Proc. of the IFIP WG11.3 Workshop on Database Security,
Shepherdstown, West Virginia, November 4-7, 1991.

SUPPORTING TIMING-CHANNEL FREE COMPUTATIONS IN MULTILEVEL SECURE OBJECT-ORIENTED DATABASES

Ravi S. Sandhu, Roshan Thomas and Sushil Jajodia

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

Abstract
In an earlier paper [3], Jajodia and Kogan proposed a message filter approach to enforcing mandatory security in multilevel object-oriented databases. The key idea in the message filter model is that all information exchange be permitted solely through messages and that security be enforced by a message filter component that mediates these messages. In a recent paper [8] the authors proposed a kernelized architecture for implementing the message filter model. A major complication in implementing this model arises due to timing channels intrinsic to the object-oriented model of computing. These channels arise because object-oriented "write-up" operations are abstract and arbitrarily complex (as opposed to primitive memory writes). One approach to closing these timing channels is to execute a logically sequential computation as concurrent pieces. Our earlier paper presented an execution model for managing such concurrent computations as well as a multiversion synchronization protocol to guarantee correctness with respect to the intended sequential execution. While our approach with asynchronous computations can close such channels, the scheduling strategy presented earlier was not totally secure as it may be exploited for timing channels under certain conditions. In this paper we present a revised execution model that not only guarantees correctness but is also timing channel free. We give proof outlines to support these claims.

1 Introduction

In recent years we have seen a considerable effort in the research and development of object-oriented databases (e.g., ORION [1], IRIS [2], GEMSTONE [6]). The driving force behind these efforts can be attributed to the advantages offered by object-oriented data models. These include the ability of objects to model the complex structure of entities in an application domain. Such modeling capabilities cannot be readily provided in conventional database systems as they have a rather flat view of data. Further, object-oriented systems provide the capability to model the behavior of real world entities in the domain through methods encapsulated in objects.

Intuitively, the object-oriented model appears attractive to environments requiring multilevel data security. This is because access control and security policies can be specified and implemented in terms of objects that map closely to the real world entities that are modeled.

Accordingly, we have seen a number of proposals for models that address mandatory security issues in object-oriented databases [3, 4, 5, 7, 9]. The model proposed by Jajodia and Kogan [3] (referred to as the *message filter model*) is distinguishable from this set as it expresses the security policy with a message filtering algorithm and relies on a message filter component to mediate all messages sent between objects. The main advantage of the message filter model is the simplicity with which mandatory security policies can be stated and enforced.

In a recent paper [8] we proposed a kernelized architecture for the message filter model that was motivated by the existing architecture of object-oriented database systems. A major complication in implementing the message filter model arises from timing channels inherent in the object-oriented computational model. This is because write-up operations on objects are no longer primitive, but rather are complex and may involve several computations taking arbitrary amounts of processing time. It is therefore insecure to report the completion of a write-up operation. Instead the process which initiated the write-up must proceed concurrently with the process which executes the write-up.

In other words these timing channels can be closed by supporting concurrent asynchronous computations. This idea was proposed in [8]. However, the scheduling strategy and execution model for concurrent computations that was presented in [8] is not totally secure as it may be exploited for timing channels under certain conditions. In this paper, we present an alternate execution model and a multiversion synchronization scheme that collectively ensure that the concurrent computations are timing channel free and serially correct (preserves the correctness of the intended logical and sequential* execution).

The rest of this paper is organized as follows. Section 2 gives a brief overview of the message filter model. This is followed in section 3 by the kernelized architecture for implementing the model. Section 4 presents the timing channel problem as well as algorithms to close these channels. Section 5 contains proof sketches for these algorithms. Section 6 gives our conclusions.

2 Message Filter Model

The main elements of the message filter model are objects and messages. Every object is assigned a single classification, which remains unchanged for the object's lifetime. Messages are assumed (and required) to be the only means by which objects can com-

*For the sake of simplicity we have throughout assumed that the computation is intended to be logically sequential. Our algorithms and protocols can be extended to the case where concurrency is present in the original computation.

municate and exchange information. Thus the central idea is that information flow can be controlled by mediating the flow of messages. Consequently, even basic object activity such as access to internal attributes, object creation, and invocation of local methods are to be implemented by having an object send messages to itself (we consider such messages to be primitive messages). The message filter takes appropriate action upon intercepting a message and examining the classifications of the sender and receiver of the message. It may let the message pass unaltered; or interpose a NIL reply in place of the actual reply; or set the status of method invocations (as restricted or unrestricted).

The message filter utilizes the filtering algorithm shown in figure 1 to mediate messages. We assume that O1 and O2 are sender and receiver objects respectively. Let L(O) denote the security level of object O. 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. Each method invocation has a status, denoted by S(t), which can be either R (for restricted) or U (for unrestricted). Each method invocation also has an attribute called rlevel which is a label from the lattice of security classes. The interpretation of status and rlevel is explained below. Both these attributes are determined when the method invocation is created and remain fixed thereafter for the duration of the method execution.

The two major cases of the algorithm correspond to whether or not m is a primitive message. Cases (1) through (4) in figure 1 deal with a non-primitive message sent from object O1 to 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. In case (2), the levels are incomparable 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. This is because a restricted method cannot update the state (attributes) of an object whereas an unrestricted method is allowed to do so. 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. Finally, it is important to note that every method invoked (such as t2 in cases (1), (3), and (4)) is executed at a fixed security level. This level is given by the variable rlevel in the algorithm. A method is given the restricted status whenever its rlevel is higher than the level of the object accessed by the method.

We illustrate these filtering functions with the help of a payroll database. 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 2. 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.

Let us see how cases (1), (3) and (4) in the filtering algorithm apply to the payroll database. Case (1) occurs when the sender and receiver are at the same level and applies

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) : % *let m pass, let reply pass*

invoke t2 **with** $\left\{\begin{array}{l} \mathrm{s(t2)} \leftarrow \mathrm{s(t1)}; \\ \mathrm{rlevel(t2)} \leftarrow \mathrm{rlevel(t1)}; \end{array}\right.$

return reply from t2 **to** t1;

(2) L(O1) $\sim$ L(O2) : % *i.e., the levels are incomparable*

% *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;

invoke t2 **with** $\left\{\begin{array}{l} \mathrm{s(t2)} \leftarrow \textbf{if } \mathrm{L(O2)}<\mathrm{rlevel(O1)} \textbf{ then } \mathrm{s(t1)} \textbf{ else } \mathrm{U}; \\ \mathrm{rlevel(t2)} \leftarrow \mathrm{lub[L(O2), rlevel(t1)]}; \end{array}\right.$

discard reply from t2;

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

invoke t2 **with** $\left\{\begin{array}{l} \mathrm{s(t2)} \leftarrow \mathrm{R}; \\ \mathrm{rlevel(t2)} \leftarrow rlevel(t1); \end{array}\right.$

return reply from t2 **to** t1;

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 : % *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) $\leftarrow$ L(O1); **return** O **to** t1;]

else **return** FAILURE **to** t1;

end case;

Figure 1: Message filtering algorithm

Figure 2: Objects in a payroll database

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 and the actual reply is discarded. 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).

Returning to the algorithm let us examine cases (5) through (7) which deal with primitive messages. READ operations always succeed as they are confined to an object's (internal) methods and the results can only be exported by messages or replies that are filtered by the message filter. In the case of WRITE and CREATE messages, the corresponding operation will succeed only if the status of the method invoking the operation is unrestricted (otherwise a FAILURE message is returned to the sender indicating failure).

3 Architecture

Figure 3 depicts the secure kernelized architecture proposed in [8] for implementing the message filter model. The architecture is motivated and built upon the typical architecture of existing object-oriented database systems. As shown in the figure this architecture is a layered one and consists of storage and object layers. The lower storage layer interfaces to the operating system and file system primitives and supports the functionality required

Figure 3: A secure kernelized architecture

for the read, write, and creation of raw bytes representing untyped objects. In contrast to the storage layer, the object layer provides a more abstract view of data by supporting the notion of objects as encapsulated units of information. Object-oriented concepts such as classes, class-hierarchies, and inheritance as well as supporting facilities such as message passing are implemented at this layer.

Our architecture calls for the entire storage layer to be trusted and thus implemented within the trusted computing base (TCB). This assumption is reasonable as this layer provides very specific functions and thus need not be very large. On the other hand, only a small portion of the object layer needs to be trusted and thus included within the TCB. The trusted functions are precisely those required to support the message filtering function and are collectively implemented by message and session manager modules.[†]

To provide support for asynchronous computations, a new message manager process is forked whenever a message is sent to a higher level object. Once forked, the message manager proceeds to execute the method invoked in the receiver object in order to enable processing of the received message. In general, several message managers may be created for a user session. A session manager process is thus created for every user session for the purpose of coordinating the various message managers. 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.

[†]We must of course also ensure that the message manager acts as a reference monitor, i.e., it cannot be bypassed in accessing objects. It the task of the underlying trusted operating system to ensure this.

In the next section we discuss in detail the algorithms required to process these FORK and TERMINATE requests in a secure and correct fashion.

4 Algorithms

In this section we discuss in details the problems posed by timing channels and elaborate on our approach to close such channels.

4.1 Addressing Timing Channels

Consider a message sent from a low object to a receiver object at a higher level. In accordance with the filtering algorithm the message filter returns a NIL value to the sender and discards 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.

Our solution to close such timing channels is to allow concurrent computations. In other words whenever a message is sent to a higher level object, we would allow the sender method as well as the receiver method to be invoked on receipt of this message, to execute concurrently. As we do not want the sender to remain blocked waiting for a reply, the NIL reply is returned immediately to the sender independent of the receiver's termination point.

Every method is executed by a message manager process that incorporates algorithms to implement the message filtering function. Thus whenever a message is sent to a higher receiver object, the receiver's method is executed by a newly forked (i.e., newly created) and concurrently running message manager process. Every message manager runs at a fixed security level that is equivalent to the rlevel of the corresponding method invocation (as determined by the message filtering algorithm). A user session may create several message managers depending on the number of messages sent upwards in security levels.

There exists a session manager process for every user session and it is charged with coordinating the various message managers created for a session. Since message managers may execute concurrently it is critical that such concurrent executions produce the same result as the intended sequential computation. In particular, a concurrent execution must ensure that the object states accessed by methods are equivalent to the states that would have been accessed in a sequential execution. Note, that we do not want arbitrary concurrency with serializability as the correctness criterion. Instead we require equivalence to the one specific serial execution which was logically intended. The session manager achieves this goal by enforcing a discipline on the concurrent execution of its message managers.

One can visualize such concurrent computations as a tree (see figure 4). 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

Figure 4: A tree of concurrent message managers

Figure 5: Progressive execution of figure 4

managers 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, 3, and 4 as the children of 1. Similarly message manager 4 at the confidential level has forked off two message managers at top-secret and secret levels respectively. The labels on the arrows in figure 4 convey the order in which the messages (sent to higher level objects) are processed if we execute this tree of computations sequentially. Note that this order can be derived by a depth-first traversal of the tree. In other words a computation to the left of another in the tree, would have been forked earlier and executed to termination ahead of the latter in a sequential execution.

4.2 The Execution Model

Any execution model to manage such a tree of concurrent computations must be motivated by two requirements: (1) the ease with which synchronization can be provided, and (2) the security properties.[‡] The synchronization protocol must ensure that we achieve the same results as the intended sequential computation. The security properties must guarantee freedom from timing and storage channels.

The execution model presented in [8] allowed only the leftmost computations in a tree to execute concurrently. This invariant guarantees that there will be only one active computation (message manager) at each ascending security level. However, the weakness of this scheme arises from the fact that new message managers are not always forked by ascending level. For example in figure 4 the message manager 4 at level confidential is forked after the higher message manager 3 at top secret. Consequently, message manager 4 will be denied execution until message manager 3 and its children terminate. If the lower message manager 4 can observe the elapsed time before it is actually started, a cooperating message manager such as 3 can introduce a timing channel.

We now present a scheme that overcomes this problem. The basic idea here is that the execution of a message manager is never delayed due to an earlier forked message manager at a higher level. A session manager now guarantees the following invariants in managing a tree of computations

- **Invariant** *A computation is started if and only if all the current as well as future computations to the left of it are guaranteed to execute at a higher level or incomparable level.*

Note that this invariant guarantees the following property: for every security level there can exist at most one executing (active) computation at that level at any given time.

Let us see the motivation and principles behind this invariant. The "only if" part is required to get correctness, i.e., equivalence to the intended logically sequential execution. A depth-first traversal of the tree specifies the order in which the message managers would terminate in a sequential execution. In particular for a given node in the tree, say n, we have the following properties in a sequential execution.

[‡]Performance must also be an important consideration. A detailed consideration of performance implications is beyond the scope of this paper. Our initial focus is on feasibility, correctness and security.

1. All nodes to the left of the path from the root to node n represent computations that would have terminated prior to n's execution.
2. The ancestor nodes of n represent computations that are blocked and waiting for their descendants (including n) to terminate.
3. All nodes to the right of n in the tree would be executed after the termination of n.

It follows that all writes performed by computations to the left of n must be visible to n, due to sequential precedence. In a multilevel context however only those writes at or below the level of n are actually visible. Thus it suffices to ensure that all computations to the left of n which are at or below the level of n have terminated. Similar observations apply to the computations to the right of n in the tree. That is, the writes performed by n must be visible to those computations to the right of n which are at or above n's level.

The "if" part of the invariant is required due to confidentiality. No computation should be delayed for some computation to its left which is at a higher or incomparable level. Such a delay opens up a timing channel. Fortunately the above invariant turns out to be necessary and sufficient to guarantee correctness and security.

To illustrate the application of this invariant consider execution of the tree in figure 4. The progress of this tree is shown in figure 5. The terminated message manager (node) which advances the computations to the next stage is highlighted. Message manager 2 being the first to be forked is allowed to execute immediately. Message manager 3 is queued up as its execution has to be delayed until message manager 2 terminates. This action is necessary because writes performed by the secret message manager 2 must be visible (i.e., readable) to the top secret message manager 3, due to sequential precedence. Our invariant guarantees that message manager 3 remains suspended at least till such time as message manager 2 and its top secret children terminate. Message manager 4 is allowed to execute immediately after being forked, since no writes from active non-terminated computations to the left of 4 in the tree will be visible at the confidential level. On the contrary, for confidentiality reasons, the start up of 4 should not be delayed (or modulated) by the higher message managers 2 (secret) and 3 (top secret) to the left of 4. Thus no potential exists for the introduction of any timing channels. Finally, message manager 5 will be started only after message manager 3 terminates and 6 will be started when 2 terminates.

4.3 Multiversion Synchronization

While our execution model above allows concurrent computations in order to close timing channels, it introduces a related synchronization problem. Synchronization schemes are needed to ensure that the concurrent execution of methods achieve the same result as a sequential one. For example in the payroll database of figure 2 a concurrent execution can lead to the message sequence (as identified by the message labels): *a, d, e, f, b, c*. Now in order to achieve the same result as a sequential execution (with sequence: *a, b, c, d, e, f*) the method in object PAY-INFO should not see any changes in object WORK-INFO that occurred after it was forked. To illustrate further, consider again the tree in figure

4. Although message managers 4 (confidential) and 6 (secret) may terminate well-ahead of 3 (top secret), our synchronization schemes must ensure that message manager 3 does not see any updates by message managers 4 and 6. This is because message managers 4 and 6 are to the right of 3 and as such would start later than 3 in a sequential execution. Similarly message manager 5 (top secret) should not see any updates by 6 (secret).

Solving this synchronization problem using classical techniques, such as those based on locking and semaphores, is unsuitable for multilevel secure systems as they introduce covert channels. Our solution instead relies on maintaining multiple versions of objects in memory. In the payroll example the processing of the (e) RESET-WEEKLY 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. Similarly in figure 4 message manager 3 (top secret) should not see any versions of confidential objects written by 4 or any secret objects written by 6.

To support the above versioning scheme, object versions are uniquely identified by time stamps. In order to incorporate the versioning scheme into our execution model, the session manager maintains the following data structure.

- **RStamp** This is a global table of time stamps with one entry per level. It identifies the initial version of objects at every level (that exists before a session starts). An individual message manager can see that portion of the table which is for levels dominated by that message manager.

The session manager also maintains a tree structure that reflects the progress of the concurrent message managers forked in a session. Every forked message manager is represented by a node (in the tree) that contains the following information attributes:

status:	this can be one of the following: active, terminated, queued;
level:	the level of the message manager;
local-stamp:	a local table where the time stamp entry at the level of the message manager indicates the version that will be written at the level; the time stamps in the other entries identify the versions that will be used to process read-down requests;
parent:	pointer to parent message manager;
object:	receiving object;
message:	message;
p:	message parameters;

4.3.1 Session Manager Algorithms

A high-level (pseudocode) specification of the session manager algorithms is shown in figures 6, 7, and 8. The algorithms make extensive use of the tree structure representing the various message managers. Let us discuss these algorithms in more detail. They are basically designed to ensure that the session manager invariant is never violated.

Procedure FORK(O2, m, p)
{
Let *parent* be the node issuing the fork;
Let *child* be a new message manager node;

Make the rightmost child of *parent*;
child.level $\leftarrow$ lub[parent.level, L(O_2)];

If in a depth-first traversal of the tree starting at the leftmost path (and until *child* is traversed) there exists a node, say n, with {n.level $\leq$ child.level and n.status = active or queued}
then child.status $\leftarrow$ queued;
else
    START(child);
end-if
}
end procedure FORK;

Figure 6: Session manager algorithm for FORK

Procedure TERMINATE(lmsgmgr)
{
Let *term* be the node that terminated at level lmsgmgr;
% *Mark this node as terminated*
term.status $\leftarrow$ terminated;

% *See if any queued nodes can be started*
Initiate a depth-first traversal to the right of *term* such that:

If for every leaf node say *leaf*, that is traversed there exists no previously traversed node p with {p.level $\leq$ leaf.level and p.status = active or queued}
then
    START(leaf);
end-if
}
end procedure TERMINATE;

Figure 7: Session manager algorithm for TERMINATE

Procedure START(nn)
{
%Let node nn represent the message manager to be started
%Update timestamps from ancestors
For every ancestor *a* of *nn* at level *l* **do**
nn.local-stamp[l] $\leftarrow$ a.local-stamp[l];

%Update timestamps from terminated message managers to the left
Initiate a depth-first search of the tree until node *nn* is traversed such that:

If the level *l* of a node *n* traversed is not a level of any of the ancestors of *nn*
and l < nn.level
then
    nn.local-stamp[l] $\leftarrow$ n.local-stamp[l];
end-if

%Update remaining local timestamp entries from the RStamp table
If there exists a level *l* lower than the level of *nn* and which is neither the level of a node traversed in the tree nor of an ancestor of *nn*
then
    nn.local-stamp[l] $\leftarrow$ RStamp[l];
end-if
}
end procedure START;

Figure 8: Session manager algorithm for START

Whenever a fork request is received, the session manager updates its tree structure by creating a node for the forked message manager and making it the right most child of the parent node issuing the fork. The session manager then checks to see if the forked node can be started immediately. To do so, a depth first traversal of the tree is made starting at the leftmost path until the newly inserted leaf node is reached. If during this traversal we find another node (either active or queued) at the same or a lower level, the newly inserted node (message manager) is queued and thus forced to wait.

The processing of a terminate request begins by updating the status of the node to terminated. We then check to see if this termination can release other nodes queued up. In determining this, our invariant leads to the property that any nodes started as a result of a termination have to be to the right of the terminated node and at a equal or higher level (and of course have to be leaves in the tree). Thus a depth first traversal to the right of the terminated node is initiated. Now as in the fork case, a leaf node is allowed to execute if and only if required by the invariant. It is important to note that a termination may result in more than one node being started. For example in figure 5(b) the termination of message manager node 2 (secret) results in nodes 3 (top secret) and 6 (secret) being started.

Both the Fork and Terminate algorithms utilize a common Start procedure. The procedure is concerned with updating the local-stamp table of a node before start up so that the message manager may know which versions of objects to use to process read down requests (at lower levels) and write operations at its level. The local-stamp-table entries may be updated with time stamps from three sources.

1. **Ancestors:** In a sequential execution, a sender method is effectively blocked until the receiver method terminates. The receiver would thus read down the latest update by the sender before it was blocked. Although in our concurrent execution a sender does not block, we can achieve the same read down result by requiring a sender to pass the last version (timestamp) it wrote at its level before issuing the fork as well as the timestamps of its ancestors, to its child (the receiver). Our scheme thus calls for the local-stamp entries at the levels of the ancestors to be updated by timestamps that have been successively passed down by the ancestors themselves (along the path on which the node to be started lies[§]).

2. **Terminated left nodes:** For levels dominated by a node's level, and for which timestamps were not obtained from the ancestors (as explained above), the start algorithm looks to the subtree of computations to the left of the node to be started. The time stamp of the last written versions at such levels is obtained from the last forked message manager (or rightmost node to the left of the node to be started) which wrote at these levels.

3. **RStamp table:** If there are levels for which time stamps could not be obtained from the above two sources, the algorithm then retrieves the time stamps from the global RStamp table maintained by the session manager. This is because objects at

[§]It does not matter in which order these timestamps are collected along this path because the level of each ancestor will be different from every other ancestor.

these levels have not been updated so far in the session. Thus the initial versions of objects (that existed before the session started) at these levels should be used by the starting message manager. The time stamps in the RStamp table identify such versions.

Thus once a message manager starts, its node in the tree will have all the timestamps necessary to process read down requests for objects classified below its level. These timestamps are never modified in the local-stamp table after start up. However, the timestamp entry at the level of the message manager is dealt with differently. On start, the timestamp is incremented unconditionally before the first write/update operation and subsequently incremented after every fork request issued to the session manager. Thus the timestamp passed on to the forked children by a message manager will vary. Each value identifies the state of the objects at the level of the message manager as of the time the fork was issued.

5 Correctness and Security Proofs

In this section we give proof sketches to show that our schemes preserve serial correctness and are free from timing channels.

5.1 Serial Correctness

We say that our multiversioning scheme preserves serial correctness if the concurrent execution as governed by our execution model guarantees the same result as a sequential (serial) execution of methods. Intuitively, serial correctness assures us that the database objects will be in the same final state (as in a sequential execution) when a session terminates. In other words the concurent execution is logically a sequential one. We approach the proof in two parts.

Part 1 *We show that write operations at any level occur in the same relative order as in a sequential execution.*

Consider a method (message manager), say nn, that is started (as a result of a fork or terminate) at a level l. We can then infer the following:

1. There exists no earlier forked message manager at level l still pending execution.
2. All current and future executions to the left of nn will be at a higher level than l.

From (2) we can see that all message managers to the left of nn but started after nn will write objects only at levels higher than or incomparable to l (as "write down" operations are not permitted). Thus from (1) and (2) we can conclude that when method nn is started at level l, no more write events at this or lower levels will be forthcoming from earlier forked methods. It follows that write events will always occur in the same relative order as in a sequential execution.

Part 2 *We show that read operations under concurrent execution obtain the same state as in a sequential execution.*

Consider again a method nn started at level l.

Case 1: Reads at level l
A read at level l would obtain the latest updates of objects at level l. This is consistent with the sequential execution.

Case 2: Reads at levels below l
Consider a second method mm at a level lower than l but which starts later than nn in a sequential execution. However, in a concurrent execution it is possible that mm may start before nn (even when mm is to the right of nn in the tree). We must now show that updates by mm are not read by nn. The proof follows from the fact that the values of timestamps assigned to nn on start up come from only these categories (as mentioned earlier).

- When obtained from the ancestors of nn the time stamps identify versions whose states correspond to the fork times of the ancestors. These are the same states as in a sequential execution where the issuers of forks are effectively suspended (until their children terminate). Since mm cannot be an ancestor of nn, the latter would not read updates by the former.

- In this category the time stamps are obtained from the latest versions created by terminated message managers to the left of nn. These latest versions reflect the same state as in a sequential execution. However, once again since mm is to the right of nn no time stamps are obtained by nn from mm.

- This last category of time stamps come from the global RStamp table maintained by the session manager. Once a session starts, the RStamp table is never updated. Hence the time stamps of versions written by mm are never reflected in the RStamp table and thus never obtained by a message manager such as nn.

From the above three categories of time stamps, we can conclude that a read by nn below its level l would always obtain the same state as in the corresponding sequential execution. This is because the timestamps obtained at every level identify versions whose states reflect the progress of computations in a sequential execution as of the time the stamps were obtained.

We have shown that writes at a level l occur in the same relative order as a sequential execution and that read operations obtain the same states of objects (again as in a sequential execution). It follows that the set of operations/updates received at the boundary of every object at level l would be the same set and in exactly the same sequence as in a sequential execution. The state transformations that occur at objects would leave them in the same final state as a sequential execution and thus preserve serial correctness.

5.2 Freedom from Timing Channels

To prove that our protocols are timing channel free, we observe that a message manager may be started only as a result of a fork or terminate request. We consider each case separately:

Case 1: Start due to fork

Consider a message manager *nn* forked at level *l*. Our algorithms ensure that *nn* can be denied immediate execution if and only if the following are true.

- Another node exists to the left of *nn* which is executing at level *l*.
- If the above condition is not true then there must exist another executing node to the left at level lower than *l*.

In both cases message manager *nn* is denied immediate execution by one or more message managers at equal or lower levels (and not by higher or incomparable levels). This cannot lead to a timing channel as the start of *nn* cannot be modulated by any message manager higher in level than *nn*.

Case 2: Start due to terminate

In this case we are guaranteed that when a message manager *nn* terminates at level *l*, any subsequent activations of message managers will be at level *l* or higher and to the right of *nn*. The proof of this follows from the fact that a message manager to the right of *nn*, say *m1* that is at a lower level, would have to be executing anyway. If *m1* is not executing then it must prevented from doing so only by an intermediate message manager *m2* to the left of *m1* and running at a level below *l*. In other words, *nn* cannot hold up the execution of *m1*. Thus in either case, the termination of *nn* cannot result in the start of another message manager at a level lower than *l*.

From the above we see that neither a fork or terminate request can cause a timing channel and this concludes our proof sketch.

6 Conclusion

In this paper we have provided an execution model that manages concurrent computations in a secure manner, free from timing channels. We also presented a multiversion synchronization scheme that guarantees the serial correctness of the concurrent computations. Although our focus in this paper has been on closing timing channels, we are currently investigating other issues related to the implementation of the message filter model. In particular we are looking into techniques for providing synchronization and concurrency control across multiple user sessions. We are currently investigating the suitability of an extended checkin/checkout model of transaction processing. We are also looking into an exception and error model that will deal with errors in the concurrent computations in a secure and covert channel free manner.

Acknowledgment

We are indebted to 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] 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.

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

[3] 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.

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

[5] 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.

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

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

[8] R.S. Sandhu, R. Thomas, and S. Jajodia. A Secure Kernelized Architecture for Multilevel Object-Oriented Databases. *Proc. of the IEEE Computer Security Foundations Workshop IV*, pp. 139-152, June 1991.

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