*Proceedings of IEEE Computer Security Foundations Workshop VI, Franconia, NH, June 1991, pages 139-152.*

# A Secure Kernelized Architecture for Multilevel Object-Oriented Databases*

*Ravi Sandhu, Roshan Thomas and Sushil Jajodia*

Center for Secure Information Systems
and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444

**Abstract**

We present a secure kernelized architecture for multilevel object-oriented database management systems. Our architecture is based on the notion of a message filter proposed by Jajodia and Kogan. It builds upon the typical architecture of current object-oriented database management systems. Since the operations mediated by the message filter are arbitrarily complex operations (as opposed to primitive reads and writes), a secure message filter requires careful attention to potential timing covert channels. Although the overall computation is logically a sequential one, to be secure we must actually execute pieces of the computation concurrently. This raises a synchronization problem for which we give a secure multiversion protocol. The fundamental problem solved in this paper is how to securely and correctly "write up" in terms of abstract operations.

## 1 Introduction

In recent years, several research and development efforts in the design and implementation of object-oriented databases have been undertaken (e.g., ORION [4], IRIS [5], GEMSTONE [10]). The impetus for these developments can be attributed to the realization of the limits of record-based data models and conventional database technologies. Object-oriented models not only allow the representation of complex object structures, but further allow modeling of the behavior of real world entities through methods encapsulated in objects.

From the security perspective, the object-oriented model has strong intuitive appeal. This is because an object in the "object-oriented" sense is an abstraction modeling a real-world entity. This enables us to specify and implement access control and security policies in terms of objects. However, mechanisms for sharing in object-oriented systems such as inheritance as well as the dynamics of method invocation add complexity to any object-oriented security model.

Several models and prototypes that address mandatory security issues for object-oriented database systems have recently been proposed [6, 7, 8, 9, 11, 14]. In most of these the security policy to be enforced is expressed by a set of properties that must be satisfied. In this respect, the model proposed by Jajodia and Kogan (referred to as the *message filter model*) [6] is unique as it expresses the security policy with a message filtering algorithm. The message filter model can be characterized as an information flow model. The central element of the model is a message filter security component that filters and controls all exchange of information between objects. Another distinguishing feature of the model is that it is not based on the notion of subjects. A database system is seen as a collection of objects that communicate and exchange information through messages. The main advantage of the message-filter model is the simplicity with which mandatory security policies can be stated and enforced.

Our focus in this paper is on implementing the message filter model with a kernelized architecture and within the framework of the Orange Book [2]. Our architecture is a layered one and utilizes the trusted computing base (TCB) subsetting approach. At the lowest layer in the TCB we rely on trusted operating system functions. The next two higher layers provide the functions for managing the persistent store and implementing the message filter. In developing our architecture, we demonstrate how in spite of the complexity of an object-oriented implementation we can still provide a high degree of assurance of the system's security with minimal trusted code. Our architecture is consistent with and builds upon the typical architecture of current object-oriented database management systems and existing trusted operating systems.

A key issue we have had to address in implementing the model is that of timing channels. These timing channels arise because operations in our model are based on abstract messages with arbitrary semantics. Models such as Bell-LaPadula [3] are able to ignore this issue since the operations in these models are primitive reads and writes. Consider a message sent from a low object to a high object in a multilevel

*Or, how to "write up" securely and correctly.

object-oriented system. Clearly the sender should not receive the actual reply from the high object. It is conceptually a simple matter to arrange for the message filter to substitute some innocuous reply (such as NIL) in place of the actual reply. However, it also should not be possible for the high object to modulate the delay in this response to the low sender (even if the response is known a priori to be NIL). In this paper, we show how to close this timing channel by concurrently executing pieces of what is logically a sequential computation. To do so correctly we develop a secure multiversion protocol for synchronizing these concurrent computations so their net effect is equivalent to the intended sequential computation.

The fundamental problem solved in this paper is how to securely and correctly "write up" in terms of abstract operations. The solution is cast in context of the object-oriented data model because it is the most flexible data abstraction model known to the authors. The essential insight is that a logically synchronous computation which "writes up" in terms of abstract encapsulated operations (rather than primitive read and write operations) must be executed asynchronously to be secure with respect to timing channels in the computation model. To achieve this and be correct (i.e., equivalent to the logically sequential computation) the actual asynchronous execution must keep available multiple versions of the data. It is necessary to be sufficiently concrete in the data model and system architecture in order to completely describe the solution. Our algorithms, being asynchronous, are inherently distributed and their precise interactions can only be described with the degree of detail given in this paper. We conjecture there is some abstract description of this problem and solution but such abstraction is beyond our current understanding of the problem.

Traditional security models such as Bell-LaPadula [3] view "write up" in a very different way and therefore this requirement of asynchronous execution does not arise in these models. If "write up" is permitted only in terms of machine language instructions, such as STORE, it is reasonable to assume that STORE operations will take a fixed amount of time independent of the data and address. In reality this assumption is an approximation of what happens in modern computers. For example, in the presence of paging we cannot predict how long a STORE operation will actually take. Timing channels similar to those identified in this paper therefore exist in paged systems. Similarly in multi-processor systems with blocking interconnection networks the execution time of a STORE instruction is no longer constant. It can be modulated to create timing channels of very high bandwidth. These timing channels have created considerable consternation in the security community [1].

Another problem with the Bell-LaPadula view of "write up" is that it only permits blind writes (i.e., modification of existing higher-level data is not permitted while its overwriting is permitted). Therefore an unclassified transaction cannot debit or credit a secret account. It can write the balance of this secret account independent of the previous balance, but this is not a very useful operation. The message-filter model on the other hand allows arbitrary abstract operations, such as credit and debit, for "write up." It is consequently a fundamentally richer model than Bell-LaPadula. It is by no means obvious that the message filter model can be implemented with a kernelized architecture and within the framework of the Orange Book. Our objective in this paper is to show that this can be done.

Figure 1: Objects in a payroll database

The rest of this paper is organized as follows. Section 2 gives an example to illustrate the problems addressed in this paper. Section 3 gives an overview of the message filter model and the message filtering algorithm. Section 4 presents a kernelized architecture built on the typical architecture of existing object-oriented database management systems. Section 5 addresses the implementation of the message filter model based on this architecture. Section 6 concludes the paper.

## 2 An Illustrative Example

We illustrate the key problems addressed in this paper by giving an example of a simple payroll database application. A typical transaction in this application is the processing of weekly payroll for an employee. The weekly pay is given by the product of the weekly accumulated hours and the employee's hourly pay rate.

Our simple object-oriented database consists of three classes of objects: (1) EMPLOYEE (Unclassified), (2) PAY-INFO (Secret), and (3) WORK-INFO (Unclassified) with the corresponding attributes as shown in figure 1. Objects EMPLOYEE and WORK-INFO are unclassified as their attributes (such as name, address, hours-worked) represent information about an employee that can be made readily available. The object PAY-INFO is secret because its attributes contain sensitive information such as hourly-rate and weekly pay.

The processing of an employee's weekly payroll requires the exchange of three messages and three replies between these objects. Processing is initiated by the EMPLOYEE object sending a PAY message to the PAY-INFO object. On receiving the PAY message, object PAY-INFO sends the message GET-HOURS to the object WORK-INFO in order to retrieve the accumulated hours for the week. These hours are returned to the PAY-INFO object in the message HOURS-WORKED. Meanwhile, object EMPLOYEE receives a NIL reply (as the actual reply cannot be returned to the low level object) for the initial PAY message and sends a RESET-WEEKLY-HOURS message to WORK-INFO. This will reset the accumulated hours to zero and enable the hours to be accumulated again for the next pay period. Finally, a reply is sent for the reset request in the message DONE.

We now elaborate on the issue of timing channels. In the object-oriented model of computation, whenever a message is received by an object a corresponding method (piece of code) is selected and executed to process the message. If we execute methods sequentially, only one method can be executed (active) at any given time. Thus in our example, after sending the PAY message, the sender method in object EMPLOYEE is effectively blocked (suspended) and the receiver method in PAY-INFO is executed. In such a situation the NIL reply for the message PAY cannot be sent until the method in PAY-INFO has terminated. Further, under these conditions the method in the high level object PAY-INFO can modulate the timing of this NIL reply and thus introduce a timing channel.

Our solution to close such timing channels is to make the timing of the NIL reply independent of the termination of the receiver method. To achieve this, we have to execute methods concurrently whenever messages are sent from a low-level object to a high level one.† In our example, immediately after the PAY message is sent, a NIL reply is returned to the blocked method in EMPLOYEE. This NIL reply resumes execution of the blocked method in EMPLOYEE. At the same time, on receipt of the PAY message, the receiver method in PAY-INFO is also executed (leading to concurrent execution).

While the above scheme closes the timing channel, it introduces a synchronization problem arising due to concurrently executing methods. To illustrate with our example, consider the scenario that could evolve when both the sender and receiver methods (in objects EMPLOYEE and PAY-INFO) are executing concurrently. Suppose an employee has accumulated 40 hours. With a pay rate of $4/hour the employee is entitled to a gross pay of $160. However, as both methods are executing concurrently, it is quite possible that the RESET-WEEKLY-HOURS message is received at object WORK-INFO before the GET-HOURS message. The message GET-HOURS will thus retrieve the reset hours (zero) instead of the actual hours accumulated for the week. This will result in the erroneous calculation of the weekly pay as zero which is clearly not acceptable. Note that this problem does not occur if the methods execute sequentially, because we are guaranteed that the GET-HOURS message is processed completely before the RESET-WEEKLY-HOURS message.

In this paper, we show how to avoid such inconsistencies arising due to the lack of synchronization by the use of multiple versions of objects. Conceptually, every update of an object results in a new version being created (in reality only a subset of these versions is actually created in the system). The crux of our multiversioning scheme is to ensure that every message and method remembers to retrieve the correct version of the object requested. By correct we mean the version representing the same state that would have existed had the methods executed sequentially. In our example the RESET-WEEKLY-HOURS message creates a new version of the object WORK-INFO rather than overwriting the accumulated hours for the week. The earlier version is also retained in memory. This is the version that existed before the message PAY was sent and is thus the correct version required to process the GET-HOURS message.

## 3 Message Filter Model

The main elements of the message filter model are objects and messages. Every object is assigned a single-level classification.‡ Objects can communicate and exchange information only by sending messages among themselves. The flow of messages and replies is mediated by the message filter. The message filter decides, upon examining a given message and the classifications of the sender and receiver, what action is appropriate. It may let the message pass unaltered; or it may interpose a NIL reply in place of the actual reply (e.g., when a low object sends a message to a high object requesting the value of one of the latter's attributes); or it may take some other action (to be discussed later). The message filter is the analog of the reference monitor in traditional access-mediation models.

The message filter must ensure that all information flows are legal. Let L(O) represent the classification of an object O. We say that information can legally flow from an object Oj to an object Ok if and only if $L(Oj) \leq L(Ok)$. All other flows are illegal.

†It is important to note that the solution presented here only addresses timing channels that arise in our computational model. Timing channels may still arise in a particular implementation of the computational model. For example, in the payroll database the system overhead associated with concurrent processing of the GET-HOURS message may result in a perceptible delay in the reply to the RESET-WEEKLY-HOURS message. These implementation dependent timing channels can ultimately be eliminated only to the extent that the underlying Operating System TCB is free of timing channels.

‡There is no loss of modeling power due to the restriction that objects be single-level as multilevel entities can still be represented [6]. A user view of multilevel objects (called a conceptual schema) is decomposed into one of single-level objects that make up the corresponding implementation schema.

The message filter has to ensure that all direct and indirect flows are prevented. It utilizes two basic schemes for achieving this. The first is to make the backward flow[§] ineffective whenever a message is sent from a sender to a receiver object with the latter having a higher classification. In this case, the actual reply is intercepted by the message filter and some innocuous reply is substituted. We wish to emphasize that a reply must always be returned for otherwise the sender will remain blocked indefinitely (waiting for the reply). Thus whenever the backward flow is to be made ineffective, a NIL is substituted for the actual reply. It is the implementation of this step which can potentially introduce timing covert channels.

The second scheme prevents illegal forward and indirect flows.[¶] This is achieved by setting the status of method invocations. Every method invocation t, has a status s(t), which is either unrestricted (U) or restricted (R). A restricted method cannot update the state (attributes) of an object whereas an unrestricted method is allowed to do so. In order to prevent indirect flows, the method invocations by a restricted method may be restricted as well.

The central idea in the security model is that information flow is controlled by mediating the flow of messages. It is thus required that all basic object activity such as access to internal attributes, object creation, and invocation of local methods, be implemented by having an object send messages to itself. The model defines built-in *primitive messages* for this purpose. These messages are READ, WRITE, and CREATE.[‖] The READ and WRITE messages request direct access to local attributes. CREATE requests allocation of space for the creation of a new object. The response to a built-in message is carried out directly by the system, according to pre-defined semantics, rather than by invocation of a user-defined method. All other messages are considered to be *non-primitive* and their semantics are defined by invocation of appropriate methods. The notion of an object sending a message to itself is conceptual. The actual implementation of primitive messages is by system calls.

### 3.1 The Message Filtering Algorithm

The message filtering algorithm is presented in figure 2. We assume that O1 and O2 are sender and receiver objects respectively. Also, let t1 be the method invocation in O1 that sent the message m, and t2 the method invocation in O2 on receipt of m. The two major cases of the algorithm correspond to whether or not m is a primitive message.

Cases (1) through (4) in figure 2 deal with non-primitive messages sent between two objects, say O1 and O2. In case (1), the sender and the receiver are at the same level. The message and the reply are allowed to pass. The status of t2 will be the same as that of t1. For the moment ignore the **rlevel** variable. In case (2), the levels are incompatible and thus the message is blocked and a NIL reply returned to method t1. In case (3), the receiver is at a higher level than the sender. The message is passed through, but a NIL reply is returned to t1 while the actual reply from t2 is discarded (thus effectively cutting off the backward flow). For case (4), the receiver is at a lower level than the sender. The message and the reply are allowed to pass. However, the status of t2 (in the receiver object) is restricted to prevent illegal flows. In other words although a message is allowed to pass from a high-level sender to a low-level receiver, it cannot cause a write-down violation as the method invocation in the receiver is restricted from updating the state of any object.

We now illustrate cases (1), (3) and (4) as they apply to the payroll database mentioned in figure 1. Case (1) occurs when the sender and receiver are at the same level and applies to the message exchange between objects EMPLOYEE and WORK-INFO. The message RESET-WEEKLY-HOURS and reply DONE are both allowed to pass by the message filter. Case (3) applies to the message exchange between objects EMPLOYEE and PAY-INFO. As the latter is classified higher, a NIL reply is returned in response to the PAY message. Case (4) involves the objects PAY-INFO and WORK-INFO. As the object WORK-INFO is classified lower than PAY-INFO the message GET-HOURS and reply HOURS-WORKED are allowed to pass. However, the method invocation in WORK-INFO is given the restricted status. This prevents the method from updating the state of object WORK-INFO (which if allowed, would cause a write-down violation).

In processing such messages, we can visualize the generation of a tree of method invocations as shown in figure 3. The restricted methods are shown within shaded regions. Suppose tk is a method for object Ok and tn a method for object On which resulted due to a message sent from tk to On. The method tn has a restricted status because $L(On) < L(Ok)$. The children and descendants of tn will continue to have a restricted status till such points as ts. The method ts is no longer restricted because $L(Os) \geq L(Ok)$ and a write by ts to the state of Os no longer constitutes a write-down. This is accounted for in the conditional assignment to s(t2) in case (3) of figure 2.[**]

The variable **rlevel** plays a critical role in determining whether or not the child of a restricted method should itself be restricted. It keeps track of the highest security level encountered as a chain of method invocations progresses. For example, consider a message sent from a Secret object to a Confidential one. The **rlevel** de-

[§]A backward flow occurs through the return value in the reply received for a previous message [6].

[¶]A forward flow occurs through the parameters in a message [6]. An indirect flow from a sender to a receiver object occurs through the parameters in messages sent by intermediate objects [6].

[‖]In practise, there will be additional primitive messages. The three identified here suffice to illustrate the main ideas.

[**]The message filtering algorithm presented here improves upon the version presented in [6]. The earlier version was unnecessarily restrictive as it required all descendants of a restricted method such as tn to be restricted as well.

**if** $O1 \neq O2 \vee m \notin \{\text{READ, WRITE, CREATE}\}$ **then case** % *i.e., m is a non-primitive message*

(1) $L(O1) = L(O2)$ : % *let m pass, let reply pass*

$$\textbf{invoke } t2 \textbf{ with} \begin{cases} s(t2) \leftarrow s(t1); \\ rlevel(t2) \leftarrow rlevel(t1); \end{cases}$$

**return** reply from t2 **to** t1;

(2) $L(O1) \sim L(O2)$ : % *block m, inject NIL reply*

**return** NIL **to** t1;

(3) $L(O1) < L(O2)$ : % *let m pass, inject NIL reply, ignore actual reply*

**return** NIL **to** t1;

$$\textbf{invoke } t2 \textbf{ with} \begin{cases} s(t2) \leftarrow \textbf{if } L(O2)<rlevel(O1) \textbf{ then } s(t1) \textbf{ else } U; \\ rlevel(t2) \leftarrow \max[L(O2), rlevel(t1)]; \end{cases}$$

**discard** reply from t2;

(4) $L(O1) > L(O2)$ : % *let m pass, let reply pass*

$$\textbf{invoke } t2 \textbf{ with} \begin{cases} s(t2) \leftarrow R; \\ rlevel(t2) \leftarrow \max[L(O1), rlevel(t1)]; \end{cases}$$

**return** reply from t2 **to** t1;

**end case**;

**if** $O1 = O2 \wedge m \in \{\text{READ, WRITE, CREATE}\}$ **then case** % *i.e., m is a primitive message*

(5) m is a READ : % *allow unconditionally*

READ value; **return** value **to** t1;

(6) m is a WRITE : % *allow if status of t1 is unrestricted*

**if** s(t1) = U **then** [WRITE; **return** SUCCESS **to** t1;]
**else** **return** FAILURE **to** t1;

(7) m is a CREATE : % *allow if status of t1 is unrestricted*

**if** s(t1) = U **then** [CREATE O **with** L(O) ← L(O1); **return** O **to** t1;]
**else** **return** FAILURE **to** t1;

**end case**;

Figure 2: Message filtering algorithm

Figure 3: Method-invocation tree with restricted paths and subtrees

rived for the method invocation at the receiver object will be Secret. The value of **rlevel** is initially equal to the classification of the first sender object in a chain and subsequently derived for every method invocation. If the **rlevel** of a method is higher than the level of the object which the method accesses, the method is given restricted status. On the other hand, if the value of **rlevel** is the same as the level of a receiver object, the method in the receiver object will be given the unrestricted status. In other words, the restricted status along the chain may be removed because write-down violations and illegal flows can no longer occur. In case (3), the status of method invocation t2 is the same as that of t1 as long as the level of the receiver is lower than the **rlevel** of the sender. The **rlevel** derived for a method can have a higher value than those of earlier methods. This happens if a sender object has a higher classification than the objects encountered so far in the chain (as shown in case (4)). If the sender and receiver objects have the same classification (as in case (1)), the **rlevel** of the receiver is the same as that of the sender. Finally, if the sender and receiver are incompatible (as in case (2)), the variable **rlevel** plays no role.

We now discuss the handling of primitive messages. READ operations (case (5)) never fail because read-up operations cannot occur. This is because read operations are confined to an object's methods and their results can only be exported by messages or replies which are filtered by the message filter. The WRITE and CREATE operations invoked on receiving the WRITE and CREATE messages (cases (6) and (7)) will succeed only if the status of the method invoking the operations is unrestricted. If a WRITE or CREATE operation fails, a failure message is sent to the sender. This failure message does not violate security since information is flowing upwards in level. In the case of a CREATE message, the new object is created at the same level as the object requesting the CREATE.

## 4 A Secure Kernelized Architecture

In this section we illustrate how our secure architecture is motivated by and built upon the architecture of existing object-oriented database systems such as ORION [4], IRIS [5], and GEMSTONE [10]. Figure 4 depicts the typical architecture of these systems. We wish to emphasize that although there are substantial differences in the features supported by the underlying object-oriented models, architecturally these systems are not that different. This is analogous to procedural programming languages which differ considerably in their syntax and semantics but are typically implemented using the same compiler architecture.

As can be seen in figure 4, this architecture is a layered one consisting of storage and object layers. We refer to the modules implementing these layers as the *storage manager* and *object server* subsystems respectively. The storage layer interfaces to the operating system and file system primitives. The functionality supported by this module enables the read, write, and

Figure 4: A layered architecture for object-oriented databases

creation of raw bytes representing untyped objects. A unique pointer (identifier) is associated with every chunk of bytes representing an object. The association between the pointers and the physical location of objects is maintained in an object table. A request to create a new object will result in the allocation of a new pointer. This module typically provides other functions such as concurrency control. In existing systems concurrency control is typically based on a check-in/check-out paradigm [5, 10].

In contrast to the storage layer which manipulates raw bytes, the object layer provides the abstraction of objects as encapsulated units of information (instances of abstract data types). By supporting the notions of messages, objects, classes, class-hierarchy, and inheritance, the object layer implements the underlying object-oriented data model. The object layer thus supports the functionality to enable objects to send messages and replies to each other, to access and update object states, as well as to create new objects. The operations to access, update, and create objects utilize the services of the lower storage layer.

We now discuss the primitive operations supported by the storage and object layers. At the storage layer these are READ, WRITE, and CREATE. These primitives are invoked to read, write, and create bytes representing objects. At the security perimeter of the object layer the primitive operations are SEND, QUIT, READ, WRITE, and CREATE. The READ, WRITE and CREATE are the primitive messages discussed in section 3. SEND and QUIT are system primitives used by methods to send messages and replies. The SEND primitive is invoked by a method to send a message to a specified object. If the SEND is permitted to pass through the message filter, it results in the invocation of the appropriate method in the receiv-

Figure 5: A secure kernelized architecture

ing object. When a method execution terminates, the QUIT primitive signals termination of the method and returns the reply to the sender.

Before an object's state is accessed or updated, the object has to be selected from the persistent store and transferred to memory. This is typically accomplished by an explicit or implicit OPEN operation. Finally, the updates to object states are made permanent by an explicit or implicit CLOSE/COMMIT operation. If the user wishes to abandon the updates made during a user session he/she can issue an ABORT request. For simplicity of exposition, we have chosen implicit OPEN and CLOSE/COMMIT operations.

In designing a secure architecture one of the critical tasks involves determining the security perimeter. In other words, we need to determine what functions and modules need to be trusted and thus implemented within the TCB. In this paper we assume that the entire storage layer is trusted. This is a reasonable assumption considering that this layer provides very specific functions and is not terribly large. However, as indicated in figure 4 only a small portion of the object layer needs to be trusted. In fact, the trusted functions are precisely those required to implement the message filter.

Figure 5 depicts our secure kernelized architecture that is derived from the architecture in figure 4. As can be seen in the figure, the storage layer is entirely within the TCB and essentially remains unchanged. We refer to the storage manager here as the *trusted storage manager*. The object layer however differs significantly from the earlier architecture. The trusted portion of the object layer now consists of a *session manager* and one or more *message manager* modules. A key aspect of our architecture is that the session manager runs as a multilevel subject while the message managers are single-level subjects with respect to the Operating System TCB. The message manager and session manager modules collectively implement the message filtering algorithm. The algorithms and implementation details of these modules are the main issues discussed in section 5.

# 5 Implementation Algorithms

In the previous section, we presented the object and storage layers that make up the typical architecture of current object-oriented database systems. In this section, we focus on the trusted functions needed to make such architectures secure as indicated in figure 5. Our objective is to stay within the general spirit of current object-oriented DBMS architectures and suggest the minimal modifications necessary for this purpose. Of course, we also seek to keep the totality of the TCB as small as possible.

## 5.1 The Trusted Storage Layer

First consider the storage layer. The storage layer layer is implemented by a trusted module called the *trusted storage manager*. This layer is responsible for maintaining the association between object identifiers, corresponding classifications of objects, and their physical addresses on persistent storage in an *object-disk table*. We assume the underlying trusted Operating System supports a segmented memory. Objects are transferred to and from this segmented memory as needed to enable the access and updates of object states. Every object resides in its own segment in memory which is labeled by the object's classification. The association between an object-identifier and its segment is stored in an *object-segment table*.

As mentioned earlier, schemes for transaction management and concurrency control are often supported at the storage layer. Existing object-oriented database systems [5, 10] typically utilize the check in/check out paradigm for this purpose. The basic idea here is that a user checks out an object from a public database into a private workspace. When the user has finished with the object (after reads and modifications) it is checked back into the public database. Such a model of transactions fits neatly into the architecture we present in this paper and can be handled by the storage manager module. In particular, our implementation guarantees that there cannot exist write-write conflicts involving users at multiple levels. In the security context, this eliminates covert storage channels that could arise by a low user's transaction being aborted or delayed due to conflicts with high-level transactions. Within a single-level, conflicts can be handled as in existing systems (such as rolling back or aborting transactions).

### 5.2 Trusted Object Layer Subset

In contrast to the storage layer the object layer must be mostly untrusted for a couple of reasons. Firstly it is fairly large and complex. Secondly it is this layer which is likely to be different from one version of a system to another. Object-oriented concepts such as inheritance and delegation are implemented within this layer. We need room for flexibility, options and configurability here. It is also inevitable that as users become familiar with object-oriented systems they will demand greater functionality in this layer. It is therefore vital that the trusted functions within the object layer be clearly identified and separated from the rest of the object layer. The conceptual identification of this component has already been accomplished in the message-filter model, i.e., the trusted function is the message filter.

Let us consider the message filtering algorithm of figure 2. This algorithm has been written in a procedural notation and indeed much of it can be interpreted in the usual sequential interpretation style of programming languages. There is however one place, viz., case (3) of the algorithm, where the usual sequential interpretation breaks down. In this case, a message is sent from an object to a receiver object at a higher level. The message filter now has to prevent any backward flow of information that may occur through the reply to this message. The message filter achieves this by returning a NIL value to the sender and discarding the actual reply. However, in order to avoid a timing channel, it should not be possible for the high method to modulate the timing of the delivery of this NIL. Thus, delivering the NIL value on the termination of the method in the receiver (and effectively suspending/blocking execution of the sender method during this period) is clearly not acceptable.

To address the above, our only alternative is to permit concurrent computations as mentioned earlier in section 2. In other words we should allow the sender method that sent a message and the receiver method to be invoked on receipt of this message, to execute concurrently. However, the sender should not block after the send waiting for a reply. Hence the NIL reply is returned immediately to the sender independent of the receiver's termination point. The receiver's method is executed by a newly created and concurrently running message manager process.

While the concurrent solution above does eliminate timing channels in our computational model, it introduces the following synchronization problem: we must ensure that the concurrent computations spawned to close the timing channels are exactly equivalent to the intended sequential execution of the methods. This requirement introduces the major complication in our implementation. A secure multiversion protocol for this synchronization will be presented in a moment.

In general, several message managers may be created for a user session, depending on the number of messages sent upwards in security levels. However there exists only one session manager per user session. A session manager coordinates the various message managers and other relevant information pertinent to a user session.

#### 5.2.1 Trusted Message Manager

The message manager algorithms are shown in figure 6 for processing SEND and QUIT requests. It is easy to see the correspondence between the cases of the filtering algorithm in figure 2 and its implementation in figure 6. Every message manager runs at a fixed security level given by lmsgmgr whose value is determined by the session manager in a manner to be described later.

The following functions in the filtering algorithm need to be implemented: (1) letting messages pass; (2) blocking messages; (3) setting return values to NIL; and (4) setting the status of method invocations.

To address the implementation of the above, we have to consider the dynamics of message propagation and method invocation in object-oriented systems. Consider a message propagation sequence involving three objects. Say, object O1 sends a message to O2 and this results in the invocation of a method in O2. The execution of this method may result in a message being sent to O3. When the method in O3 finishes execution, it will return a reply to O2. The general approach used (as for example in [10]) to manage contexts for such sequences of message propagations is to use a call stack. Our architecture calls for such context management to be done in a trusted fashion and thus by the message manager. Each stack frame stores various information, such as the message parameters, regarding a message. The integrity of this information has to be guaranteed by the message manager.

We now describe the individual cases in figure 6. When a message m with parameters p is allowed to pass, as in cases (1), (3), and (4), a new context frame is pushed onto the call stack. In case (3), a new message manager is created (by a FORK call to the session manager as described later), and this operation is done by the session manager while initializing the stack of the new message manager. When a message is blocked, as in case (2) (due to the incompatibility between sender and receiver levels), no new frame is pushed onto the stack. When the filtering algorithm calls for the return of a NIL value as a reply (as in cases (2) and (3)), the message manager writes a NIL value on the top frame of its stack. This value is subsequently returned to the method that sent the message. A method terminates by issuing a QUIT call with the return value (r) as a parameter. The message manager then pops the call stack. If the stack is empty the message manager issues the TERMINATE call to its session manager to terminate itself; otherwise it writes the return value into the top frame of the stack and resumes execution of the method which was waiting for this reply.

Having discussed the non-primitive cases let us look at the primitive messages. For a READ (case (5)), there exists two possibilities. A read at the same level

**procedure** SEND(m, p, O1, O2)
**if** O1 $\neq$ O2 $\vee$ m $\notin$ {READ, WRITE, CREATE} **then case** % *i.e., m is a non-primitive message*
(1) L(O1) = L(O2) : PUSH-STACK(p); t2 $\leftarrow$ **select** method for O2 **based on** m; **start** t2;
(2) L(O1) $\sim$ L(O2) : WRITE-STACK(NIL); RESUME;
(3) L(O1) < L(O2) : FORK(lmsgmgr, O2, m, p, WStamp); WStamp $\leftarrow$ WStamp + 1;
WRITE-STACK(NIL); RESUME;
(4) L(O1) > L(O2) : PUSH-STACK(p); t2 $\leftarrow$ **select** method for O2 **based on** m; **start** t2;
**end case**;
**if** O1 = O2 $\wedge$ m $\in$ {READ, WRITE, CREATE} **then case** % *i.e., m is a primitive message*
(5) m is a READ : **if** L(O1)= lmsgmgr **then** v $\leftarrow$ WSTAMP **else** v $\leftarrow$ RSTAMP(L(O1));
**read** O1 with max{version: version $\leq$ v};
(6) m is a WRITE : **write** O1 with version $\leftarrow$ WStamp;
(7) m is a CREATE : **create** O with L(O) $\leftarrow$ L(O1) and version $\leftarrow$ WStamp;
**end case**;
**end procedure** SEND;

**procedure** QUIT(r)
POP-STACK;
**if** EMPTY-STACK **then** TERMINATE(lmsgmgr,WStamp) **else** [WRITE-STACK(r); RESUME;]
**end procedure** QUIT;

Figure 6: Message manager algorithms for SEND and QUIT

of the message manager will result in the reading of the latest version in memory. However, if it is a read down request the version read will be the latest that existed as of the time the message manager was forked. In the case of a WRITE (case (6)), if a version with time stamp WStamp already exists, it is overwritten in place. If no such version exists, a new version with time stamp Wstamp is created. A CREATE (case (7)) results in the new object being created with the current time stamp of the message manager. It is important to point out here that primitive messages are processed by direct system calls and as such the performance penalty incurred is minimal.

It remains to show how the effect of setting the status of method invocations to restricted or unrestricted is achieved. Recall that the message manager runs at a fixed level given by lmsgmgr. Every method executed by the message manager thereby also runs at the level lmsgmgr. A method is prevented from updating an object's state if it violates the standard $\star$–property of the Bell-Lapadula security model [3]. Thus, for example, a method invocation at the secret level is prevented from writing into an unclassified object's segment thereby achieving the effect of a restricted method invocation in the unclassified object. An attempt such as this to violate mandatory security will result in the return of a "FAILURE" message to the sender's method.

The security level of a message manager is derived at its time of creation and is equivalent to the **rlevel** derived for the corresponding method invocation using the message filtering algorithm. Thus, the level is derived from the level of the receiver object and the level of the parent message manager issuing the fork (as shown in the algorithm for FORK in figure 7).

The interface between a message manager and its session manager is made up of two calls: (1) FORK issued by a message manager to its session manager to request the creation of a new message manager, and (2) TERMINATE issued by a message manager to its session manager to terminate itself. This brings us to consideration of the session manager.

#### 5.2.2 Trusted Session Manager

A session manager is a trusted multilevel subject instantiated on behalf of every active user session in the system. It is charged with the task of coordinating the various message managers that are forked. The algorithms implemented by the session manager are shown in figure 7 and are: (i) FORK and (ii) TERMINATE, and are used to respectively process fork and terminate requests from message managers. Both algorithms utilize a common procedure START to initiate execution of message managers. A fork request may not result in the immediate execution of a new message manager. This is because a session manager enforces a discipline on the concurrent execution of its message managers so as to achieve the desired equivalence to the intended sequential execution. Correspondingly, the termination of a message manager may result in initiating the execution of a queued one.

```
procedure FORK(lmsgmgr, O2, m, p, WStamp);
mm ← pointer to msmgr node with level = lmsgmgr and status = active;
% there is a unique msgmgr node with these properties
% create a new msgmgr node with appropriate values
nn ← pointer to a new msgmgr node; nn.level ← max{lmsgmgr, L(O2)}; nn.lcreator ← lmsgmgr;
nn.tcreation ← WStamp; nn.object ← O2; nn.message ← m; nn.p ← p;
case
    mm.child ≠ nil: % enqueue new msgmgr node at tail of mm.queue
            nn.status ← queued; nn.child ← nil; nn.parent ← nil; nn.queue ← nil; enqueue nn on mm.queue;
    mm.child = nil: % insert new msgmgr node as child of mm and activate it
            nn.status ← active; nn.child ← nil; nn.parent ← mm; nn.queue ← nil; START(nn);
end case
end procedure FORK;

procedure TERMINATE(lmsgmgr, WStamp);
mm ← pointer to msmgr node with level = lmsgmgr and status = active;
% there is a unique msgmgr node with these properties
% mark this msgmgr as terminated and record its termination time
mm.status ← terminated; mm.tterminate ← WStamp;
% attempt to start a queued message manager or end-session if appropriate
loop ← true;
while loop do begin case
mm.child ≠ nil: % quit while loop
        loop ← false;
mm.child = nil ∧ mm.queue = nil: % update RStamp, delete mm and look to parent (if any)
        RStamp[mm.level] ← mm.tterminate;
        if mm.parent = nil then END-SESSION else [mm ← mm.parent; mm.child ← nil];
mm.child = nil ∧ mm.queue ≠ nil % activate head of mm.queue as child of mm
        hh ← dequeue mm.queue; hh.child ← nil; hh.parent ← mm; hh.queue ← nil;
        START(hh); loop ← false;
end case end while;
end procedure TERMINATE;

procedure START(nn);
%activate msgmgr node nn and run it concurrently
nn.status ← active;
fork message manager process with
    STACK ← EMPTY-STACK; PUSH-STACK(nn.p);
    lmsgmgr ← nn.level; RStamp[lcreator] ← nn.tcreation; WStamp ← RStamp[nn.level];
    t2 ← select method for nn.O2 based on nn.m; start t2;
end with;
end procedure START;
```

Figure 7: Session manager algorithms for FORK and TERMINATE

Consider the tree of message managers shown in figure 8. The labeled nodes (circles) in the figure represent computations (message managers executing methods) while the arrows represent messages. The figure shows a snapshot of a tree of message managers (computations) with message manager 1 at the unclassified level having sent messages to one secret object, one top-secret object and one confidential object in this sequence. These receiver objects are at a higher level than the sender and this has resulted in the forking of message managers 2, 5, and 6 as the children of 1. Similarly message manager 2 at the secret level has forked off two message managers at the top-secret level.

In managing such a tree of message managers the session manager guarantees the following invariant.

- *Only the leftmost computations are allowed to execute concurrently.*

The leftmost computations consist of the message managers on the path from the root of the tree to the leftmost leaf (message managers 1, 2, and 3 in figure 8), i.e., the leftmost path. The progress of the tree of computations in figure 8 as governed by the session manager is shown in figure 9. In each successive diagram, a terminated message manager that advances the computation to the next stage is highlighted. Note that the leftmost path has at most one message manager at each ascending security level. This property is the key foundation on which our synchronization protocol is built.

The implementation of a protocol that guarantees the above invariant requires the session manager to maintain various bits of state information for each message manager. In particular the status of a message manager is one of the following.

1. *Active:* If a message manager is one of the leftmost computations, it is allowed to execute and is thus considered to be active.
2. *Queued:* If a message manager is not one of the leftmost computations it is not allowed to be active and is queued for later execution.
3. *Terminated:* When an active message manager terminates its status is changed to terminated until such point as it can be deleted from the tree. Deletion is permitted only if all descendants of this message manager have themselves terminated and been deleted from the tree.

To summarize, the history of a message manager is as follows: (i) possibly queued, (ii) active, (iii) terminated, and (iv) purged from the tree.

Our objective is to ensure, in a secure manner, that the concurrent computations managed by a session manager achieve precisely the same result as the intended sequential execution. The labels on the arrows in figure 8 convey the order in which the messages (sent to higher level objects) are processed if we execute this

Figure 8: A tree of concurrent message managers

Figure 9: Progressive execution of figure 8

tree of computations sequentially. This order can be derived by a depth-first traversal of the tree. We illustrate the synchronization problem that arises when methods are executed concurrently. In the payroll database of figure 1, consider a concurrent execution of methods that led to the message sequence (as identified by the message labels): *a, d, e, f, b, c*. In order to achieve the same result as a sequential execution (with message sequence: *a, b, c, d, e, f*) the method in object PAY-INFO should not see any changes in WORK-INFO that occurred after it was forked.

### 5.3 Multiversion Synchronization

Solving such synchronization problems using classical techniques such as those based on locking and semaphores is of course known to be unsuitable for multilevel secure systems (as they introduce covert channels). Our solution instead relies on retaining multiple versions of objects in memory. Thus in the above scenario the processing of the (e) RESET-WEEKLY-HOURS message would result in the creation of a new version of object WORK-INFO with the reset hours. However, an earlier version of object WORK-INFO that existed before the method in PAY-INFO was forked is used to process the (b) GET-HOURS message. The versioning scheme is generalized for a tree of concurrent message managers and must ensure that read-down requests see exactly the object states that would have existed had the tree of concurrent message managers executed sequentially in a depth-first manner.

We now discuss the management of concurrent message managers and the multiversion synchronization scheme in detail as incorporated in the algorithms implemented by the message and session managers. The following timestamps are used for this purpose:

- **WStamp.** This is used by each message manager to time stamp versions of objects written by the message manager. Its initial value is determined as the value of its parent's **WStamp** at the instant that this message manager was forked. It is incremented following every fork executed by this message manager.
- **RStamp.** This is a table of time stamps, one per level, used by the active message managers to read the appropriate version at each level below lmsgmgr (the version read at lmsgmgr is always the most recent one). This table is updated on every START operation and some TERMINATE operations. An individual message manager only sees that portion of this table which is for levels strictly dominated by the message manager. During the execution of a message manager this portion of the table remains constant. The session manager is responsible for maintaining the **RStamp** table.

The session manager maintains a data structure to keep information about the computation tree illustrated in figure 8. This data structure is a doubly linked list of msgmgr nodes with root pointing to the head of the list. The list keeps track of the currently active computations in the computation tree, i.e., the leftmost path. Each msgmgr node in this list also points to a queue of queued msgmgr nodes waiting to become active. Each msgmgr node stores the information shown in table 1 about the corresponding message manager. These attributes should be self-explanatory at this point.

| *Attribute* | *Comment* |
|---|---|
| status | active, terminated or queued |
| level | level of the message manager |
| lcreator | level of creator |
| tcreation | WStamp of creator |
| tterminate | WStamp at termination |
| child | pointer to child msgmgr |
| parent | pointer to parent msgmgr |
| queue | pointer to queue of msgmgr nodes |
| object | receiving object |
| message | message |
| p | message parameters |

Table 1: msmgr node data structure

Let us walk-through the algorithms beginning with a fork request from a message manager (as shown in case (3) in figure 6). On issuing the fork a message manager passes its security level (lmsgmgr) and WStamp among other information to the session manager. The message manager then increments the value of WStamp. This ensures that it will write a new version after issuing each fork. On receiving the fork the session manager records the Wstamp received from the message manager that issued the fork, in the tcreation attribute of a new msgmgr node (see figure 7). If the parent message manager currently has an active or terminated child in the tree, this fork request is queued. Otherwise execution of the new message manager is initiated by calling the procedure START. START initializes a new stack for the created message manager and updates the RStamp at the level of its parent (creator) as well as its own WStamp. The update of the RStamp entry of the parent ensures that read down requests from the newly created message manager (and its potential descendants) read the version that existed before the new message manager was forked. The new message manager updates its own WStamp by looking up the RStamp entry at its own level. The Wstamp is then subsequently used for processing read and write requests from local methods (and will be incremented after any fork call).

On receiving a terminate request (see figure 7), the session manager records the new status of the message manager as terminated and further records the WStamp in tterminate. The session manager then checks to see if the terminated message manager has an active or terminated child. If it does, no new computations can be started and so the terminate algorithm is exited. If the terminating message manager has no child but has a non-empty queue of forked message managers pending execution, the first one in the queue is started by calling the START procedure. If the above two conditions are not true, i.e., if there exists no child and no queue, then this message manager can be purged from the data structure. The algorithm then looks to the parent of this message manager to see if execution of some queued message manager can be initiated. Eventually the tree becomes empty and the session terminates.

### 5.4 Analysis

We now informally sketch a proof of correctness for the synchronization protocol presented in this paper. Formal proofs are quite feasible but are beyond the space constraints of this paper. We begin by making clear what we mean by correctness. We can model computations in our system as a sequence of events as invoked by our algorithms. The events of interest to us include: read, write, send, fork, quit, start, and terminate. The events generated by the algorithms clearly depend on the sequence in which methods are executed. Now consider the sequence of events generated by a sequential execution of methods. In such an execution once a method sends a message it waits for the reply to this message before proceeding. We can compare the results of such a sequential computation to the results obtained by corresponding methods when executed concurrently under the supervision of a session manager. If these results are equivalent, we can conclude that our algorithms are correct (modulo correctness of the sequential execution). In particular, all read events in the concurrent execution should read the same states of objects as in a sequential execution.

We make the following claims.

1. There exists only one active (executing) message manager at a security level at any given time. This has been discussed earlier.
2. There exists no write-write conflicts at objects. This is a natural outcome of 1.
3. Consider two write operations in an executing method. We are guaranteed by our algorithms that there will never exist an interleaving write operation from a higher level method. This follows from the fact that write-down operations are not permitted.
4. If two methods are at the same **rlevel** and an interleaving write operation (caused by the second method) occurs between the writes of an executing method, then the relative order of these writes will be the same in both the sequential and concurrent executions. This is because methods at the same **rlevel** are executed sequentially.

From (1) to (4) we see that when a method terminates under the supervision of a session manager, it leaves all the objects accessed in the same state as it would have in a sequential execution. To make sure that such consistent states of objects are made available to the rest of the system, the session manger updates the RStamp at the level of the terminated message manager to reflect the latest state of objects available. This is done in the TERMINATE algorithm of figure 7 by assignment of the variable tterminate to RStamp.

It now remains to show that a read operation obtains the same state of objects as in a sequential execution. If two methods m1 and m2 are executed sequentially (i.e., m2 is invoked as a result of a SEND from m1) m2 will always read the state of objects that existed before the SEND. This is achieved in our synchronization scheme by making a message manager remember the state by recording its parent's WStamp at the time of fork (i.e., nn.tcreation $\leftarrow$ WStamp in figure 7). This remembered state is then requested at the time of execution (i.e., RStamp[lcreator] $\leftarrow$ nn.tcreation, as shown in figure 7). This completes our proof sketch.

Turning to security we have demonstrated how our filtering algorithms enforce the security policy by preventing illegal flows of information. In the implementation of our algorithms we have also addressed the problems of timing channels. Our implementation also eliminates covert channels that could arise due to conflicting operations issued by users at various security levels. This is because our schemes for executing methods guarantee that write-write conflicts between multiple users at different security levels will not occur.

To get a perspective on the performance implications of our schemes, we can look at the overhead introduced. New concurrent computations are requested only when messages are sent to higher levels. Also, at most one message manager (computation) is active at any level. This limits the overhead considerably in comparison to schemes that require the creation of a new process for every method to be executed in the system. Also our synchronization scheme calls for new versions to be created only if write operations occur after a fork request by a message manager.

## 6 Conclusion

In this paper we have identified the fundamental requirement that secure "writing up" in terms of abstract operations requires logically sequential computations to be executed asynchronously. Furthermore to achieve this and be correct (i.e., equivalent to the logically sequential computation) the actual asynchronous execution must keep available multiple versions of the data. Our solution is cast in context of the object-oriented data model because it is the most flexible data abstraction model known to the authors. The algorithms, being asynchronous, are inherently distributed and their interactions can be described precisely only in a concrete context such as given in this paper.

As a consequence of our concrete algorithmic description, we have given the conceptual implementation of the message filter security model using a traditional TCB. This demonstrates that the complexity of an object-oriented implementation is tractable. We have mapped the message filtering functions to the traditional system calls of a secure operating system kernel. The trusted functions in our architecture are object access, management from memory and persistent store, message management, and context management. We have presented algorithms for managing asynchronous method execution and multi-version synchronization. These algorithms eliminate timing channels that arise in the logically sequential computational model of the message filtering algorithm. The synchronization protocols hold potential for optimizations such as reducing the number of versions that need to be retained. We will be looking at such optimizations in future work.

## Acknowledgment

We are indebted to John Campbell, Joe Giordano and Howard Stainer for their support and encouragement, making this work possible. The opinions expressed in this paper are of course our own and should not be taken to represent the views of these individuals.

## References

[1] Minutes of the First Workshop on Covert Channel Analysis, *Cipher*, Special Issue, July 1990.

[2] Trusted computer system evaluation criteria. *DoD Computer Security Center*, December 1985.

[3] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. *Technical Report MTR-2997, The Mitre Corp.*, 1976.

[4] W. Kim et al. Features of the ORION object-oriented database system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publ. Co., Inc., Reading, MA, 1989.

[5] D. Fisherman. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):pp. 48–69, January 1987.

[6] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multi-level security. *Proc. of the 1990 IEEE Symposium on Security and Privacy*, pp. 76–85, May 1990.

[7] T.F. Keefe and W.T. Tsai. Prototyping the SODA security model. *Proc. 3rd IFIP WG 11.3 Workshop on Database Security*, September 1989.

[8] T.F. Keefe, W.T. Tsai, and M.B. Thuraisingham. A multilevel security model for object-oriented systems. *Proc. 11th National Computer Security Conference*, pp. 1–9, October 1988.

[9] B. Kogan, S. Jajodia, and R. Sandhu. Implementation issues in multilevel security for object-oriented databases. *Proc. of the Workshop on Security for Object-oriented Systems*, April 1990.

[10] D. Maier. Development of an object-oriented DBMS. *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 472–482, 1986.

[11] J.K. Millen and T.F. Lunt. *Secure Knowledge-based Systems.* Technical Report, Computer Science Laboratory, SRI International, August 1989.

[12] A. Skarra, S. Zdonik, and S. Reiss. An object server for an object-oriented database. In *Proc. 1986 Intl. Workshop on Object-Oriented Database Systems*, ACM/IEEE, 1986.

[13] A. Synder. Encapsulation and inheritance in object-oriented programming languages. *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 38–45, 1986.

[14] M.B. Thuraisingham. A multilevel secure object-oriented data model. *Proc. 12th National Computer Security Conference*, pp. 579–590, October 1989.