A principled approach to object deletion and garbage collection in multilevel secure object bases

Elisa Bertino Elena Ferrari
Dipartimento di Scienze dell'Informazione
Università di Milano
Via Comelico 39/41, 20135 Milano,Italy
e-mail: {bertino,ferrarie}@dsi.unimi.it

Abstract
This paper introduces guidelines to prevent illegal information flows due to object deletion in multilevel secure object database management systems (ODBMSs). The guidelines are formally stated as security principles. We also show how to design a garbage collection mechanism in a multilevel secure ODBMS that ensures both security and referential integrity.

Keywords
Object Database Management Systems, Object Deletion, Garbage Collection

1 INTRODUCTION

Object-oriented database management systems and recent object-relational database management systems (in what follows we will refer to both kind of systems as object database management systems - ODBMSs for short) continue to be an active research area for both the academic and the industrial world. Issues related to security and privacy have been investigated in the area of ODBMSs and models have been proposed for both mandatory access controls (Jajodia et al. 1990, Millen et al. 1992, Thuraisingham 1989) and discretionary access controls (Rabitti et al. 1991). However, much work is still needed in this area. In particular, even if some approaches, developed for relational DBMSs, can be directly applied to ODBMSs, new security problems arise that are specific to object manipulation in ODBMSs. We believe that addressing such security issues is important given the relevance of object technology in the current and next generations of DBMSs.

An important issue in ODBMSs is related to object deletion. Two different approaches are used by existing ODBMS to enforce object deletion; under

Database Security XI T.Y. Lin and Shelly Qian (Eds)
© 1998 IFIP. Published by Chapman & Hall

the first users are allowed to explicitly delete objects; under the second a garbage collection mechanism is used by which an object is removed by the system when no longer reachable by other objects. Because under the latter approach no explicit delete operation is provided at application level, referential integrity is ensured. However, object deletion and garbage collection are operations that, if not properly implemented, could be exploited as covert channels. An important requirement for those operations is therefore to be secure from covert channels and, at the same time, to ensure referential integrity among objects in the database. Because of the relevance of garbage collection in object systems, several algorithms have been proposed for both centralized and distributed systems (Kolodner et al. 1989, Moss 1992). Here, we continue our investigation in object deletion and secure garbage collection (Bertino et al. 1994). We first show how object deletion and garbage collection could be illegally exploited to perform unauthorized data accesses; then, we introduce some principles ensuring a secure delete operation. Moreover, we present a garbage collection protocol which is secure against covert channels. The main differences between this work presented and our previous work (Bertino et al. 1994) can be summarized as follows. First, here we provide a formal setting to address secure object delete operations. Second, in our previous paper the copying approach to garbage collection was considered. Here, we consider a different approach, based on the mark-and-sweep technique, and show how the proposed approach is secure wrt the formal setting. The formal setting we propose consists of a number of principles forming the basic guidelines for secure object deletion and garbage collection. These guidelines provide a concrete *embodiment* of the general Bell-LaPadula principles (Bell et al. 1975) for the specific case of object deletion and garbage collection. We believe that such guidelines are an important step towards the development of secure object systems.

The remainder of this paper is organized as follows. Section 2 recalls the basic concepts of multilevel security and outlines the object model we refer to in this paper. Section 3 introduces the problem of object deletion and provides rules for secure object deletion. Section 4 analyzes the mark-and-sweep garbage collection protocol with respect to the principles presented in Section 3. Section 5 provides conclusions.

2 PRELIMINARY CONCEPTS

We first recall the basic concepts of multilevel security and describe the message filter model (Jajodia et al. 1990). Then, we characterize the object model we refer to in the paper.

2.1 The multilevel security model

The system consists of a set O of objects, a set S of subjects, and a set Lev of security levels with a partial ordering relation $\leq$. A level L_i is *dominated* by a level L_j if $L_i \leq L_j$. A level L_i is *strictly dominated* by a level L_j (written $L_i < L_j$) if $L_i \leq L_j$ and $i \neq j$. We say that two levels L_i and L_j are *incomparable* (written $L_i <> L_j$) if neither $L_i \leq L_j$ nor $L_j \leq L_i$ holds. A total function $\mathcal{L}$, called *security classification function*, is defined from $O \cup S$ to Lev. Given an object o, function $\mathcal{L}$ returns the security classification of o. Similarly, given a subject s, $\mathcal{L}(s)$ denotes the security classification of s.

A secure system enforces the Bell-LaPadula restrictions (Bell et al. 1975) that can be stated as follows: *1)* a subject s is allowed to read an object o iff $\mathcal{L}(o) \leq \mathcal{L}(s)$ (no-read-up); *2)* a subject s is allowed to write an object o iff $\mathcal{L}(s) \leq \mathcal{L}(o)$ (no-write-down).

The second property is also known as the **-property* and prevents leakage of information due to Trojan Horses.

2.2 The reference object model

An object database consists of a set of objects exchanging information via messages. An object consists of a unique object identifier (OID), which is fixed for the whole life of the object, and a set of attributes, whose values represent the state of the object. The value of an attribute can be an object or a set of objects. Moreover, an object has a set of methods encapsulating the object state. An object can be primitive (like an integer, or a character), or can be built from other objects (either primitive or non-primitive). We denote a non-primitive object as a triple (*oid, state,meths*), where: *oid* is the object identifier; $state = (a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n)$, where a_i is an attribute name (the names of object attributes must be distinct), and v_i is the value of attribute a_i, $1 = i, \ldots, n$; *meths* is a set of method names.

Let o and o' be two objects. We say that o is a *high-level* (*low-level*) object with respect to o', if $\mathcal{L}(o') < \mathcal{L}(o)$ ($\mathcal{L}(o) < \mathcal{L}(o')$). Similarly, let o and o' be two objects such that o stores in one of its attribute the OID of o'. We say that the OID of o' is a *high-level* (*low-level*) OID with respect to o if $\mathcal{L}(o) < \mathcal{L}(o')$.

Whenever an object o has as value of one of its attributes the OID of an object o', we say that o *references* o'.*

In our model, no reference is allowed among objects at incomparable security levels. Moreover, we make the assumption, common to most proposals,

*An object o may reference a high-level object o' provided that: (a) the creation of o' has been requested by o (or by another object at a level lower than or equal to the level of o); (b) the OID of o' is generated according to a covert-channel free OID generation mechanism (see (Bertino et al. 1994) for such a mechanism).

that all objects are single-level and, therefore, all attributes of an object have the same security level. Multilevel objects can easily be represented in terms of single-level objects; we refer the reader to (Bertino et al. 1997b) for a detailed discussion on this issue.

The methods of an object can be invoked by sending a message to the object. Upon the reception of the message, the corresponding method is executed, and a reply is returned to the object sending the message. The reply can be either an OID, a primitive object, or a special *nil* value, denoting that no information is returned.

The invocation of a method m on the reception of a message can be either synchronous or asynchronous. In the former case, the sender waits for the reply value, that is, it is suspended until the invoked method terminates. In the latter case, a *nil* reply value is immediately returned to the sender which will be executed concurrently with the receiver; the sender will be able to get the reply value successively. In this paper we do not distinguish between synchronous and asynchronous method invocations. Moreover, we assume that method invocations are performed sequentially during a user session within the system.

The fact that messages are the only means by which objects can exchange information makes information flow in object systems have a very concrete and natural embodiment in terms of messages and their replies (Jajodia et al. 1990). Thus, information flow in object systems can be controlled by mediating message exchanges among objects.

2.3 The message filter model

The Bell-LaPadula model has been applied to the object model by means of the message filter (Jajodia et al. 1990). Under this approach all messages exchanged among objects in the system are filtered according to the following rules: *1)* if the sender of the message is at a level strictly dominating the level of the receiver, the method invoked by the message is executed by the receiver in *restricted mode*, that is, no update can be performed. More precisely, a restricted mode execution at a level l should be *memoryless* at level l. Therefore, even though the receiver can see the message, the execution of the corresponding method on the receiver should leave the state of the receiver (as well as of any other object at a level not dominated by the level of the receiver) as it was before the execution. *2)* If the sender of the message is at a level strictly dominated by the level of the receiver, the method is executed by the receiver in *normal mode*, but the returned value is *nil*. To prevent timing channels, the *nil* value is returned to the sender before actually executing the method.

The first principle ensures that an object does not write-down, whereas the second one ensures that an object does not read-up. The *message filter* is a trusted component of the object system in charge of enforcing the above principles on all message exchanges among objects. Note that, according to the reference object model, an object is allowed to reference a high-level object; this means that an object may have as value of one of its attributes a high-level OID. This possibility allows an object to send information to objects at higher levels. However, since every message is intercepted by the message filter, this possibility does not violate the overall security of the system since read-up operations will always return a *nil* response value. Moreover, an object of level L_i may only stores the OIDs of the high-level objects whose creation has been requested by a level lower than or equal to level L_i. The mechanism we adopt for OIDs generation is described in (Bertino et al. 94).

3 DELETE OPERATION

Existing ODBMSs use different approaches with respect to the delete operation. There are two categories of systems: systems supporting explicit delete operations (like Orion (Kim et al. 1990) and Iris (Fishman et al. 1989)), and systems using a garbage collection mechanism (like O2 (Deux et al. 1990) and Gemstone (Maier et al. 1986)). A garbage collector is a piece of software that deletes objects no more accessible. There is a special object, called *root*, which is always persistent. All objects that can be reached from the root by traversing object references, are persistent. An object is removed when it can no longer be reached from the root.

If the delete operation is not properly executed, covert channels may be established.

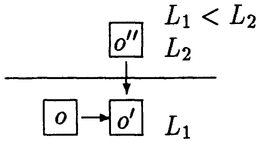

Figure 1 An object o' with references from its own level and a higher level

Example 1 *Consider two objects o and o' such that $\mathcal{L}(o) = \mathcal{L}(o') = L_1$, and an object o'' such that $\mathcal{L}(o'') = L_2$, with $L_1 < L_2$. Suppose that both o and o'' reference o' (see Figure 1). Suppose that the reference from o to o' is removed, because o has been deleted. If a garbage collection approach is used, o' would not be deleted since there is still another object (i.e., o'') referencing it. Therefore two subjects at levels L_1 and L_2, respectively could exploit this fact to establish a covert channel. The subject at level L_1 would create two* *objects o and o', at its own level, such that o references o'. Then, the subject at level L_2 would create an object o'', at its own level, such that o'' references o'. Then, after an amount of time pre-defined by the two subjects, the subject at level L_1 would remove the reference from o to o'. If, after the reference has been removed, object o' still exists, this situation is interpreted as 1. By contrast, if object o' is removed, this situation is interpreted as 0. Note that the subject at level L_1 would simply need to check storage occupancy to determine whether o' still exists.*

Exploiting the above covert channel requires collusion of two subjects at different levels. Note, however, that this is a common situation for many types of covert channels. See as an example, covert channels exploiting concurrency control mechanism in DBMS (Jajodia et al. 1992).

Moreover, whenever storage is deallocated because of object deletion, the problem of *dangling references* may arise. In systems with explicit delete operations dangling references may arise since an object can be removed even if there are references to it. In a garbage collection environment, an untrusted collector could intentionally remove an object to create a dangling reference.

A security problem is that dangling references can be used to establish covert channels, as the following example shows.

Example 2 *Consider Figure 2(a). If object o' is deleted by a subject at level L_2, a dangling reference appears in object o at level $L_1 < L_2$ (Figure 2(b)). A subject at level L_1 could infer the deletion of object o' by trying to send a write message to the object. On the basis of the result of such operation, the subject at level L_1 gets one bit of information from a higher security level.*

Thus, the deletion of objects referenced by low-level objects can be exploited by low-level subjects to infer information from high-level objects. A subject at a security level L_2 could delete a subset of the objects referenced by objects at a security level L_1, with $L_1 < L_2$. Then a subject at level L_1 could try to access all high-level objects resulting in a set of unsuccessful-successful accesses. Hence, an arbitrary string of bits of reserved information could be transmitted from a higher security level. Note that, in a garbage collection environment an untrusted collector could intentionally remove the objects at level L_2 referenced by low-level objects in order to establish a covert channel.

As Examples 1 and 2 above show, there are many ways in which a delete operation can be exploited to establish a covert channel. However, it is important to note that, when an object is deleted, the only side effects on the database are a state transition of the database itself and, possibly, the generation of dangling references. Therefore, the only means to establish a covert channel exploiting object deletion are those stated by the following definition.

Definition 1 (Delete Covert Channel) *A delete covert channel is a co-*

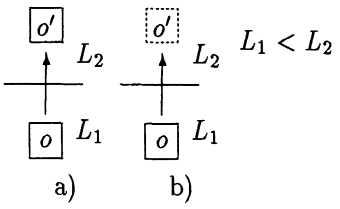

Figure 2 An object o referencing a high-level object

vert channel established by one of the following means: (i) exploiting dangling references; (ii) monitoring the state of the system with respect to object or memory allocation; or (iii) performing intentional data scavenging.**

In the above definition, with the term *state of the system* we refer to the state of all objects stored in the system and all the information related to the system itself, such as memory allocation and method error codes.

Moreover, delete operations can be regarded as a form of write operations: depending on the specific delete approach used, information may have to be set into the object being deleted (such as reference counts). It is therefore necessary to avoid that they can be exploited to establish a Trojan Horse. The Trojan Horse is simply established by having at a high-level some piece of code allowing or disallowing deletions of lower level objects or by plainly writing sensitive data into lower-level system information, used when removing the object (we call it *Delete Trojan Horse*). The above considerations lead to the following definition of *secure delete operation.*

Definition 2 (Secure Delete Operation) *A delete operation is secure iff it cannot be exploited to establish a delete covert channel or a delete Trojan Horse.*

An important question is whether there could be other circumstances, besides the ones considered in Definition 2, leading to insecure delete operations. We believe not. Indeed, illegal information flow may arise in two cases: 1) write-down operations, which for delete operations mean that a high-level subject may cause or prevent the deletion of a low-level object, or write sensitive data into lower-level system information. This case has been identified as Delete Trojan Horse in the above definition; 2) read-up operations, which for delete operations mean that dangling references* may arise, or that some low-level subjects may read high-level information about memory occupancy or

*When we speak of object allocation rather than memory allocation, we mean information about whether or not memory is allocated to a given object regardless of the amount of memory it uses (e.g. the result of a query looking for the instances of a given class).

*Accesses to system resources such as memory pages and disk sectors no more allocated. See also *object reuse* in (Chokhani 1992).

*This is because the delete operation just removes objects.

de-allocated areas. All these situations have been identified as delete covert channels in Definition 1. Note that completeness of Definitions 1 and 2 is based on the observation that a delete operation consists of two steps: (a) logically removing the object (and then checking references to the object); (b) physically removing the object (and thus de-allocating the storage allocated to the object). Our definitions are based on analysis of security threats that can arise in the above steps.

Even though the delete operation can be thought of as a form of write, because of the many ways the delete operation is implemented in object systems, it is important to establish some basic principles, enforcing secure delete operations, underlying any possible implementation of the delete operation. These principles are the topic of the following subsection.

3.1 Security principles for object deletion

In the following we define a set of *security principles*, ensuring the security of a delete operation. These principles state *what* needs to be done by the *Trusted Computing Base* (*TCB*) (Chokhani 1992) to prevent illegal flows of information due to object deletion, rather than *how* it will actually be implemented. For instance, Figure 2 only shows how to exploit dangling references to establish a delete covert channel, regardless of implementation details. Indeed, the *TCB* can easily block these illegal flows by simply using a strategy like the one discussed in Subsection 2.3 for handling messages sent to high-level objects.

We do not make any assumption whether deletion is implicit or explicit or whether referential integrity is enforced. In order to make our approach widely applicable, we do not assume a particular mandatory security model and start from the basic mandatory access control principles introduced in Subsection 2.1. Moreover, our principles do not assume any system architecture (single-subject vs kernelized).

Since delete operations can be regarded as a form of write operations, deleting a low-level object can be interpreted as a violation of the *-property. Hence, we suggest the following principle:

Principle 1 (No Delete Down) *An object o can cause the deletion of an object o' if and only if $\mathcal{L}(o) \leq \mathcal{L}(o')$.*

Note that we have used 'can cause the deletion' rather than 'can delete', because in a garbage collection environment an object can only cause the deletion of another object by updating all references to the given object, causing its deletion by the collector. In systems supporting explicit deletions, an object can cause the deletion of another object by issuing a delete command.

As we have seen in Example 1, an object o at level L could be referenced by several high-level objects and these references from high-level objects to low-level objects could be used to establish a delete covert channel. In order to prevent this type of problem, the following principle is established:

Principle 2 (No Interference from High to Low) *If an object o is referenced by high-level objects and by no object of level L', $L' < \mathcal{L}(o)$, a delete operation invoked on object o from level $\mathcal{L}(o)$ cannot be prevented.*

In Example 2, dangling references from low-level objects to high-level objects are used to infer higher level data. However, OIDs referencing high-level objects are needed if write-up is allowed by the security model. Therefore, we propose the following principle:

Principle 3 (No Dangling References from Low to High because of High-Level Deletions) *A delete operation invoked from level L on an object o' of level L', $L \leq L'$, must not be allowed if there exists at least an object o such that $\mathcal{L}(o) < L$ and o references o'.*

Note that Principle 3 does not forbid the deletion of an object o, referenced by low-level objects, if this deletion is required by an object at a level dominated by all the levels of the objects referencing o. Indeed, the dangling references arising from this deletion cannot be exploited as a covert channel trying to access the deleted object.

As stated by Definition 1, dangling references are not the only mean of establishing a delete covert channel. For instance, using an untrusted garbage collector that acts on the entire database, an object at level L could infer the deletion of an object at level L', $L < L'$, by monitoring the system resources. Hence, system resources must be controlled according to the following principle:

Principle 4 (No Global Information) *Information about system resources at security level L can be made available to an object o if and only if $L \leq \mathcal{L}(o)$.*

It is important to note that the no read-up principle is normally intended as a restriction imposed on the operations acting on the database, whereas Principle 4 states a more general rule to avoid also leakage of information due to *system* information, like memory allocation.

Example 3 *The interplay among the given principles is illustrated with the help of Figure 3. Suppose that a subject at level L_2 invokes the deletion of object o_2. Under the security principles, o_2 is not deleted, since its deletion would violate Principle 3. Note that, even if the deletion of o_2 is not allowed, Principle 2 is satisfied too, because object o_2 is referenced by both a high-level object (i.e., o_3)* and *a low-level object (i.e., o_1). By contrast, if a subject at*

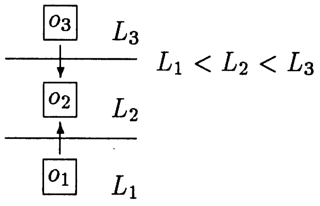

Figure 3 An object referenced from multiple levels

level L_1 invokes a delete operation on o_2, this operation is allowed since the deletion of o_2 does not violate any principle.

The correctness of the above security principles is stated by the following proposition.

Proposition 1 *A delete operation is secure iff Principles 1- 4 are satisfied.*

We refer the reader to (Bertino et al. 1997a) for the formal proof.

3.2 Implementation issues

The four principles guarantee the security of the delete operation. The description of a detailed implementation for the delete operation is outside the scope of this paper. Nevertheless, it is possible to make some considerations to help in securely designing and implementing such operation.

- The delete operation can be considered a special case of write operation. Therefore a *TCB* enforcing the *-property verifies Principle 1.
- The delete operation must neither update the state of an object nor read it, but it must physically remove an object. An important requirement for a system to be secure is that the basic storage elements (e.g. disk sectors, memory pages, etc.) be cleared prior to their assignment to an object so that no intentional or unintentional data scavenging takes place. The storage elements can be cleared when deallocated, that is, when an object is deleted. The security principles are defined disregarding implementation details. Hence we require the physical deletion to be performed by the *TCB*: when a delete operation is invoked on an object, the *TCB* calls a trusted procedure to perform the deletion.
- There are two possible approaches to enforce Principle 3: *1)* upwards dangling references are masked by concatenating the OIDs with the security level where the object is allocated (Millen et al. 1992) and making such dangling references *ineffective* (Bertino et al. 1994). This means that an object trying to access a high-level object is returned a default reply value even if the target object has been previously deleted. This can be per-

formed by the *TCB* that determines the security level from the OID and can, therefore, recognize a high-level OID. This approach requires that the OID contains the security level of the object (cfr. Bertino et al. 1994); *2)* the deletion of an object like o' in Figure 2 is prevented by the *TCB*.*
In both cases Principle 3 is verified. In particular, the first solution makes the upwards dangling references ineffective. Hence the principle is merely satisfied since no low-to-high dangling reference can compromise security. Note that this strategy can also be applied to incomparable security levels. Moreover, according to the first of the above approaches, object o_2 in Figure 3 can be deleted by a subject at level L_2 and Principle 3 is satisfied because the dangling reference from object o_1 to object o_2, arising from the deletion of o_2, is masked by the *TCB*. The first approach can also be adopted for the create operation. An object o requesting the creation *immediately* receives the new OID and the creation itself is executed asynchronously: errors possibly occurred during the creation do not prevent object o from immediately receiving the new OID.

A *TCB* satisfying the requirements stated above can be designed on the basis of the message filter approach described in Subsection 2.3. Finally, note that referential integrity is not preserved by the security principles, because they deal only with security. In particular, because of Principle 2, dangling references can arise which, however, cannot be exploited as delete covert channels.

4 SECURE GARBAGE COLLECTION

Our aim is to achieve referential integrity in multilevel databases by means of garbage collection. A serious drawback with conventional garbage collection mechanisms is that the garbage collector would have to access objects at various security levels. The garbage collector would therefore have to be trusted. We describe here a different approach, based on the *mark-and-sweep* technique; under this approach the collector is structured so that the trusted part is minimized. We analyze this approach with respect to the four principles for secure delete operations and we show that the garbage collector satisfies the four security principles.

A mark-and-sweep collector follows pointers in the heap marking any object that is reached (*marking phase*), then it collects all the non-marked objects (*sweeping phase*) scanning the heap sequentially. The marking phase starts from the *root objects* (objects containing information always needed). The root objects are the "entry points" for a security level.

One way to implement a multilevel trusted collector is to employ a *TCB* (Shockley et al. 1987) which controls the behavior of single-level untrusted collectors and enforces the Bell-LaPadula principles. Therefore, we require

*Auxiliary information about low-level OIDs should be stored by the *TCB* for this purpose.

root objects and a marking collector MC_L for each security level L. The marking collector MC_L is an object at level L in the database, which is activated and controlled by the *TCB*. The marking collector MC_L executes the marking phase for level L, while all sweeping phases are performed by the *TCB* to avoid data scavenging. MC_L does not mark an object o at level L or higher iff all references to o from other objects at level L have been removed (that is, the object is *non-locally reachable*). Garbage collection is managed according to the stop-the-world approach: activities are suspended, garbage is collected and then activities are restarted.

Since marking collectors are untrusted objects, each marking collector can only read objects or system information at its security level or at lower levels. In the following we show how to prevent the marking collectors from being exploited as storage covert channels. Storage covert channels are illegal channels established via the exploitation of the dynamic allocation of memory or via data scavenging. For example, a high-level subject could establish such a covert channel by saturating the memory, to prevent the normal computation of a low-level subject, which in turn could infer high-level information. To overcome this drawback, we adopt the following solution. System memory (volatile and non-volatile) is divided into a number of partitions of fixed size, one for each security level. Subjects at level L can allocate memory only from the partition assigned to L and the creation of a high-level object is performed at the level requested for the new object. This allocation scheme prevents storage covert channels from being established.

The marking collector MC_L executes a write operation in order to mark an object, hence it is only able to mark objects at level L or higher. The marking collector MC_L cannot be aware of references from objects at security levels higher than L because of the no read-up restriction. Therefore, dangling references could arise at security levels higher than L after the garbage collection is completed. The approach we propose to avoid dangling references is based on copying operations. Under this approach, an object o, non-locally reachable at its security level, is copied at higher security levels as needed. This mechanism does not need to be trusted; therefore it can be implemented by the marking collectors. The marking collector MC_L* builds a table called *Copy Table* to store pairs of related OIDs of the form (*old-oid*, *new-oid*), where *old-oid* is the OID of a low-level non-marked object while *new-oid* is the OID of its copy created at level L by the marking collector MC_L. The Copy Table at level L is read by the marking collectors at levels higher than L to avoid redundant copies. It is sufficient to create a copy of an object o, non-locally reachable, at the lowest levels where o is needed and update all high-level objects referencing o. When dealing with incomparable levels, a copy is generated for each level as needed.

The marking collectors are activated by visiting the security lattice on the basis of a sequence $(L_1, \ldots, L_n)$ called *visit-sequence*, where L_1 is the lowest

*Except for the lowest level in the security lattice.

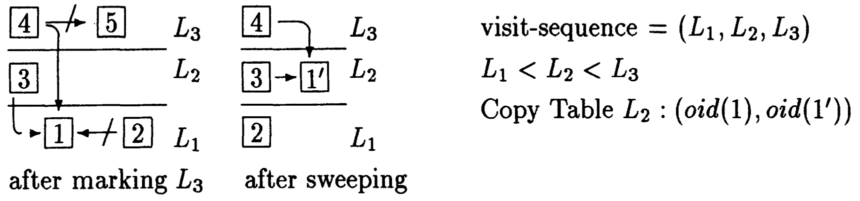

Figure 4 Achieving referential integrity with garbage collection

level, L_n the highest and for each L_i, L_j, $1 < i \leq n, 1 \leq j < i$, $L_j < L_i$ or $L_j <> L_i$. The visit-sequence is a static list associated with a given database. When level L is visited, the marking collector MC_L is activated and after its termination the next security level in the visit-sequence is visited. The sweeping phase must be postponed till the end of the marking phases, otherwise an object could be removed before being copied: when the last collector completes its execution, the sweeping phase is performed for all security levels.

Figure 4 shows an example of this approach: object 1 at level L_1 is not marked by the marking collector MC_{L1}; hence it is copied at level L_2 by the marking collector MC_{L2} that adds the pair $(oid(1), oid(1'))$ to the Copy Table at level L_2. The OID stored in object 4 at level L_3 and referencing object 1 is updated with the OID of the copy $1'$ generated at level L_2. This update is performed by the marking collector MC_{L3} by reading the Copy Table at level L_2.

This approach satisfies the security principles previously stated. Suppose that the marking collectors correctly execute the marking phase. Principle 1 is satisfied. The marking collectors cannot perform explicit deletions. Moreover, they are objects under the control of the *TCB*; hence they cannot violate the *-property by causing a low-level object to be deleted. Principle 2 is satisfied. A non-marked low-level object is copied at higher security levels, if needed, then it is deleted by the *TCB*. By contrast, if an object is locally reachable, it is marked and cannot be deleted. Principle 3 is satisfied. No low-to-high dangling reference appears if the marking collectors execute correctly the marking phase. Principle 4 is satisfied. The rule stated by this security principle is enforced by the *TCB*; hence we can assume that the principle is satisfied. Even if a marking collector incorrectly executes the marking phase, no security violation arises. The marking collector MC_L cannot read information at higher or incomparable levels; hence an incorrect marking phase can only generate dangling references from objects at level L. The only dangling references that could be exploited to establish a delete covert channel are those referencing high-level objects that have been deleted during the sweeping phases; these dangling references can be made ineffective by the *TCB* (cfr. Subsection 3.2, solution 1). Moreover, the system memory is divided into partitions of fixed size which is always the same. Therefore, it is not possible to establish covert channels due to memory saturation, overflows and so on. Finally, timing channels due to the execution of marking collectors can be avoided by properly controlling their execution time. For instance, the execution time of each marking collector can be forced to be longer than a pre-defined lower bound, which can be the same for each security level.

5 CONCLUDING REMARKS

We have analyzed issues related to object deletion in multilevel secure ODBMSs and we have stated principles ensuring a secure delete operation. These principles should be observed in designing and implementing garbage collection mechanisms and mechanisms for data manipulation in object systems. We have shown how a multilevel garbage collector algorithm, based on the mark-and-sweep technique, can be analyzed with respect to the above principles.

An important question concerns the computational overhead associated with each principle. The overhead is proportional to the number of references that each object has and how such references are distributed across levels. Such considerations are confirmed by some experimental evaluations performed on the copying garbage collection algorithm. The experiments have shown that the performance mainly depends on the number of copying operations of objects from lower to higher levels. This number depends on the number of references from higher-level objects to lower-level objects. Another factor impacting the performance is the structure of the security levels lattice. If the number of incomparable levels is high, the performance is good, because the first phase of the collector can be activated in parallel for all the incomparable levels. By contrast, the performance is not optimal when all levels are totally ordered. It is easy to see that these considerations apply also to the mark-and-sweep collector.

REFERENCES

[1] Bell, D. and LaPadula, L. (1975) Secure computer systems: unified exposition and multics interpretation. TR ESD-TR-75-306, MTR-2997, MITRE.

[2] Bertino, E. and Ferrari, E. (1997a) A Principled Approach to Object Deletion and Garbage Collection in Multilevel Secure Object Bases. *Pre-Proceedings of the 11th Annual IFIP WG 11.3 Working Conf. on Database Security*, pages 75–86, Lake Tahoe, CA.

[3] Bertino E., Ferrari, E. and Samarati, P. (1997b) A multilevel entity model and its mapping onto a single-level object model. *Theory and Practice of Object Systems*, to appear.

[4] Bertino, E. Mancini, L. V. and Jajodia, S. (1994) Collecting garbage in multilevel secure object stores. In *Proc. IEEE Symp. on Research in Security and Privacy*, Oakland, CA.

[5] Chokhani, S. (1992) Trusted products evaluation. *Communications of the ACM*, **35**(7):66–76.

[6] Deux, O. et al. (1990) The story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1):91–108.

[7] Fishman, D. et al. (1989) Overwiew of the Iris DBMS. *Object-oriented concepts, databases, and applications.* Addison-Wesley, pages. 219-50.

[8] Kim, W. et al. (1990) Architecture of the ORION next-generation database system. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1):109–24.

[9] Kolodner, E., Liskov B. and Weihl, W. (1989) Atomic garbage collection: managing a stable heap. In *Proc. ACM-SIGMOD Conf.*.

[10] Jajodia, S. and Atluri, V. (1992) Alternative correctness criteria for concurrent executions of transactions in multilevel secure database systems. In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA.

[11] Jajodia, S. and Kogan, B. (1990) Integrating an object-oriented data model with multilevel security. In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA.

[12] Maier, D. et al. (1986) Development of an object-oriented DBMS. In *Proc. of the 1st OOPSLA Conference*, Portland, Oregon.

[13] Millen, J. K. and Lunt, T. F. (1992) Security for object-oriented database systems. In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA.

[14] Moss, J. E. (1992) Working with persistent objects: to swizzle or not to swizzle. *IEEE Trans. on Software Engineering*, **18**(8).

[15] Rabitti, F. Bertino, E. Kim, W. and Woelk, D. A model of authorization for object-oriented and semantic database systems. *ACM Trans. on Database Systems*, **16**(1).

[16] Shockley, W. R. and Schell, R. R. (1987) TCB subsets for incremental evaluation. In *Proc. of the 2nd AIAA Conference on Computer Security.*

[17] Thuraisingham, M.B. (1989) Mandatory security in object-oriented database systems. In *Proc. of the OOPSLA Conference*, New Orleans, Louisiana.

BIOGRAPHY

Elisa Bertino is professor in the Department of Computer Science of the University of Milan. She has also been on the faculty in the Department of Computer and Information Science of the University of Genova, Italy. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose and at the Microelectronics and Computer Technology Corporation in Austin, Texas. Prof. Bertino is a co-author of the book "Object-Oriented Database Systems - Concepts and Architectures" (Addison-Wesley, 1993) and is on the editorial board of the IEEE Transactions on Knowledge and Data Engineering.

Elena Ferrari received a MS degree in Information Sciences from the University of Milan in 1992. Since November 1993, she has been a PhD student at the Department of Computer Science of the University of Milan. Her main research interests include database security, temporal data models and multimedia databases. On these topics she has published several papers. She has been a visiting researcher at George Mason University, VA and at Rutgers University, NJ.