Exception-Based Information Flow Control in Object-Oriented Systems

ELISA BERTINO, SABRINA DE CAPITANI DI VIMERCATI, and ELENA FERRARI
Università di Milano
and
PIERANGELA SAMARATI
Università di Milano and SRI International

We present an approach to control information flow in object-oriented systems. The decision of whether an information flow must be permitted or denied depends on both the authorizations specified on the objects and the process by which information is obtained and transmitted. Depending on the specific computations executed, a process accessing sensitive information could still be allowed to release information to users not allowed to directly access this information. Exceptions to the permissions/restrictions stated by the authorizations are specified by means of exceptions associated with methods. Two kinds of exceptions are considered: *invoke exceptions*, applicable during a method execution, and *reply exceptions*, applicable to the information returned by a method. Information flowing from one object into another object or returned to the user is subject to the different exceptions specified for the methods enforcing the transmission. In the paper, we formally characterize information transmission and flow in a transaction and define the conditions for an information flow to be safe. We define security specifications and characterize safe information flows. We propose an approach to control unsafe flows and present an algorithm enforcing it. We also illustrate an efficient implementation of our controls and present some experimental results evaluating its performance.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General - *Security, integrity, and protection*; H.2.4 [**Database Management**]: Systems - *Object-oriented databases*; H.2.7 [**Database Management**]: Database Administration - *Security, integrity, and protection*

General Terms: Security

Additional Key Words and Phrases: Access control, confidentiality, information flow control, object-oriented databases and systems

This research was supported in part by NATO under Collaborative Research Grant number 930888. Samarati's work was also partially supported by DARPA/Rome Laboratory under contract F30602-96-C-0337 and by the National Science Foundation under grant CCR-94-22688.
A preliminary version of this paper appeared under the title "Providing Flexibility in Information Flow Control for Object-Oriented Systems" in the *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1997.
Authors' addresses: Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico 39/41, 20135 Milano, email: ⟨(bertino|decapita|ferrarie|samarati)@dsi.unimi.it⟩; P. Samarati, Computer Science Laboratory, SRI International, EL235, 333 Ravenswood Ave., Menlo Park, CA 94025, email: ⟨samarati@csl.sri.com⟩ (on leave from Università di Milano).

1. INTRODUCTION

Discretionary access control policies ensure protection of information by controlling each request against authorizations specifying the accesses to be allowed or denied. Access control based on discretionary authorizations, while being very flexible, is vulnerable to Trojan Horses. The reason of this weakness is that no control is enforced on the use or dissemination of the information once this information is released to an authorized user or, more precisely, to a process running on his/her behalf. It is thus possible for a process to write information into objects accessible by users not authorized to access the objects from which the information has been read. Mandatory policies prevent such illegal flows through the use of labels. Each subject and object is assigned a label. Read and write operations are allowed only if the labels of the involved subject and object satisfy specific constraints [Bell and LaPadula 1976; Castano et al. 1995]. The use of labels, however, severely reduces the flexibility proper of the discretionary access control.

Several proposals have tried to overcome the vulnerability of discretionary access control by complementing it with some form of flow control. Some approaches restrict programs to access only objects satisfying given patterns [Karger 1987] or owned by specific users [Boebert and Ferguson 1985]. Other approaches [Graubart 1989; McCollum et al. 1990; Stoughton 1981] associate with each object an access control list that is propagated together with the information in the object. Hence, information always carries along access restrictions specified on it. Specifically targeted to the protection against unauthorized leakage of information are the approaches proposed in [Samarati et al. 1997; Walter et al. 1974]. In these approaches a process can write an object o only if o is at least as protected in reading as all the objects read by the process up to that point. (An object o is at least as protected in reading as another object o' if the set of subjects allowed to read o is contained in the set of subjects allowed to read o'.) Intuitively, the policy behind these approaches, which we call *strict* policy, is based on the same principles as the mandatory policy. Access control lists are used in place of labels, and the inclusion relationship between sets is used in place of the dominance relationship between labels. Although the discretionary flexibility of explicitly specifying authorized accesses is not lost, the overall flexibility of the system is definitely reduced by the application of the strict policy: after having read an object o, a process is unable to write any object less protected in reading than o, even if this write operation would not result in any improper information leakage. The approach proposed in [Samarati et al. 1997], developed to support flow control in object-oriented systems, partially overcomes this problem in two ways. First, by supporting different interaction modes between objects, it does not force strictly sequential execution of operations. Second, it allows a method execution to specify that no information has to be returned to it if this information cannot be written in the object on which the execution operates (thus avoiding blocking subsequent write operations). These two aspects, although providing some flexibility, do not overcome the main limitation of the strict policy: information flow is controlled only by considering read and write operations without taking into account how the information has been obtained and/or transmitted.

None of the above proposals take into account an important aspect of the problem: whether or not a write operation required by a process may be allowed ultimately should depend on the information such an operation releases, not on the information accessed by the process. A process may access sensitive data and yet not release any sensitive information through the write operations it executes. Such write operations should therefore be allowed, thus representing an *exception* to the strict policy. On a different perspective, information released by a process may be more sensitive than the information that the process has accessed. Although they do not violate the strict policy, write operations releasing this sensitive information should be blocked, again representing an exception (restrictive in this case) to the strict policy.

Exceptions are very important in many practical applications where access control is used. Everyday activities often require the strict policy to be bypassed, and exceptions to the strict policy can be enforced without compromising security. As an example, consider a procedure accessing personal information regarding employees of an organization and returning the benefits to be granted to each employee. Possibly, the benefits of each employee can be released to users who are not authorized to read the information provided as input of the benefit calculation.

The development of an access control model with information flow control and exceptions entails several issues. The most difficult one is perhaps that extensive code verification is not feasible in practice. A possible solution is to confine exceptions to specific procedures, with reference to the information they return or they store. Even in the application of the mandatory policy, recent work has pointed out the need for supporting exceptions within the execution of specific trusted procedures [Informix Software, Inc. 1993; Oracle Corporation 1996; Sybase, Inc. 1993]. Other important issues concern the specification of such exceptions and the performance of the overall access control mechanism.

In this paper we propose a discretionary access control model with information flow control and exceptions. Each procedure may have associated a set of exceptions to the restrictions/permissions stated by the authorizations. Like previous proposals we couch our work in object-oriented systems. There are several advantages in considering object-oriented systems. The fact that all procedures are represented by methods associated with objects allows users to identify specific pieces of *trusted* code for which exceptions can be enforced. Moreover, the specification of exceptions for specific methods naturally fits with the encapsulation and modularity principles of object-oriented systems. We consider two kinds of exceptions: *invoke exceptions*, applicable within a method execution, and *reply exceptions*, applicable to the information returned by a method. Exceptions can be *permissive* or *restrictive*. A permissive exception overrides a restriction imposed by the strict policy, thus permitting an information flow which would otherwise be blocked. A restrictive exception overrides a permission stated by the strict policy, thus forbidding an information flow which would otherwise be allowed. It is important to note that in this paper we consider only protection from explicit flows due to the execution of read and write operations. We do not consider implicit flows which can be enforced by sophisticated forms of covert channels (like timing or storage channels) [Kemmerer 1982]. It is also important to note that we are only concerned with controlling information flow from the point of view of protecting the confidentiality of the information (thus ensuring that sensitive information is not leaked to users not allowed to access it). We are not concerned with the integrity of the information, that is, of whether a write operation enforcing information flow executes a proper update. An approach similar to ours and controlling indirect write operations caused by information flow can be applied for that.

The remainder of the paper is organized as follows. Section 2 describes the object-oriented model to which our flow control is applied and introduces the notations that will be used throughout the paper. Section 3 illustrates a motivating example. Section 4 formally defines information transmission and flow. Section 5 describes security specifications and their semantics. Section 6 formally defines safe information flow. Section 7 illustrates how unsafe flows can be controlled by a message filter intercepting every message exchanged between objects and presents an algorithm enforcing it. Section 8 illustrates the techniques that we have devised to efficiently implement the message filtering algorithm and reports some performance results of the message filter controls. Section 9 illustrates the administrative policy regulating the specification of exceptions and discusses possible approaches to method certification. Finally, Section 10 presents the conclusions. The Appendix reports the proofs of the theorems stated in the paper.

2. BASIC ASSUMPTIONS, NOTATIONS, AND DEFINITIONS

We characterize the reference object-oriented data model and introduce definitions and notations which will be used throughout the paper [Samarati et al. 1997].

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. Let I be a set of *object identifiers*, A a set of *attribute names*, M a set of *method names*, and V a set of *values* defined as $V = D \cup I$, where D is a finite set of domains denoting primitive objects (e.g., integer, real, string). An object is formally defined as follows.

Definition 1. (*Object*) An *object* o is either a primitive object or a 4-tuple (*oid*, *attrs*, *vals*, *meths*), where $oid \in I$, $attrs = (a_1, \ldots, a_n)$, with $a_j \in A$ for all $j = 1, \ldots, n$, $vals = (v_1, \ldots, v_k)$, with $v_j \in V$ for all $j = 1, \ldots, k$, and $meths \subseteq M$.

Objects interact with each other via message exchanges. Whenever a message is received by an object, a method lookup is performed to determine the method to be executed. After the execution of the method, a reply is eventually returned to the sender of the message. Messages can be sent under any of the following interaction modes.

—*Synchronous* (`s`). The sender e_h of the message requiring execution e_k stops executing and waits for the invoked execution to terminate and the reply to return. Upon reception of the reply, the sender resumes.

—*Asynchronous* (`a`). The sender e_h of the message requiring execution e_k immediately receives a null reply (`nil`) from the system and continues executing independently from e_k. No reply is sent by e_k to e_h.

Syntax	Description
$(\texttt{read},(a_j),\texttt{s},rp)$	It returns the value of attribute a_j, if a_j is an attribute of the receiver object; it returns `failure` otherwise.
$(\texttt{write},(a_j,v_j),\texttt{s},rp)$	It assigns the value v_j to attribute a_j, if a_j is an attribute of the receiver object; it returns `failure` otherwise.
$(\texttt{create},(v_1,\ldots,v_n),\texttt{s},rp)$	It creates a new object and assigns the values $(v_1,\ldots,v_n)$ to its attributes; it returns the oid of the new object, if the creation succeeds; it returns `failure` otherwise.
$(\texttt{get},mid,\texttt{s},rp)$	It returns the reply of the execution invoked by the message whose identifier is mid.

Table 1. Primitive messages

—*Deferred reply* (`dr`). Upon sending the message, the sender e_h 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 invoked execution is returned upon explicit request (`get` message) of e_h.

If a message is sent requiring the execution of a method in a given interaction mode md, we say that the method is invoked and executed under mode md. The invoker object specifies the interaction mode when sending the message. If nothing is specified, the synchronous mode is assumed. Interaction modes can be changed by modifying the method code and re-compiling it.

Messages are formally defined as follows.

Definition 2. (*Message*) A *message* is a 4-tuple (*name,pars,mode,reply*), where $name \in M$ is the name of the invoked method, $pars = (par_1,\ldots,par_k)$, with $par_j \in V \cup A$ for all $j = 1,\ldots,k$, is an ordered set of values called message parameters, $mode \in \{\texttt{s},\texttt{a},\texttt{dr}\}$ is the mode under which the corresponding method must be executed, and $reply = (rp_1,\ldots,rp_n)$, with $rp_j \in V \cup \{\texttt{nil},\texttt{failure},\texttt{success}\}$ for all $j = 1,\ldots,n$, is a sequence of return values.

Read, write, and create operations, and retrieval of replies for executions invoked under deferred reply mode are enforced by means of *primitive messages* invoking the corresponding system-defined *primitive* methods. Primitive methods are elementary in the sense that they cannot invoke any other methods, that is, they cannot send any messages. Primitive messages, which are summarized in Table 1, must always be invoked under the default synchronous interaction mode.

Each activity in the system is started by a message sent by a user to an object. We term *transaction* the set of all method executions invoked (directly or indirectly) as a consequence of the receipt of a message sent by a user. The user who sends the message is the *initiator* of the transaction. Each method execution (execution, for short) is associated with a pair (m,o), where m is the method being executed, and o is the object on which m is executed, respectively. We write $(m,\_)$ when the object is not relevant for the discussion. Given an execution e_h, we denote with $m(e_h)$ and $o(e_h)$ the method being executed and the object on which the method is executed, respectively. We use `r`, `w`, `c`, and `g` to denote the execution of `read`, `write`, `create`, and `get` methods, respectively. We denote the fact that execution e_h invokes execution e_k by an arrow connecting e_h to e_k with a superscript indicating the interaction mode. Thus, $\xrightarrow{\texttt{s}}$, $\xrightarrow{\texttt{a}}$, and $\xrightarrow{\texttt{dr}}$ denote synchronous, asynchronous, and deferred reply invocation, respectively. We omit the specification of the interaction mode when its value is not of interest in the explanation.

In the remainder of this paper, we assume that every execution e_i invoking a deferred reply execution e subsequently invokes the corresponding request for reply, which we denote with $\widehat{e}$. We do not consider $\widehat{e}$ as an actual execution by itself, but rather as a completion of e.

We define two order relationships on executions.

Definition 3. (*Invocation order*) Let e_h and e_k be two executions invoked by the same execution e_j. We say that e_h ***is invoked before*** e_k, written $e_h <_i e_k$, if the message requiring the invocation of e_h is sent before the message requiring the invocation of e_k.

Note that since executions can run under asynchronous or deferred reply modes, the order in which executions are invoked does not necessarily coincide with the order in which 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*) Let e_h and e_k be two executions, invoked by the same execution e_j, such that $e_h <_i e_k$. We say that e_h ***is executed before*** e_k, written $e_h <_e e_k$, if and only if any of the following conditions holds

(1) $e_j \stackrel{\mathtt{s}}{\rightarrow} e_h$;

(2) $e_j \stackrel{\mathtt{dr}}{\rightarrow} e_h \ \wedge \ \widehat{e_h} <_i e_k$.

The above definition states that if execution e_j invokes e_h and, subsequently, e_k then e_h is executed before e_k if: 1) e_h is invoked synchronously, or 2) e_h is invoked under deferred reply mode and the message requiring the reply of e_h is sent before invoking e_k. We graphically represent a transaction by a *transaction execution tree*. The root of the tree is the execution invoked by the message sent by the initiator of the transaction. Each execution e_h invoked by execution e_j is a child of e_j in the tree. The arc connecting e_j to e_h is labeled with the interaction mode under which e_h is invoked. To preserve the order between invocations, for any two executions e_h and e_k, if $e_h <_i e_k$, e_h appears on the left of e_k in the tree. Note that since no messages can be sent as part of the execution of primitive methods, primitive executions are always leaves of the tree.

3. MOTIVATING EXAMPLE

To illustrate the need of exceptions to the strict policy and how our approach enforces them we refer to the following example. We consider an object-oriented database storing information regarding an organization (departments, employees, projects). Figure 1 illustrates the schema of this database. In the figure, an arrow between an attribute a and a class c indicates that a has domain c. Also, $*$ denotes multi-valued attributes. We assume that at the end of the month method `MonthlyReport` is invoked on instances of class `Department` to produce reports on the activities of the departments. `MonthlyReport`, in turn, invokes methods `PrjReport`, `WrkReport`, and `ExpReport` whose execution queries related objects to produce summary information about projects, work activities, and expenses of the

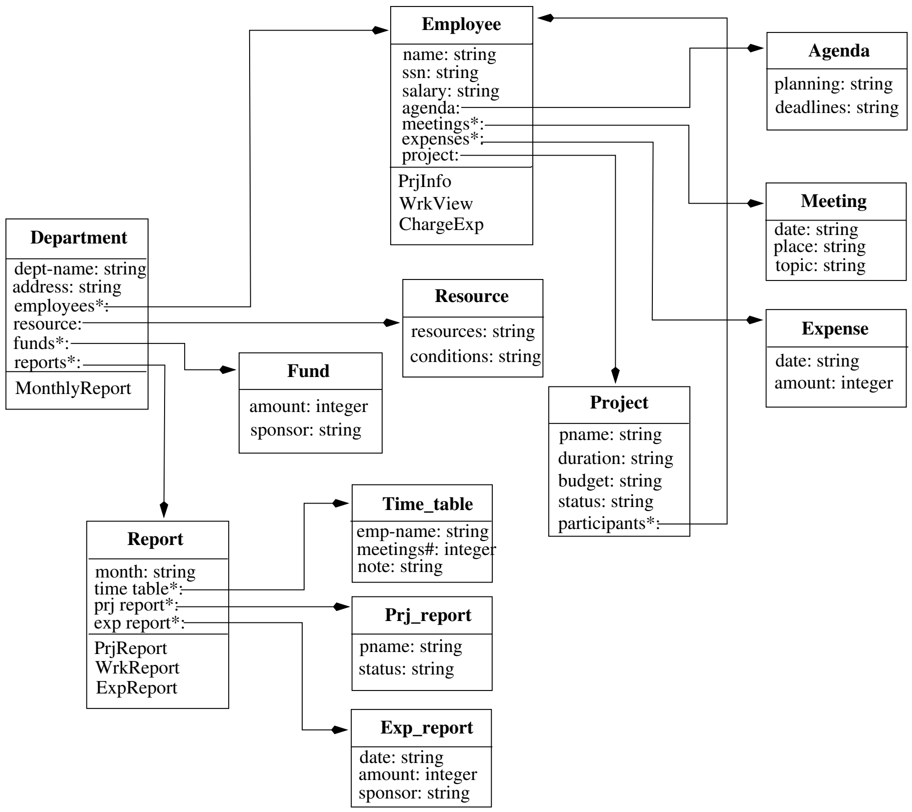

Fig. 1. A sample source schema

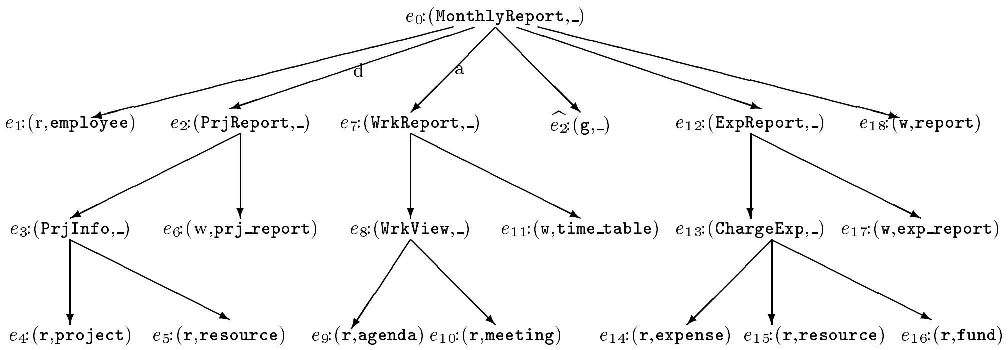

Fig. 2. An example of transaction execution tree

department employees. Figure 2 illustrates a possible execution tree for such a transaction where, for the sake of simplicity, we assume the existence of a single department and a single employee, `jim`. In particular, with reference to the schema of Figure 1, each class has only one instance, which, for simplicity, we denote with the same name as the class. To keep Figure 2 simple, no label is associated with arcs corresponding to the default synchronous mode invocation. Let us briefly discuss how exceptions may need to be enforced in this transaction.

Suppose that `agenda`, `meeting`, `expense`, and `project` can be accessed only by `jim`, while all the objects containing information at the department level can be accessed also by other users. According to the strict policy, the transaction of Figure 2 will never be able to complete successfully: the write operations would always be blocked because they are requested after some read operations on more protected objects. However, these write operations may not necessarily actually enforce unauthorized information disclosure. In particular, the information that `PrjInfo` returns may be less sensitive than the information read from `project` and could in this case be returned to some users not allowed to directly read `project`. A reply exception can be used to specify that the information that `PrjInfo` returns can be released to such users. As a result, subsequent write operations on objects accessible by the specified users, which would be denied under the strict policy, would be allowed. A condition may also be associated with the exception. For instance, the exception may be applicable only if the execution has not read attribute `budget` of object `project`. While reply exceptions apply to the information that a method execution returns through the reply, invoke exceptions apply to the information that a method execution passes to the execution it invokes. For instance, the information `MonthlyReport` passes to the write execution on object `report` (execution e_{18}) may be less sensitive than the information it has read or received by other executions. An invoke exception can capture this concept. Exceptions can also be used to deny write operations which obey the strict policy. Consider for instance method `ChargeExp`, which determines and returns the funds to which the expenses must be charged. Suppose that the assignment of the expenses to the funds is determined in the code itself. In this case, the information that `ChargeExp` returns may be more sensitive than the information read and may not be released to some users, even if these users are allowed to read the objects read by the execution. Such a situation can be modeled by attaching to method `ChargeExp` a restrictive reply exception for these users on the objects read.

One may wonder whether the application of a more sophisticated finer-grained form of access control would suffice to satisfy the requirements that our exception-based approach addresses. The answer is no. The important point and expressive advantage of exceptions is to explicitly consider the procedures with which information is accessed, transmitted, and released, and consequently grant or deny an information flow accordingly. This is not possible through the specification of authorizations on objects or their attributes. For instance, suppose that the names of a project's `participants` cannot be released to some users but the information of the total number of them can. This information, which can be known by reading attribute `participants`, is therefore less sensitive than the value of the attribute itself. If method `PrjInfo` is trusted not to release the value of attribute `participants` but only the total number of them, then the information `PrjInfo` returns can be released to users not allowed to read the value of attribute `participants`. This situation, which is easily modeled through exceptions as illustrated above, cannot be modeled through authorizations, even if applied at the attribute level. Note also that conditions on attributes are not to be interpreted as a means to enforce attribute-level authorizations. The semantics of the conditions is simply to restrict the applicability of the exceptions. They may also serve as a way to make sure that an exception is applied only if the method has not read given attributes, instead of trusting the method to not do so (see Section 9).

4. INFORMATION TRANSMISSION AND FLOW

We characterize the transmission of information between executions and the consequent flow of information that can be enforced between objects.

4.1 Information transmission

Each message between objects results in a *bi-directional* 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 enforce uni-directional transmission of information, from the sender to the receiver. Transmission of information between two executions can also be carried out without any message exchanges between the two (indirect transmission). For instance, if execution e_i invokes execution e_h which in turn invokes execution e_k, there is a direct transmission, through the message parameters, from e_i to e_h and from e_h to e_k and, as a consequence, an indirect transmission from e_i to e_k. Similarly, if executions e_h and e_k are not asynchronous, there is a direct transmission, through the replies, from e_k to e_h and from e_h to e_i, and thus an indirect transmission from e_k to e_i.

We call *channel* a direct or indirect transmission of information. Depending on the "direction" along which information is transmitted, we distinguish between *forward* and *backward* channels.

Definition 5. (*Forward channel*) Let e_h and e_k be two executions in a transaction T. There exists a *forward channel* from e_h to e_k if and only if any of the following conditions holds

(1) $e_h \rightarrow e_k$;

(2) there exists an execution e_i in T such that $e_h \rightarrow e_i$ and there exists a forward channel from e_i to e_k.

Definition 6. (*Backward channel*) Let e_h and e_k be two executions in a transaction T. There exists a *backward channel* from e_h to e_k if and only if any of the following conditions holds

(1) $e_k \stackrel{\texttt{s}}{\rightarrow} e_h$ or $e_k \stackrel{\texttt{dr}}{\rightarrow} e_h$;

(2) there exists an execution e_i in T such that there exists a backward channel from e_h to e_i and either $e_k \stackrel{\texttt{s}}{\rightarrow} e_i$ or $e_k \stackrel{\texttt{dr}}{\rightarrow} e_i$.

The channels in the above definitions are uni-directional, in the sense that each of them considers only one direction in which information can be transmitted. A transmission can exploit both backward and forward channels. For instance, if an execution e_i invokes an execution e_h and then, after receiving e_h's reply, invokes execution e_k, there is a transmission of information from e_h to e_k due to the combination of the backward channel from e_h to e_i and the forward channel from e_i to e_k. The following definition covers all transmission channels between executions.

Definition 7. (*Transmission channel*) Let e_h and e_k be two executions in a transaction T. There exists a *transmission channel* from e_h to e_k if and only if any of the following conditions holds

(1) there exists a forward channel from e_h to e_k;
(2) there exists a backward channel from e_h to e_k;
(3) there exists a backward channel from e_h to e^* and a forward channel from e^* to e_k, and $e'_h <_e e'_k$; where e^* is the least common ancestor of e_h and e_k in the execution tree of T and e'_h and e'_k are the ancestors of e_h and e_k, respectively, invoked by e^*.

It is easy to see that the above definition implies that for any pair of executions e_h and e_k there exists at most one transmission channel from e_h *to* e_k (of course there can also be the reverse channel from e_k to e_h). We denote such a channel with the pair (e_h, e_k). We denote with *ch-back*(e_h, e_k) and *ch-forw*(e_h, e_k) the sequence of executions enforcing backward transmission and forward transmission, respectively.

Example 1. Consider the transaction illustrated in Figure 2. Some channels in it are as follows: e_4 to e_0 (backward), e_0 to e_{17} (forward), e_4 to e_{17} (combination of the previous backward and forward channels). Moreover, *ch-back*$(e_4, e_{17}) = (e_4, e_3, e_2)$, and *ch-forw*$(e_4, e_{17}) = (e_0, e_{12}, e_{17})$.

Note that transmission channels in Definition 7 capture the *potential* transmission of information that can occur between two executions e_h and e_k. Transmission is only potential because the information received by e_k through the channel may actually be independent from the information transmitted from e_h. Determining whether a channel corresponds to an actual transmission of information would require analyzing how the information received by e_k has been obtained [Denning 1976]. This analysis is outside the scope of this paper. We therefore make the pessimistic assumption that all transmission channels result in transmission of information. Under this assumption, the following theorem holds.

THEOREM 1. *Let e_h and e_k be two executions in a transaction T. There exists transmission of information from e_h to e_k if and only if there exists a transmission channel from e_h to e_k.*

4.2 Information flow

There exists an information flow between an object o_h and an object o_k when information about the state of o_h is stored into o_k. An information flow between two objects obviously requires a transmission of information between some executions operating on the two objects. However, a transmission of information between executions operating on different objects does not necessarily enact a flow. 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 [Jajodia and Kogan 1990] different assumptions are made. In particular, an information flow from an object may simply arise if the object sends, or replies to, some messages, regardless of whether the object has been read. Our assumption is based on the fact that the information about the values of an object's attributes can be known only by reading the object. Hence, if no read operation is executed on the object, no flow from the object can be enacted. To ensure that no information about the values of an object's attributes can be hidden inside methods' specifications, we do not allow methods to change their own code or the code of other methods. Changes to methods' code must be requested directly by the users (*trusted path*)[1]. Analogously, an object can acquire information only by changing its internal state (i.e., by writing some of its attributes). Thus, if no changes occur in an object, no information flow to the object can be enacted. Summarizing, information flow from an object o_h to another object o_k requires: *1)* a *read* operation on the "source" object o_h, *2)* a *write/create* operation on the "destination" object o_k, and *3)* a *transmission channel* from the read to the write/create execution. This is formalized by the following definition.

Definition 8. (*Information flow*) Let e_h be an execution on object o_h and let e_k be an execution on object o_k in a transaction T. There exists an information flow from o_h to o_k enacted by e_h and e_k, written (o_h, e_h, e_k, o_k), if and only if all the following conditions hold

(1) $m(e_h) = \texttt{read}$;
(2) $m(e_k) = \texttt{write}$ or $m(e_k) = \texttt{create}$;
(3) there exists a transmission channel from e_h to e_k.

In the following, given a transaction T, $FLOWS(T)$ denotes the set of all flows in T. Figure 3 lists all the flows enacted by the transaction in Figure 2.

It is important to note that our definition of flow also captures flows which can be generated by the evaluation of conditions (e.g., in **if-then-else** statements [Denning 1982]) on an object's attributes. As a matter of fact, evaluating a condition on the value of an attribute requires a read operation on the attribute or, more precisely, on the object to which the attribute refers. Hence, if a condition on an attribute is evaluated in a method execution, the object is read and the information contained in the object is considered to be transmitted to subsequent executions.

5. SECURITY SPECIFICATIONS

We illustrate how *security specifications*, that is, accesses to be or not to be allowed, and exceptions are stated in our model. Like other approaches, we associate with each object an *access control list* (ACL) listing the users authorized to access the object. For the sake of simplicity, we consider a single access control list and do not distinguish between read and write authorizations.

In the following, we assume that the discretionary access control is always performed on write and create operations, which are therefore taken into consideration

[1]Improper modifications to code can be controlled, for instance, by allowing only valid compilers, created with a privileged action, to create and modify methods [Bingham 1974; Wilkinson et al. 1981] or by applying integrity checksum techniques [Denning 1982].

o_h	e_h	e_k	o_k
project	e_4	e_6	prj_report
project	e_4	e_{17}	exp_report
project	e_4	e_{18}	report
resource	e_5	e_6	prj_report
resource	e_5	e_{17}	exp_report
resource	e_5	e_{18}	report
agenda	e_9	e_{11}	time_table
meeting	e_{10}	e_{11}	time_table
expense	e_{14}	e_{17}	exp_report
expense	e_{14}	e_{18}	report
resource	e_{15}	e_{17}	exp_report
resource	e_{15}	e_{18}	report
fund	e_{16}	e_{17}	exp_report
fund	e_{16}	e_{18}	report
employee	e_1	e_6	prj_report
employee	e_1	e_{17}	exp_report
employee	e_1	e_{18}	report

Fig. 3. Flows (o_h, e_h, e_k, o_k) enacted by the transaction in Figure 1

by the information flow control only if authorized. By contrast, we do not require discretionary access control on read operations. We instead see the release of the transaction reply to the initiator analogous to a write operation, and control the flow enforced by it accordingly. The reason for this approach is flexibility. In particular, in this way a transaction can be allowed to read objects and operate on information which the initiator cannot access. The flow control ensures that this information will not be released to the initiator unless an exception is in place. In our model each method m can have associated two sets of exceptions: *reply exceptions* and *invoke exceptions*. The reply exceptions are applied on the information that m returns by replying to its invoker. The invoke exceptions are applied on the information that m passes to other executions through invocation. Reply and invoke exceptions can be either *permissive* or *restrictive*. Permissive exceptions relax the restrictions imposed by the strict policy. Restrictive exceptions impose, instead, additional restrictions. Intuitively, a permissive exception applies when the information returned or passed by an execution is less sensitive than the information that the execution has potentially received by its invoker or by the executions it has invoked. A restrictive exception applies when the execution itself adds sensitivity to the information. Of course, not all methods will have exceptions attached to them, only privileged methods will. Moreover, since exceptions can allow information flows which would otherwise be blocked, or block information flows which would otherwise be allowed, methods with exceptions must be trusted for the information they return or pass to other methods. We will illustrate possible approaches to trusted method certification in Section 9. Exceptions can have conditions associated with them: an exception is applicable *if and only if* the condition associated with it is verified. The conditions on exceptions specified for a method m are expressed on the basis of the set O_READ(m) of objects and on the set A_READ(m) of attributes read during the execution of m. Conditions can require, for instance, that the execution had (not) read some attributes or objects, or had (not) read

```
⟨exception_cond⟩    ::=  ⟨simple_cond⟩ | ⟨exception_cond⟩ ∧ ⟨exception_cond⟩|
                         ⟨exception_cond⟩ ∨ ⟨exception_cond⟩
⟨simple_cond⟩       ::=  ⟨meth_o_read⟩ ⟨incl_op⟩ set_of_oids |
                         ⟨meth_o_read⟩ ⟨eq_op⟩ set_of_oids |
                         oid ⟨memb_op⟩ ⟨meth_o_read⟩ |
                         count(⟨meth_o_read⟩) ⟨comp_op⟩ nat_number |
                         count(⟨meth_o_read⟩) ⟨eq_op⟩ nat_number |
                         ⟨meth_a_read⟩ ⟨incl_op⟩ ⟨set_of_attr_expr⟩ |
                         ⟨meth_a_read⟩ ⟨eq_op⟩ ⟨set_of_attr_expr⟩ |
                         ⟨attr_expr⟩ ⟨memb_op⟩ ⟨meth_a_read⟩ |
                         count(⟨meth_a_read⟩) ⟨comp_op⟩ nat_number |
                         count(⟨meth_a_read⟩) ⟨eq_op⟩ nat_number
⟨attr_expr⟩         ::= oid.attr_name
⟨set_of_attr_expr⟩  ::= {⟨attr_expr⟩} | {⟨attr_expr⟩, ⟨set_of_attr_expr⟩}
⟨incl_op⟩           ::= ⊆ | ⊈
⟨eq_op⟩             ::= = | ≠
⟨memb_op⟩           ::= ∈ | ∉
⟨comp_op⟩           ::= > | < | ≥ | ≤
⟨meth_o_read⟩       ::= O_READ(meth_name)
⟨meth_a_read⟩       ::= A_READ(meth_name)
```

Fig. 4. Syntax of the language to express exception conditions

only given attributes or objects.

Figure 4 reports the syntax, in BNF form, of the language to express conditions. Terminal symbols are as follows: `set_of_oids` and `oid` denote any element in 2^I and I, respectively, `nat_number` denotes any natural number, `attr_name` denotes any element in A, and `meth_name` denotes any element in M. To uniquely identify attributes in the system, we express them in the form `oid.attr_name`, where `oid` is the identifier of an object and `attr_name` is the name of any of its attributes.

Example 2. The following are examples of conditions:

(1) `resource` $\notin$ O_READ(`PrjInfo`). It is satisfied for an execution of `PrjInfo` if the execution has not read object `resource`.

(2) O_READ(`WrkView`) $=$ {`meeting,agenda`} $\wedge$ `meeting.place` $\notin$ A_READ(`WrkView`). It is satisfied for an execution of `WrkView` if the execution has read only objects `meeting` and `agenda` and has not read attribute `place` of object `meeting`.

In the following, $\mathcal{C}$ denotes the set of conditions that can be specified, that is, conditions expressible through the language given, and $\mathcal{U}$ the set of users in the system. Exceptions associated with methods are formally defined as follows.

Definition 9. (*Method's Exception*) A method's exception is a 4-tuple (C, o, U, tp) where:

—$C \in \mathcal{C} \cup \{true\}$ is the condition specifying when the exception is applicable;
—$o \in I$ is the identifier of the object for which the exception is specified;
—$U \subseteq \mathcal{U}$ is the set of users for which the exception is specified;

Access Control Lists

ACL(`employee`) $=$ {`ada`,`jim`,`meg`}
ACL(`agenda`) $=$ ACL(`meeting`) $=$ {`bob`,`jim`}
ACL(`expense`) $=$ ACL(`project`) $=$ {`jim`}
ACL(`time_table`) $=$ ACL(`prj_report`) $=$ {`ada`,`bob`,`jim`,`sam`,`tom`}
ACL(`resource`) $=$ ACL(`fund`) $=$ {`ada`,`bob`,`jim`,`meg`,`sam`,`tom`}
ACL(`report`) $=$ ACL(`exp_report`) $=$ {`ada`,`meg`}

Exceptions

RE(`WrkView`) $=$ {({*true*},{`agenda`},{`ada`,`sam`,`tom`},+), ({*true*},{`meeting`},{`ada`,`sam`,`tom`},+)}
RE(`ChargeExp`) $=$ {({*true*},{`expense`},{`ada`,`meg`},+), ({*true*},{`fund`},{`bob`,`jim`,`sam`,`tom`},−)}
RE(`PrjInfo`) $=$ {{`resource` $\notin$ O_READ(`PrjInfo`)},{`project`}, {`ada`,`bob`,`sam`,`tom`},+)}
IE(`MonthlyReport`) $=$ {({*true*},{`project`},{`ada`,`meg`},+), ({*true*},{`employee`},{`bob`,`sam`,`tom`},+)}

Fig. 5. An example of security specifications

—$tp \in \{+,-\}$ is the type (permissive vs. restrictive) of the exception.

An exception with a *true* condition is an exception which is always applicable.

The semantics of the set of reply, $RE(m)$, and invoke, $IE(m)$, exceptions associated with a method m is as follows.

—A reply exception $(C, o, U, tp) \in RE(m)$ states that, if condition C is satisfied, the restrictions imposed by the strict policy are waived (if tp = '+'), or restricted (if tp = '−') for users in U with reference to the information returned by m and which has been read from object o during m's execution. Intuitively, if C is satisfied the information in o can be released (if tp = '+'), or cannot be released (if tp = '−'), to users in U through m's reply, regardless of whether users in U are authorized to read o.

—An invoke exception $(C, o, U, tp) \in IE(m)$ states that, if condition C is satisfied, the restrictions imposed by the strict policy are waived (if tp = '+'), or restricted (if tp = '−'), for read operations on object o executed before or during m's execution, and write operations executed by m, with respect to users in U. In other words, if C is satisfied, method m is allowed (if tp = '+'), or denied (if tp = '−'), to write information read from object o into objects accessible by users in U, regardless of whether these users appear in the ACL of o. Note that the exception only applies to users in U. The strict policy still applies to other users.

In the following, we denote with $RE(m).O$ the set of the object identifiers in $RE(m)$, that is, the objects on which a reply exception is specified for m. Similarly, we denote with $IE(m).O$ the set of object identifiers in $IE(m)$.

Example 3. Figure 5 illustrates an example of security specifications referred to the objects and methods of the example illustrated in Section 3. Here `ada`, `bob`, `jim`, `meg`, `sam`, and `tom` are all employees of the department. In particular, `ada` is the department's director, `bob` is the collaborator of `jim`, and `meg` is the administrative supervisor. The ACLs and exceptions reported in Figure 5 reflect this scenario. The ACLs are of immediate interpretation. Let us discuss the exceptions. Method `WrkView` can always release, through its reply, information read from objects `agenda` and `meeting` to `ada`, `sam`, and `tom`. `ChargeExp` can release information read in `expense` to `ada` and `meg`, but cannot release information read in `fund` to `bob`, `jim`, `sam`, and `tom` (see discussion in Section 3). Method `PrjInfo` can release information read in `project` to `ada`, `bob`, `sam`, and `tom`, provided that object `resource` has not been read during execution. Finally, `MonthlyReport` can release information read from `project` to `ada` and `meg`, and information read from `employee` to `bob`, `sam`, and `tom`.

Note that the possibility of expressing both permissive and restrictive exceptions may lead to conflicts between exceptions specified for the same method. Intuitively, a conflict arises when, with reference to a method execution, both a restrictive and a permissive exception can be applied with respect to a given object for a given user. For instance, the set of exceptions $\{(\{\mathit{true}\},\{\texttt{fund}\}, \{\texttt{jim,sam}\},-)$, $(\{\text{count}(\text{O\_READ}(\texttt{ChargeExp}) > 3\}, \{\texttt{fund}\}, \{\texttt{jim}\}, +)\}$ produces a conflict when the condition associated with the permissive exception is satisfied. The specification of conflicting exceptions should always be avoided. We note, however, that many exceptions which are potentially in conflict may actually only seldom result in a conflict at run time. Moreover, requiring their reformulation such that they never conflict (by appropriately restating the conditions in them) may complicate the administrative task. For this reason, we do not explicitly forbid conflicting or potentially conflicting exceptions. We provide instead a conflict resolution policy to solve conflicts at run time. In particular, we adopt the denials-take-precedence policy: in case of a conflict the restrictive exception prevails. The reason for this solution is that in case of a conflict the safest choice, with respect to information release, is taken.

6. SAFE INFORMATION FLOW

We define safe information flows, that is, information flows which do not leak information to users who are not authorized to read it according to the security specifications. A flow of information from an object to another object obeys the security specifications if and only if the users authorized to read the object being written are allowed to access the information flowing into it, either by explicit authorizations in the ACL or by permissive exceptions encountered during information transmission, and there is no explicit denial to release the information to them caused by restrictive exceptions encountered during information transmission.

Since exceptions can be subject to conditions, we need to distinguish the exceptions specified for a method from the exceptions actually applicable to a particular execution of this method. Exceptions of an execution are maintained in a compact form by associating with each object o for which an exception must be enforced the sets DU (*denied users*) and AU (*allowed users*) of users for which a restrictive and permissive, respectively, exception on o applies.

Definition 10. (*Execution reply exceptions*) Let e be an execution. The set of reply exceptions associated with e, denoted by $RE(e)$, is a set of triples $\{(o_1, DU_1, AU_1), \ldots, (o_k, DU_k, AU_k)\}$, where $o_i \in RE(m(e)).O$, $i = 1, \ldots, k$, and DU_i and AU_i are obtained as follows. $DU_i = \{U \mid (C, o_i, U, -) \in RE(m(e))$ and C is satisfied$\}$, and $AU_i = \{U \mid (C, o_i, U, +) \in RE(m(e))$, and C is satisfied$\} \setminus DU_i$.

Definition 11. (*Execution invoke exceptions*) Let e be an execution. The set of invoke exceptions associated with e, denoted by $IE(e)$, is a set of triples $\{(o_1, DU_1, AU_1), \ldots, (o_k, DU_k, AU_k)\}$, where $o_i \in IE(m(e)).O$, $i = 1, \ldots, k$ and DU_i and AU_i are obtained as follows. $DU_i = \{U \mid (C, o_i, U, -) \in IE(m(e))$ and C is satisfied$\}$, and $AU_i = \{U \mid (C, o_i, U, +) \in IE(m(e))$, and C is satisfied$\} \setminus DU_i$.

Note that, according to Definitions 10 and 11, the set of reply/invoke exceptions associated with an execution e contains at most one tuple for each object. Note also that the denials-take-precedence conflict resolution policy is enforced in the definition of the exceptions by subtracting the set of users to whom a negative exception applies (i.e., appearing in DU_i) when constructing AU_i. This ensures that for each triple (o_i, DU_i, AU_i) in the set of exceptions, DU_i and AU_i are disjoint. In the following, given a set $E = \{(o_1, DU_1, AU_1), \ldots, (o_m, DU_m, AU_m)\}$ of execution exceptions, we denote with $\Pi_{o_i}(E)$ the pair (DU_i, AU_i) of users to whom a permissive/restrictive exception on o_i applies. For instance, with reference to Figure 5, $\Pi_{\texttt{fund}}(RE(e_{13})) = (\{\texttt{bob,jim,sam,tom}\},\{\emptyset\})$ and $\Pi_{\texttt{project}}(IE(e_0)) = (\{\emptyset\},\{\texttt{ada,meg}\})$.

Information transmitted along a channel is subject to the exceptions (reply or invoke, depending on the direction along which the information is transmitted) associated with each execution in the channel. The set of users to whom the information transmitted along a channel can be released can therefore increase (decrease) due to permissive (restrictive) exceptions encountered during the transmission.

Applying an execution exception to the information read from an object means adding to the set of users to whom the information can be released those users to whom a permissive exception applices (i.e., the users belonging to AU) and removing from them the users to whom a restrictive exception applies (i.e., the users belonging to DU). We represent this combination by means of operator $\pm$, defined as follows. Let o be an object, U the set of users to whom the information in o can be released, and (o,DU,AU) an exception applicable to o. The set of users to whom the information can be released after the enforcement of (o,DU,AU) is $U \pm (DU, AU) = (U \cup AU) \setminus DU$.

We are now ready to formally define safe information flows.

Definition 12. (*Safe information flow*) Let $f = (o_h, e_h, e_k, o_k)$ be an information flow with $\textit{ch-back}(e_h, e_k) = (e_1, \ldots, e_n)$ and $\textit{ch-forw}(e_h, e_k) = (e'_1, \ldots, e'_m)$. f is safe if and only if $\text{ACL}(o_k) \subseteq \text{ACL}(o_h) \pm \Pi_{o_h}(RE(e_1)) \pm \ldots \pm \Pi_{o_h}(RE(e_n)) \pm \Pi_{o_h}(IE(e'_1)) \pm \ldots \pm \Pi_{o_h}(IE(e'_m))$.

In other words, an information flow is safe if all the users allowed to read the object into which the information is written are allowed to see the information flowing into it (according to the ACL of the object from which the information has been read and the exceptions encountered along the transmission).

Example 4. Consider the transaction in Figure 2 and the security specifications in Figure 5. Flow $(\texttt{agenda}, e_9, e_{11}, \texttt{time\_table})$ is safe. In fact, $\text{ACL}(\texttt{time\_table}) = \{\texttt{ada,bob,jim,sam,tom}\} \subseteq \{\texttt{ada,bob,jim,sam,tom}\} = \text{ACL}(\texttt{agenda}) \pm \Pi_{\texttt{agenda}} (RE(e_9)) \pm \Pi_{\texttt{agenda}} (RE(e_8)) \pm \Pi_{\texttt{agenda}} (IE(e_7)) \pm \Pi_{\texttt{agenda}} (IE(e_{11}))$.

A write/create operation is *safe* if and only if all flows enacted by it are safe. This is formalized by the following definition.

Definition 13. (*Safe write/create*) Let e_k be a write or create execution on o_k in a transaction T. e_k is safe if and only if each flow $(o_h, e_h, e_k, o_k) \in FLOWS(T)$ is safe.

For instance, write execution e_6 is not safe since one of the flows it enacts, namely (`project`,e_4, e_6, `prj_report`), is not safe.

In the following section we illustrate an approach to ensuring that only executions that do not enact any unsafe flows are performed.

7. INFORMATION FLOW CONTROL

In our approach, a *message filter* [Jajodia and Kogan 1990] intercepts every message and its reply to guarantee that no unsafe flow takes place. To determine whether a write or create operation should be blocked, the message filter keeps track of the information transmitted between executions together with the users to whom the information can be released. Information transmitted and applicable exceptions are represented through *folders*, whose construction and use are illustrated here.

7.1 Construction of folders

Information transmitted between executions is denoted by using the objects from which the information has been read. To keep track of the users to whom information read from an object can be released, we introduce the concept of *folder*. A folder is a set of pairs (o,U), where o is an object identifier and $U \subseteq \mathcal{U}$ is a set of users. Set U associated with object o represents the set of users allowed to access information in o, according to o's ACL and the exceptions encountered along the transmission channels that the information read from o has traversed.

Intuitively, we can imagine that folders are passed between executions upon message exchanges. Folders are incrementally built by means of functions *release_back*(): $\mathcal{E} \times 2^{I\times 2^{\mathcal{U}}} \rightarrow 2^{I\times 2^{\mathcal{U}}}$ and *release_forw*(): $\mathcal{E} \times 2^{I\times 2^{\mathcal{U}}} \rightarrow 2^{I\times 2^{\mathcal{U}}}$, where $\mathcal{E}$ denotes the set of all possible executions. Given a folder F associated with an execution e at a given point, function *release_back*() determines the folder to be returned by e to its invoker; function *release_forw*() determines the folder to be passed by e to the execution it next invokes. Let $F = \{(o_1, U_1), \ldots, (o_n, U_n)\}$ be a folder and e an execution. Then,

$release\_back(e, F)$=$\{(o_1, U_1'), \ldots, (o_n, U_n')\}$, where U_i'=$U_i \pm \Pi_{o_i}(RE(e))$, $i = 1, \ldots, n$.

$release\_forw(e, F)$=$\{(o_1, U_1'), \ldots, (o_n, U_n')\}$, where U_i'=$U_i \pm \Pi_{o_i}(IE(e))$, $i = 1, \ldots, n$.

An execution e receives information, and corresponding folders, from the different executions it invokes. These folders must be combined to determine the folder associated with e. The information collected by e is the union of the information sent back to e by all the executions it has invoked. Accordingly, the set of objects in e's folder must be equal to the union of the set of objects appearing in the folders that e received by the executions it invoked. Let us now discuss how users are combined. Information read from an object can be released to a user only if the user is authorized to access the information along *all* the transmission channels from which the information is flowing. According to this principle, when combining folders returned by the executions it invokes, an execution e must consider the *intersection* of the users indicated in the different folders. To compute this combination of folders, we introduce the operation of *union-and*, denoted as

$\cup^\wedge$. Let $F'=\{(o'_1,U'_1),\ldots,(o'_m,U'_m,)\}$, $F''=\{(o''_1,\ U''_1),\ldots,(o''_n,U''_n)\}$ be two folders. $F' \cup^\wedge F'' = \{(o_1,U_1),\ldots,(o_k,U_k)\}$, where $\{o_1,\ldots,o_k\} = \{o'_1,\ldots,o'_m\} \cup \{o''_1,\ldots,o''_n\}$. Moreover, $U_i = U'_i$ if $o_i \in \{o'_1,\ldots,o'_m\} \setminus \{o''_1,\ldots,o''_n\}$; $U_i = U''_i$ if $o_i \in \{o''_1,\ldots,o''_n\} \setminus \{o'_1,\ldots,o'_m\}$; $U_i = U'_i \cap U''_i$ if $o_i \in \{o'_1,\ldots,o'_m\} \cap \{o''_1,\ldots,o''_n\}$.

We are now ready to define backward and forward folders.

Definition 14. (*Backward folder*) Let e_h and e_k be two executions such that $e_h \rightarrow e_k$. The folder transmitted from e_k to e_h through the message reply, denoted as BACK(e_k,e_h), is recursively defined as follows:

$$\mathrm{BACK}(e_k,e_h) = \begin{cases} \{(o(e_k),\mathrm{ACL}(o(e_k)))\} & \text{if } m(e_k) = \texttt{read} \\ \emptyset & \text{if } m(e_k) \in \{\texttt{write},\texttt{create}\} \\ & \quad \text{or } e_h \overset{\texttt{a}}{\rightarrow} e_k \\ release\_back(e_k, \bigcup_z^\wedge \mathrm{BACK}(e_z,e_k)) & \text{otherwise} \end{cases}$$

Definition 14 states that the folder transmitted from execution e_k to its invoker e_h is: *i)* the set composed of the pair (object read, its access control list), if e_k is a read execution, *ii)* the empty set, if execution e_k is either a write or create execution or it is invoked in asynchronous mode, *iii)* the result of the application of the reply exceptions of execution e_k to the *union-and* of the information e_k has received by the executions it has invoked, otherwise.

Definition 15. (*Forward folder*) Let e_h and e_k be two executions such that $e_h \rightarrow e_k$. The folder transmitted from e_h to e_k through the message requiring the invocation of e_k, denoted as FORW(e_h,e_k), is recursively defined as follows.

$$\mathrm{FORW}(e_h,e_k) = \begin{cases} \emptyset & \text{if } o(e_h) \text{ is a user} \\ release\_forw(e_h,F) & \text{otherwise} \end{cases}$$

where $F = \bigcup_z^\wedge\{\mathrm{BACK}(e_z,e_h) \mid e_z <_e e_k\} \cup^\wedge \mathrm{FORW}(e_j,e_h)$ and e_j is the execution that invoked e_h.

Definition 15 states that the folder transmitted from execution e_h to execution e_k it invokes is: *i)* the empty set, if e_k is invoked directly by a user[2], *ii)* the result of the application of the invoke exceptions of e_h to the information available to e_h at the time of e_k's invocation, otherwise. This information is equal to the *union-and* of the folder e_h received from its invoker with the folders that e_h has received, before invoking e_k, by the executions invoked.

Figure 6 shows the backward and forward folders for the transaction in Figure 2, assuming the security specifications in Figure 5. For the sake of simplicity, empty folders are omitted from the figure.

According to Definition 12, the set of users to whom the information in an object o_i can be released through a write/create operation is the set of all the users who belong to the ACL of o_i or to whom a permissive exception applied during information transmission, and for whom there does not exist an explicit denial to access information in o_i caused by a restrictive exception encountered during information

[2] 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.

Backward folders

BACK(e_1, e_0) $=$ {(employee,{ada,jim,meg})}
BACK(e_4, e_3) $=$ {(project,{jim})}
BACK(e_5, e_3) $=$ {(resource,{ada,bob,jim,meg,sam,tom})}
BACK(e_3, e_2) $=$ BACK(e_2, e_0) $=$ {(resource,{ada,bob,jim,meg,sam,tom}), (project,{jim})}
BACK(e_9, e_8) $=$ {(agenda,{bob,jim})}
BACK(e_{10}, e_8) $=$ {(meeting,{bob,jim})}
BACK(e_8, e_7) $=$ {(agenda,{ada,bob,jim,sam,tom}), (meeting,{ada,bob,jim,sam,tom})}
BACK(e_{14}, e_{13}) $=$ {(expense,{jim})}
BACK(e_{15}, e_{13}) $=$ {(resource,{ada,bob,jim,meg,sam,tom})}
BACK(e_{16}, e_{13}) $=$ {(fund,{ada,bob,jim,meg,sam,tom})}
BACK(e_{13}, e_{12}) $=$ BACK(e_{12}, e_0) $=$ { (resource,{ada,bob,jim,meg,sam,tom}), (expense,{ada,jim,meg}), (fund,{ada,meg})}

Forward folders

FORW(e_0, e_2) $=$ FORW(e_2, e_3) $=$ FORW(e_3, e_4) $=$ FORW(e_0, e_7) $=$ FORW(e_7, e_8) $=$ FORW(e_8, e_9) $=$ {(employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_3, e_5) $=$ {(project,{jim}), (employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_2, e_6) $=$ {(project,{jim}), (employee,{ada,bob,jim,meg,sam,tom}), (resource,{ada,bob,jim,meg,sam,tom})}
FORW(e_8, e_{10}) $=$ {(agenda,{bob,jim}), (employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_7, e_{11}) $=$ {(agenda,{ada,bob,jim,sam,tom}), (meeting,{ada,bob,jim,sam,tom}), (employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_0, e_{12}) $=$ FORW(e_{12}, e_{13}) $=$ FORW(e_{13}, e_{14}) $=$ {(project,{ada,jim,meg}), (resource, {ada,bob,jim,meg,sam,tom}), (employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_{13}, e_{15}) $=$ FORW(e_{13}, e_{16}) $=$ {(project,{ada,jim,meg}),(expense,{jim}), (resource, {ada,bob,jim,meg,sam,tom}), (employee,{ada,bob,jim,meg,sam,tom})}
FORW(e_{12}, e_{17}) $=$ FORW(e_0, e_{18}) $=$ {(project,{ada,jim,meg}), (expense,{ada,jim,meg,}), (resource,{ada,bob,jim,meg,sam,tom}), (fund,{ada,meg}), (employee,{ada,bob,jim,meg,sam,tom})}

Fig. 6. Backward and forward folders for the transaction in Figure 2 assuming the security specifications in Figure 5

transmission. The following theorem states that these users are exactly the users in set U_i associated with object o_i in the folder passed to the write/create operation.

THEOREM 2. *Let e_h and e_k be two executions in a transaction T such that $e_h \rightarrow e_k$, and e_k is a write/create execution on object o_k. e_k is safe if and only if $\forall (o_i, U_i) \in$ FORW(e_h, e_k): ACL(o_k) $\subseteq U_i$.*

Example 5. Consider the transaction in Figure 2, whose folders are reported in Figure 6, and the security specifications in Figure 5. Execution e_6=(w,prj_report) is unsafe because the condition of Theorem 2 is not satisfied for pair (project,{jim}) in FORW(e_2, e_6). In fact, ACL(prj_report)={ada,bob,jim,sam,tom} $\not\subseteq$ {jim}. All the other write/create executions in the transaction are safe.

7.2 The message filtering algorithm

We present an algorithm enforcing the message filter controls. The algorithm receives as input a message and determines how to deal with the message depending on the method to be executed, on the interaction mode, and on the flows that the

The message filtering algorithm

INPUT: A message $msg = (name,pars,mode,reply)$ sent by execution e_h, within a transaction initiated by user u, running on object o_h to require an execution e_k on object o_k.
OUTPUT: It allows e_k and returns its reply if e_k does not enforce unsafe flows. It blocks e_k and returns `failure` otherwise.

METHOD:

If $name \neq$ `get`
then $B_{e_k} := \emptyset$
  If $o_h = u$ /* the message is sent by a user to start a transaction */
  then $F_{e_k} := \emptyset$
    execute e_k
    $reply$:= reply from e_k
    If $\forall\ (o_i, U_i) \in B_{e_k}$: $u \in U_i$
    then return $reply$ **to** e_h
  else $F_{e_k} := release\_forw(e_h, F_{e_h} \cup^{\wedge} B_{e_h})$
endif
Case msg **of**
1) $(\texttt{read}, (a_j), mode, reply)$:
    execute e_k
    $reply$:= reply from e_k
    $B_{e_k} := \{(o_k, \mathrm{ACL}(o_k))\}$
    $B_{e_h} := B_{e_h} \cup^{\wedge} B_{e_k}$
    return $reply$ **to** e_h

2) $(\texttt{write}, (a_j, v_j), mode, reply)$:
    If $\forall (o_i, U_i) \in F_{e_k}$: $\mathrm{ACL}(o_k) \subseteq U_i$
    then execute e_k
      $reply$:= reply from e_k
    else $reply$:= `failure`
    return $reply$ **to** e_h

3) $(\texttt{create}, (v_1, \ldots, v_k), mode, reply)$:
    If $\forall (o_i, U_i) \in F_{e_k}$: $u \in U_i$
    then execute e_k
      let o be the object created by e_k
      $\mathrm{ACL}(o) := \{u\}$
      $reply$:= oid of o
    else $reply$:= `failure`
    return $reply$ **to** e_h

4) $(\texttt{get}, mid, mode, reply)$:
    retrieve $(mid,\ e_h,\ reply,\ B'_{e_k})$ from the reply queue
    $B_{e_h} := B_{e_h} \cup^{\wedge} B'_{e_k}$
    return $reply$ **to** e_h

5) *(name,pars,mode,reply)* and $name \notin \{\texttt{read}, \texttt{write}, \texttt{create}, \texttt{get}\}$:
  case $mode$ **of**
  a) `s`: execute e_k
    $reply$:= reply from e_k
    $B_{e_h} := B_{e_h} \cup^{\wedge} release\_back(e_k, B_{e_k})$
    return $reply$ **to** e_k

  b) `a`: **return** `nil` **to** e_h
    execute e_k
    discard reply from e_k

  c) `dr`: produce a message identifier mid
    return mid **to** e_h
    execute e_k
    $reply$:=reply from e_k
    $B'_{e_k} := release\_back(e_k, B_{e_k})$
    insert $(mid,\ e_h,\ reply,\ B'_{e_k})$ into the reply queue
endcase

Fig. 7. The message filtering algorithm

execution would enact. For each execution e, the algorithm maintains two folders: a *forw-folder*, denoted by F_e, and a *back-folder*, denoted by B_e. F_e is the forward folder that e has received from its invoker e_i, that is, $F_e = \text{FORW}(e_i, e)$. B_e at a given point in e's execution is equal to the *union-and* of the backward folders received by e up to that point from the executions it has invoked. In particular, the B_e associated with e at the time e invokes an execution e_k is $B_e = \bigcup_z^{\wedge}\{\text{BACK}(e_z, e) \mid e_z <_e e_k\}$. To support the deferred reply interaction mode, a *reply queue* stores the replies of executions invoked under `dr` mode. When a deferred reply execution terminates, its reply is inserted into the reply queue. The reply is retrieved from the queue and returned to the invoker upon interception of the corresponding `get` message. Figure 7 reports the message filtering algorithm. Suppose that an execution e_h, running on object o_h, sends a message *msg=(name,pars,mode,reply)* to object o_k invoking execution e_k. If the message is not a `get` message, the following initialization process is executed. First, B_{e_k} is initialized to the empty set. Then, if the message has been sent by the transaction initiator, that is, no protected information is passed to e_k, F_{e_k} is initialized to the empty set and execution e_k is performed. The reply of e_k is returned to the transaction initiator only if, for each pair $(o_i, U_i) \in B_{e_k}$, the transaction initiator belongs to U_i. The reason for this control is that, since no discretionary control is executed on read operations, the flow control must ensure that the information flow to the transaction initiator is safe (see Section 6). If the message is not sent by the transaction initiator, F_{e_k} is set equal to the result of function *release_forw*() applied to execution e_h and to the *union-and* of F_{e_h} and B_{e_h}. After this initialization process, executed only for messages different from `get`, the algorithm determines how to execute e_k and update folders depending on both the method to be executed and the interaction mode. Cases 1 through 4 consider the execution of primitive methods. Cases 5a-c consider the execution of non-primitive methods under the different interaction modes. Let us examine the different cases.

(1) The message is a read message. The read method is invoked and its reply is returned to e_h. B_{e_h} is updated by taking its *union-and* with $\{(o_k, \text{ACL}(o_k))\}$.

(2) The message is a write message. The algorithm determines whether the write is safe by checking if for each pair $(o_i, U_i) \in F_{e_k}$ the ACL of object o_k is contained in U_i. If this condition is satisfied the write operation is executed and its reply returned. Otherwise, the write execution is blocked and a `failure` is returned.

(3) The message is a create message. The algorithm determines whether the create is safe by checking if for each pair $(o_i, U_i) \in F_{e_k}$ the transaction initiator u belongs to U_i. If this condition is satisfied, the create method is executed, the ACL of the created object is initialized to the set containing only the transaction initiator, and the identifier of the new object is returned to e_h. Otherwise, the create execution is blocked and a `failure` is returned.

(4) The message is a get message requiring the reply of execution e_k previously invoked under deferred reply mode. The algorithm retrieves the tuple corresponding to execution e_k from the reply queue. B_{e_h} is updated by taking its *union-and* with the folder contained in the tuple. The reply retrieved from the queue is returned to e_h.

(5) The message is a non-primitive message requiring the execution of e_k. De-

pending on the possible interaction modes under which e_k can be executed, we consider the following cases.
5a) e_k is invoked under synchronous mode. Execution e_k is performed and its reply is returned to e_h. B_{e_h} is updated by taking its *union-and* with the result of *release_back*(e_k, B_{e_k}).
5b) e_k is invoked under asynchronous mode. A `nil` reply is immediately returned to e_h. Execution e_k is performed and its reply discarded.
5c) e_k is invoked under deferred reply mode. The algorithm generates a message identifier and returns it to e_h. Then, e_k is executed. Folder B'_{e_k} to be stored in the reply queue is computed as *release_back*(e_k, B_{e_k}). Finally, a tuple composed of the reply of e_k, together with the generated message identifier, the execution e_h, and folder B'_{e_k} is inserted into the reply queue.

The following theorem states the correctness of the message filtering algorithm.

THEOREM 3. *The message filtering algorithm blocks a write or create execution if and only if the execution is not safe.*

8. COMPLEXITY, IMPLEMENTATION, AND PERFORMANCE EVALUATION

The message filtering algorithm intercepts every message exchange within a transaction. One may therefore wonder whether such a fine-grained control is feasible in practice or the resulting overhead would instead prove too high. A detailed complexity analysis allows the stating of the following theorem.

THEOREM 4. *The running time of the message filtering algorithm is* linear *in the number of read, write/create, and non-primitive executions enacted by the input message.*

We refer the reader to the Appendix for the formal proof. Here we note that the overall worst case complexity is $O((c_{rf} + c_{ua}) \cdot N_m + c_{ua} \cdot (n_{np} + n_r) + c_{rb} \cdot n_{np} + n_{wc} \cdot n_r + n_r)$ where n_r, n_{wc}, n_{np} are the number of read, write/create and non-primitive executions in the transaction, respectively, $N_m = n_r + n_{wc} + n_{np}$ is the total number of executions, c_{rf} is the cost of function *release_forw*$()$, c_{rb} is the cost of function *release_back*$()$, and c_{ua} is the cost of the *union-and* operation.

From Theorem 4 it descends that an efficient implementation of the message filtering algorithm greatly depends on the efficient implementation of the *release_back*, *release_forw*, and *union-and* operations. In the remainder of this section we illustrate the techniques we have devised to implement these operations and report some experimental results on their performance. The results obtained allow us to conclude that the overhead imposed by the message filtering algorithm controls is very limited with respect to the cost of the transaction execution.

8.1 Implementation

We assign to each user a number that uniquely identifies him/her in the system. Each ACL is implemented as a string of boolean values of length N, where N is the total number of users in the system. Entry i in the ACL of an object indicates whether user i is authorized (value 1) or not (value 0) to access the object. Exceptions and folders associated with executions are encoded in an analogous way. Exceptions are represented as lists of triples $[(o_1, DU_1, AU_1), \ldots, (o_n, DU_n, AU_n)]$, where $o_1, \ldots, o_n$ are the objects for which an exception is specified, and DU_i and AU_i are strings of boolean values of length N. Value 1 in the *j-th* entry of string DU_i indicates that a restrictive exception applies on object o_i for user j. Similarly, value 1 in the *j-th* entry of string AU_i indicates that a permissive exception applies to object o_i for user j. Folders are represented as lists of pairs $[(o_1, U_1), \ldots, (o_n, U_n)]$, where $o_1, \ldots, o_n$ are the objects from which information has been read and U_i is a string of boolean values of length N. Value 1 (0, respectively) in the *j-th* entry of U_i indicates that user j can (cannot, respectively) access information in o_i. It can be trivially proved that functions *release_back()* and *release_forw()* and the *union-and* operation translate on operations on strings as follows. Let e be an execution and let $F = [(o_1, U_1), \ldots, (o_n, U_n)]$ and $F' = [(o'_1, U_1), \ldots, (o'_m, U_n)]$ be two folders; then

—$release\_back(e, F) = [(o_1, U'_1), \ldots, (o_n, U'_n)]$, where $U'_i = (U_i \vee AU_i) \wedge (\neg DU_i)$ and AU_i and DU_i are the elements in $\Pi_{o_i}(RE(e))$, $i = 1, \ldots, n$.

—$release\_forw(e, F) = [(o_1, U'_1), \ldots, (o_n, U'_n)]$, where $U'_i = (U_i \vee AU_i) \wedge (\neg DU_i)$ and AU_i and DU_i are the elements in $\Pi_{o_i}(IE(e))$, $i = 1, \ldots, n$.

—$F \cup^{\wedge} F' = [(o_1, U_1), \ldots, (o_k, U_k)]$, where $\{o_1, \ldots, o_k\} = \{o'_1, \ldots, o'_m\} \cup \{o''_1, \ldots, o''_n\}$. Moreover, $U_i = U'_i$ if $o_i \in \{o'_1, \ldots, o'_m\} \setminus \{o''_1, \ldots, o''_n\}$; $U_i = U''_i$ if $o_i \in \{o''_1, \ldots, o''_n\} \setminus \{o'_1, \ldots, o'_m\}$; $U_i = U'_i \wedge U''_i$ if $o_i \in \{o'_1, \ldots, o'_m\} \cap \{o''_1, \ldots, o''_n\}$.

The following proposition states how the definition of safe write/create operation translates in terms of string encoding.

PROPOSITION 1. Let $[(o_1, U_1), \ldots, (o_n, U_n)]$ be the folder that a write/create execution e receives from its invoker. Execution e is safe if and only if all the entries in $\neg(U_1 \wedge \ldots \wedge U_n) \wedge$ ACL$(o(e))$ have value 0.

In other words, a write/create operation is safe if the logical AND between the ACL of the object being written with the negation of the logical AND of the strings in the folder of the execution produces the null string.

It is easy to prove the correctness of Proposition 1. Let us first consider write executions. Intuitively, the string resulting from the logical AND of all the strings contained in the folder that execution e receives from its invoker has value 1 in entry i if and only if the information of all the objects in the folder can be released to i. Accordingly, its negation has a 0 for such a user. The logical AND of this negated string with the ACL of the object has value 0 in entry i if and only if user i either cannot read the object being written or she/he can receive the information from all the objects in the folder. This is exactly the definition of safe flow. In the case of create executions, the ACL of the created object has value 1 only in correspondence of the entry of the transaction initiator. Therefore, the condition in Proposition 1 is verified if and only if the transaction initiator can receive the information from all the objects in the folder. Again, this is the definition of safe flow.

Example 6. With reference to our running example, suppose that the order of the subjects in the ACLs of the objects is `ada`, `bob`, `jim`, `meg`, `sam`, and `tom`. Consider execution e_{17}. $F_{e_{17}} = \text{FORW}(e_{12}, e_{17}) = [($`project`, [101100]), (`employee`, [111111]), (`resource`, [111111]), (`expense`, [101100]), (`fund`, [100100])]. The ACL

Number of non-primitive executions	Number of primitive executions	Number of users				
		10	*50*	*100*	*150*	*200*
5	*2*	0.020	0.020	0.200	0.220	0.230
	4	0.022	0.023	0.024	0.025	0.027
	6	0.025	0.028	0.030	0.030	0.031
	8	0.030	0.032	0.035	0.036	0.037
	10	0.033	0.037	0.039	0.041	0.041
10	*2*	0.035	0.040	0.042	0.044	0.046
	4	0.047	0.055	0.064	0.071	0.079
	6	0.074	0.084	0.097	0.111	0.127
	8	0.094	0.113	0.137	0.165	0.194
	10	0.114	0.147	0.188	0.221	0.259
20	*2*	0.080	0.088	0.096	0.104	0.113
	4	0.135	0.161	0.194	0.230	0.258
	6	0.212	0.257	0.324	0.389	0.452
	8	0.306	0.395	0.505	0.607	0.725
	10	0.395	0.534	0.716	0.881	1.052
30	*2*	0.152	0.163	0.192	0.204	0.229
	4	0.264	0.326	0.399	0.473	0.539
	6	0.420	0.535	0.686	0.834	0.973
	8	0.642	0.811	1.064	1.324	1.568
	10	0.878	1.161	1.554	1.946	2.311
40	*2*	0.240	0.276	0.322	0.357	0.397
	4	0.478	0.563	0.692	0.820	0.951
	6	0.744	0.962	1.210	1.478	1.744
	8	1.118	1.419	1.885	2.325	2.789
	10	1.508	2.052	2.747	3.449	4.114
50	*2*	0.345	0.396	0.453	0.513	0.579
	4	0.688	0.843	1.046	1.248	1.451
	6	1.125	1.426	1.855	2.285	2.692
	8	1.632	2.165	2.856	3.578	4.267
	10	2.276	3.023	4.085	5.205	6.303

Table 2. Running time, expressed in seconds, taken by the message filtering algorithm controls for the different transactions considered

of the object `exp_report` to be written is [100100]. Hence, $\neg([101100] \wedge [111111] \wedge [111111] \wedge [101100] \wedge [100100]) \wedge [100100] = [000000]$, meaning the write is safe. Consider now execution e_6. $F_{e_6} = \text{FORW}(e_2, e_6) =$ [(`project`,[001000]), (`employee`, [111111]), (`resource`, [111111])]. The ACL of the object `prj_report` to be written is [111011] then, $\neg([001000] \wedge [111111] \wedge [111111]) \wedge [111011] = [110011]$. This means that the flow is unsafe because it would release to `ada`,`bob`,`sam`, and `tom` information they cannot access.

8.2 Performance evaluation

We describe the results of the experiments that we have conducted to evaluate the performance of the message filtering algorithm. The experiments have been carried out on a 166-MHz Pentium with 128 Mb of RAM, under the Linux 2.0.30 operating system. The experiments have been performed in a shared environment. However, we made sure that the system workload was more or less the same when all the experiments were run. Moreover, we ran each experiment ten times. The results reported here are the average of the results produced by the ten experiments. We assumed as a cost factor the time taken by the message filtering algorithm for executing the *release_forw*(), *release_back*(), and *union-and* operation. We considered as parameters of the experiments the number of users in the system and the number of executions in the transaction (size of the transaction). The first parameter, by affecting the length of the ACLs, affects the cost of all the *release_forw*(), *release_back*(), and *union-and* operations. The second parameter affects the number of these operations.

Note that the number of *release_back*(), *release_forw*(), and *union-and* to be executed in a transaction depends only on the size of the transaction and not on the shape of its execution tree. Given this observation, for the sake of simplicity, in our experiments we considered transactions with execution trees of height three, that is, with one level of non-primitive executions (children of the root). The reason for considering an intermediate level of non-primitive executions is to execute meaningful *release_back* operations. Moreover, we assumed the following conditions:

—the number of methods with exceptions is 50% of the total number of methods;

—the number of objects on which an exception is specified is 50% of the total number of objects accessed by the transaction;

—all non-primitive executions, with the only exception of the root, invoke the same number of primitive executions (half of which are read executions);

—each non-primitive execution has a reply exception on 50% of the objects it reads;

—each non-primitive execution has an invoke exception on 50% of the objects in its forward folder;

—the conditions associated with exceptions always evaluate to true.

We ran experiments for the following different cases: 5, 10, 20, 30, 40, and 50 non-primitive executions; 2, 4, 6, 8, and 10 primitive executions for each non-primitive execution; 10, 50, 100, 150, and 200 users. The possible combinations of the execution cases produce 30 different transactions and, by varying the number of users, a total of 150 experiments. Table 2 lists the results of the experiments. The entry corresponding to a given number of executions and users reports the seconds taken by the message filtering algorithm controls for the corresponding transaction. As already anticipated, both the number of users in the system and the number of executions affect the performance of the message filtering algorithm. However, the number of executions has a greater impact on the algorithm performance. This accords with the fact that the number of executions determines the number of *release_forw*(), *release_back*(), and *union-and* to be executed, while the number of users determines only the length of the strings on which these operations apply.

These observations are better illustrated by the graphs in Figures 8 and 9 reporting the running time of the message filtering algorithm with respect to the number of users and executions, respectively. In particular, Figure 8 illustrates the time taken by the message filtering algorithm for transactions with 5 (Figure 8(a)) and 50 (Figure 8(b)) non-primitive executions, as the number of users varies. Figure 9 illustrates the time employed by the message filtering algorithm for 2 (Figure 9(a)) and 10 (Figure 9(b)) primitive executions, as the number of non-primitive execu-

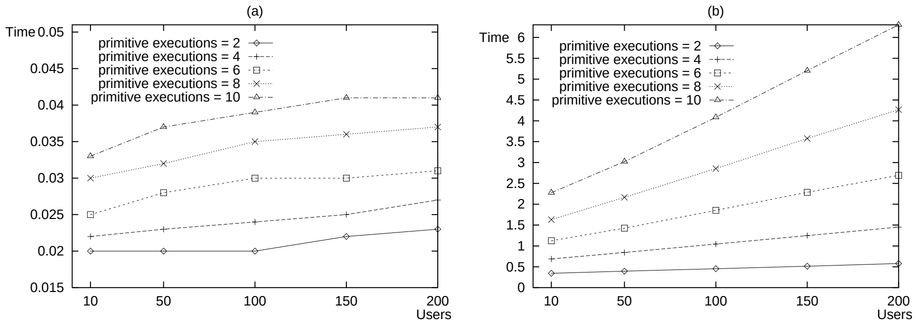

Fig. 8. Time expressed in seconds, taken by the message filtering algorithm controls for the case of 5 (Figure 8(a)) and 50 (Figure 8(b)) non-primitive executions as the number of users varies

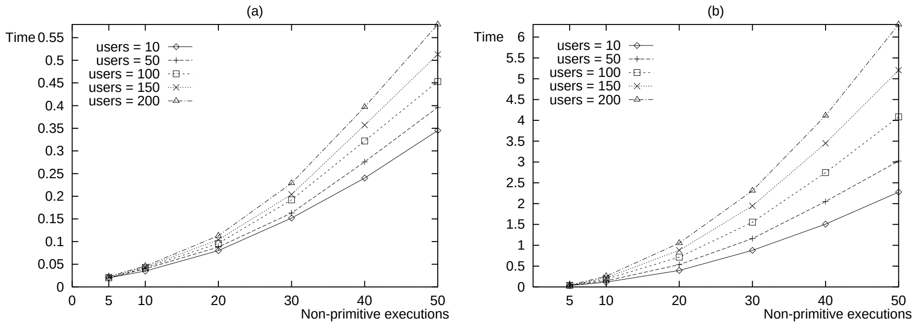

Fig. 9. Time taken by the message filtering algorithm controls for the cases of 2 (Figure 9(a)) and 10 (Figure 9(b)) primitive executions as the number of non-primitive executions varies

tions varies. As the graphs show, the number of users affects the time taken by the message filter controls only for transactions with a high number of executions (Figure 8(b)) and it is not very relevant for small-size transactions (Figure 8(a)). By contrast, the total number of executions affects the time taken by the message filter controls with similar curves (Figure 9(a) and 9(b)) for the cases with a low and a high number of users.

What we have reported above are "absolute" times. To determine whether these times are acceptable in real-world situations, we need to evaluate them with respect to the time taken by normal transaction execution. The most well-known benchmark developed for object-oriented database systems, namely 007 [Carey et al. 1993], considers transactions composed of at most 10 executions. Therefore, typical object-oriented transactions have a number of exceptions, which is located in the initial range of the graphs in Figure 9. Moreover, according to the benchmark, a transaction execution may take between 20 and 40 seconds for small databases, with a much longer time for larger databases. We can therefore conclude that the impact of the message filtering algorithm controls on the overall transaction execution time is negligible.

9. EXCEPTION SPECIFICATION AND METHOD CERTIFICATION

Exceptions provide flexibility in the information flow control enforcement by avoiding the rigid application of the strict policy. Conditions associated with exceptions further increase this flexibility. Since exceptions allow flows that would otherwise be blocked and forbid flows that would otherwise be allowed, it is important that the specification and enforcement of exceptions be strictly controlled to ensure that users will not be able to improperly subvert the security constraints by specifying exceptions. First of all, since exceptions essentially allow users to state to whom information in an object can or cannot be released, it is obvious that exceptions on an object be specified only by users who can administer the object (i.e., users who can specify authorizations). In our reference discretionary authorization model we have not made any assumption on who can specify authorizations on an object. Our model is general and can be applied to systems with different administrative policies. For instance, if ownership is enforced, only the owner of an object will be allowed to specify authorizations and exceptions on the object. If administration is centralized, only the administrator will be allowed to specify authorizations and exceptions on all the objects. Finally, if a decentralized policy is enforced, by which the owner of an object can grant other users the privilege of specifying authorizations, such a policy can be applied to exceptions as well; thus allowing the owner of an object to grant other users the privilege of specifying exceptions on the object. Regardless of the specific administrative policy countered, the important restriction to enforce is that exceptions be specified and modified only by users explicitly authorized for that. Note that, by contrast, it is not necessary to regulate the specification of exceptions with respect to the methods to which they refer. In principle, users allowed to specify exceptions on an object could specify such exceptions with respect to any method. A more sophisticated policy can, however, include explicit authorizations for users to attach exceptions to methods. In this case a user will be allowed to specify an exception for a method of an object only if she/he has both the authorization to specify exceptions on the object and the authorization to attach exceptions to the method.

In this paper we have considered that each object has associated an ACL expressing the set of authorizations on the object. This assumption does not necessarily require the user/administrator to explicitly specify authorizations on each single object in the system. High-level languages can be devised which allow users to state authorizations based on objects properties or classifications. From these high level authorizations, authorizations on single objects can be derived. For instance, authorizations could be specified at the class level and authorizations specified for a class would be considered applicable to all instances.

A user specifying an exception on an object for a method must "trust" the method to properly manage the information read from the object. Different approaches can be applied to provide method certification.

A first possible approach to method certification is to specify, by means of a suitable formal language, the program behavior and then check this formal specification against the system security policy. This approach entails two main tasks: the first is to devise a mechanism for verifying a method against its specification; the second is to check whether the program specification conforms to the security policy. An example of this approach is the Automatic Policy Checker (APC) [King 1991], which allows the automatic verification of program specifications with respect to a security policy. Both method and security policy specifications are expressed by means of a formal language based on regular expressions. The verification of a method against its formal specification is performed by using a filter program which inspects the method's binary code [Crocker and Pozzo 1989].

Other approaches to method certification are based on the automatic analysis of the method source code. The pioneering work in this direction is the compile-time algorithm developed by Denning [Denning 1982]. Andrews and Reitman [Andrews and Reitman 1980] present a compile-time certification technique based on Hoare's logic. Mizuno and Schmidt [Mizuno and Schmidt 1990] propose a certification technique specifically tailored to modular systems. The method consists of two main steps. In the first step, executed at compile time, an algorithm performs a data flow analysis of each individual module. The algorithm enforces Denning's information flow control policy. The output of this phase is a data structure that describes the module's behavior. The second step, executed at run time, combines the data structures of all the modules to verify the security of intermodular flows. Another possible approach to method certification [Sybase, Inc. 1993] is to allow users who specify exceptions for a method to inspect the method code. We believe that all those approaches can be combined with our access control mechanism.

We note that conditions associated with methods may be used to provide some form of certification on the methods' behavior. For instance, suppose that an exception on an object can be applied for a method only if the method does not read a specific attribute a. Instead of requiring certification on the method to ensure that the method does not read attribute a, a condition can be associated with the exception stating that the exception can be applied only if a does not belong to the set of attributes read during the method execution.

We conclude this section by noting that, according to our definitions, it is not possible for a combination of independently specified exceptions to allow flows which should instead be blocked. If an exception is enforced it means that there is trust in the fact that the method to which the exception refers does not pass sensitive information. Therefore, the information this method releases, through reply or invocation, is subject to the exception and can be released to the users for whom the exception is specified (by assumption, each specified exception is correct). It is true that along a transmission channel the information might encounter other exceptions, and therefore be released to other users not originally considered. However, if this is the case, it is because other exceptions are in place and there is trust in the method enforcing them not to release the sensitive information received.

10. CONCLUSIONS

We have proposed a flexible information flow control model for object-oriented systems which allows users to specify exceptions to the strict policy. Exceptions, specified with reference to specific objects and users, are associated with methods. Two types of exception are supported: invoke exceptions, applicable during a method execution, and reply exceptions, applicable to the information returned by a method. Exceptions can be either permissive or restrictive. A permissive exception states that the strict policy can be waived with reference to the execution of specific methods. A restrictive exception imposes, instead, restrictions in addition to the ones imposed by the strict policy. Another relevant feature of our model is that it permits limiting the applicability of method exceptions by associating conditions with them. An exception is applicable only if the corresponding condition is verified. The resulting system is flexible since it takes into account how the information flowing between objects has been retrieved and transmitted. We have formally defined information transmission and flow and given conditions for an information flow to be safe. We have also proposed an algorithm that mediates all the execution requests within a transaction and blocks unsafe write and create operations, and we have discussed its performance.

Our model leaves space for further work. First, the application of exceptions can be restricted to specific operations within a method instead of holding through its whole execution. Another issue concerns the combination of exceptions. For instance, a method can specify that the information it returns cannot be subject to any exceptions, besides those of the method itself. Another important issue to be addressed concerns administration. The specification of authorizations, exceptions, and the understanding of their interworking requires a considerable effort from the administrator. Tools should be developed that guide the specification of exceptions and help visualize their combined effect.

APPENDIX

A. PROOFS OF THEOREMS

Proof of Theorem 1

We first prove that if there exists a transmission channel from e_h to e_k then there exists transmission of information from e_h to e_k. Let us consider the different cases of Definition 7.

Consider first case 1, that is, suppose that there exists a forward channel from e_h to e_k. Then, by Definition 5, either $e_h \rightarrow e_k$, or there exists a sequence of executions $e_{h+1}, e_{h+2}, \ldots, e_{k-1}$ such that $e_h \rightarrow e_{h+1} \rightarrow \ldots \rightarrow e_{k-1} \rightarrow e_k$. In both cases, transmission of information from e_h to e_k is enacted through the message parameters that each execution sends to the execution it invokes.

Consider now case 2, that is, suppose that there exists a backward channel from e_h to e_k. Then, by Definition 6, either $e_k \rightarrow e_h$ and the invocation is not asynchronous, or there exists a sequence of executions $e_{k-1}, e_{k-2}, \ldots, e_{h+1}$ such that $e_k \rightarrow e_{k-1} \rightarrow \ldots \rightarrow e_{h+1} \rightarrow e_h$, and each invocation in the sequence is not asynchronous. In both cases there exists transmission of information from e_h to e_k through the reply each execution sends to its invoker.

Consider finally case 3. Since by hypothesis there exists a backward channel from e_h to e^*, by case 2 above there exists transmission of information from e_h to e^*. Analogously, since there exists a forward channel from e^* to e_k, by case 1 above, there exists transmission of information from e^* to e_k. By hypothesis $e'_h <_e e'_k$ which, by Definition 4, implies that $e'_h <_i e'_k$, and either: *i)* $e^* \xrightarrow{\mathtt{s}} e'_h$, or *ii)* $e^* \xrightarrow{\mathtt{dr}} e'_h$ and $\widehat{e'_h} <_i e'_k$, where $\widehat{e'_h}$ is the execution requiring the reply of e'_h. Consider case *i.* According to the definition of synchronous mode execution, e^* stops executing upon the invocation of e'_h until it receives the reply from e'_h. Hence, when execution e^* invokes e'_k it has already received the information from e_h. Therefore, transmission of information from e_h to e_k is enacted through the mediation of e^*. Consider case *ii*. Again, since $\widehat{e'_h} <_i e'_k$, and, by hypothesis, $\widehat{e'_h}$ is executed in synchronous mode, e^* invokes e'_k after receiving the information from e_h. Therefore, transmission of information from e_h to e_k is enacted.

We now prove that if there exists transmission of information from e_h to e_k then there must exist a transmission channel from e_h to e_k. We distinguish two cases according to whether e_h and e_k are, or are not, connected by a path in the transaction execution tree.

Suppose that e_h and e_k are connected by a path. Suppose first e_h is an ancestor of e_k. Then, by case 1 of Definition 7, there exists a forward channel from e_h to e_k and therefore the implication is satisfied. Suppose instead that e_h is a descendant of e_k. Since by assumption there is transmission, all invocations along the path connecting e_k to e_h are not asynchronous. Then, by case 2 of Definition 7 there exists a backward channel from e_h to e_k and therefore the implication is satisfied. Suppose that e_h and e_k are not connected by a path. Since there is transmission, execution e^*, least common ancestor of e_h and e_k, mediates the transmission. Hence, there must be (backward) transmission from e_h to e^* and subsequent (forward) transmission from e^* to e_k. Then, for what is said above there exists a backward channel from e_h to e^* and a forward channel from e^* to e_k. Moreover, since transmission of information is enforced from e_h to e_k, the transmission of information from e^* to e_k must be subsequent to the reception of information by e^* from e_h. Thus, either $e^* \stackrel{\mathtt{s}}{\rightarrow} e'_h$ or $e^* \stackrel{\mathtt{dr}}{\rightarrow} e'_h$ and $\widehat{e'_h} <_i e'_k$, where e'_h and e'_k are the ancestors of e_h and e_k invoked by e^*. Then, by case 3 of Definition 7, there exists a transmission channel from e_h to e_k.

Proof of Theorem 2

In the following we use $\pm_{e \in ch\text{-}back(e_i,e_k)}$ as a short hand for $\pm\Pi_{o_i}(RE(e_{i-j})) \pm \ldots \pm \Pi_{o_i}(RE(e_i))$, where *ch-back*$(e_i, e_k) = (e_{i-j}, \ldots, e_{i-1}, e_i)$. We use $\pm_{e \in ch\text{-}forw(e_i,e_k)}$ as a short hand for $\pm\Pi_{o_i}(IE(e^*)) \pm \ldots \pm \Pi_{o_i}(IE(e_k))$, where *ch-forw*$(e_i, e_k) = (e^*, \ldots, e_{i-1}, e_k)$.

We first prove the "only if" implication. We suppose that the implication does not hold and derive a contradiction. Suppose that e_k is not safe and that the condition on the right hand side of the if and only if is satisfied. Since e_k is not safe, by Definition 13 there exists an unsafe flow $(o_i, e_i, e_k, o_k) \in FLOWS(T)$, such that e_i is a read execution on object o_i. Hence, by Definition 12, $\mathrm{ACL}(o_k) \not\subseteq (\mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e))) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$. By Definition 15, it is easy to verify that if $(o_i, e_i, e_k, o_k) \in FLOWS(T)$ then there exists a pair $(o_i, U_i) \in FORW(e_h, e_k)$. By hypothesis, $\mathrm{ACL}(o_k) \subseteq U_i$. Hence, there must exist a user u such that $u \in U_i$ and $u \notin \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$.

Let e^* be the least common ancestor of e_i and e_k and let $e_{i-j}, \ldots, e_{i-1}, e_i$ be the sequence of executions in *ch-back*(e_i, e_k), that is, such that $e^* \rightarrow e_{i-j} \ldots \rightarrow e_{i-1} \rightarrow e_i$. Let us consider the folder that each execution in *ch-back*(e_i, e_k) returns to its invoker. By Definition 14, $\mathrm{BACK}(e_i, e_{i-1}) = \{(o_i, \mathrm{ACL}(o_i))\}$, since, by hypothesis, e_i is a read execution on object o_i. Consider now $\mathrm{BACK}(e_{i-1}, e_{i-2})$. By Definition 14 this folder is the result of function *release_back* with, as input, e_{i-1} and the *union-and* of the folders that e_{i-1} has received from the executions it has invoked. Since $e_{i-1} \rightarrow e_i$, $\mathrm{BACK}(e_i, e_{i-1}) = \{(o_i, \mathrm{ACL}(o_i))\}$ is one of these folders. Since the set of users associated with an object as a result of the *union-and* operation is equal to the intersection of the set of users associated with the object in the folder in which the object appears, the folder resulting from the *union-and* operation contains a pair (o_i, U_i) such that $U_i \subseteq \mathrm{ACL}(o_i)$. Hence, by the definition of *release_back*, $\mathrm{BACK}(e_{i-1}, e_{i-2})$ contains a pair (o_i, U_i') such that $U_i' = U_i \pm \Pi_{o_i}(RE(e_{i-1}))$. Using analogous considerations for each execution $e_j \in$ *ch-back*(e_i, e_k) we can easily prove that folder $\mathrm{BACK}(e_{i-j}, e^*)$ contains a pair (o_i, U_i) such that $U_i \subseteq \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e))$.

Let us consider the forward channel from e_i to e_k and let $e^*, e_{h-l}, \ldots, e_h, e_k$ be the sequence of executions in *ch-forw*(e_i, e_k), that is, such that $e^* \rightarrow e_{h-l} \rightarrow \ldots e_h \rightarrow e_k$. By Definition 15, $\mathrm{FORW}(e^*, e_{h-l})$ is the output of function *release_forw* with, as input, e^* and the *union-and* of the folders that e^* has received back, before invoking e_{h-l}, from the executions it has invoked and with the folder e^* has received from its invoker. Since by hypothesis there exists a transmission channel from e_i to e_k, then e^* receives folder $\mathrm{BACK}(e_{i-j}, e^*)$ before invoking e_{h-l}. $\mathrm{BACK}(e_{i-j}, e^*)$ contains a pair (o_i, U_i) such that $U_i \subseteq \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e))$. Moreover the folder returned by the *union-and* operation contains a pair (o_i, U_i') such that $U_i' \subseteq U_i$. Thus, by definition of *release_forw*, $\mathrm{FORW}(e^*, e_{h-l})$ contains a pair (o_i, U_i'') such that $U_i'' = U_i' \pm \Pi_{o_i}(IE(e^*)) \subseteq \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm \Pi_{o_i}(IE(e^*))$. By applying the same reasoning to each execution $e_j \in$ *ch-forw*(e_i, e_k) we can easily prove that folder $\mathrm{FORW}(e_h, e_k)$ that e_h passes to e_k contains a pair (o_i, U_i) such that $U_i \subseteq \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$. Hence, there cannot exist a user u such that $u \in U_i$ and $u \notin \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$, which contradicts the assumption.

We now prove the "if" implication. We suppose that the implication does not hold and derive a contradiction. Suppose that e_k is safe and $\exists (o_i, U_i) \in \mathrm{FORW}(e_h, e_k)$ such that $\mathrm{ACL}(o_k) \not\subseteq U_i$. By Definition 15, if $(o_i, U_i) \in \mathrm{FORW}(e_h, e_k)$, then there exists at least a flow $(o_i, e_i, e_k, o_k) \in FLOWS(T)$ from object o_i to object o_k. Since e_k is safe, for each flow $(o_i, e_i, e_k, o_k) \in FLOWS(T)$ then $\mathrm{ACL}(o_k) \subseteq \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$. Therefore, there exists a user u such that $u \notin U_i$ and $u \in \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$ for each read execution e_i on object o_i connected through a channel to e_k.

Let us consider $\mathrm{FORW}(e_h, e_k)$. By Definition 15, $\mathrm{FORW}(e_h, e_k)$ is the output of function *release_forw* with, as input, e_h and the folder $F = \bigcup_z^{\wedge} \{\mathrm{BACK}(e_z, e_h) \mid e_z <_e e_k\} \cup^{\wedge} \mathrm{FORW}(e_j, e_h)$, where e_j is the execution that invoked e_h. By the definition of function *release_forw*, the pair $(o_i, U_i) \in \mathrm{FORW}(e_h, e_k)$ is obtained by taking the pair $(o_i, U_i') \in F$ and combining it with $\Pi_{o_i}(IE(e_h))$ by means of the $\pm$ operator. Let $(DU_i, AU_i) = \Pi_{o_i}(IE(e_h))$. By definition sets AU_i and DU_i are disjoint. Thus, $u \notin U_i$ implies that $u \notin AU_i$, since $U_i = (U_i' \cup AU_i) \setminus DU_i$. Two cases can arise: 1) $u \in U_i' \cap DU_i$, or 2) $u \notin U_i'$. Let us first consider case 1. It is immediate to verify that $u \in DU_i$ implies that $u \notin \mathrm{ACL}(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$ for each read execution e_i on object o_i connected through a channel to e_k. Indeed, for each execution e_i, $e_h \in$ *ch-forw*(e_i, e_k), since by hypothesis $e_h \rightarrow e_k$. This implies that ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e)) =$ ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_h)} \Pi_{o_i}(IE(e)) \pm \Pi_{o_i}(IE(e_h))$. Let $U =$ ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_h)} \Pi_{o_i}(IE(e))$. ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e)) = (U \cup AU_i) \setminus DU_i$ and thus $u \notin$ ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_h)} \Pi_{o_i}(IE(e))$, since $u \in DU_i$, which contradicts the thesis. Let us consider case 2. By the definition of $\cup^{\wedge}$, at least one set between $\bigcup_z^{\wedge}\{\text{BACK}(e_z,e_h) \mid e_z <_e e_k\}$ and FORW(e_j,e_h) contains a pair (o_i,U_i) such that $u \notin U_i$. Suppose that this set is $\bigcup_z^{\wedge}\{\text{BACK}(e_z,e_h) \mid e_z <_e e_k\}$. Then, there exists an execution e_z, $e_z <_e e_k$, and BACK(e_z,e_h) contains a pair (o_i,U_i) such that $u \notin U_i$. Hence, there exists a read operation e_i on object o_i which is connected through a channel to e_z. Thus, there exists a transmission channel from e_i to e_k through their least common ancestor e_h. By definition of function *release_back*, the pair $(o_i,U_i) \in$ BACK(e_z,e_h) is obtained by taking the pair (o_i,U_i') in the *union-and* of the folders that e_z has received from the executions it has invoked and combining it with $\Pi_{o_i}(RE(e_z))$ by means of the $\pm$ operator. Let $(DU_i,AU_i) = \Pi_{o_i}(RE(e_z))$. By definition, sets AU_i and DU_i are disjoint. Thus, $u \notin U_i$ implies that $u \notin AU_i$, since $U_i = (U_i' \cup AU_i) \setminus DU_i$. Two cases can arise: 1) $u \in (U_i' \cap DU_i)$, or 2) $u \notin U_i'$. Let us first consider case 1. By hypothesis execution e_h is such that u does not appear in any set $\Pi_{o_i}(IE(e_h))$. It is easy to show that $u \in DU_i$ implies that $u \notin$ ACL$(o_i) \pm_{e \in ch\text{-}back(e_i,e_k)} \Pi_{o_i}(RE(e)) \pm_{e \in ch\text{-}forw(e_i,e_k)} \Pi_{o_i}(IE(e))$ for each read execution e_i on object o_i connected through a channel to e_k through the mediation of e_h. Thus, we have a contradiction. If case 1 is false, then case 2 holds. We can apply recursively the same steps we are illustrating for e_z until either we find an execution e_j such that u does not appear in any set $\Pi_{o_i}(IE(e_h))$ or we find a read execution e_i on object o_i such that $u \notin$ACL(o_i). In both cases we have proved the thesis. Thus, we have a contradiction.

Proof of Theorem 3

We first need to prove some lemmas ensuring the correctness of the variables used by the message filtering algorithm and their relationships with the definitions introduced in subsection 7.1.

LEMMA 1. *Let e_h and e_k be two executions such that $e_h \rightarrow e_k$. At the end of execution e_k, $BACK(e_k,e_h) = release\_back(e_k, B_{e_k})$.*

If the message is not a `get` message, B_{e_k} is initialized to be the empty set and then changed according to the method executed by e_k. We consider each case. If $m(e_k) =$ `read`, B_{e_k} is set equal to $\{o_k, \text{ACL}(o_k)\}$, where $o_k = o(e_k)$. Thus, the lemma is trivially satisfied, since $release\_back(e_k, B_{e_k}) = B_{e_k}$. If $m(e_k) \in$ {`write`,`create`} or e_k is an asynchronous execution, B_{e_k} keeps its initial value $\emptyset$. Thus, the lemma is satisfied. For all the other cases, we prove that the lemma holds by induction. The base of the induction is represented by the cases in which $m(e_k)$ is a primitive method, for which we have shown that the lemma holds. Let $e_1,\ldots,e_n$ be the executions invoked by e_k. We suppose that the lemma holds for $e_1,\ldots,e_n$ and we prove it also holds for e_k. The algorithm updates B_{e_k} by taking its *union-and* with *release_back*(e_i, B_{e_i}) at the completion of each execution e_i called by e_k such that $m(e_i)$ is a non-primitive method or `get` execution and e_i is not executed in an asynchronous mode. B_{e_k} is instead updated to be its union with B_{e_i} in case of a read operation. It is not updated in case of `write` or `create` executions. However, if $m(e_i) =$ `read`, $release\_back(e_i, B_{e_i}) = B_{e_i}$. If $m(e_i) \in$ {`write`,`create`} or e_i is an asynchronous execution, $release\_back(e_i, B_{e_i}) = \emptyset$. Hence, in all the cases $B_{e_k} = B_{e_k} \cup^\wedge release\_back(e_i, B_{e_i})$. Therefore, for any execution e_k, $B_{e_k} = \bigcup_z^\wedge \{release\_back(e_i, B_{e_i}) \mid e_k \rightarrow e_i\}$. By inductive hypothesis, $release\_back(e_i, B_{e_i}) = \mathrm{BACK}(e_i, e_k)$. Hence, the lemma is satisfied.

Lemma 2. *Let e_h be an execution. For any execution e_k such that $e_h \rightarrow e_k$, B_{e_h} before the invocation of e_k is equal to the* union-and *of the backward folders received by e_h up to that point from the executions it has invoked, that is, $B_{e_h} = \bigcup_i^\wedge \{\mathrm{BACK}(e_i, e_h) \mid e_i <_e e_k\}$.*

With the process followed for the previous lemma, we obtain that at the time e_k is invoked, $B_{e_h} = \bigcup_i^\wedge release\_back(e_i, B_{e_i})$ where i ranges over all executions e_i preceding e_k such that $e_i <_e e_k$. By Lemma 1, at the end of execution e_i, $release\_back(e_i, B_{e_i}) = \mathrm{BACK}(e_i, e_h)$. Therefore, at the time e_k is invoked, $B_{e_h} = \bigcup_i^\wedge \{\mathrm{BACK}(e_i, e_h) \mid e_i <_e e_k\}$, and hence the lemma is satisfied.

Lemma 3. *Let e_h and e_k be two executions such that $e_h \rightarrow e_k$, then $F_{e_k} = \mathrm{FORW}(e_h, e_k)$.*

We prove that F_{e_k} agrees with Definition 15 by induction. The base of the induction is the execution corresponding to the root of the transaction execution tree, that is, the execution invoked directly by a user. We first prove the lemma for the basic case. Then we prove that if it is satisfied for a generic execution e_h it is also satisfied for any execution e_k such that $e_h \rightarrow e_k$. Consider the case where e_h is the root of the transaction execution tree. F_{e_h} is set equal to the empty set by the algorithm and is never changed subsequently. Hence, the lemma holds. We now prove that if the lemma is satisfied for an execution e_h it is also satisfied for any execution e_k such that $e_h \rightarrow e_k$. The algorithm initializes F_{e_k} to the result of $release\_forw(m(e_h), B_{e_h} \bigcup^\wedge F_{e_h})$ and never changes it subsequently. By Lemma 2, $B_{e_h} = \bigcup_z^\wedge \{\mathrm{BACK}(e_z, e_h) \mid e_z <_e e_k\}$, before the invocation of e_k. By inductive hypothesis $F_{e_h} = \mathrm{FORW}(e_j, e_h)$, where e_j is the execution that invokes e_h. Therefore, the lemma is satisfied, since, according to Definition 15, $\mathrm{FORW}(e_h, e_k) = release\_forw(e_h, F)$, where $F = \bigcup_z^\wedge \{\mathrm{BACK}(e_z, e_h) \mid e_z <_e e_k\} \cup^\wedge \mathrm{FORW}(e_j, e_h)$.

We are now ready to prove the theorem.

Consider first the case of write executions. Let e_k be a write execution. We first prove the "if" implication. We assume that it does not hold and derive a contradiction. Suppose that the algorithm blocks an execution e_k which is safe. By Theorem 2, if e_k is safe then $\forall (o_i, U_i) \in \mathrm{FORW}(e_h, e_k)$: $\mathrm{ACL}(o_k) \subseteq U_i$, where e_h is the execution that invokes e_k and $o_k = o(e_k)$. By Lemma 3, $F_{e_k} = \mathrm{FORW}(e_h, e_k)$. Therefore, the condition in the **if** statement of case 2 of the algorithm is satisfied, and e_k is executed, which contradicts the assumption.

To prove the "only if" implication we assume that it does not hold and derive a contraction. Suppose that execution e_k is not safe and that the algorithm does not block it. By Theorem 2, if e_k is not safe, then $\exists (o_i, U_i) \in \mathrm{FORW}(e_h, e_k)$ such that $\mathrm{ACL}(o_k) \not\subseteq U_i$, where e_h is the execution that invokes e_k and $o_k = o(e_k)$. By Lemma 3, $F_{e_k} = \mathrm{FORW}(e_h, e_k)$. Therefore, the condition in the **if** statement of case 2 of the algorithm is not satisfied. Then the **else** statement is executed which implies that e_k is blocked and a `failure` is returned, which contradicts the assumption.

Consider now create executions. Let e_k be a create execution. We first prove the "if" implication. We assume that it does not hold and derive a contradiction. Suppose that execution e_k is blocked by the message filtering algorithm even if e_k is safe. By Theorem 2, if e_k is safe then $\forall(o_i, U_i) \in \text{FORW}(e_h, e_k)$, $\text{ACL}(o_k) \subseteq U_i$, where e_h is the execution that invokes e_k and $o_k = o(e_k)$. By Lemma 3, $F_{e_k} = \text{FORW}(e_h, e_k)$, whereas $\text{ACL}(o_k)$ is set equal to the set containing only the transaction initiator. Therefore, the condition in the **if** statement of case 3 of algorithm is true, and e_k is executed, which contradicts the assumption.

To prove the "only if" implication we assume that it does not hold and derive a contradiction. Suppose that execution e_k is not safe and that it is not blocked by the algorithm. By Theorem 2, if e_k is not safe then $\exists(o_i, U_i) \in \text{FORW}(e_h, e_k)$ such that $\text{ACL}(o_k) \not\subseteq U_i$, where e_h is the execution that invokes e_k, and $o_k = o(e_k)$. By Lemma 3, $F_{e_k} = \text{FORW}(e_h, e_k)$. Moreover, $\text{ACL}(o_k) = \{u\}$. The condition in the **if** statement of case 3 of the algorithm is not satisfied, and the **else** statement is executed. Hence, e_k is not executed and a `failure` is returned, which contradicts the assumption.

Proof of Theorem 4

Let T be a transaction. The complexity of the message filtering algorithm is expressed in terms of the following parameters:

—n_r: number of read executions in T;

—n_{wc}: number of write/create executions in T;

—n_{np}: number of non-primitive executions in T;

—$N_m = n_r + n_{wc} + n_{np}$: total number of executions in T;

—c_{rf}: cost of *release_forw*() function;

—c_{rb} : cost of *release_back*() function;

—c_{ua}: cost of *union-and* operation.

We assume that the execution of any instruction but those listed above takes time $O(1)$. We also assume that all executions are invoked under synchronous mode. The reason for this assumption is that synchronous executions cost more than asynchronous executions, and, if we do not consider the cost of maintaining the reply queue, they have the same cost of executions invoked under deferred reply mode.

The message filtering algorithm is composed of an **if** statement followed by a **case** statement. The **if** statement is executed for all messages different from `get`, that is, for all the executions in our case. For execution e_k invoked directly by the transaction initiator, the message filtering algorithm checks if $u \in U_i$ for all $(o_i, U_i) \in B_{e_k}$. In the worst case, B_{e_k} contains at most n_r different objects. Hence, the total cost of this operation is $O(n_r)$.

For all the other $N_m - 1$ executions a *union-and* and a *release_forw*() operation are performed. Hence, the total cost of the **if** statement (**else** branch) for all these executions is $\text{O}((c_{rf} + c_{ua}) \cdot (N_m - 1))$, which implies a complexity of $\text{O}(n_r + (c_{rf} + c_{ua}) \cdot N_m)$.

Consider now the **case** statement and its cost for the different executions in T.

—n_r read executions. Each read execution requires a *union-and* operation. The total cost is $c_{ua} \cdot n_r$.

—n_{wc} write/create executions. Each of these executions requires the evaluation of the condition $\mathrm{ACL}(e_k) \subseteq U_i$, in case of write operations, or $u \in U_i$, in case of create operations, for each pair (o_i, U_i) in F_{e_k}. Since there can be at most n_r pairs in F_{e_k}, the cost in the worst case is n_r. The total cost is $n_r \cdot n_{wc}$.

—n_{np} non-primitive executions. Each of them requires performing a *release_back*() and a *union-and* operation. The total cost is $(c_{rb} + c_{ua}) \cdot n_{np}$.

The complexity of the **case** statement is therefore $O(c_{ua} {\cdot} (n_{np} + n_r) + c_{rb} {\cdot} n_{np} + n_{wc} \cdot n_r)$.

The complexity of the message filtering algorithm obtained as the sum of the complexity of the **if** and **case** statement is $O((c_{rf} + c_{ua}) {\cdot} N_m + c_{ua} {\cdot} (n_{np} + n_r) + c_{rb} {\cdot} n_{np} + n_{wc} \cdot n_r + n_r)$.

REFERENCES

ANDREWS, G. R. AND REITMAN, R. P. 1980. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems 2*, 1, 56–76.

BELL, D. E. AND LAPADULA, L. J. 1976. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, Massachusetts.

BINGHAM, H. W. 1974. Access controls in burroughs large systems. Privacy and Security in Computer Systems TR 404, National Bureau of Standards Special Publication.

BOEBERT, W. E. AND FERGUSON, C. T. 1985. A partial solution to the discretionary trojan horse problem. In *Proc. of the* 8^{th} *Nat. Computer Security Conf.* (Gaithersburg, MD, 1985), pp. 141–144.

CAREY, M., DEWITT, D., AND NAUGHTON, J. 1993. The 007 benchmark. In *Proc. of the 1993 ACM SIGMOD Conference* (Washington, DC, May 1993), pp. 12–21.

CASTANO, S., FUGINI, M. G., MARTELLA, G., AND SAMARATI, P. 1995. *Database Security.* Addison-Wesley.

CROCKER, S. AND POZZO, M. 1989. A verification-based virus filter. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1989), pp. 319–324.

DENNING, D. E. 1976. A lattice model of secure information flow. *Comm. of the ACM*, 236–243.

DENNING, D. E. 1982. *Cryptography and Data Security.* Addison-Wesley, ACM Press.

GRAUBART, R. 1989. On the need for a third form of access control. In *Proc. of the* 12^{th} *Nat. Computer Security Conf.* (Gaithersburg, MD, June 1989), pp. 296–303.

Informix Software, Inc. 1993. INFORMIX-*Online/Secure. Security Features User's Guide.* Informix Software, Inc.

JAJODIA, S. AND KOGAN, B. 1990. Integrating an object-oriented data model with multilevel security. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1990), pp. 76–84.

KARGER, P. A. 1987. Limiting the damage potential of discretionary Trojan horses. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1987), pp. 32–37.

KEMMERER, R. A. 1982. A practical approach to identifying storage and timing channels. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, April 1982), pp. 66–71.

KING, M. M. 1991. Identifying and controlling undesirable program behaviors. In *Proc. of the* 14^{th} *Nat. Computer Security Conf.* (October 1991), pp. 283–194.

McCollum, C. J., Messing, J. R., and Notargiacomo, L. 1990. Beyond the pale of MAC and DAC - Defining new forms of access control. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1990), pp. 190–200.

Mizuno, M. and Schmidt, D. 1990. A security flow control algorithm and its denotational semantics correctness proof. Technical report (December), Computing and Information Sciences Dept, Kansas State University.

Oracle Corporation. 1996. *Trusted* Oracle *Administrator's Guide.* Oracle Corporation.

Samarati, P., Bertino, E., Ciampichetti, A., and Jajodia, S. 1997. Information flow control in object-oriented systems. *IEEE Trans. on Knowledge and Data Engineering 9*, 4, 524–538.

Stoughton, A. 1981. Access flow: A protection model which integrates access control and information flow. In *Proc. of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, April 1981), pp. 9–18.

Sybase, Inc. 1993. sybase *Secure* SQL *Server.* Sybase, Inc.

Walter, K. G., Ogden, W. F., Rounds, W. C., Bradshaw, F. T., Ames, S. R., and Sumaway, D. G. 1974. Primitive models for computer security. Technical Report TR ESD-TR-4-117, Case Western Reserve University.

Wilkinson, A. L., Anderson, D. H., Chang, D. P., Hin, L. H., Mayo, A. J., Viney, I. T., Williams, R., and Wright, W. 1981. A penetration analysis of a burroughs large system. *ACM SIGOPS Oper. Syst. Reviews 1*, 15 (January), 14–25.