Information Flow Control using Versions in Object-Oriented Systems

A. Fellah *
Dept. of Math. and Computer Science, University of Lethbridge, Lethbridge, AB, Canada T1K 3M4

A. Maamir
Dept. of Computer Science, University of Sharjah, Sharjah, United Arab Emirates

I. Rahwan
Dept. of Information Systems, University of Melbourne, Parkville 3010, Australia

Abstract. One of the main features of information flow control is to ensure the enforcement of privacy, secrecy, and confidentiality. However, most information flow models that have been proposed are too restrictive, overprotected, and inflexible. This paper presents an approach to control flow information in object-oriented systems using versions, thus allowing considerable flexibility without compromising system security by disclosing and leaking sensitive information. Models based on message filtering intercept every message exchanged among objects to control the flow of information. Versions are proposed to provide flexibility and avoid unnecessary and undesirable blocking of messages during the filtering process. Two options of operations are supported by versions – cloning reply and non-cloning reply. Furthermore, we present an algorithm enforcing the message filtering through these operations.

Keywords: Information flow control, discretionary access control, mandatory access control, role based access, security.

1. Introduction

Applications requiring secrecy and confidentiality are growing in numbers – electronic commerce, mobile computing, intranets and large network systems such as commercial multiusers database systems and object-oriented database systems are a few examples. Secrecy ensures that users access only information that they are allowed to see. Confidentiality ensures the protection of private information, such as payroll data, employees or customers records, as well as sensitive corporate data, such as internal memos or competitive strategy documents. There has been a general consensus that security is the key to the success of these applications. Therefore we need effective mechanisms and policies to prevent accidental destructions and malicious attacks, and to control the disclosure and propagation of the information to users who are not

* Corresponding author (fellah@cs.uleth.ca)

allowed access to the information. There has been much research on information flow controls and access control models to design and implement secure object-oriented systems. Various kinds of access control models have been studied in literature, see for example (Samarati et al., 1997; Sandhu and Samarati, 1996; Denning, 1976; Ferrari et al., 1997; Myers and Liskov, 1998). Information flow models are intended to address secrecy and privacy problems, however most of them are too restrictive to be used. Decentralized label model (Myers and Liskov, 1997; Myers and Liskov, 1998) have been introduced to improve traditional models in several ways, making them more flexible by attaching flow policies to pieces of data. Another method to loosen the strict information flow policies of the work in (Samarati et al., 1997) has been proposed by the same authors in (Ferrari et al., 1997). The method allows exceptions (waivers) which can be specified with reference to specific objects and users without disclosing sensitive information. It has been pointed out in previous and recent work, see for example (Samarati et al., 1997; Ferrari et al., 1997; Myers and Liskov, 1997; Fellah et al., 1999), the need for improving flexibility in information flow policies without compromising system security by disclosing sensitive information.

Information discretionary and mandatory controls are the most common types of access control mechanism used in today's computer systems. In the access control model, an access rule is specified in a form $\prec s, o, op \succ$, where a subject s is allowed to access or manipulate an object o through an operation op, e.g., read, write, execute. The permission to perform a certain operation on an object is said to be an *access right*. In general, access control mechanisms are discretionary which restrict access to objects based solely on the identity of subjects which are trying to access them. However, this basic principle of discretionary access control (DAC) contain an illegal flow of information, that is, a subject which is granted access to an object can propagate and pass on the information along to another subject. Data in an object may be obtained via other objects by unauthorized subjects of the object. For example, if user u_1 is allowed to read u_2'*s* data, but on the other hand u_2'*s* does not allow u_3 to read it. u_2 cannot control how u_1 distributes the information it has read. u_1 can read the information from u_2 and then propagates it to u_3 since a copy of the information is now owned by u_1. A *Trojan horse* is a computer program which works in a similar way, leaking information despite the discretionary access control. The main drawback of DAC mechanisms is that they do not impose any control on the flow of information in the systems. Thus, this makes discretionary policies vulnerable to Trojan horses and DAC cannot deter hostile attempts to access sensitive information. A DAC

mechanism allows users to grant or revoke access privileges to any of the objects under their controls. Such users can be corporations or agencies which are the actual owner of system objects as well as the programs that process them.

Another access control model is the mandatory access control (MAC) which restricts access to objects based on the sensitivity of the information and the authorization of subjects to access such informations. Security labels (i.e., clearance) are associated with each object and subject reflect the subject's trust level not to disclose sensitive information to subjects not cleared to see it. Every entity, i.e., subject and object, is classified to a security class. These mandatory policies are not particularly well suitable for the requirements of organizations that process unclassified but sensitive information.

Recently a family of reference models for role based access control (RBAC) have been proposed and investigated in research, see for example (Sandhu, 1998; Sandhu et al., 1996; Ferrariolo et al., 1995; Ferrariolo, Barkley and Khuh, 1999). RBAC can be viewed as an alternative to traditional discretionary (DAC) and mandatory access control (MAC) policies that is particularly attractive for commercial applications. The RBAC model has extended the framework access model to include role hierarchies. In RBAC, access decisions are based on the roles and responsibilities of each user in the organization's structure. A *role* can be defined to be a collection of access rights, which represent a set of job functions in the organization. Each user is assigned one or more roles, and each role is assigned one or more privileges. For example, within a hospital system access rights and decisions are based on the roles that medical personnel can play in the organization. The potential role of a Doctor can include prescribing medications, provide treatments and interpret an imaging diagnosis; and the roles of a nurse can include provide care for patients, measure vital signs, and monitor drug administration; and the role of a medical assistant may include take health histories, and perform laboratory tests. Roles can be hierarchical, mutually exclusively structured or collaborative or overlapping. For example, in a hospital some roles are hierarchical. The doctor role may include all privileges granted to the nurse role, which in turn includes all privileges granted to the medical assistant role. Role hierarchies are a natural generalization of organizing roles for granting responsibilities and privileges within an organization. RBAC is used particularly for commercial applications, because it reduces the cost of security administration and the complexity of managing large networked systems. For example, RBAC has been implemented on web servers and particularly to an intranet computing environment in (Ferrariolo, Barkley and Khuh, 1999).

Although access control is still discretionary in the model of (Samarati et al., 1997), the overall flexibility is reduced by the application of the very tight and strict policy. In this paper, we propose an approach based on versions (cloneability) to overcome the problem of flexibility of the work in (Samarati et al., 1997). Versions make access control improve flexibility without reducing the potential for information leaks and disclosure. Access right controls described in (Samarati et al., 1997) have several features and can be considered as an applicable methodology for security, however it has some drawbacks which will be discussed in Section 3. The remainder of the paper is organized as follows. Section 2 describes the object-oriented model and introduces the basic terminology used throughout the paper. Section 3 summarizes the overall inflexibility of the strict policy of the work in (Samarati et al., 1997) using different access control lists. Section 4 adds a great deal of flexibility to the information flow model of (Samarati et al., 1997) through versions while keeping tighter control for security problems. We propose the use of versions and further extend the authorization model (Samarati et al., 1997) to avoid unnecessary and undesirable blocking. Section 5 illustrates our approach for controlling the flow of information using a message filter intercepting messages exchanged among objects. It also presents an algorithm using versions for enforcing such control. Finally, in section 6 we conclude our work and give an outlook to future work.

2. Object-Oriented Model

Object-oriented systems are composed of objects. There are many definitions of an object, such as *objects* are encapsulation of data (*state*) and related functionality *methods* for manipulating the data. *Classes* are prototypes of objects. An object is a physical implementation of a class, an *instance* of a class. A class is defined to be a set of *attributes* and *methods*. A class may have many instance objects. Objects interact and communicate by passing messages. A method of an object is invoked by sending a message to the object. Access to attributes of an object is also based on the message-passing paradigm. Messages are the means by which objects communicate. For each message, a corresponding method is executed. If an object wants to access an attribute of another object, it sends a message requiring the execution of a method that reads that attribute and returns it to the sender. If the sender object is authorized to read that attribute, the reply(*success* or *failure*) is allowed to pass, otherwise it is blocked. New classes can be created by reusing (i.e., inheriting) attributes and methods from

other existing classes. However, inheritance is beyond the scope of this paper. Further research considerations with respect to the propagation of access rights through inheritance hierarchies is being investigated and addressed in a paper in preparation.

In this paper, we assume that the system is composed of multiple objects. Each object o in the system is characterized by a unique object identifier o_{id} that is associated with a set of attributes and methods. A *transaction* is a set of method executions invoked as a result of the receipt of a message sent by an object. Let e_i denote the execution of the i^{th} method invoked during a transaction call.

3. Discussion of the Authorization Model

Access control lists (ACL) associate an object with a list of users U and groups to specify security policy operations on that object. An operation on the object may be performed by a user if that user or a group to which that user belongs is listed in the ACL associated with the object. ACL specify an object, a list of users, together with a set of granted or denied operations (i.e., access rights). This allows a particular user to read, write, and create instance of the objects. Access to ACL should be protected just as other object are protected. ACL-based mechanisms have been conveniently been used in (Samarati et al., 1997) to overcome the vulnerability of discretionary access control to Trojan horses leaking information to unauthorized users.

In the authorization model (Samarati et al., 1997), three kinds of access control lists, read access control list (RACL), write access control list (WACL), and create access control list (CACL), have been considered, each of which is associated with an object. RACL(o) contains all users who are allowed to read object o. WACL(o) and CACL(o) are defined similarly. Both synchronous and asynchronous executions are allowed in (Samarati et al., 1997) to provide some kind of flexibility in the application of the policy. Asynchronous executions are allowed to permit independence between operations. That is, after the execution is invoked no reply is expected or returned, and the invoking execution could progress without being blocked until a reply comes. Another type of executions allowed are deferred reply executions, in which the invoking execution progresses after sending the message up to a point where it needs the reply. At this stage, the reply is retrieved in response to an explicit request. Moreover, the model in (Samarati et al., 1997) considered executions that can be restricted or unrestricted. Suppose an object o_1 invokes an execution e_1, which in turn invokes another execution e_2 on another object o_2. If e_2 is executed as restricted, the

reply of that method is blocked if the object waiting for it, o_1 in this case, is accessible to users not allowed access to information in that reply (i.e., not allowed to access object o_2). By contrast, if a method is executed as unrestricted, no constraints are enforced on the reply, but the corresponding write operation on the object o_1, performed later by e_1, will still be blocked. This explains the notion of blocking information flow at the time of information transmission and at the time of information acquisition in order to allow for even more flexibility in the system. The main advantage of using the unrestricted option is to allow information to propagate through the methods, which might need that information for local computation, while ensuring this information will still be protected against illegal acquisition. Checking whether the information in o_2 can be written to o_1 is done by testing whether the read access control list of o_2, denoted by RACL(o_2), contains RACL(o_1). If RACL(o_2) $\supseteq$ RACL(o_1) then the information in o_2 can be written to o_1. In this case, even o_2 acquires that information no other unauthorized users will be able to read it since they are a subset of the users allowed to read o_2 . Otherwise, if RACL(o_2) $\subseteq$ RACL(o_1) then either the reply will be blocked or the actual corresponding write operation will not be allowed, depending on whether the method has been sent as restricted or unrestricted.

Access right controls, as described in (Samarati et al., 1997) have several features and can be considered as an applicable methodology for security, however it has some drawbacks which can be illustrated in the following example. Assume we have the following four objects: o_1, o_2, o_3, and o_4 such that RACL(o_1) = $\{o_1, o_3, o_4\}$, RACL(o_2) = $\{o_1, o_2, o_4\}$. Both RACL are shown in Fig. 1, where an arrow from object o_i to object o_j means that o_i can read the information contained in o_j.

If the object o_1 tries to read the information in o_2, then o_1 will never be allowed to perform such operation according to the strict policy of (Samarati et al., 1997). The reasons are: first, o_3 is allowed access to o_1 but not to o_2; second, the object o_1 is not trusted by the object o_2 to disclose and release the information to o_3. Therefore no write operation will never be performed on o_1 that contains any information from o_2. This information flow blocking will not only affect o_1 but it will also block o_4 from accessing it through o_1. This information might be critical to o_4 and should be accessed through o_1 but not directly from o_2 because o_1 may operate on the information and update the data. Therefore the write operation is blocked by the strict policy (Samarati et al., 1997). In this case, o_4 is unnecessarily blocked from accessing that information because the object o_3 is not allowed to read it. In object-oriented systems, a method invoked by a subject may invoke other methods. Thus, many methods can be invoked in a nested manner.

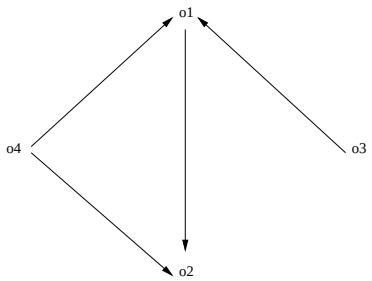

Figure 1. Information flow using RACLs.

Therefore in large systems with many objects such method invocation can trigger a blocking mechanism where write operations on objects are extremely unnecessarily limited. We add more flexibility to the work in (Samarati et al., 1997) by using versions to avoid undesirable blocking. We propose the use of *versions* and further extend the reference authorization model of (Samarati et al., 1997).

4. Adding Non-Sterilized Versions

Our approach uses the concept of *versions* which allow considerable flexibility to overcome the restrictive policy and undesirable blocking of the model in (Samarati et al., 1997). For reasons which become apparent below, we denote objects by $o_i{:}\,0$ instead of just o_i and we also assume that all objects are "cloneable". For example, cloning in Java is not automatically available even though the *clone*( ) method is defined in the base-of-all classes *Object*. Let $o_i{:}\,0$ denotes the source (i.e., parent) version. We associate with each object an access control list (ACL), listing all users and groups who are authorized to access the object. For the sake of simplicity, we only consider read authorization in our example. Analogously, write and execute authorizations can be performed in a similar way. Suppose that a subject s invokes a method m on object $o_i{:}0$; m is allowed to write the information read from $o_i{:}\,0$

into another object $o_j:0$ accessible by other objects O_i that are not allowed to read the source object $o_i:0$. This sequence of operations is not allowed as a result of the strict and non reconciling policy proposed in (Samarati et al., 1997). Thus, the *write* operation to $o_j:0$ will never happen and will be blocked because the information can be leaked and released to O_i. We use versions to overcome this unnecessary blocking problem which makes the system inflexible, but without reducing the potential for information leaks and declassifications. We create another version of the object $o_j:0$. Let $o_j:1$ be the new created object, that is an identical copy of $o_j:0$ re-blessed with different RACL. That is, the write operation can be performed on $o_j:1$. The RACL of the new version excludes the objects in $o_j:0$*'s* RACL that are not allowed to read the information from $o_i:0$. To keep a safe information flow from object o_i to object o_j ($\mathrm{RACL}(o_j) \subseteq \mathrm{RACL}(o_i)$), information transmitted and the sequence of executions are subject to the intersection of the operations of all objects participating in the transmission. In the above example, $\mathrm{RACL}(o_j:1) = \mathrm{RACL}(o_j:0) \cap \mathrm{RACL}(o_i:0)$. Similarly, if a method whose execution is allowed to write the information of some objects $o_k:0$ into another object $o_j:0$ then the execution will only update versions (assume there exists at least one version) of the object $o_j:0$. These versions should not be exposed to other objects that are not allowed to access $o_k:0$. If no such version(s) exists that satisfies the information flow control policy then a new version of $o_j:0$ is created as explained above.

In order to enforce the control of the policy all versions are tracked down by a message filter. For each object, the message filter maintains a list of all its versions called the object's *version list* denoted by $VL(o_j)$. Let $vlen(o_j)$ be a version length variable which denotes the number of versions created from the object o_j. For reasons which become apparent below, we find it useful to distinguish between the first (oldest) and last (youngest) created versions of an object. Let $VL(o_j)[0]$ and $VL(o_j)[\mathrm{vlen}]$ denote respectively the first and last created versions from the object o_j. Operations on objects are provided as object methods. The methods of an object are executed only if the transaction's initiator has the correct authorization. For simplicity we assume that all methods required in a transaction be authorized. By authorized, we mean that the object requesting the execution exists in one of the version's RACL of the object being read. We introduce two options for reading objects called *cloning reply* and *non-cloning reply* described as follows.

- *Cloning reply* (CR): Allows a read method execution on the last version of the object. In this case, we assume the initiator object o_t of the read execution does not require information to be written

to it. Otherwise o_t is forced to clone itself, thus creating another version in order to acquire the new information without violating the security policy.

– *Non-cloning reply* (NCR): This option allows a read method execution on any versions except the last version. The version read does not forces its initiator to clone itself. Thus creating a new version that abides by the information security policy and where the access control checks are actually still taking place.

For example, assume the object $o_t:0$ is the initiator of an execution that requires reading the information in $o_j:0$. If the execution is requested under CR option then the read method is executed on the object $VL(o_j)$[vlen], the last created version of the object $o_j:0$. Otherwise, if the execution is requested under NCR option then the message filter performs a search on the version list starting backward, from the last version scanning down to the first version. The message filter is used to screen out unsolicited executions. All messages have to pass the filter checks. In this example, the message filter checks the reading condition that is, if RACL($VL(o_j)[p]$) $\supseteq$ RACL($VL(o_t)[q]$), where $p = vlen(o_j)$,$vlen(o_j)$ - 1,...,0, and $q = vlen(o_t)$,vlen(o_t) - 1,...,0 to find the latest version such that $o_t:0$ can read without provoking it to clone itself. If a version is found in VL then the read method is executed on that version. On the other hand, if no versions were found then the latest version is read since at this point the object is cloned anyway and the creation of a new version is imminent in this case.

5. Message Filtering

The *message filter* (Jajodia and Kogan, 1990) is a trusted system component which has the ability to intercept every message exchanged among the objects to control the flow of information. In this section we elaborate on the information flow control policies using message filtering. As stated in Section 4, we use o_i:vn to denote the version vn associated with object o_i. The original version of an object o_i is denoted by $o_i:0$. Suppose we have the following scenario: an object o_1 with a read access control list RACL($o_i:0$) = {$o_i:0$, o_1, o_2, o_3, o_6} acquiring information from another object o_j. Let o_j:0 be the only single version that exists at this time such that RACL($o_j:0$) = {o_j, o_i, o_1, o_2, o_4}. Also, assume that another version of o_i has to be created during some transaction calls. Let such a version be o_i:1 where RACL(o_i:1) = {o_i:1, o_1, o_2}. The new information has been written to o_i:1 to prevent $o_i:0$ to propagate it along to o_3. Now o_i wants to acquire information from

another object o_k which has only one version, where RACL(o_k:0) = {o_k:0, o_i, o_1, o_3, o_5}. A new version of o_i has to be created. Let such a version be o_i:2, where RACL(o_i:2) = {o_i:2, o_1, o_3}. At this stage o_i:2 contains the latest information where VL(o_i) = {o_i:0, o_i:1, o_i:2}. But if o_i wants to acquire information from o_j:0 again, this information will be written to the version o_i:1 which is considered as the latest version. In this case, the version list is updated and o_i:1 is moved to the end of the list VL(o_i) = {o_i:0, o_i:2, o_i:1}. The latest version is always accessible because the version length variable *vlen* reflects the number of entries in the version list. If *vlen* is equal to zero at any time then it means there are no extra versions of the object o_i. The object o_i is the only source object without any descendents.

Another issue that should be considered when organizing and accessing versions is when the write operation is executed on more than one version. In the example above, if o_i acquires information from the object o_t which has only one version, where RACL(o_t:0) = {o_t:0, o_i, o_1, o_2, o_3} then both o_i:1 and o_i:2 can accommodate that information. In this case, there is no need for duplicating the information; the read access control list of o_i:1 is updated to be RACL(o_i:1) = {o_i:1, o_1, o_2, o_3} and o_i:2 is removed from the list.

The message filter keeps track of two types of information – *forward information* and *backward information* which we borrow from the terminology of reference (Samarati et al., 1997).

- *Forward Information:* The information that the execution has received from its invoker, through the parameters of the message requiring the invocation.
- *Backward Information:* The information that the execution has received from the executions it has invoked, through the message replies.

Consider a message sent by execution e_i running on object o_i. Suppose the message requires an execution e_j on object o_j. The backward information B(e_j) of execution to be invoked is initialized to Øand its RACL is initialized to U, the set of all users. If the message has been sent directly by a user to start a transaction, the forward information of e_j is set equal to Ø. Otherwise, if the message has been sent by another execution e_i the forward information F(e_j) is set equal to the union of the forward and backward information of e_i. Then, e_j is executed and the reply returned is determined by the message filter according to the following rules:

1. The message is a write message. For all versions of o_j, starting from o_j:vlen down to o_j:0, if the RACL of o_i contains the RACL of

that version the write operation is executed on the version. After processing all versions, if at least one version has acquired the information then RACL of the last version acquiring the information is set to the union of the RACL's of all other versions that acquired it. All other versions are removed from the list, *vlen* is updated, and a success message is returned. Otherwise, if no version was able to acquire the information then a new version is created given the first available identifier and the information is written to it. The version list is then updated to include the new version. The variable *vlen* is incremented and a success message is returned.

2. The message is a read message with cloning reply option. The following steps take place.
   (a) The read method is executed on the latest version $o_j{:}_{\text{vlen}}$.
   (b) The RACL of the execution e_j is set equal to the RACL of $o_j{:}_{\text{vlen}}$.
   (c) The RACL of e_i is modified by intersecting it with the RACL of e_j.
   (d) The backward information of e_j is set equal to $\{o_j{:}_{\text{vlen}}\}$. The backward information of e_i is updated by taking its union with the backward information of e_j.
   (e) The reply of the message is then returned to e_i.
3. The message is a read message with non-cloning reply option. The following steps take place.
   (a) The read method is executed on the latest version $o_j{:}_{\text{vlen}}$.
   (b) For all versions of o_j, starting from the latest $o_j{:}_{\text{vlen}}$ down to $o_j{:}_0$. If the RACL of e_i includes the RACL of that version, say version number x, then the read method is executed on that version and no more versions are examined.
   (c) The RACL of e_j is set equal to the RACL of the version $o_j{:}x$ and the RACL of e_i is modified by intersecting it with the RACL of e_j.
   (d) The backward information of e_j is set equal to $\{o_j : x\}$ and the backward information of e_i is updated by taking its union with the backward information of e_j.
   (e) The reply of the message is then returned to e_i.
4. The message is a create message. A new object whose identifier is o_{id} is created and the user running the transaction is given privileges

on the object. The number of versions *vlen* is set to zero and o_{id} is returned to the execution e_i.

The code of the message filtering algorithm is summarized as follows. The reader should refer to (Fellah et al., 1999) for more details on the algorithm and its proof.

Input: Message (*msg)* sent by execution e_i, running on object o_i.
Output: Returns success or failure by enforcing the security policy on versions.

begin
AcquiredList := Ø; // Keep track of versions on which the *write* has been executed
B(e_j) := Ø;
RACL(e_j) := U;
if $o_i \in U$ **then** // The message is sent by a user starting a transaction
F(e_j) := Ø;
else // The message has not been sent by a user
F(e_j) := F(e_j) ∪ B(e_j);
endif
case *msg* **of**

(1) (write,(a_{tt}, *val*), null) **do** // *val*: value of attribute a_{tt}
for n=*vlen*(o_j) **downto** 0 **do**
if RACL(o_i) ⊇ RACL(o_j:n) **then**
o_j:n.att:= *val*
AcquiredList:= AcquiredList ∪ {n }
LastAcquired:= n
// Keep track of the version with the smallest *id* that has been aquired the information.
// All others are removed from the version list
acquired:= true
endif
end for
if acquired = true **then**
RACL(o_j:*LastAcquired*)= ∪$_h$ {RACL(o_j:h) | h ∈ AcquiredList}
// Remove from *VL* all o_j:h | h ∈ AcquiredList except o_j:*Lastaquired*
else
vlen:=*vlen*+1
create (o_j:vlen)
VL(o_j):= VL(o_j) ∪ {o_j:vlen}
endif

reply:= success
return reply **to** e_j

(2) (read,(a_{tt}),CR) **do** //a_{tt} $\in$ set of attributes
reply:= o_j:vlen.att
RACL(e_j):= RACL(o_j:vlen)
RACL(e_i):= RACL(e_i) $\cap$ RACL(e_j)
B(e_j):= {o_j:vlen}
B(e_j):= B(e_i) $\cup$ B(e_j)
return *reply* **to** e_i

(3) (read,(a_{tt}),NCR) **do**
for n=*vlen*(o_j) **downto** 0 **do**
if RACL(e_i) $\supseteq$ RACL(o_j:n) **then**
reply:= o_j:vlen.att
RACL(e_j):= RACL(o_j:n)
RACL(e_i):= RACL(e_i) $\cap$ RACL(e_j)
B(e_j):= {o_j:n}
B(e_i):= B(e_i) $\cup$ B(e_j)
return *reply* to e_i
exit loop
end if
end for

(4) (create,(v_{o}, v_{t},...,v_{k}), null) **do**
create o with values v_{o}, v_{t},...,v_{k}
reply:= o_{id} // identifier of o
return *reply* **to** e_i
end case
end

6. Conclusion

We have proposed a flexible and nonrestrictive information flow model for object-oriented systems using versions without compromising system security or reducing the potential for information leaks and disclosure. By focussing on flexibility rather than on object redundancy, we can get rid of all unnecessary message blocking that the message filter in (Samarati et al., 1997) strictly enforces. It is clear under certain operations – cloning reply and non-cloning reply, the message filtering has control over versions and does not allow any information transfer between different versions of the same object.

This work leaves some issues unaddressed and open to further research. A first issue to be investigated concerns the large number of versions that could be created during several transaction calls. Even though the number of versions created for a given object has conveniently contributed to the flexibilty and expressiveness of the information flow control model, conditions and restrictions on creating versions should be made more precise and investigated carefully.

References

Samarati, P., Bertino, E., Ciampichetti, A. and Jajodia S. Information Flow Control in Object-Oriented Systems. *IEEE Trans. on Knowledge and Data Engineering*, 9(4):524–538, July/August 1997.

Sandhu, R. Role Activation Hierarchies. *Proceedings of 3rd ACM Workshops on Role-Based Access Control*. 22–23, Fairfax, Virginia, October 1998.

Sandhu, R., Coyne, E.J., Feinstein, H.L and Youman, C.E. Role Based Access Control Models. *IEEE Computer*, 29 (2), 38–47, February 1996.

Sandhu, R. and Samarati, P. Authentication, Access Control, and Audit. *ACM Computing Surveys*, 28 (1), March 1996.

Thomas, R.K. and Sandhu, R. Implementing the Message Filter Object-Oriented Security Model without Trusted Subjects. *Proceedings of the IFIP WG11.3 Workshop on Database Security*, 19–21, Vancouver, Canada, August 1992.

Denning, D.E. A Lattice Model of Secure Information Flow. *Comm. of the ACM*, 19(5), 236–243, 1976.

Abadi, M. Secrecy by Typing in Security Protocols. *Proceedings Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

Ferrari, E., Samarati, P., Bertino, E. and Jajodia, S. Providing Flexibility in Information Flow Control for Object-Oriented Systems. *Proceedings IEEE Symposium on Security and Privacy*, 130–140, Oakland, CA, USA, May 1997.

Ferraiolo, D.F., Gugini, J.A. and Kuhn, D.R. Role-Based Access Control: Features and Motivations. *Proceedings 11th Annual Computer Security Applications Conference*, New Orleans, LA, December 1995.

Ferraiolo, D.F., Barkley, J.F. and Kuhn, D.R. A Role Based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and Systems Security*, 2(1), February, 1999.

Fellah, A., Maamir, A. and Rahwan, I. Information Flow Control Using Versions in Object-Oriented Systems. *Technical Report (CS#9P), Dept. of Math. & Computer Science*, UAE University, Al-Ain, UAE, June 1999.

Myers, A.C. and Liskov, B. Complete, Safe Information Flow with Decentralized Labels. *Proceedings of IEEE S&P'98*, Oakland, California, May 1998.

Myers, A.C. and Liskov, B. A Decentralized Model for Information Flow Control. *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, 129–142, Saint-Malo, France, 1997.

Jajodia, S. and Kogan, B. Integrating an Object-oriented Data Model with Multilevel Security. *Proc. IEEE Symp. on Security and Privacy*, 76–85, Oakland, California, 1990.

Denning, D.E and Denning, P.J. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7), 504–513, 1977.