustrates the information transmission and flow for the transaction in Fig. 1, assuming the RACLs in Fig. 2.

Note that flows in Definition 8 are potential flows. The information written in \( o_i \) may be independent of the information read in \( o_i \). Moreover, information about \( o_j \) may have

6. Note that this does not mean that no input parameters are specified. The information of which we keep track is the information read by the transaction during its execution. We do not consider the information introduced by the user upon invocation of the transaction.
7 THE MESSAGE FILTER

The message filter is a trusted system component which intercepts every message exchanged among the objects in a transaction execution to guarantee that no unsafe flow takes place. The concept of message filter was first introduced by Jajodia and Kogan [8] for the enforcement of the mandatory policy in object-oriented systems. The basic ideas behind this approach are valid in our framework as well. The message filter basically works by blocking

1) every write execution which may enact unsafe flows, and

7. For the sake of simplicity, only forward transmission to write executions is given.

2) every reply of restricted executions carrying information for which users allowed to access the object receiving the reply are not authorized.8

Unsafe flows can therefore be blocked at two different times: at the time of information transmission, by enforcing restricted executions, or at the time of information acquisition, by blocking the corresponding write operations. Note that if an object is passed some information it is not allowed to store, the object will not be able to execute any further write operations. Restricted executions avoid this by preventing information transmission to an object if this would imply that the object will not be allowed to execute subsequent write operations. Note that both kinds of information blocking, that through write operations and that through restricted execution, are useful. Note also that it is completely legitimate to return an object information it is not allowed to store. This is, for example, the case of an object providing some services which needs to make some computation over sensitive information. The sensitive information will be passed to the object which, however, will not be allowed to store it.

To enforce deferred reply execution, the message filter maintains a reply queue storing the reply of executions invoked under deferred reply mode. When an execution invoked under deferred reply mode completes, its reply is inserted in the queue. The reply is retrieved from the queue and returned to the invoker upon explicit request of the invoker. To determine possible flows, for each execution e, the message filter keeps track of

1) the information that the execution has received from its invoker, through the parameters of the message requiring the invocation;

2) the information that the execution has received from the executions it has invoked, through the message replies;

3) the RACL of the execution.

8. Note that the synchronous-restricted approach is vulnerable to timing channels [14]. Indeed, a Trojan horse embedded in a method of an object can transmit information to a less protected object by appropriately modulating the delay in sending the reply. To eliminate the possibility of timing channels, the message filter can introduce random delay when returning a null reply (“nil”) for a restricted execution whose actual reply is being blocked. An approach to eliminate timing channels by delaying the communication between processes has been proposed by Kang and Moskowitz in [9].

<table>
<thead>
<tr>
<th>Backward Transmission</th>
<th>Forward Transmission7</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_3 \Delta e_2 = \emptyset )</td>
<td>( \nabla w_{o_3} = {o_2} )</td>
<td>( o_2 \gg o_1 )</td>
</tr>
<tr>
<td>( e_2 \Delta e_1 = {o_2} )</td>
<td>( \nabla w_{o_4} = {o_2, o_5} )</td>
<td>( o_2 \gg o_3 )</td>
</tr>
<tr>
<td>( e_4 \Delta e_1 = \emptyset )</td>
<td>( \nabla w_{o_5} = {o_2, o_5} )</td>
<td>( o_2 \gg o_4 )</td>
</tr>
<tr>
<td>( e_5 \Delta e_1 = {o_5} )</td>
<td>( \nabla w_{o_6} = {o_2, o_6} )</td>
<td>( o_2 \gg o_5 )</td>
</tr>
<tr>
<td>( e_7 \Delta e_6 = {o_7} )</td>
<td>( \nabla w_{o_1} = {o_2, o_5} )</td>
<td>( o_2 \gg o_6 )</td>
</tr>
<tr>
<td>( e_6 \Delta e_1 = \emptyset )</td>
<td></td>
<td>( o_4 \gg o_4 )</td>
</tr>
<tr>
<td>( r_{o_i} \Delta = {o_i}, i = 2, 4, 5, 6, 7 )</td>
<td></td>
<td>( o_5 \gg o_1 )</td>
</tr>
<tr>
<td>( w_{o_i} \Delta = \emptyset, i = 1, 3, 4, 5, 6 )</td>
<td></td>
<td>( o_5 \gg o_5 )</td>
</tr>
</tbody>
</table>

Fig. 3. Information transmission7 and flow for transaction in Fig. 1, assuming the RACLs in Fig. 2.
We refer to the information in 1) as forward information and denote it with \( F(e) \). We refer to the information in 2) as backward information and denote it with \( B(e) \). \( F(e) \) corresponds to the forward information (\( V_e \)) of Definition 7. \( B(e) \) is determined incrementally and its value at a given time is the union of the information returned by each execution \( e_i \) (i.e., \( e_i \Delta \)) invoked by \( e \) up to that time, as specified in Definition 5. The relationship between the variables used by the message filter and the definitions introduced in Section 6 will be formally stated in Section 7.1.

We now illustrate how the message filter works upon interception of a message. Remember that an elementary method is executed only if the transaction’s initiator has the corresponding authorization; otherwise, the operation does not have any effect. For simplicity we consider all elementary methods required in a transaction to be authorized and correct, where by correct we mean that the parameters in the methods invoking the methods correspond to existing attributes of objects.

Consider a message \( g = (\text{name, pars, mode}) \) being sent by execution \( e_i \) running on object \( o_i \). The message is intercepted by the message filter. Suppose the message requires an execution \( e_j \) on an object \( o_j \). The backward information \( B(e_j) \) of execution \( e_j \) to be invoked is initialized to \( \varnothing \), and its RACL is initialized to the set \( U \) of all users. If the message has been sent directly by a user to start a transaction, the forward information of \( e_j \) is set equal to \( \varnothing \). Then, execution \( e_j \) is performed and its reply returned to the user. Otherwise, if the message has been sent by another execution \( e_p \), the forward information \( F(e_p) \) of \( e_p \) is set equal to the union of the forward and the backward information of \( e_i \). Then, the message filter determines how to execute \( e_p \) depending on the method to be executed and on the interaction mode. Cases 1-3 consider the execution of the built-in read, write, and create methods. Cases 4-8 consider the execution of non-primitive methods under different interaction modes:

1) The message is a read message. The read method on \( o_j \) is invoked. The RACL of \( e_i \) is set equal to the RACL of object \( o_j \) read, and the RACL of \( e_i \) is updated to be the RACL of \( e_j \) intersected with the RACL of \( e_i \). The backward information of \( e_j \) is set equal to \( \{ o_i \} \) and the backward information of \( e_i \) is updated by taking its union with the backward information of \( e_i \). The reply of the message is then returned to \( e_i \).

2) The message is a write message. If the RACL of every object in the forward information of \( e_i \) contains the RACL of object \( o_j \) no unsafe flow can be enacted by execution \( e_j \). Then, the write method is executed and a “success” is returned. Otherwise, no method is executed and a “failure” is returned.

3) The message is a create message. A new object is created and the user running the transaction is given privileges on the object created. The oid of the created object is returned to execution \( e_i \).

4) \( e_i \) is a synchronous-unrestricted non-primitive execution. Execution \( e_i \) is invoked. The RACL of \( e_i \) is modified by intersecting it with the RACL of \( e_i \). The backward information of \( e_j \) is updated by taking its union with the backward information of \( e_i \). The reply produced by \( e_i \) is then returned to \( e_j \).

5) \( e_j \) is a synchronous-restricted execution. Execution \( e_i \) is invoked. If the RACL of \( e_i \) contains the RACL of \( o_i \) the reply is allowed to pass and RACL(\( e_i \)) is updated as in case 4. Otherwise, a NIL is returned to \( e_i \) and the backward information of \( e_j \) is set equal to \( \varnothing \). The backward information of \( e_i \) is updated by taking its union with the backward information of \( e_j \).

6) \( e_j \) is an asynchronous execution. A NIL is returned to \( e_i \). Then, \( e_i \) is invoked and its actual reply discarded. The backward information of \( e_j \) is set equal to \( \varnothing \).

7) \( e_i \) is a deferred reply unrestricted execution. A message identifier is generated by the filter and returned to \( e_i \). Execution \( e_j \) is invoked. The five-tuple composed of the identifier of the message invoking \( e_j \), the identifier of \( e_j \), the reply of \( e_j \), the backward information of \( e_j \) and the RACL of \( e_j \) is inserted in the reply queue.

8) \( e_i \) is a deferred reply restricted execution. A message identifier is generated by the filter and returned to \( e_i \). Execution \( e_j \) is invoked. If the RACL of \( e_i \) contains the RACL of \( o_j \), the reply to be returned is set equal to the reply produced by \( e_j \); otherwise, the reply to be returned is set equal to NIL and the backward information of \( e_i \) is set equal to \( \varnothing \). Then, the message filter proceeds as in case 7, and the five-tuple composed of the identifier of the message invoking \( e_j \), the identifier of \( e_j \), the reply to be returned, the backward information of \( e_i \) and the RACL of \( e_j \) is inserted in the reply queue.

If the message is a get message requiring the reply of an execution \( e_i \) previously invoked under deferred reply mode, the message filter examines the reply queue to determine the information about the interested invocation. If the reply is different from NIL, the RACL of execution \( e_i \) is updated by intersecting it with the RACL of \( e_j \). Then, the backward information \( B(e_j) \) of \( e_j \) is updated by taking its union with the backward information of \( e_i \).

The code for the message filtering algorithm is given next.

**Message filtering algorithm**

/* Input: Message \( g = (\text{name, pars, mode}) \) sent by execution \( e_p \) running on object \( o_i \) */

begin

If name\( (g) \neq \text{GET} */ \( g \) is a message invoking an execution \( e_j \) on \( o_j \) */

then

\( B(e_j) \leftarrow \varnothing \)

\( \text{RACL}(e_j) \leftarrow U \)

if \( o_i \in U \) /* the message is sent by a user to start a transaction */

then

\( F(e_j) \leftarrow \varnothing \)

invoke \( e_j \)

reply \leftarrow \text{reply from } e_j

return reply to \( e_i \)

else /* the sender of the message is not a user */

\( F(e_j) \leftarrow F(e_j) \cup B(e_i) \)

end
case
(1) \( g = (\text{READ}, (a_i), \text{mode}) \) do
   reply \( \leftarrow \) value of \( a_i \)
   RACL\((e)\) \( \leftarrow \) RACL\((o)\)
   RACL\((e)\) \( \leftarrow \) RACL\((e)\) \( \cap \) RACL\((e)\)
   \( B(e) \leftarrow \{o\} \)
   \( B(e) \leftarrow B(e) \cup B(e) \)
   return reply to \( e \)
(2) \( g = (\text{WRITE}, (v_i), \text{mode}) \) do
   if \( \forall o \in F(e), \text{RACL}(o) \supseteq \text{RACL}(o) \)
      then \( [a_i \leftarrow v_i; \text{reply} \leftarrow \text{success}] \)
      else reply \( \leftarrow \) failure
endif
   return reply to \( e \)
(3) \( g = (\text{CREATE}, (v_1, \ldots, v_k), \text{mode}) \) do
   CREATE \( o \) with values \( v_1, \ldots, v_k \),
   RACL\((o)\) \( \leftarrow u \), WACL\((o)\) \( \leftarrow u \),
   CACL\((o)\) \( \leftarrow u \),
   reply \( \leftarrow \) oid of \( o \)
   return reply to \( e \)
(4) \( \text{mode} = \text{SU} \) and \( \text{name}(g) \notin \{\text{READ, WRITE, CREATE}\} \) do
   /* \( e \) is a non-primitive execution invoked
   in synchronous mode */
   invoke \( e \)
   reply \( \leftarrow \) reply from \( e \)
   RACL\((e)\) \( \leftarrow \) RACL\((e)\) \( \cap \) RACL\((e)\)
   \( B(e) \leftarrow B(e) \cup B(e) \)
   return reply to \( e \)
(5) \( \text{mode} = \text{SR} \) do /* \( e \) is invoked in
   synchronous-restricted mode */
   invoke \( e \)
   if \( \text{RACL}(e) \supseteq \text{RACL}(o) \)
      then reply \( \leftarrow \) reply from \( e \)
      RACL\((e)\) \( \leftarrow \) RACL\((e)\) \( \cap \) RACL\((e)\)
   else reply \( \leftarrow \) NIL, \( B(e) \leftarrow \{\} \)
endif
   \( B(e) \leftarrow B(e) \cup B(e) \)
   return reply to \( e \)
(6) \( \text{mode} = \text{A} \) do /* \( e \) is invoked in
   asynchronous mode */
   reply \( \leftarrow \) NIL
   return reply to \( e \)
   invoke \( e \)
   \( B(e) \leftarrow \{\} \)
   discard reply from \( e \)
(7) \( \text{mode} = \text{DU} \) do /* \( e \) is invoked in
   deferred reply mode */
   produce a message identifier \( \text{mid} \)
   return \( \text{mid} \) to \( e \)
   invoke \( e \)
   reply \( \leftarrow \) reply from \( e \)
   insert \( (\text{mid}, e, \text{reply}, B(e), \text{RACL}(e)) \) to the reply queue
(8) \( \text{mode} = \text{DR} \ldots \)
LEMMA 2. Let \( e_1 \) and \( e_2 \) be two executions such that \( e_1 \rightarrow e_2 \). At the end of execution \( e_2 \), \( B(e_2) \neq e_1 \Delta e_1 \).

PROOF. We prove this lemma by proving that \( B(e_2) \) agrees with Definition 5. We consider each case in the definition in turn. \( B(e_2) \) is initialized to \( \emptyset \) by the algorithm and then changed according to the method to be executed and the execution mode of \( e_2 \). If \( \text{name}(e_2) = \text{read} \), \( B(e_2) \) is set equal to \( o(e_2) \) by the algorithm and the lemma is satisfied. Similarly, if \( \text{name}(e_2) \in \{\text{write}, \text{create}\} \), \( B(e_2) \) keeps the value \( \emptyset \) with which it was initialized. Thus, the lemma is satisfied. If \( e_1 \) is an asynchronous or a restricted execution such that \( \text{RACL}(e_1) \nsubseteq \text{RACL}(o(e_2)) \), \( B(e_2) \) is set equal to \( \emptyset \) by the algorithm at the end of the execution. Hence, the lemma is satisfied. Consider now the last case of the definition. We prove that the lemma holds in this case using induction. The base of the induction is represented by the first two cases of the definition, for which the lemma is satisfied. We then prove that if it holds for executions \( e_1, \ldots, e_n \) invoked by execution \( e_p \), it also holds for \( e_p \). The algorithm updates \( B(e_p) \) by taking its union with \( B(e_z) \) at the completion of each execution \( e_z \) called such that \( \text{name}(e_z) \notin \{\text{write}, \text{create}\} \) or \( \text{mode}(e_z) \neq \text{asynchronous} \). However, in these cases \( B(e_z) = \emptyset \) and hence \( B(e_z) = B(e_z) \cup B(e_z) \). Hence, for any execution \( e_p, B(e_p) = \bigcup \{B(e_z) \mid e_z \rightarrow e_p\} \). Since the lemma holds for \( e_z, B(e_z) = e_z \Delta e_z \). Hence, \( B(e_p) = \bigcup \{e_z \Delta e_z \mid e_z \rightarrow e_p\} = e_p \Delta e_p \), and the lemma is satisfied. □

LEMMA 3. Let \( e_1 \) be an execution. For any execution \( e_w \) such that \( e_1 \rightarrow e_w \), \( B(e_w) \) before the invocation of \( e_w \) is equal to the union of all the information received by \( e_w \) from the execution preceding \( e_z \), i.e., \( B(e_w) = \bigcup \{e_z \Delta e_z \mid e_z \leq e_w\} \).

PROOF. The algorithm updates \( B(e_z) \) to \( B(e_z) \cup B(e_z) \) at the completion of each execution \( e_z \) invoked by \( e_1 \) such that \( \text{name}(e_z) \notin \{\text{write}, \text{create}\} \) and \( \text{mode}(e_z) = \text{asynchronous} \), and leaves \( B(e_z) \) unchanged otherwise. However, in these cases \( B(e_z) = \emptyset \) and hence \( B(e_z) = B(e_z) \cup B(e_z) \). Hence, at the time of invocation of \( e_z \), \( B(e_z) = B(e_z) \) for all \( e_z \) preceding \( e_w \), i.e., such that \( e_z \leq e_w \). Since at the completion of each execution \( e_z, B(e_z) = e_z \Delta e_z \) from Lemma 2, \( B(e_w) \) can be rewritten as \( B(e_w) = \bigcup \{e_z \Delta e_z \mid e_z \leq e_w\} \), and the lemma is satisfied. □

LEMMA 4. Let \( e_1 \) and \( e_2 \) be two executions such that \( e_1 \rightarrow e_2 \), then \( F(e) = e_2 \Delta e_1 \).

PROOF. We use induction to prove that \( F(e_2) \) agrees with Definition 7. We first show that the lemma is satisfied for the execution corresponding to the root of the transaction execution tree, and if it is satisfied for an execution \( e_p \), it is also satisfied for any execution \( e_z \) such that \( e_z \rightarrow e_p \). Let us consider the case where \( e_z \) is the root of the transaction execution tree, i.e., it has been invoked directly by the user. \( F(e_z) \) is set equal to \( \emptyset \) by the message filter and is never changed subsequently and, therefore, the lemma is satisfied. Assume next that the lemma is satisfied for \( e_z \). We will show that it is also satisfied for any execution \( e_z \) such that \( e_z \rightarrow e_p \). According to Definition 7, \( e_z \Delta e_p = \bigcup \{e_z \Delta e_z \mid e_z < e_z \} \). From Lemma 3, we can substitute the first part of the expression on the right side of the equality with \( B(e_z) \), and since the lemma is satisfied for \( e_z \), we can substitute the second part of the expression with \( F(e_z) \). Thus, \( e_z \Delta e_p = B(e_z) \cup F(e_z) \), where \( B(e_z) \) is the backward information of \( e_z \) at the time of the invocation of \( e_z \). Since \( F(e_z) \) is initialized by the algorithm to \( B(e_z) \cup F(e_z) \) and never changed subsequently, the lemma is satisfied. □

THEOREM 1. The message filter correctly enforces restricted executions. That is, given executions \( e_1 \) and \( e_2 \) such that \( e_1 \rightarrow e_2 \), \( e_1 \Delta e_2 = \emptyset \) if \( \text{RACL}(e_1) \nsubseteq \text{RACL}(o(e_2)) \).

PROOF. Suppose that execution \( e_1 \) is restricted (case 5 if \( e_1 \) is synchronous-restricted and case 8 if \( e_1 \) is deferred reply-restricted). The message filter checks \( \text{RACL}(e_1) \) and injects a NIL reply iff condition \( \text{RACL}(e_1) \nsubseteq \text{RACL}(o(e_2)) \) evaluates to false, where \( o \circ \circ \). The theorem is, therefore, trivially satisfied. □

THEOREM 2. The message filter blocks those (and only those) write and create operations that may enact unsafe flows.

PROOF. Consider first the case of write executions. The message filter returns “failure” for the write executions blocked and “success” otherwise. We suppose all operations to be correct and authorized, i.e., failures can be caused only by the message filter. We prove that the executions for which a “failure” is returned are exactly those executions which may enact unsafe flows. Formally, \( \forall w : \text{reply}(w) = \text{“failure”} \Rightarrow \exists o \in \text{W}, \text{RACL}(o(\circ \circ)) \nsubseteq \text{RACL}(o(w)) \).

We prove the \( \Rightarrow \) implication by assuming that it does not hold and deriving a contradiction. Suppose then that \( \text{reply}(w) = \text{“failure”} \) and \( o \circ \circ \in \text{W}, \text{RACL}(o(\circ \circ)) \nsubseteq \text{RACL}(o(w)) \). Since \( F(w) = \emptyset \) from Lemma 4, condition in the if statement of case 2 is evaluated to true. But then, \( w \) is executed and \( \text{reply}(w) \) is assigned “success” by the algorithm which contradicts the assumption.

To prove the \( \Leftarrow \) implication, we again assume that it does not hold and then derive a contradiction. Suppose then that \( \exists o \in v w, \text{RACL}(o(\circ \circ)) \nsubseteq \text{RACL}(o(w)) \) and
reply(\(w\)) = “success.” Since \(F(w) = \forall w\) from Lemma 4, the condition in the if statement of case 2 is evaluated to false. But then, reply(\(w\)) = “failure,” which contradicts the assumption.

Consider now the case of create executions. Since create executions are never blocked by the algorithm, we need to prove that all create executions are safe; i.e., for any create execution \(c, \exists o, o \in Vc\) such that RACL(\(o\)) \(\not\subseteq\) RACL(\(o(c)\)). Upon completion of create execution \(c, \text{RACL}(o(c))\) is assigned value \(u\) by the message filter, where \(u\) is the identifier of the initiator of the transaction. According to Property 1, \(u \in \text{RACL}(o)\) for any object \(o\) accessed by the transaction. Thus, \(u \in \text{RACL}(o),\) for all \(o \in Vc\). Hence, \(\text{RACL}(o(c)) = [u]\) for all objects \(o \in Vc\), and the theorem is satisfied.

8 Administration of Authorizations

Whether a flow of information between two objects is safe depends on the authorizations of the users on the two objects, because these authorizations determine the decisions of the message filter on how to handle the messages. Hence, the correctness of the message filter controls strongly depends on the correctness of the authorizations.

To ensure the correctness of authorizations, we impose some restrictions on the administration (i.e., granting or revoking) of authorizations.

The first restriction is that authorizations may not be granted or revoked during execution of normal transactions. To ensure the satisfaction of this restriction, we do not allow methods to change authorizations. Changes to the authorizations must be requested directly by the users (trusted path). To understand the importance of this requirement, suppose that authorizations can be granted or revoked during normal execution. A malicious user \(v\) could then embed in a method a Trojan horse that, when executed by another user \(u\), grants \(v\) authorizations on the objects that are owned by \(u\). Since \(u\) is authorized to grant someone else authorizations on these objects, these grants would be considered as legitimate. However, their execution was hidden and not desired by \(u\).

The second restriction concerns the assignment of authorizations on objects created inside a transaction. Since we propose to keep authorization administration outside the normal transaction execution, upon creation of an object, our approach gives only the user who started the transaction the authorizations to access the object. Authorizations on the created object can be granted by its owner to other users once the transaction has completed. Note that, as a consequence of a grant operation, a user previously not authorized to read an object can then be allowed to read it. This fact accords with our model. Grants given by a user express his explicit desire to give other users the access. We assume that the owner of an object will behave correctly with respect to the dissemination of the information stored in the object. The basic assumption underlying our model is that users are trusted while processes executing on their behalf are not.

The third restriction is that the authorizations on an object cannot change while the object is being accessed by some transaction. This restriction ensures the consistency of the ACLs of an object at any point during the execution of a transaction. Therefore, it avoids the scenario in which an object that was not accessible for reading by some users is then made accessible to them, or vice versa, thereby compromising the correctness of the message filter controls. Note that, for our purposes (i.e., avoiding information to be disclosed to unauthorized users), it would have been sufficient to prevent the read set of an object to be changed during normal execution. However, we believe that it is a good principle to consider administrative operations as separate from the normal activity. Note that this restriction is used in most of the current relational systems in which a transaction accessing a relation receives the lock on the relation and the related information, including the ACL.

We note, however, that our restriction of not allowing the authorizations of an object to change if the object is being accessed by some transaction may be undesirable in some cases. It is possible to relax this restriction by allowing some changes to the authorizations. We elaborate on different approaches that can be adopted in this respect.

Suppose that the authorizations on an object are changed while the object is being accessed by some transaction, the system can accept the change and:

1) Immediately apply the change. In this case, because the message filter checks the RACL of all objects read before a write operation at the time the write occurs, the message filter will always consider the latest RACL, i.e., the RACL produced by the last change.
2) Consider the updated RACL for new transactions which may require to access the object. However, if the object is being currently accessed by a transaction, only the RACL the object had at the time it was first accessed by the transaction is considered.
3) Immediately apply the change and modify the message filter to remember the RACL of every object when it is read. Flow control will then be enforced by considering the RACL of an object at the time it was read; if a transaction executes different read operations on an object, the RACL at the time of each read is considered.

To clarify the difference between those approaches consider an object \(o\) with RACL equal to RACL1. Suppose a transaction \(T\) reads \(o\). Suppose the RACL of \(o\) is subsequently changed to RACL2. At this point, \(T\) reads \(o\) again and then wishes to write object \(o\). The RACL that must be considered to determine possible information flows are

1) RACL2, if the first approach is adopted;
2) RACL1, if the second approach is adopted; and
3) both RACL1 and RACL2, if the third approach is adopted.

9 Implementation Issues

Enforcing the access control for each single object and controlling flows of information between objects may introduce some performance problems. These problems can be overcome by providing efficient implementation of the controls enforced by the message filter. Implementation issues to be taken into consideration concern both space requirements and processing overhead.
We are currently working on a prototype for the message filter proposed in this paper. Although it is not our goal to address implementation and performance issues in the current paper, we briefly discuss how space and processing requirements can be reduced by techniques we plan to use in our implementation. To reduce the space overhead, we plan to use the approach proposed by Kelter in [11]. It exploits the fact that, generally, many objects in the system share the same access control list (ACL). As a consequence the number of different ACLs appearing in an object base is usually smaller than the number of objects. On the basis of this observation, Kelter proposes to assign each ACL a number and to associate with each object the number of the corresponding ACL, instead of the ACL itself. Thus, one copy of each ACL will be stored, which is then shared by different objects. In the approach proposed by Kelter, each ACL is a concatenation of entries. An ACL entry is the concatenation of the subject identifier and two arrays of Boolean for the access values of the different modes. An entry for a subject exists in the ACL of an object only if the subject has some authorization on the object.

We also assign numbers to ACLs. However, instead of keeping a single ACL for all access modes, we associate with each object three ACLs (RACL, WACL, and CACL), one for each access mode. Each ACL is a string of Boolean values. The number of elements in the ACL is equal to the number of users in the system. Each user is assigned a number which uniquely identifies the user in the system. Entry \( i \) in an ACL of an object \( o \) indicates whether user \( i \) has the authorization (value 1) for the corresponding access mode or not (value 0) on \( o \). This approach is not space consuming. For example, with an ACL 20 bytes long, we can encode 160 users. In most cases, however, the length will be smaller, because the number of users in current OODB applications is generally small [5]. As in [11], ACL numbers and ACLs are transformed into one another using an ACL directory that can be implemented as a dynamic table. The dynamic table consists of an array of pointers. The \( j \)th pointer points to an independently allocated area that contains the ACL whose number is \( j \).

ACLs and ACL numbers can also be used to reduce the processing time needed to enforce the control on information flows. For instance, the control to be executed upon each write operation can be efficiently implemented as follows: Instead of keeping the forward information \( F(e) \) of an execution \( e \) as a set of objects, \( F(e) \) is expressed as the set of the corresponding RACL numbers. In this case, even if several objects are accessed, with the same RACL numbers, the RACL number will appear only once in \( F(e) \). Upon each write operations \( w \), the logical AND of all RACLs whose number is in \( F(w) \) is generated.\(^{10}\) The resulting string will have 1 in position \( i \) if and only if user \( i \) can read all the objects in \( F(w) \). The negation of this string is then calculated and logically ANDed with the RACL of the object to be written. The result of this operation is a string where entry 1 in position \( i \) indicates that user \( i \) has the authorization to read the object to be written and does not have the authorization to read an object in \( F(w) \). Hence, if an entry 1 exists in such a string, the flow is unsafe. A similar technique can be used for computing the backward information of executions.

To illustrate this approach further, consider the RACLs in Fig. 2 and the transmission of information to write operations given in Fig. 3. Considering the order of the subjects in the RACL to be \( x, y, \) and \( z \), the RACL, expressed as explained above, are as follows:

- \( \text{RACL}(o_1) = \text{RACL}(o_2) = \text{RACL}(o_3) = \text{RACL}(o_4) = [110] \)
- \( \text{RACL}(o_5) = \text{RACL}(o_6) = [101] \)
- \( \text{RACL}(o_7) = [100] \)

Consider now \( w_{o_5} \) and \( w_{o_6} \), the two write operations that would enact unsafe flows, where \( \forall w_{o_5} = \{o_2\} \) and \( \forall w_{o_6} = \{o_2, o_3\} \). As for \( w_{o_5} \), we have

\[ \text{RACL}(o_5) \land \lnot(\text{RACL}(o_2)) = [001] \]

i.e., the flow is unsafe because user \( z \), who cannot read some object in the forward information of \( w_{o_5} \), can read \( o_3 \). Similarly, for \( w_{o_6} \), we have

\[ \text{RACL}(o_6) \land \lnot(\text{RACL}(o_2) \lor \text{RACL}(o_3)) = [010], \]

meaning the flow is unsafe because user \( y \), who cannot read some object in the forward transmission of \( w_{o_6} \), can read \( o_1 \).

\(^{10}\) The value does not be to be recomputed at each write operation. It can be stored and incrementally updated.

10 Conclusions and Future Research

In this paper, we have proposed the use of a strict need-to-know policy for overcoming the vulnerability of discretionary access controls to Trojan horses leaking information to unauthorized users. We have investigated such an approach in the framework of an object-oriented model. In this model, there is an information flow whenever information stored in an object is transmitted (via message exchanges) to another object and acquired by this object through a write operation. To control information flow, the discretionary access control is complemented by a message filter, whose task is to intercept every message to prevent any illegal flows. Our model provides different options for the application of this strict need-to-know policy, thus making the policy more flexible and adaptable to different situations.

We have cast our discussion in terms of object systems based on the message passing paradigm. Our reason for this choice is that the Corba [12] standard is based on this paradigm. Since one of goals of Corba is facilitating the interoperability of object systems, the scope of our access control model can be extended to the case of heterogeneous object systems. Note, however, that some object-oriented database systems use a procedure-call approach for method invocation. We believe that our approach can be used in this framework by making the filter controls part of the procedure-call mechanism. We leave issues related to this question for further work.

The work presented in this paper can be extended in several directions. A first direction concerns the consideration of authorizations on methods, instead than on ele-
mentary read, write, and create operations. Authorizations on methods fit better with the object-oriented paradigm since they do not affect information hiding. If authorizations on methods are considered, the ACL of the method in which an object has been read may have to be considered, instead of the RACL of the object read, to determine information flows. A second direction concerns the use of trusted methods to support exceptions to the strict need-to-know policy. Issues related to trusted methods concern formal verification of methods and their use in combination with methods that are not trusted. Further directions include: the development of techniques to inspect the code of methods in order to determine actual information flows; the investigation of different filtering policies, such as for example that of blocking read operations instead than transmission or acquisition; and the use of versions to control information flow, for instance write operations to a less protected object can be allowed and result in the creation of a more protected version of the object.

ACKNOWLEDGMENTS

The work of Pierangela Samarati and Elisa Bertino was partially supported by the Italian National Research Council (CNR) under Grant No. 94.00450.CT12, by NATO under Collaborative Research Grant No. 930888, and by the Italian M.U.R.S.T. The work of Sushil Jajodia was partially supported by the National Science Foundation under Grant No. IRI-9303416 and by NATO Collaborative Research Grant No. 930888.

REFERENCES


Alessandro Ciampichetti received the master’s degree in computer science from the University of Genova, Italy, in October 1993. From November 1993 to July 1994, he worked as a research consultant at the University of Genova. He is currently working as a consultant in the area of client/server databases with the Andersen Consulting Worldwide Organization. His research interests include access control and object-oriented databases.

Sushil Jajodia received his PhD from the University of Oregon, Eugene. He is now director of the Center for Secure Information Systems and a professor of information and software systems engineering at George Mason University, Fairfax, Virginia. He joined GMU after serving as the director of the Database and Expert Systems Program at the National Science Foundation. Before that, he was head of the Database and Distributed Systems Section at the Naval Research Laboratory in Washington. He has also been a visiting professor at the University of Milan, Italy, and at the Isaac Newton Institute for Mathematical Sciences, Cambridge University, England.

Dr. Jajodia’s research interests include information security, temporal databases, and replicated databases. He has published more than 150 technical papers in refereed journals and conference proceedings, and has edited or coedited 10 books, including Advanced Transaction Models and Architectures (Kluwer, 1997), Multimedia Database Systems: Issues and Research Directions (Springer-Verlag Artificial Intelligence Series, 1996), Information Security: An Integrated Collection of Essays (IEEE Computer Society Press, 1995), and Temporal Databases: Theory, Design, and Implementation (Benjamin/ Cummins, 1993). He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of information security.

Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding co-editor-in-chief of the Journal of Computer Security. He is on the editorial boards of IEEE Concurrency and the International Journal of Cooperative Information Systems, and is a contributing editor of the Computer & Communication Security Reviews. He served as program chair of the 1996 IFIP Working Conference on Integrity and Control in Information Systems and the 1997 IFIP Working Conference on Database Security. He has been named a Golden Core member for his service to the IEEE Computer Society. He is a past chairman of the IEEE Computer Society Technical Committee on Data Engineering and the Magazine Advisory Committee. He is a senior member of the IEEE and a member of the IEEE Computer Society and the Association for Computing Machinery. The URL for his web page is http://www.isse.gmu.edu/~csis/faculty/jajodia.html.