# Database I

# Mark-and-Sweep Garbage Collection in Multilevel Secure Object-Oriented Database Systems*

Alessandro Ciampichetti[1], Elisa Bertino[2] and Luigi Mancini[1]

[1] Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova, Viale Benedetto XV 3, 16132 Genova, Italy
email: {ciampi,mancini}@disi.unige.it
[2] Dipartimento di Scienze dell'Informazione,
Università di Milano, Via Comelico 39, 20135 Milano, Italy
email: bertino@hermes.mc.dsi.unimi.it

**Abstract.** In this paper, the introduction of garbage collection techniques in a multilevel secure object-oriented database system is discussed; in particular, the attention is focused on mark-and-sweep collectors. A secure garbage collection scheme guarantees referential integrity and avoids potential covert channels arising from object deletion.
*Keywords:* object-oriented database systems, mandatory access control, garbage collection, object deletion, mark-and-sweep.

## 1 Introduction

Issues related to security and to privacy have not been widely investigated in the area of Object-Oriented Database Systems (OODBSs). Models have been proposed for mandatory access control [11] and for discretionary access control OODBSs [17]. In particular, discretionary access control provided by several commercial OODBSs is not able to protect data against certain types of attacks, known as *Trojan Horses*. Mandatory access control is safer then discretionary access control in that the former is able to protect data from Trojan Horses operating at different security levels. However, even mandatory access control is not able to always protect data from illegal accesses that exploit *covert channels* [3]. Therefore, recent research in database security has been directed towards defining architectures, techniques and algorithms to protect data against covert channels. An example along this direction is represented by concurrency control mechanisms specifically designed for multilevel secure relational DBSs [14]. The problem of developing DBSs able to guard against sophisticated type of illegal accesses, has been dealt so far only in the area of relational DBSs. Some of the solutions developed for relational DBSs can be directly applied to OODBSs; however, new security problems may arise that are

* The work reported in this paper is partially supported by CNR under Grant n. 94.00450.CT12, by NATO under Collaborative Research Grant n. 930888, and by the Italian M.U.R.S.T.

specific to OODBSs. In particular, object deletion and garbage collection are operations that could be exploited as covert channels.

Because of the relevance of garbage collection in object stores, several algorithms both centralized and distributed have been proposed (see [10, 15], for example). This paper continues the work in secure garbage collection reported in [2]. The main differences are summarized as follows. First, some issues related to storage covert channels and delete operations are analyzed in more detail here. Second, the problem of timing channels is solved by taking into account integrity, availability and confidentiality requirements. Finally, in [2] the copying approach to garbage collection was used, whereas the scheme discussed here is based on the mark- and-sweep technique. The use of different techniques shows that the approach based on untrusted collectors [2] is valid in general regardless of the particular garbage collection technique employed. Other solutions making various assumptions on the structure of the object store are discussed in [4].

The remainder of this paper is organized as follows. Section 2 summarizes the message filter approach. Section 3 introduces object deletion in multilevel environments. Section 4 describes our secure garbage collection scheme, whereas Section 5 draws some conclusions from the research reported in this paper and provides directions for future research efforts.

## 2 The Message Filter Approach

This section recalls the basic concepts of multilevel security and describes the message filter approach [11], since our aim is to introduce a garbage collector in a secure OODBS employing the message filter approach. In the Bell-LaPadula model [1], the system consists of a set $O$ of *objects* (passive entities), a set $S$ of *subjects* (active entities), and a partially ordered set $Lev$ of *security levels* with a partial ordering relation $\leq$. A (security) level $l_i$ is said to be *dominated* by another level $l_j$ if $l_i \leq l_j$. Moreover, a level $l_i$ is strictly dominated by a level $l_j$, denoted as $l_i < l_j$, if $l_i \leq l_j$ and $i \neq j$. Two mappings $F$ and $F_S$ are defined from $O$ and $S$, respectively, to $Lev$ that associate each object as well as every subject with a security level. That is, $\forall o \in O, F(o) \in Lev$, and $\forall s \in S, F_S(s) \in Lev$. A secure system enforces the Bell-LaPadula principles that can be stated as follows:

1. a subject $s$ is allowed to read an object $o$ only if $F(o) \leq F_S(s)$ (no read-up);
2. a subject $s$ is allowed to write an object $o$ only if $F_S(s) \leq F(o)$ (no write-down).

The second principle is also known as the **-property* and prevents leakage of information due to Trojan Horses. Additional details can be found in [5].

The Bell-LaPadula model has been applied to the object model by means of the message filter approach. In the following, this approach is described. An OODBS is defined as a set of objects exchanging information via messages. An object consists of an object identifier (OID) and of a set of attributes, whose values represent the *state* of the object. Moreover, an object has a set of methods that are used to manipulate the state of the object. Hence, an object is both a subject and an object with respect to the Bell-LaPadula terminology [3]. An object can be primitive (like an integer, or a character) or non-primitive. A non-primitive object is an object in which the value of an attribute can be a primitive object or an OID. Whenever an object $o$ has as value of one of its attributes the OID of an object $o'$, we say that $o$ *references* $o'$ and that $o'$ is *referenced* by $o$. Given an object $o$, a security level is assigned to the entire object $o$.

A message consists of three components: *name, parameters* and *reply*. Each message is associated with a method that is activated upon the reception of the message. A special type of object, called *user object*, represents a user session within the system, and can invoke methods spontaneously. An object, as part of the execution of a method, can send messages to other objects. Each message has a *sender* and a *receiver* object. Access to internal attributes, object creation and invocation of internal methods are all performed by having an object sending a message to itself. In particular, there exist three *primitive* messages, named *read, write* and *create*, which an object sends to itself in order to access internal attributes or to perform object creation respectively. An object sends a message $m$ by invoking a system primitive $send(m, id)$, where $id$ is the OID of the receiver object.

The state of an object can only be accessed by sending messages. This consideration is the basis of the message filter approach. Indeed, a message filter intercepts every message sent by any object in the system and decides how to handle the message. Mandatory access control is enforced by the message filter on the basis of the following two rules:

1. if the sender of a message is at a strictly higher level than the level of the receiver, the method is executed by the receiver in *restricted mode* (no update can be performed);
2. if the sender of a message is at a strictly lower level then the level of the receiver, the method is executed in *unrestricted (normal) mode*, but the returned value is always *nil*. To prevent timing channels, the *nil* value is returned to the sender before actually executing the method.

Principle 1 ensures that a subject does not write down, whereas principle 2 ensures that a subject does not read up. The message filter approach introduces two additional constraints: (1) the security level of an instance must dominate the class security level, and (2) the security level of a subclass must dominate the superclass security level.

Since its purpose is to enforce the security, the message filter has to be trusted, that is, it is embedded in the *Trusted Computing Base* (*TCB*) [3]. It is worth noting that the implementation of the message filter must also meet the following requirements, discussed in more detail in Section 4.3: (i) the creation of high-level objects could be exploited to establish timing channels; and (ii) OIDs must be generated in a secure fashion [16].

[3] In the following, the term "subject" will still be used to emphasize the actual method executions rather than the data hold by an object.

## 3 Implicit and Explicit Deletion

Existing OODBSs use different approaches for the implementation of the delete operation. There are two categories of OODBSs: those that allow subjects to perform explicit delete operations (e.g., Orion [8]) and those that employ a garbage collection mechanism to remove objects that are not any more reachable from other objects (e.g., $O_2$ [7] and Gemstone [13]). Note that a garbage collector is a piece of software that deletes objects no more reachable.

Whenever storage can be deallocated, the problem of dangling references may arise. A *dangling reference* occurs when there is a reference to storage that has been deallocated. It is a logical error to use dangling references, and it makes the system rise run-time errors.

The following example shows how dangling references can be exploited to establish a covert channel. If object $o'$ in Figure 1(a) is deleted by a subject at level $L_2$, a dangling reference appears in object $o$ at level $L_1$, Figure 1(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 an operation (run-time error or successful update), the subject receives one bit of information from a higher security level. A subject at level $L_2$ could delete a subset of the objects referenced by the objects at level $L_1$, and a subject at level $L_1$ could try to access all higher-level objects resulting in a set of unsuccessful-successful updates. Hence, an arbitrary string of bits of reserved information could be transmitted from a higher security level. In a garbage collection environment, an untrusted collector, which acts on the entire database, could intentionally remove the object $o'$ at level $L_2$ to establish a covert channel. Hence, upwards dangling references must be avoided. Note that this simple example only states *what* needs to be done by a secure system to prevent such covert channels.

In the following, by "high-level (low-level) object" it is meant "object with higher (lower) classification than those of the objects considered".

## 4 Secure Mark-and-Sweep Garbage Collection

In the following, the attention is focused on the interaction between garbage collection and the message filter. The garbage collection is concerned with automatically reclaiming storage that was once used but is no longer needed. The collector is invoked periodically or when a memory overflow arises. Our goal is to achieve referential integrity in multilevel OODBSs by means of mark-and-sweep garbage collection without compromising security. A mark-and-sweep collector follows the pointers in the heap marking any object that is reached (*marking phase*), then it collects all unmarked objects (*sweeping phase*) scanning the heap sequentially. The marking phase starts from the *root objects*, that is, the objects containing information always needed. The root objects are the "entry points" for a security level. A serious drawback with conventional garbage collection mechanisms is that the collector would have to access objects at various security levels; this would require the collector to be trusted. However, theoretically secure solutions could produce complex implementations impractical to be verified

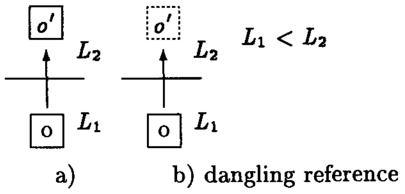


**Fig. 1.** Dangling references and covert channels

secure. We propose a different approach which does not require the collector to be trusted.

A mark-and-sweep collection algorithm can be augmented with *generations* to improve performance [12, 22]. Hence, a marking collector can also be used to manage large stable heaps; this is an important requirement for a collection algorithm to be widely employed. Moreover, various algorithms have been developed for efficiently scanning the heap [18, 20, 21].

## 4.1 Overview of the Approach

The overall garbage collection scheme is similar to that described in [2]. A multilevel trusted collector can be implemented employing a $TCB$ which controls the behaviour of single-level untrusted collectors, that is, root objects and an untrusted garbage collector $GC_l$ for each security level $l$ are required. The untrusted collectors are objects whose behaviour is under the control of the message filter. They cooperate among themselves and behave as a multilevel trusted collector. As a consequence of this fact, each collector can only read user or system information at its security level or at lower levels. Moreover, since the collector $GC_l$ executes a write operation in order to mark an object, it is only able to mark an object at levels higher or equal to $l$.

In Figure 2, the interactions among the collectors and the message filter are shown with the help of a simple security lattice. In the following, an object at level $l$ is said to be *(non) locally reachable* if it can(not) be reached starting from the root objects of level $l$. The garbage collection is managed in a stop-the-world fashion: activities are suspended, garbage is collected and then activities are restarted. Each collector $GC_l$ executes the marking phase for the security level $l$, while the unmarked objects are collected (i.e. deleted) subsequently in all security levels by the $TCB$ during the sweeping phase. In particular, the collector $GC_l$ does not mark an object $o$ at level $l$ or higher if and only if all references to $o$ from other objects at level $l$ have been removed.

In [2] the interactions among the message filter and the collectors were not addressed, and a *Trusted Collection Monitor* controlled the activations and the behaviour of the collectors. Here, the message filter is logically subdivided into two modules: the Garbage Collection Module ($GCM$) that activates the collectors and deletes the garbage, and the Message Filtering Module ($MFM$) that

$Unclassified < Confidential < Secret < TopSecret$

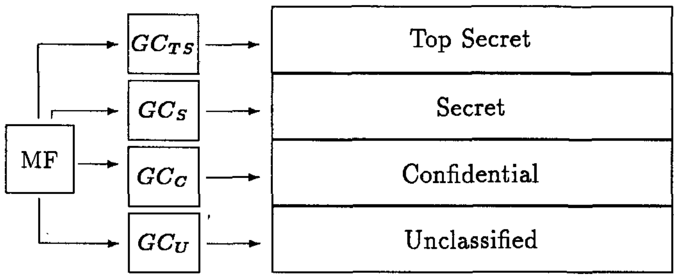


**Fig. 2.** Message Filter and Garbage Collectors

acts as the basic message filter. The $GCM$ activates the collectors $GC_l$ by sending them a message, say DETECT. A method *detect* which performs the marking phase is associated with this message.

The collector $GC_l$ cannot consider the OIDs from objects at security levels higher than $l$ because of the no read-up constraint. Therefore, dangling references could appear at security levels higher than $l$ after the garbage collection is completed. Indeed, an object at level $l$ which is not locally reachable and which is not referenced by any low-level object, is not marked by the collectors at levels lower or equal to $l$; hence, such an object will be deleted during the sweeping phase at level $l$. The key of our approach to maintain referential integrity is that whenever an object $o$ at level $l$, which is to be deleted from its security level, is referenced by an object at level $l'$, $l < l'$, the collector $GC_{l'}$ creates a copy of $o$ at level $l'$, that is, a new object is created at level $l'$ and data are copied into the new object. The following subsection will show in more detail how the proposed garbage collection scheme works.

## 4.2 Untrusted Marking Collectors

Since the aim of our approach is to avoid explicit deletion, it does not make sense to define a deletion message for each object (i.e. a primitive message). Indeed, in a garbage collection environment the system (here, the $GCM$) must collect garbage and delete objects, while subjects can only read and update objects. Moreover, an important requirement for a system to be secure is the *object reuse* [3], that is, the basic storage elements (e.g., disk sectors, memory pages, etc.) must be cleared prior to their assignment to a subject so that no intentional or unintentional data scavenging [4] takes place. The storage elements can be cleared when deallocated. Hence, the delete operation is required to be a

[4] Accesses to system resources such as memory pages and disk sectors no more allocated.

trusted operation implemented as part of the $TCB$ and it is invoked by the $GCM$ and not by the collectors as in [2]. However, in order to avoid all the storage covert channels, some issues have to be analyzed that in [2] were not addressed. 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, the following solution is proposed. The system memory (volatile and non volatile) is divided into a number of fixed partitions, one for each security level. Moreover, subjects at level $l$ can allocate memory only from the partition assigned to level $l$ and the creation of a high-level object is performed at the level requested for the new object. Hence, there is no way for a subject at level $l$ to interfere with the memory allocation performed at a lower or incomparable security level.

**Achieving Referential Integrity** In Section 4.1, we have outlined the problem of potential dangling references in high-level objects arising from the garbage collection. The solution we propose is the following. The collector $GC_l$ is activated if and only if all the collectors $GC_{l'}$, $l' < l$, have terminated the local marking phase. If an object is unmarked after the marking phase is ended at its security level, it is copied [5] at higher security levels once it is reached during the marking phase at such levels, that is, when a downwards OID referencing it is found at higher levels. Note that the copying operations can be performed by the collectors, since they obey the Bell-LaPadula principles. The copy $c$ created at level $l$ is marked by the collector $GC_l$ since it is a useful information at that level. The marking phase at level $l$ continues by traversing the OIDs eventually stored in $c$.

To avoid redundant copies, it is sufficient to create a copy of an unmarked low-level object $o$ at the lowest levels where $o$ is needed. For example, if object $o$ is referenced from levels $L_1$ and $L_2$, such that $L_1 < L_2$, a copy of $o$ could be created only at level $L_1$. Then, the OID referencing $o$ and stored at level $L_2$ should only be updated with the OID of the copy stored at level $L_1$. When dealing with incomparable levels, a copy is generated for each level, as the example in Figure 3 shows. Note that the sweeping phases must be postponed till the marking phase ends in all security levels, otherwise the copying of a low-level object $o$ could not be executed, since $o$ could have been previously deleted.

If an object $o$ if referenced only from incomparable levels, a copy of $o$ is created at each of such levels. If the consistency among the copies of $o$ is not needed, then the copies can be treated as independent objects. Otherwise, the copies of $o$ must be kept consistent, and any further update must be denied. For the former case, the garbage collection scheme proposed so far works well. In the latter case, a problem arises. Indeed, the fact that a copy of a low-level is read-only can be exploited to signal a bit of information between two subjects

[5] In the following, as previously stated, by "copy" we mean the creation of a new object and the copying of data into it.

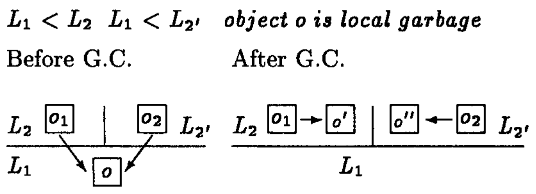


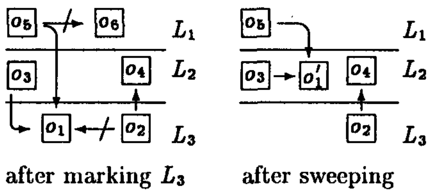
**Fig. 3.** Copying objects at incomparable security levels

at incomparable security levels. A simple and radical solution is the following. When a collector $GC_l$ creates a copy of a low-level object $o$, $GC_l$ sets the copy of $o$ as read-only, regardless of the existence of other objects referencing $o$ from levels incomparable with $l$. This information is used by the $MFM$ to avoid further write accesses. If a malicious collector $GC_l$ does not set a copy of a low-level object as read-only, no covert channel can be established. Indeed, the collector $GC_l$ has no information about the objects stored at levels higher (or incomparable) than $l$. Note that if each subject in the system is allowed to create read-only objects, data can be kept consistent without compromising confidentiality.

The copying operations can be performed with the help of a hash table associated with each security level. Such tables, called *Copy Tables*, are read and updated by the collectors. In particular, the collector $GC_l$ builds at level $l$ a Copy Table [6] to store pairs of related OIDs of the form (*old-oid*, *new-oid*), where *old-oid* is the OID of a low-level unmarked object, and *new-oid* is the OID of its copy created at level $l$ by the collector $GC_l$. The Copy Table *ct* at level $l$ is read by the collectors at levels higher than $l$ to avoid redundant copies and does not survive a collection cycle, that is, during the marking phase the collector $GC_l$ does not mark *ct*. Moreover, we assume that when *ct* is created its OID is available at higher levels. Hence, the collectors at higher levels can read *ct* and do not copy it as any other unmarked low-level object is.

**Visiting the Security Lattice** The collectors are activated visiting the security levels hierarchy on the basis of a sequence $(l_1, \ldots, l_n)$ called *visit-sequence*, such that $l_1$ is the lowest security level, $l_n$ the highest, and for each $l_i$ $(1 < i \leq n)$ and for each $l_j$ $(1 \leq j < i)$, $l_j < l_i$ or $l_j <> l_i$ (incomparable). A visit-sequence is a static list associated with a given database. When level $l$ is visited, the collector $GC_l$ is activated and after its termination the next security level in the visit-sequence is visited. As the marking phase is ended for all the security levels, the $GCM$ deletes all unmarked objects, that is, performs the sweeping phase for each security level. In Figure 4, an example of this approach is shown. Object $o_1$ at level $L_3$ is not marked by the collector $GC_{L3}$. Hence, a copy $o_1'$ of object $o_1$

[6] Except for the lowest level in the security lattice. Indeed, this level has no downwards OIDs.



Visit sequence = $(L_3, L_2, L_1)$

$L_3 < L_2 < L_1$

Copy Table $L_2$ : $(oid(o_1), oid(o'_1))$

Objects $o_1, o_4, o_6$: local garbage

**Fig. 4.** Achieving referential integrity with garbage collection

is created at level $L_2$ by the collector $GC_{L2}$ that adds the pair $(oid(o_1), oid(o'_1))$ in the Copy Table at level $L_2$. The OID stored in object $o_5$ at level $L_1$ and referencing object $o_1$ is updated with the OID of the copy $o'_1$ stored at level $L_2$. This update is performed by the collector $GC_{L1}$ that reads the OID of object $o'_1$ from the Copy Table at level $L_2$. Note that object $o_4$ is not locally reachable at level $L_2$, but it is referenced by object $o_2$. Hence, the collector $GC_{L3}$ marks object $o_4$ during the marking phase at level $L_3$ (write-up is allowed); thus, object $o_4$ is not deleted during the sweeping phase at level $L_2$.

**The Garbage Collection Module** In Figure 5, the *GCM* is described. The procedure *Mark* carries on the marking phase for each security level on the basis of the *visit-sequence*. The collectors are invoked by sending them the message DETECT (variable *msg*); for this purpose the OID of each collector is needed (variable *id*). The message filter can directly access any object and its methods, hence the message DETECT can be sent in a simpler way to the relevant garbage collector. In particular, the *GCM* does not use the system primitive *send* used by subjects; rather, it uses a system primitive *mf-send* that performs asynchronous method invocations.

Since the marking phase is executed autonomously by all collectors that are untrusted components, a collector $GC_l$ could delay the completion of the marking phase to signal some information to low-level collectors which, in turn, can evaluate the time elapsed between two consecutive garbage collection cycles. In order to meet the NCSC covert channel capacity guidelines [6] for B3/A1 classes, we propose the following solution. The overall garbage collection time must be long enough to prevent timing channels with high-capacity bandwidth. To accomplish this requirement, the execution time of each marking phase is forced to be longer than a pre-defined lower bound, called *min-run*, which can be the same for each security level. Since the upper bound for the bandwidth of timing channels is 100 bit per second (bps) [6], *min-run* can be assigned a value that does not cause performance penalty.

The performance of the marking phase can be improved by allowing the parallel execution of collectors at incomparable security levels. The *GCM* should be modified for this purpose; in particular, the visit-sequence should be used in a different manner. Indeed, it is only needed that the collector $GC_l$ starts the local

```
Mark( )
{
    let lev = first(visit-sequence);
    while lev $\neq$ null do
    {
        let msg = (DETECT, ( ), detect-reply);
        let id = oid($GC_l$);
        mf-send(msg, id);
        if min-run not expired then wait(min-run);
        let lev = next(visit-sequence);
    };
};

Sweep($l$: level) {
    for each unmarked object $o$ in $l$ do {
        let id = oid($o$);
        clear-mem(id); % clears memory
        delete-object(id); % deallocates memory
    };
    if min-sweep not expired then wait(min-sweep);
};

Garbage Collection Module
        Marking Phase : Mark()
        Sweeping Phase : forall security level $l$ do Sweep($l$)
```

**Fig. 5.** Garbage Collection Module

marking phase if and only if all the collectors at level $l'$, $l' < l$, have terminated.

The sweeping phase starts after the termination of the *Mark* procedure, and is performed in parallel at all security levels by the trusted procedure *Sweep*. Note that there is no synchronization requirement among the sweeping phases. Since the procedure *Sweep* is executed under the control of the $GCM$, a collector cannot signal information during the sweeping phase by means of timing channels. For example, a solution similar to the above based on the use of a lower bound on execution time (*min-sweep*) can be adopted. In order to reduce the bandwidth of timing channels to 0 bps, the $GCM$ can fix a time limit for the execution of each collector. After such a time-out is expired, the $GCM$ aborts the collector at level $l$ which has not yet completed the marking phase. If a collector ends the marking phase within the time limit, the $GCM$ waits for the time-out to be expired. In such a way, any illegal behaviour of a collector is hidden to all the lower (or incomparable) security levels. However, this solution has several drawbacks. First, it is needed that the sweeping phase is executed for a security level $l$ only if the collector $GC_l$ ends its marking phase within the time-out fixed by the $GCM$. Indeed, since the procedure *Sweep* is executed under the control of the $GCM$, a collector cannot signal information during the sweeping phase by means of timing channels. For example, a solution similar to the above one based on the use of time-outs can be adopted by evaluating the maximum time needed for the sweeping phase of a single security level. Second, if a time-out occurs at level $l$, there is no certainty that the corresponding collector has already marked every high-level object according to existing low references. Hence, the sweeping phase cannot be executed for all levels that dominate $l$. Finally, the time limit for level $l$ is not simple to be chosen, since it is rather difficult to evaluate the time a collector needs to create the copies of low-elevel unmarked objects. Moreover, increasing the time limits may lead to serious performance penalty.

## 4.3 Remarks on the Message Filter Approach

The $MFM$ is similar to the original message filter. However, in this section we outline some implementation issues.

- The fact that OIDs must be unique may lead to a covert channel. When an object is created, the new OID must be different from the OIDs that are already in use, including those belonging to higher or incomparable level objects. How much information the object requesting the creation gets depends on the mechanism used to assign OIDs to a new object. This problem and some solutions are well known [16]. Here, we assume that an OID is concatenated with the security level (or some encoding of it) at which the referenced object is stored; hence, the message filter knows the security level associated with an OID without accessing the referenced object. Therefore, run-time errors caused by accesses to maliciously deleted objects are avoided. Such run-time errors could generate a covert channel like that described in Section 3.
- Create messages must be properly controlled. Indeed, a timing channel could be established on the basis of the timing of the reply (i.e. the OID of the new object). The solution we propose is that a subject $s$ at level $l$ requesting a creation at level $l'$, $l < l'$, *immediately* receives the new OID, while the creation itself is executed asynchronously. Potential dangling references can be managed by the message filter as explained before, and errors possibly occurred during the creation do not prevent subject $s$ from immediately receiving the new OID.

The solutions stated above have a drawback. Indeed, no acknowledge is returned to a subject sending a message through an upwards OID or creating an object at higher levels. Hence, communication from a low- to a high-level subject may be unreliable in that it is based on assurance of high probability that the information will arrive error free and will be processed correctly. One way to circumvent this drawback is to provide a controlled stream of acknowledgements to the low- level as described in [9]. However, upwards dangling references, possibly generated by Trojan Horses planted inside the collectors, must be managed providing either an acknowledgement and not filtering a run-time error, or avoiding object deletion. In the latter case, the $TCB$ can record the references pointing an object at level

$l$ from objects at level $l'$, $l' < l$, via a trusted reference counting stored at level $l$. In this way, the *GCM* is able to avoid the deletion of an object $o$ when $o$ is referenced by low-level objects.

In some proposals, write-ups are not allowed. In this case, the message filter can be simplified, since no upwards OID is allowed. Hence, all messages sent to high-level objects are treated as messages sent among incomparable levels, that is, a *nil* reply is immediately returned to the sender of the message [11]. Moreover, a collector does not need to mark high-level objects. Consider Figure 4; object $o_2$ could not reference object $o_4$ and this object would be deleted during the sweeping phase at level $L_2$. However, the OIDs and create messages must be properly managed. In particular, a subject $s$ at level $l$ can still create an object at level $l'$, $l < l'$, but the OID of the new object will not be returned to $s$.

## 4.4 Correctness

The problem of defining a secure garbage collection scheme has been analyzed so far disregarding implementation details. However, most covert channels arise from the exploitation of system resources for a given implementation. Hence, the implementation must be accurate and must satisfy the requirements of the security policy [3]. Nevertheless, single-level collectors can be easily layered onto a *TCB*; the advantages of a layered implementation are analyzed in [19]. In particular, no additional security proofs for the collectors is required because they are controlled by the message filter as any other object in the database. We make this more clear with the following considerations.

Each collector can only read information (user or system information) at its security level or at lower levels. Moreover,

1. the collector $GC_l$ cannot execute explicit delete operations, nor it can cause the deletion of an object at level $l'$, $l' < l$, (the *-property should be violated for this purpose);
2. all messages sent by means of *illegal references* [7] are ineffective because they are blocked by the *MFM* that returns immediately the value *nil* as reply, even if the receiver object has been deleted.

The above considerations assume that the collectors are correct with respect to memory consistency. Suppose now that a collector incorrectly executes the marking phase:

- the collector $GC_l$ could intentionally avoid marking an object reachable only at level $l$. Then, the *GCM* would delete it. Therefore, the collector $GC_l$ could generate dangling references in memory. However, such dangling references would only concern objects at the same security level or higher. Hence, no covert channel could be established. By contrast, the collector $GC_l$ cannot mark an unreachable object because it cannot access the object at all.

[7] OIDs between incomparable levels.

- The collector $GC_l$ could avoid the marking of an object $o$ at level $l'$, $l < l'$ reachable only from level $l$. In this case, an upwards dangling reference could appear at level $l$ since $o$ would be deleted during the sweeping phase of level $l'$ if no restriction is applied on objects referenced by low-level objects (cf. Section 4.3). Nevertheless, such a dangling reference could not be exploited to establish a covert channel because of two reasons: (i) the collector $GC_l$ itself creates the dangling reference and (ii) the $MFM$ avoids run-time errors (cf. Section 4.3).
- The collector $GC_l$ could avoid the copying of an unmarked low-level object, but this would only generate dangling references at level $l$.
- Information stored in a Copy Table at level $l$ concern level $l$ or lower levels; hence, a Copy Table cannot be used for the illegal transfer of information.
- The overall garbage collection time has a lower bound. Hence, the collector $GC_l$ has no way to establish a timing channel with high-capacity bandwidth.
- The behaviour of the collector $GC_l$ with respect to memory allocation is not different from that of any other subject at level $l$.

In conclusion, the confidentiality is achieved even if the collectors are intentionally designed and implemented incorrectly. Note that dangling references, possibly occurring when activities restart, are similar to programming errors in the source code of a collector. That is, if a dangling reference occurs, then there exist a collector that does not work properly. Hence, since collectors must be tested before being employed, it seems difficult that such a problem occurs. Moreover, the programmers developing the collectors know the restrictions enforced on the collectors activity, hence there is no reason to implement a collector that generates dangling references, since it is known by the programmers that there is no way to establish a covert channel.

## 5 Conclusions

In this paper, the introduction of garbage collection techniques in multilevel secure OODBSs has been discussed. Such an approach guarantees referential integrity and avoids covert channels due to object deletion via a careful control of the garbage collection execution. The use of untrusted garbage collectors simplifies design, implementation and testing (formal and operative) of the $TCB$. Moreover, untrusted collectors give the system more flexibility. Indeed, the collectors can be thought as "black boxes" that manage memory and that are controlled by the $TCB$, disregarding implementation and garbage collection techniques.

The basic ideas under the scheme proposed can be used with different garbage collection techniques [2]. In particular, the garbage collection scheme proposed in [2] can be reused. There is still interesting work to be do in this area, such as the design and efficient implementation of the message filter and the garbage collectors, and the study of different activation strategies for the collectors. We are currently investigating a garbage collection scheme in which the collectors acts concurrently.

## References

1. Bell D., LaPadula L., *"Secure Computer Systems: Unified Exposition and Multics Interpretation"*, Technical Report ESD-TR-75-306, MTR-2997, MITRE, Bedford, Massachusetts, 1975.
2. Bertino E., Mancini L. V., Jajodia S., *"Collecting Garbage in Multilevel Secure Object Stores"*, Proc. IEEE Symp. on Research in Security and Privacy, Oakland, CA, May 1994.
3. Chokhani S., *"Trusted Products Evaluation"*, Comm. of the ACM, vol. 35, no. 7, July, 1992, pp. 66-76.
4. Ciampichetti A., *"Object Deletion and Garbage Collection in Secure Object-Oriented DBMSs"*, (in Italian), Master Thesis, Department of Computer Science, University of Genova, Italy, October 1993.
5. Denning D. E., *"Cryptography and Data Security"*, Addison Wesley Editions, Reading, Massachusetts, 1982.
6. Department of Defense, *"Trusted Computer System Evaluation Criteria"*, DOD 5200.28-STD, Washington DC, Usa, December 1985.
7. Deux O., et al., *"The Story of $O_2$"*, IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, 1990, pp. 91-108.
8. Kim W., et al., *"Architecture of the ORION Next-Generation Database System"*, IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, 1990, pp. 109-124.
9. Kang H. M., Moskowitz I. S., *"A Pump for Rapid, Reliable, Secure Communication"*, 1st ACM Conf. - Computer and Comm. Security, pp. 119-129, Va, Usa, November 1993.
10. Kolodner E., Liskov B., Weihl W., *"Atomic Garbage Collection: Managing a Stable Heap"*, Proc. ACM-SIGMOD International Conference on Management of Data, Boston, Oregon, May-June 1989.
11. Jajodia S., Kogan B., *"Integrating an Object-Oriented Data Model with Multilevel Security"*, Proc. 1990 IEEE Computer Society Symp. on Research in Security and Privacy, May 1990.
12. Lieberman H., Hewitt C., *"A real-time Garbage Collector based on the Lifetime of Objects"*, Comm. of the ACM, Vol. 26, No. 6, June 1983.
13. Maier D., et al., *"Development of an Object-Oriented DBS"*, Proc. OOPSLA 1st Conference, Portland, Oregon, October 1986.
14. Maimone W. T., Greenberg I. B., *"Single-Level Multiversion Schedulers for Multilevel Secure Database Systems"*, Proc. IEEE Computer Society Symp. on Research in Security and Privacy, Oakland, California, May 1990.
15. Mancini L. V., Shrivastava S. K. , *"Fault-Tolerant Reference Counting for Garbage Collection in Distributed Systems"*, The Computer Journal, vol. 34, no. 6, 1991.
16. Millen J. K., Lunt T. F., *"Security for Object-Oriented Database Systems"*, Proc. IEEE Computer Society Symp. on Research in Security and Privacy, Oakland, California, May 1992.
17. Rabitti F., Bertino E., Kim W., Woelk D., *"A Model of Authorization for Object-Oriented and Semantic Database Systems"*, ACM Transactions on Database Systems, vol. 16, no. 1, March 1991.
18. Schorr H., Waite W. M., *"An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures"*, Comm. of the ACM, vol. 10, n. 8, pp. 501-506, 1967.

19. Shockly W. R., Schell R. R., *"TCB Subsets for Incremental Evaluation"*, Proc. 2nd AIAA Conference on Computer Security, December 1987.
20. Thorelli L. E., *"Marking Algorithms"*, Bit, vol. 12, n. 4, pp. 555-568, 1972.
21. Thorelli L. E., *"A Fast Compactifying Garbage Collector"*, Bit, vol. 16, n. 4, pp. 426-441, 1976.
22. Zorn B., *"Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection"*, Comm. of the ACM, 1990, pp. 87-98.