Providing Flexibility in Information Flow Control for Object-Oriented Systems *

Elena Ferrari, Pierangela Samarati, Elisa Bertino
Dipartimento di Scienze dell'Informazione
Università di Milano, 20135 Milano, Italy
{ferrarie,samarati,bertino}@dsi.unimi.it

Sushil Jajodia
ISSE Department, George Mason University
Fairfax, VA 22030-4444, U.S.A.
jajodia@isse.gmu.edu

Abstract

This paper presents an approach to control information flow in object-oriented systems that takes into account, besides authorizations on objects, also how the information has been obtained and/or transmitted. These aspects are considered by allowing exceptions to the restrictions stated by the authorizations. Exceptions are specified by means of waivers associated with methods. Two kinds of waivers are supported: invoke-waivers, specifying exceptions applicable during a method's execution, and reply-waivers, specifying exceptions applicable to the information returned by a method. Information flowing from one object into another object is subject to the different waivers of the methods enforcing the transmission. We formally characterize information transmission and flow in a transaction taking into consideration different interaction modes among objects. We then define security specifications, meaning authorizations and waivers, and characterize safe information flows. We formally define conditions whose satisfaction ensures absence of unsafe flows and present an algorithm enforcing these conditions.

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 simply on authorizations, however, has the serious drawback of being vulnerable to Trojan Horses Even if each access is controlled against the authorizations and granted only if explicitly allowed by them, information leakage to unauthorized users is possible. The reason for this is that no control is enforced on the use or dissemination of the information once this is released to an authorized user or, more precisely, to a process running on his behalf. Hence, improper, although authorized, write operations can store the information into objects accessible by users not authorized to read this information. Mandatory policies avoid this kind of flow by introducing ordered 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. With the use of labels, however, the flexibility proper of the discretionary access control is lost.

Some proposals try to overcome the vulnerability of discretionary access control by complementing it with some form of flow control [1, 7, 8, 10, 11, 13]. Some approaches restrict a program to access only objects satisfying given patterns [7] or owned by specific users [1]. Other approaches [3, 8, 11] associate with each object an access control list that is propagated together with the information in the object. Specifically targeted to the protection against unauthorized leakage of information are the proposals in [10, 13]. In these approaches a process can write an object o only if o is is protected in reading at least 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 specifying accesses is not lost, the overall flexibility is definitely reduced by the application of the strict policy. As a matter of fact, after having read an object o a process is completely unable to write any object less protected in reading than o, even if the write operation would not result in any improper information leakage. The approach in [10], developed to support flow control in object-oriented sys-

*The work of Elisa Bertino, Elena Ferrari, and Pierangela Samarati was partially supported by the Italian M.U.R.S.T. under grant 40%. The work of Sushil Jajodia was partially supported by National Science Foundation under grants IRI–9633541 and INT–9412507 and by National Security Agency under grants MDA904–96–1–0103 and MDA904–96–1–0104.

tems, partially overcomes this problem in two ways. First it takes into account the fact that operations are not always executed sequentially, because of the different interaction modes with which messages can be sent. 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 that invoked the method (thus avoiding blocking subsequent write operations). These two aspects however, do not overcome the main limitation of the strict policy. Information flow is still controlled only by considering read and write operations without taking into account how the information has been obtained and/or transmitted.

Everyday activities often require the strict policy to be bypassed and several situations can be found where exceptions to the strict policy can be enforced without compromising security. Generally, these exceptions can be confined to specific procedures with reference to the information they return or they store. As matter of fact, even in the application of mandatory policy, recent work has pointed out the need for supporting exceptions within the execution of specific trusted procedures [5, 9, 12]. In this paper we extend the work in [10] to the consideration of two kinds of exceptions to the strict policy: *i)* exceptions on information released by a procedure, and *ii)* exceptions for write operations executed within a procedure.

The motivation for the first kind of exceptions comes from the observation that often the information that a procedure returns is not as sensitive as the information the procedure accesses during execution. To illustrate, consider the following four objects: *jim-personal-agenda*, *jim-working-agenda*, *jim-flights*, and *jim-absences*. Suppose that the first object can be accessed only by *jim*, the second and third by *jim* and *jim-secretary* and the latter by everybody. Consider now a procedure *holiday-planning* that invokes two sub-procedures *get-free-days* and *fill-absences*. The first procedure takes some dates as inputs, reads *jim-personal-agenda* and *jim-working-agenda* and returns the dates for which availability has been obtained. The second procedure writes *jim-absences* accordingly. According to the strict policy, such a process will never be able to complete successfully. The write operation on *jim-absences* would always be blocked because it is requested after some read operations on more protected objects. However, these read operations have all been executed within *get-free-days*, which returns information that can be released to everybody, without disclosing the content of the specific objects read. Therefore an exception can be enforced for the read operations executed within such a procedure so that information it returns can be released to everybody.

The motivation for the second kind of exceptions comes from the observation that the information a process writes into objects may be less sensitive than the information the process has read during its execution. These write operations could therefore be allowed. To illustrate, suppose that procedure *get-free-days* above, before completing, invokes *book-jim-flight*, which writes object *jim-flights*. This write operation is blocked by the strict policy. Note that this is true even with the exception illustrated above, which applies only *after* *get-free-days* has completed, on its reply and not to write operations requested *during* *get-free-days*' execution. To allow the write operation, we therefore need to specify exceptions applicable *within* the execution of the procedure.

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 allowed. Moreover, the specification of exceptions for specific methods naturally fits with the encapsulation and modularity principles of object-oriented systems. The two kinds of exceptions introduced above are specified in our model by means of *waivers* associated with methods: *invoke-waivers*, for specifying exceptions applicable during a method's execution, and *reply-waivers*, for specifying exceptions applicable to the information returned by a method.

The remainder of the paper is organized as follows. Section 2 describes the object-oriented model to which control is applied and introduces the notations that will be used throughout the paper. Section 3 formally defines information transmission and flow. Section 4 describes security specifications and their semantics. Section 5 formally defines safe information flow. Section 6 illustrates how unsafe flows can be controlled by a message filter intercepting every message exchanged between objects and presents an algorithm enforcing this control. Section 7 concludes the paper.

2 Basic assumptions, notations, and definitions

Each real-world entity is represented by an object. Each object has a unique object identifier (*oid*), a set of instance attributes, 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*, N a set of *message names*, and V a set of *values* defined as $V = D \cup I$, where D is a finite set of domains denoting primitive objects. 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*) such that *oid* $\in I$, *attrs*$=(a_1, \ldots, a_n)$, $a_j \in A$ $j = 1, \ldots, n$, *vals*$= (v_1, \ldots, v_k)$, $v_j \in V$ $j = 1, \ldots, k$, and *meths*$\subseteq M$.

Syntax	Description
(read,(a_j),*rp*)	It returns the value of attribute a_j, if a_j is an attribute of the receiver object; it returns failure otherwise
(write,(a_j, v_j),*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
(create,($v_1, \ldots, v_n$),*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
(get,*mid*,*rp*)	It returns the reply of the execution invoked by the message whose identifier is *mid*

Table 1. Primitive messages

Objects interact via message exchanges. Whenever an object receives a message, a 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 the 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 the execution e_k immediately receives a null reply ("nil") from the system and continues executing independently of e_k. No actual reply to the message is sent by e_k to e_h.

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 execution e_k invoked is returned upon explicit request of e_h.

If a message is sent requiring the execution of a method in a specified 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 (n, p, md, rp) where $n \in N$ is the message *name*, $p = (p_1, \ldots, p_k)$, $p_j \in V \cup A$ $j = 1, \ldots, k$, is a list of message *parameters*, $md \in \{\texttt{s,a,dr}\}$ is the *mode* under which the corresponding method must be executed, and $rp = (rp_1, \ldots, rp_n)$, with $rp_j \in V \cup \{\texttt{nil}, \texttt{failure}, \texttt{success}\}$ $j = 1, \ldots, n$, is a list of return values (*reply*).

Read, write, and create operations, and retrieval of replies for executions invoked under deferred mode, are enforced by means of *primitive messages* invoking the corresponding system-defined methods, called *primitive* methods. Primitive methods are elementary in the sense that they cannot invoke other methods, i.e., they cannot send any message. Primitive messages, which are summarized in Table 1, are always 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 denote with the 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 execution is associated with a pair (m, o), where m is the method being executed, and o is the object on which m is executed. The notation $(m, \_)$ is used when the object is not relevant for the discussion. Given an execution e_h, notations $m(e_h)$ and $o(e_h)$ denote the method being executed and the object on which the method is executed, respectively. We will use r, w, and g instead of m, to denote the execution of read, write, and get methods. We denote invocation of execution e_k by e_h by an arrow connecting e_h to e_k with a superscript indicating the interaction mode. Thus, $\rightarrow^{\texttt{s}}$, $\rightarrow^{\texttt{a}}$, $\rightarrow^{\texttt{dr}}$, denote the synchronous, asynchronous, and deferred reply invocation respectively. We will omit the specification of the interaction mode, when its value is not of interest in the explanation.

We assume that whenever a deferred reply execution is required by an execution e_i, the corresponding reply request is subsequently invoked by e_i. In the following, we do not consider the execution invoked by a get message as an actual execution by itself, but rather as a completion of the corresponding deferred reply execution. Given an execution e invoked under deferred reply mode, we use the notation $\widehat{e}$ to denote the execution requiring the reply produced by e.

Two order relationships are defined on executions.

Definition 3 (Invocation order) Let e_h and e_k be two executions in a transaction T, 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 and deferred reply modes, the order with which the executions are invoked does not necessarily coincide with the order with which the executions are completed. For instance, an asynchronous execution invoked before another execution cannot be considered as being completed before the latter. The following definition characterizes the execution order.

Definition 4 (Execution order) Let e_h and e_k be two executions in a transaction T, invoked by the same execution e_j, such that $e_h <_i e_k$. We say that e_h *precedes* e_k, written $e_h <_e e_k$, if and only if one of the following conditions holds: *1)* $e_j \rightarrow^{\mathtt{s}} e_h$; *2)* $e_j \rightarrow^{\mathtt{dr}} e_h \wedge \widehat{e_h} <_i e_k$.

The above definition states that if execution e_j invokes e_h and, subsequently e_k, then e_h precedes 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.[1]

The relationships among executions in a transaction, are graphically represented by a *transaction execution tree* whose root is the execution invoked by the initiator of the transaction. If execution e_h is invoked by execution e_j, e_h is inserted as 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. Primitive method executions are leaves of this tree. An example of transaction execution tree is illustrated in Figure 1. In the figure, the node corresponding to each execution e_i is of the form $e_i : (m_j, o_k)$. No label is associated with arcs corresponding to the default synchronous mode.

3 Information transmission and flow

In this section we characterize the transmission of information between executions and the consequent flow of information between objects.

3.1 Information transmission

Each message enforces *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.

Information between two executions can also be transmitted without any direct message exchanges between the two executions (indirect transmission). For instance if e_i invokes e_h which in turns invokes 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 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.

[1] Remember that `get` messages, like all messages invoking primitive methods, are always sent under `synchronous` mode.

We call *channel* a direct or indirect transmission of information. We distinguish *forward* and *backward* channels according to the "direction" along which information is transmitted.

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 one of the following conditions holds: *1)* $e_h \rightarrow e_k$; *2)* $e_h \rightarrow e_i$ 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_k to e_h if and only if one of the following conditions holds: *1)* $e_h \rightarrow^{md} e_k$, where $md \in \{\mathtt{s}, \mathtt{dr}\}$; *2)* $e_h \rightarrow^{md} e_i$, where $md \in \{\mathtt{s}, \mathtt{dr}\}$, and there exists a backward channel from e_k to e_i.

Information can also be transmitted by employing backward and forward channels jointly. For instance, if e_i invokes e_h and then, after receiving e_h's reply, invokes e_k, there is 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.

Transmission channels can then be defined as follows.

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 one 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)* Let e^* be the least common ancestor of e_h and e_k in the transaction execution tree of T, and let e'_h and e'_k be the ancestors of e_h and e_k invoked by e^*. Then, there exist: *i)* a backward channel from e_h to e^*, *ii)* a forward channel from e^* to e_k, and *iii)* $e'_h <_e e'_k$.

Note that transmission channels characterize *potential* transmission of information that can occur between two executions e_h and e_k. Indeed, if the information received by e_k through the channel is independent from the information transmitted from e_h, then there is no *actual* transmission of information between the two. Determining whether a channel actually enforces transmission of information would require analyzing how the information received by e_k has been obtained [2]. This analysis is outside the scope of this paper. In this paper, we therefore make the assumption that all transmission channels actually enforce transmission of information. Given this assumption, we can now state the following theorem.

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.

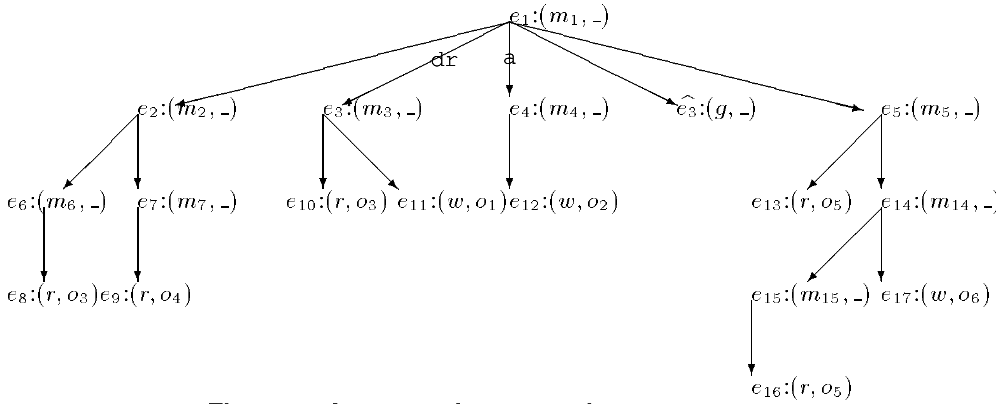

Figure 1. A transaction execution tree

PROOF. (We refer the reader to [4] for the formal proof.)

3.2 Information flow

With the term information flow we denote the fact that information about the status, i.e., the content of an object is acquired and stored by another object. Information flow between two objects obviously requires transmission of information from one object to the other object through some executions. However, transmission of information between two executions operating on different objects does not always enact a flow. We assume that there is a flow only when information is actually read from the *source* object and is stored, through either a write or create operation, in the *recipient* object. In other words, there can exist information flow *from* an object only if information is *read* from the object, and 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 [6] 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 the object's attributes cannot be known unless a read operation is executed on the object. Then, if no read operation is executed on the object no flow from the object is enacted. To ensure that no information about the value of an object's attribute can be hidden inside method specifications, we do not allow methods to change their own code or the code of other methods. Changes to method code must be requested directly by the users (*trusted path*).

Summarizing, information flow from an object o_h to another object o_k requires: *1)* a *read* method execution on the "source" object o_h, *2)* a *write* method execution on the "destination" object o_k, and *3)* a *transmission channel* from the read to the write 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)* $m(e_h)$ = `read`; *2)* $m(e_k)$ =$\{$`write, 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 the flows in T.

Example 1 The transaction illustrated in Figure 1 enacts the following flows: (o_3, e_8, e_{11}, o_1), (o_3, e_8, e_{12}, o_2), (o_3, e_8, e_{17}, o_6), (o_4, e_9, e_{11}, o_1), (o_4, e_9, e_{12}, o_2), (o_4, e_9, e_{17}, o_6), $(o_3, e_{10}, e_{11}, o_1)$, $(o_3, e_{10}, e_{17}, o_6)$, $(o_5, e_{13}, e_{17}, o_6)$, $(o_5, e_{16}, e_{17}, o_6)$.

Note that a flow between two objects may be enacted by different pairs of executions. For instance, in the example above (o_3, e_8, e_{11}, o_1) and $(o_3, e_{10}, e_{11}, o_1)$ are both flows from o_3 to o_1. Also, an execution may enact flows from or to different objects. For instance, execution e_{10} participates in the flows from o_3 to o_1 and to o_6; execution e_{11} participates in the flows from o_3 and o_4 to o_1.

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

4 Security specifications

In this section we illustrate how *security specifications*, i.e., accesses to be or not to be allowed and exceptions, are stated.

We associate with each object an *access control list* (ACL), listing the users who are authorized to access the object. Note that, since we are interested in controlling information flows, we care only of read authorizations.[2] For this reason, we consider a single access control list and do not distinguish between read and write authorizations. For the purpose of this paper, we assume that a user allowed to access an object can read and write the object.

As discussed previously, to take into consideration the methods that participate in the retrieval or transmission of information, we allow *exceptions*, called *waivers*, to the strict policy. Intuitively, these exceptions state that restrictions of the strict policy can be *waived* (from which the name) with reference to the execution of specific methods. Each method m can have associated two kinds of waivers: *reply waivers* (*RW*(m)), which specify exceptions to the information that m returns, and *invoke waivers* (*IW*(m)), which specify exceptions that can be applied on the information that m passes to the executions it invokes. Of course not all methods will have waivers attached to them, only privileged *trusted* methods will. Both *RW* and *IW* are sets of pairs (O, U), where O is a set of objects, and U is a set of users. The semantics of *RW* and *IW* is as follows. If $(O_i, U_i) \in RW(m)$, then the restriction of the strict policy is waived for users in U_i with reference to read operations on objects in O_i executed during m's execution, and write operations to which information has been transmitted through m's reply. In other words, the information in O_i can be released to users in U_i through m's reply (even if users in U_i are not authorized to read objects in O_i). If $(O_i, U_i) \in IW(m)$, then the restriction of the strict policy is waived for read operations on objects in O_i executed before or during m's execution, and write operations executed by m. In other words, method m is allowed to write information read from an object in O_i into another object accessible by users in U_i even if these users were not allowed to read the source object. Note that the strict policy is still applied for users not belonging to U_i.

Figure 2 illustrates an example of security specifications referred to the objects and methods of the transaction in Figure 1. Methods whose reply and invoke waivers are both empty are omitted.

Example 2 According to the security specifications of Figure 2, method m_5 can release, through its reply, information read from object o_5 to Bob. Moreover, m_5 can, during its execution, write information read from o_3 and o_5 into objects accessible by Eric and Frank.

[2] Discretionary access control can be coupled with the information flow control model presented here. For the sake of simplicity, we do not consider discretionary access control here and assume that each operation required in a transaction is authorized.

5 Safe information flow

As illustrated in the previous section, our model supports three different and complementary ways of specifying to which users the information in an object can be released, that is: *i)* by attaching to the object the list of users authorized to access it; *ii)* by including the object and a set of users in the reply waivers of a method; and *iii)* by including the object and a set of users in the invoke waivers of a method.

A flow of information between two objects obeys the security specifications if and only if the users authorized to read the object into which information is written are allowed to access the information flowing into it because either *i)* they are authorized to access the object read, or *ii)* the information can be released to them because of waivers applied during information transmission.

Information transmitted along a channel is subject to the union of the waivers (reply or invoke according to the direction along which the information is transmitted) specified for each method whose execution participates in the transmission.

To formally define safe information flows we first need to introduce some notations. Given a transmission channel from e_h to e_k, we denote with *ch-back*(e_h, e_k) the sequence of executions enforcing the transmission in the channel through their replies, and with *ch-forw*(e_h, e_k) the sequence of executions enforcing the transmission in the channel through the parameters they pass. More precisely, let e^* be the least common ancestor of e_h and e_k: *ch-back*(e_h, e_k) is the sequence of executions in the path from e_h to e^* (this latter excluded); *ch-forw*(e_h, e_k) is the sequence of executions in the path from e^* to e_k. Given a sequence E of executions, $RW^*(E)$ denotes the union of the reply (invoke) waivers of the methods to which the executions in E refer. Analogously, $IW^*(E)$ denotes the union of the invoke waivers of the methods to which the executions in E refer. Formally, $RW^*(e_1, \ldots, e_n)$= $\bigcup_{i=1}^{n} RW(m(e_i))$; $IW^*(e_1, \ldots, e_n)$= $\bigcup_{i=1}^{n} IW(m(e_i))$. Finally, given a set of waivers W, $\Pi_o(W)$ denotes the users for which the restrictions on o are waived. Formally: $\Pi_o(W)$=$\{U \mid (O, U) \in W, o \in O\}$.

Safe information flows can then be defined as follows.

Definition 9 (Safe information flow) An information flow (o_h, e_h, e_k, o_k) is *safe* iff:
ACL(o_k) $\subseteq$ACL(o_h) $\cup$ $\Pi_{o_h}(RW^*($*ch-back*$(e_h, e_k)))$ $\cup$ $\Pi_{o_h}(IW^*($*ch-forw*$(e_h, e_k)))$.

Example 3 Consider the transaction illustrated in Figure 1, whose flows are reported in Example 1, and the security specifications in Figure 2. Flow (o_3, e_8, e_{17}, o_6) is safe, since ACL(o_6)=$\{$Carol,Eric$\}$, and
ACL(o_3) $\cup$ $\Pi_{o_3}(RW^*($*ch-back*$(e_8, e_{17})))$ $\cup$
$\cup$ $\Pi_{o_3}(IW^*($*ch-forw*$(e_8, e_{17})))$ = $\{$Ann,Bob,Carol,

$\mathbf{o}_i$	$\mathbf{ACL(o}_i\mathbf{)}$
o_1	{Ann,Bob,Carol,David}
o_2	{Ann,David}
o_3	{Ann,Bob,Carol,David}
o_4	{Carol,Eric}
o_5	{Ann,Carol}
o_6	{Carol,Eric}

$\mathbf{m}_i$	reply waivers $\mathbf{RW(m}_i\mathbf{)}$	invoke waivers $\mathbf{IW(m}_i\mathbf{)}$
m_2	{({o_3},{Eric}),({o_4},{Ann,Bob})}	$\emptyset$
m_3	{({o_1, o_3},{Frank})}	{({o_4, o_5},{David})}
m_5	{({o_5},{Bob})}	{({o_3, o_5},{Eric,Frank})}
m_6	{({o_2},{Eric,Frank}),({o_3},{Eric})}	$\emptyset$
m_{15}	{({o_4},{Frank})}	$\emptyset$

Figure 2. An example of security specifications

David,Frank,Eric}. Flow $(o_5, e_{16}, e_{17}, o_6)$ is not safe, since ACL(o_6)= {Carol,Eric}, and
ACL(o_5) $\cup\, \Pi_{o_5}(RW^*$(*ch-back*(e_{16}, e_{17}))) $\cup$
$\cup\, \Pi_{o_5}(IW^*$(*ch-forw*(e_{16}, e_{17}))) = {Ann,Carol}.

We say that a write or create operation is *safe* if and only if all flows it enacts are safe.

Definition 10 (Safe write/create) Let e_k be a write or create execution on o_k in a transaction T. e_k is safe iff:
$\forall f \in$ *FLOWS*(T) : $f = (o_h, e_h, e_k, o_k) \Rightarrow f$ is safe.

For instance, with reference Example 3, write execution e_{17} is not safe.

6 The message filter

The *message filter* [6] is a trusted system component which intercepts every message exchanged among the objects in a transaction to guarantee that no unsafe flow takes place. To check whether a write or a create operation should be blocked, the message filter keeps track of the information transmitted between executions and of the waivers applicable to it. Information transmitted between executions is denoted by using the identifiers of the objects from which the information has been read. To keep track of the waivers applicable to the information, we introduce the concept of *folder* attached to an execution. A folder is a set of pairs (o_i,Ξ_i), where o_i is an object identifier, and $\Xi_i = \{(\Psi_{1i}, \Phi_{1i}), \ldots, (\Psi_{mi}, \Phi_{mi})\}$ a set of boolean formulae whose variables are methods' names. Each pair (Ψ_{ji}, Φ_{ji}) indicates the reply (Ψ_{ji}) and invoke (Φ_{ji}) waivers that apply to the information in o_i according to the transmission channel the information has traversed. Intuitively, the fact that a method m appears in the Ψ_{ji}/Φ_{ji} in a folder means that the reply/invoke waivers of m are to be applied for the information in o_i carried by the execution to which the folder refers. The boolean connectives in the formula indicate whether the waivers of the methods must be considered in conjunction or in disjunction.

In the following, we denote with $\mathcal{F}$ the set of all the boolean formulae whose variables are method names in M. We use ϵ to denote the empty formula. Given a folder F, we denote with $F.O$ the set $\{o \mid (o, \Xi) \in F\}$. Analogously, given a set of waivers W, we denote with $W.O$ the set $\{o \mid o \in O, (O, U) \in W\}$.

Example 4 The folder $F = \{(o_3, \{(m_6 \vee m_2, \epsilon)\}),$ $(o_4, \{(m_2, m_3 \vee m_4), (m_5, m_1)\}\}$ indicates that information from objects o_3 and o_4 is being transmitted. The information read from o_3 is subject to the union of the reply waivers of m_6 and m_2. The information read from o_4 is subject to: *i)* the union of the reply waivers of m_2 and the invoke waivers of m_3 and m_4 along one channel, and *ii)* the union of the reply waivers of m_5 and the invoke waivers of m_1 along the other channel.

Each execution passes to the executions it invokes, and receives back from them, a folder indicating the information collected during traversal of the channels and the corresponding waivers to be applied to it. Folders transmitted between executions are incrementally built by means of two functions, namely *release_back*() and *release_forw*(). 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.

Function *release_back*() is formally defined as follows.

Function 1 release_back(): $M \times 2^{I\times\mathcal{F}\times\mathcal{F}} \rightarrow 2^{I\times\mathcal{F}\times\mathcal{F}}$
Let $F = \{(o_1, \Xi_1), \ldots, (o_n, \Xi_n)\}$ be a folder and e an execution, where $\Xi_i = \{(\Psi_{1i}, \Phi_{1i}), \ldots, (\Psi_{mi}, \Phi_{mi})\}$, $i = 1, \ldots, n$. *release_back*$(e, F) = \{(o_1, \Xi'_1), \ldots, (o_n, \Xi'_n)\}$, where $\Xi'_i = \{(\Psi'_{1i}, \Phi_{1i}), \ldots, (\Psi'_{mi}, \Phi_{mi})\}$, $i = 1, \ldots, n$, and each Ψ'_{ji} is as follows:

$$\Psi'_{ji} = \begin{cases} \Psi_{ji} \vee m(e) & \text{if } o_i \in RW(m(e)).O \\ \Psi_{ji} & \text{otherwise} \end{cases}$$

Intuitively the function modifies the folder F by taking into consideration the reply waivers of execution e. In particular, since these reply waivers must be considered in addition to those waivers already applicable, the folder is modified by connecting method $m(e)$ in *or* to each formula Ψ_{ji} associated with each object o_i for which some waivers are to be applied.

Function *release_forw*() is defined as follows.

Function 2 release_forw(): $M \times 2^{I \times \mathcal{F} \times \mathcal{F}} \rightarrow 2^{I \times \mathcal{F} \times \mathcal{F}}$
Let $F = \{(o_1, \Xi_1), \ldots, (o_n, \Xi_n)\}$ be a folder and e an execution, where $\Xi_i = \{(\Psi_{1i}, \Phi_{1i}), \ldots, (\Psi_{mi}, \Phi_{mi})\}$, $i = 1, \ldots, n$. *release_back*$(e, F) = \{(o_1, \Xi'_1), \ldots, (o_n, \Xi'_n)\}$, where $\Xi'_i = \{(\Psi_{1i}, \Phi'_{1i}), \ldots, (\Psi_{mi}, \Phi'_{mi})\}$, $i = 1, \ldots, n$, and each Φ'_{ji} is as follows:

$$\Phi'_{ji} = \begin{cases} \Phi_{ji} \vee m(e) & \text{if } o_i \in IW(m(e)).O \\ \Phi_{ji} & \text{otherwise} \end{cases}$$

Intuitively, the function modifies the folder F, indicating the information available to e up to that point, by taking into consideration the invoke waivers of e. In particular, each formula Φ_{ji} in F is modified by connecting $m(e)$ in *or* to each formula Φ_{ji} associated with an object o_i for which some invoke waivers are applicable.

Example 5 Consider execution e_2 in the transaction execution tree of Figure 1.
Let $F = \{(o_3, \{(m_6, \epsilon)\}), (o_4, \{(\epsilon, \epsilon)\})\}$ be the folder execution e_2 has collected from e_6 and e_7. The folder e_2 passes to e_1 is F' =*release_back*(e_2, F) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, \epsilon)\})\}$, since the reply waivers of method m_2 apply to both o_3 and o_4. Suppose now that F' is the folder indicating the information available to e_3 before invoking e_{11}. The folder m_3 passes to e_{11} is: *release_forw*(e_3, F') = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, m_3)\})\}$.

An execution e receives information, and corresponding folders, from the different executions it invokes. These folders must therefore 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 objects appearing in the folder describing the information collected by e during execution must be equal to the union of all the objects appearing in the folders that e received back from the executions it invoked. Let us now see how waivers must be combined. Intuitively, when an execution collects the information returned by the executions it has invoked, it merges different transmission channels. The reply waivers in the folder sent back by an execution apply only along the channel the information has traversed for arriving to that execution. These waivers do not apply to other channels the information read in the same object through other read operations may have traversed. According to this principle, when combining folders returned by the execution it invokes, an execution e must consider the *intersection* of the reply waivers indicated in the different folders. To calculate the combination of folders, we introduce the operation of *union-and*, denoted as $\cup^\wedge$. The operation of *union-and* applies only to folders of the form $F = \{(o_1, \Xi_1), \ldots, (o_n, \Xi_n)\}$, such that each Ξ_i consists of a unique pair (Ψ_{1i}, Φ_{1i}), with $\Phi_{1i} = \epsilon$.[3] The union-and operator merges two folders F' and F'' by producing a new folder which contains the union of the objects in $F'.O$ and $F''.O$ together with the intersection of the waivers applicable to them. This is formally defined as follows.

Definition 11 (Union-and operator) Let
F'=$\{(o'_1, \Xi'_1), \ldots, (o'_m, \Xi'_m)\}$, F''=$\{(o''_1, \Xi''_1), \ldots, (o''_n, \Xi''_n)\}$ be two folders such that $\Xi'_i = \{(\Psi'_{1i}, \epsilon)\}$, $i = 1, \ldots, m$ and $\Xi''_j = \{(\Psi''_{1j}, \epsilon)\}$, $j = 1, \ldots, n$.
$F' \cup^\wedge F'' = \{(o_1, \Xi_1), \ldots, (o_k, \Xi_k)\}$, where $F.O = F'.O \cup F''.O$. For each $i = 1, \ldots, k$, $\Xi_i = \{(\Psi_{1i}, \epsilon)\}$ where

$$\Psi_{1i} = \begin{cases} \Psi'_{1i} & \text{if } o_i \in F'.O \setminus F''.O \\ \Psi''_{1i} & \text{if } o_i \in F''.O \setminus F'.O \\ \Psi'_{1i} \wedge \Psi''_{1i} & \text{if } o_i \in F''.O \cap F'.O \end{cases}$$

We are now ready to define backward and forward folders.

Definition 12 (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: *i)* $\{(o(e_k), \{(\epsilon, \epsilon)\})\}$, if $m(e_k)$ = `read`; *ii)* the empty set, if $m(e_k)$ $\in$\{`write`, `create`\} or $e_h \rightarrow^{\mathtt{a}} e_k$; *iii)* $release\_back(e_k, \bigcup^\wedge_z \mathrm{BACK}(e_z, e_k))$, otherwise.

Definition 12 states that the folder transmitted from execution e_k to its invoker e_h is: *i)* the set composed of the object read and of the set containing the two empty formulae, 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 waivers of execution e_k to the union-and of the information e_k has received by the executions it has invoked, otherwise. Note that, according to Definition 12, for each pair of executions e_h and e_k, the folder BACK(e_k, e_h) has the form $\{(o_1, \{(\Psi_{11}, \epsilon)\}), \ldots, (o_k, \{(\Psi_{1k}, \epsilon)\})\}$. This explains why the union-and operator has been defined on waivers of this form.

Definition 13 (Forward folder) Let e_h and e_k be two method 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: *i)* the empty set, if $o(e_h)$ is a user in U; *ii)* $release\_forw(e_h, F)$, otherwise, where $F = \bigcup^\wedge_z \{\mathrm{BACK}(e_z, e_h) \mid e_z <_e e_k\} \cup \mathrm{FORW}(e_j, e_h)$, with e_j the execution that invoked e_h.

Definition 13 states that the folder transmitted from execution e_h to the invoked execution e_k is equal to: *i)* the

[3] The reason for defining the union-and operator only for these particular folders is that the folder an execution sends back to its invoker is of this type. We will clarify this in the remainder of the discussion.

Backward folders	**Forward folders**
BACK(e_8, e_6) = $\{(o_3, \{(\epsilon, \epsilon)\})\}$	FORW(e_1, e_3) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, \epsilon)\})\}$
BACK(e_6, e_2) = $\{(o_3, \{(m_6, \epsilon)\})\}$	FORW(e_3, e_{10}) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, m_3)\})\}$
BACK(e_9, e_7) = $\{(o_4, \{(\epsilon, \epsilon)\})\}$	FORW(e_3, e_{11}) = $\{(o_3, \{(\epsilon, \epsilon), (m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, m_3)\})\}$
BACK(e_7, e_2) = $\{(o_4, \{(\epsilon, \epsilon)\})\}$	FORW(e_1, e_4) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, \epsilon)\})\}$
BACK(e_2, e_1) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}),$ $(o_4, \{(m_2, \epsilon)\})\}$	FORW(e_4, e_{12}) = $\{(o_3, \{(m_6 \vee m_2, \epsilon)\}), (o_4, \{(m_2, \epsilon)\})\}$
BACK(e_{10}, e_3) = $\{(o_3, \{(\epsilon, \epsilon)\})\}$	FORW(e_1, e_5) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, \epsilon)\}), (o_4, \{(m_2, \epsilon)\})\}$
BACK(e_3, e_1) = $\{(o_3, \{(m_3, \epsilon)\})\}$	FORW(e_5, e_{13}) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, m_5)\}), (o_4, \{(m_2, \epsilon)\})\}$
BACK(e_{13}, e_5) = $\{(o_5, \{(\epsilon, \epsilon)\})\}$	FORW(e_5, e_{14}) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, m_5)\}), (o_4, \{(m_2, \epsilon)\}), (o_5, \{(\epsilon, m_5)\})\}$
BACK(e_{16}, e_{15}) = $\{(o_5, \{(\epsilon, \epsilon)\})\}$	FORW(e_{14}, e_{17}) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, m_5)\}), (o_4, \{(m_2, \epsilon)\}),$ $(o_5, \{(\epsilon, m_5), (\epsilon, \epsilon)\})\}$
BACK(e_{15}, e_{14}) = $\{(o_5, \{(\epsilon, \epsilon)\})\}$	FORW(e_{14}, e_{15}) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, m_5)\}), (o_4, \{(m_2, \epsilon)\}), (o_5, \{(\epsilon, m_5)\})\}$
BACK(e_{14}, e_5) = $\{(o_5, \{(\epsilon, \epsilon)\})\}$	FORW(e_{15}, e_{16}) = $\{(o_3, \{((m_6 \vee m_2) \wedge m_3, m_5)\}), (o_4, \{(m_2, \epsilon)\}), (o_5, \{(\epsilon, m_5)\})\}$
BACK(e_5, e_1) = $\{(o_5, \{(m_5, \epsilon)\})\}$	

Figure 3. Backward and forward folders for the transaction in Figure 1 assuming the security specifications in Figure 2

empty set, if e_k is invoked directly by a user,[4] *ii)* the result of the application of the invoke waivers 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 of the folder that e_h obtains by combining those received back before invoking e_k and the folder that e_h received from its invoker.

Figure 3 shows the backward and forward folders for the transaction of Figure 1, assuming the security specifications of Figure 2. For the sake of simplicity, empty folders are omitted.

Waivers applicable to o_i are represented by the set Ξ_i associated with the object o_i in the folder passed to the write/create operation. For each pair (Ψ_{ji}, Φ_{ji}) in Ξ_i the set of users to whom the information in o_i can be released because of reply waivers, denoted by *rep_waive*$(o_i, \Psi_{ji}, \Phi_{ji})$, is computed by taking formula Ψ_{ji} and replacing each method m with $\Pi_o(RW(m))$, each symbol $\vee$ with $\cup$, and each symbol $\wedge$ with $\cap$. Analogously, the set of users to whom information can be released because of invoke waivers, denoted by *inv_waive*$(o_i, \Psi_{ji}, \Phi_{ji})$, is computed by taking formula Φ_{ji} and replacing each method m with $\Pi_o(IW(m))$, each symbol $\vee$ with $\cup$, and each symbol $\wedge$ with $\cap$.

We can now express safe write/create operations.

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 write/create execution on object o_k. e_k is safe iff: $\forall(o_i, \Xi_i) \in$ FORW(e_h, e_k), $\forall(\Psi_{ji}, \Phi_{ji}) \in \Xi_i$: ACL$(o_k)$ $\subseteq$ACL$(o_i)\cup$ *rep_waive*$(o_i, \Psi_{ji}, \Phi_{ji})\cup$ *inv_waive*$(o_i, \Psi_{ji}, \Phi_{ji})$.

PROOF (We refer the reader to [4] for the formal proof.)

[4] 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.

Example 6 Consider the transaction of Figure 1, the folders in Figure 3, and the security specifications of Figure 2. $e_{11} = (w, o_1)$ is safe since all the pairs in FORW(e_3, e_{11}) satisfy the condition of Theorem 2. $e_{12} = (w, o_2)$ is not safe since the pair $(o_4, \{(m_2, \epsilon)\})$ $\in$FORW(e_4, e_{12}) does not satisfies the condition in Theorem 2. Indeed, ACL(o_2)=$\{$`Ann`,`David`$\}$, and ACL$(o_4)\cup$ *rep_waive*$(o_4, m_2, \epsilon)\cup$ *inv_waive*(o_4, m_2, ϵ) = $\{$`Ann`,`Bob`,`Carol`,`Eric`$\}$.
$e_{17} = (w, o_6)$ is not safe because $(o_5, \{(\epsilon, m_5), (\epsilon, \epsilon)\}) \in$ FORW(e_{14}, e_{17}) and ACL$(o_6) \not\subseteq$ ACL(o_5).

6.1 The message filtering algorithm

In this subsection we present an algorithm enforcing the message filtering. 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 execution would enact. Execution of any method different from write is always allowed by the message filter. However, every message must be intercepted to properly construct folders and to enforce the different interaction modes. Safety of create operations is guaranteed by initializing the ACL of the created object to the set containing only the transaction initiator. Execution of a write operation is allowed only if the safety condition given in Theorem 2 is satisfied.

In order to enforce the control the message filter maintains, for each execution e, 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 the execution e_i invoking it, i.e., $F_e =$ FORW(e_i, e). $B(e)$ at a given point in e's execution is equal to the backward folders received by e up to that point from the executions it has invoked. The B_e associated with e at the time it invokes an execution e_k is

Algorithm 1 The message filtering algorithm

INPUT: A message $msg = (n, p, md, rp)$ sent by e_h, running on o_h to require an execution e_k on o_k.
OUTPUT: It returns the reply according to the interaction mode specified in msg if e_k does not enforce unsafe flows. It denies e_k and returns `failure` otherwise.

If $n \neq$ `get`
  then $B_{e_k} := \emptyset$
    If o_h is a user
      then $F_{e_k} := \emptyset$
        execute e_k
        rp := reply from e_k
        return *rp* **to** e_h
      else $F_{e_k} :=$ *release_forw*$(m(e_h), F_{e_h} \cup B_{e_h})$
endif
Case msg **of**
  1) $($`read`$, a, md, rp)$:
    execute e_k
    rp := reply from e_k
    $B_{e_h} := B_{e_h} \cup^{\wedge} \{(o_k, \{(\epsilon, \epsilon)\})\}$
    return *rp* **to** e_h

  2) $($`write`$, (a, v), md, rp)$:
    If $\forall (o_i, \Xi_i) \in F_{e_k}, \forall (\Psi_{ji}, \Phi_{ji}) \in \Xi_i$:
      ACL$(o_k) \subseteq$ ACL$(o_i) \cup$
      rep_waive$(o_i, \Psi_{ji}, \Phi_{ji}) \cup$
      inv_waive$(o_i, \Psi_{ji}, \Phi_{ji})$
      then execute e_k;
        rp := reply from e_k
    else *rp*:= `failure`
      return *rp* **to** e_h

  3) $($`create`$, (v_1, \ldots, v_k), md, rp)$:
    execute e_k
    ACL$(o) :=$ {transaction initiator}
    rp:= oid of o
    return *rp* **to** e_h

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

  5) (n, p, md, rp) and $n \notin \{$`read`, `write`, `create`, `get`$\}$:
    case md **of**
      a) `s`:
        execute e_k
        rp := reply from e_k
        $B_{e_h} := B_{e_h} \cup^{\wedge}$ *release_back*$(m(e_k), B_{e_k})$
        return *rp* **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
        rp:=reply from e_k
        $B'_{e_k} :=$ *release_back*$(m(e_k), B_{e_k})$
        insert $(mid,\ e_h,\ rp,\ B'_{e_k})$ into the reply queue
endcase

Figure 4. The message filtering algorithm

therefore $B_e = \bigcup_z^{\wedge} \{\text{BACK}(e_z, e) \mid e_z <_e e_k\}$. To support deferred reply interaction mode, the message filter maintains a *reply queue* which stores the replies of executions invoked under `dr` mode.

The message filtering algorithm reported in Figure 4 works as follows. Suppose that an execution e_h, running on o_h, sends a message *msg*=(*n*,*p*,*md*,*rp*) to o_k to invoke 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, F_{e_k} is initialized to the empty set, execution e_k is performed and its reply is returned to the user, if the message has been sent by the transaction initiator. Otherwise, F_{e_k} is set equal to the result of function *release_forw*$()$ applied to method $m(e_h)$ and to the union of the forw-folder and back-folder of e_h.

After this initialization process executed only for messages different from get, the algorithm determines how to execute e_k depending on the method to be executed and the interaction mode. Cases 1-4 consider the execution of primitive methods, whereas cases 5a-5c consider the execution of non-primitive methods under the different interaction modes. Let us examine the different cases.

1. *Read* message. The read method is invoked and its reply is returned to e_h. The back-folder of e_h is updated by taking its union-and with $\{(o_k, \{(\epsilon, \epsilon)\})\}$.

2. *Write* message. The algorithm determines if the write is safe by checking the satisfaction of the condition given in Theorem 2. If the condition is satisfied, the write operation is executed and its reply returned, otherwise the write execution is blocked and a `failure` is returned.

3. *Create* message. The create method is executed and the identifier of the new object is returned to e_h. The ACL of the created object is initialized to the set containing only the identifier of the transaction initiator.

4. *Get* message requiring the reply of execution e_k. The algorithm retrieves the tuple corresponding to execution e_k from the reply queue. The back-folder of 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. *Non-primitive* message requiring the execution of e_k. If e_k is invoked under synchronous mode, e_k is performed and its reply is returned to e_h. The back-folder of e_h is updated by taking its union-and with the result of function *release_back*$()$ with, as input, $m(e_k)$ and the back-folder of e_k (step 5a). If e_k is invoked under asynchronous mode, a `nil` reply is returned to

e_h. Execution e_k is performed and its actual reply discarded (step 5b). If 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 the result of function *release_back*() with, as input, $m(e_k)$ and the back-folder of e_k. Finally, a tuple composed of the reply of e_k, together with the generated message identifier, e_h, and folder B'_{e_k} is inserted in the reply queue (step 5c).

7 Conclusions

We have proposed a flexible information flow model for object-oriented systems which allows exceptions to the strict policy. Exceptions, which can be specified with reference to specific objects and to specific users, are stated by means of waivers associated with methods. Two types of waivers are supported: invoke waivers, specifying exceptions applicable during a method's execution, and reply waivers, specifying exceptions applicable to the information returned by a method. We have formally defined information transmission and flow and given conditions for an information flow to be safe. We have proposed an algorithm that blocks unsafe write operations.

Our model leaves space for further work. A first issue to be investigated concerns the enrichment of exception specifications. For instance, conditions can be added to waivers and the exceptions indicated in them applied only if these conditions are satisfied. In the paper we have assumed that the objects on which exceptions are to be applied are always explicitly specified. The use of variables in the specification can also be investigated. Moreover, the application of waivers can be restricted to specific operations within a method instead of holding through its whole execution. Another issue concerns the combination of waivers. In our model information is subject to the union of all the waivers encountered along a channel. Different combination operators, such as those proposed in [2] for the combination of access classes, can instead be desired. A further issue to be investigated concerns the certification of trusted methods.

References

[1] W.E. Boebert and C.T. Ferguson. A partial solution to the discretionary Trojan horse problem. In *Proc. of the* 8^{th} *Nat. Computer Security Conf.*, pp. 141–144, Gaithersburg, MD, 1985.

[2] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, pp. 236–243, May 1976.

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

[4] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing Flexibility in Information Flow Control for Object-Oriented Systems. Tech. Report, Dip. di Scienze dell' Informazione, Università di Milano, Nov. 1996.

[5] INFORMIX-*Online/Secure. Security Features Users's Guide*. Informix Software,Inc., 1993.

[6] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *Proc. IEEE Symp. on Security and Privacy*, pp. 76–85, Oakland, CA, 1990.

[7] P.A. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proc. IEEE Symp. on Security and Privacy*, pp. 32–37, Oakland, CA, 1987.

[8] C.J. McCollum, J.R. Messing, and L. Notargiacomo. Beyond the pale of MAC and DAC - Defining new forms of access control. In *Proc. IEEE Symp. on Security and Privacy*, pp. 190–200, Oakland, CA, 1990.

[9] *Trusetd* ORACLE *Administrator's Guide*. Oracle Corporation, 1996.

[10] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *IEEE-Trans. on Knowledge and Data Engineering*. To appear.

[11] A. Stoughton. Access flow: A protection model which integrates access control and information flow. In *Proc. IEEE Symp. on Security and Privacy*, pp. 9–18, Oakland, CA, 1981.

[12] SYBASE *Secure* SQL *Server*. Sybase, Inc., 1993.

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

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