Proc. of the IFIP WG11.3 Workshop on Database Security, Vancouver, Canada, August 19-21, 1992

Implementing the Message Filter Object-Oriented Security Model without Trusted Subjects

Roshan K. Thomas and Ravi S. Sandhu[1]

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

Abstract
We propose a new architectural framework and implementation scheme, for the message filter multilevel security model for object-oriented databases. Major complications in implementing the model arise from the intrinsic nature of object-oriented computations which are abstract and often involve arbitrarily complex write-up actions. Dealing with the timing of write-up operations has broad implications on security (due to the potential for signaling channels), integrity, and performance. A fundamental insight, gained in the course of our research, has been to close these channels by allowing concurrent computations in what is otherwise a logically sequential computation. However in closing these channels one has to meet the conflicting goals of integrity and performance. Our earlier work investigated an architecture that called for a trusted subject (session manager) to manage a tree of concurrent multilevel computations generated by a user session. In this paper we provide an alternate achitecture that eliminates the need for trusted subjects and the associated central coordination and management of concurrent computations. This revised architecture is a kernelized one as no subject is exempted from the simple-security and $\star$ properties. Hence security comes for free while we continue to meet the additional conflicting requirements for integrity and performance.

Keyword Codes: D.1.5; D.4.6; K.6.5
Keywords: Object-oriented Programming, Security and Protection

[1]The work of both authors was partially supported by the National Security Agency through contract MDA904-92–C-5140. We are indebted to Howard Stainer and Mike Ware for making this work possible.

1 INTRODUCTION

A message filter approach to integrating mandatory security in multilevel object-oriented databases was originally proposed in [3]. The main elements of the model are objects and messages. Security is enforced by a message filter component that controls information flow by mediating message exchanges. The original message filter specification is a step in the right direction in modeling and integrating security in a way natural to the object-oriented paradigm. However, it gives no clue as to how such a specification model can be implemented. This has led the authors to investigate implementation aspects of the message filter model [8, 9].

Although the message filtering actions ensure that mandatory access controls cannot be bypassed, they open up the potential for timing channels. In fact, these channels arise due to the abstract nature of write-up computations in the object-oriented model. When a message is sent upwards in the security lattice, the resulting computation (method) in a higher level object may take an arbitrary amount of processing time before returning a reply. Although the actual reply (in terms of its contents) cannot be returned to the lower level, the timing of a substituted innocuous reply such as a NIL, can be exploited for timing channels. A fundamental insight, gained in the course of investigating implementation issues for the message filter model, has been to close such channels by executing an otherwise logically sequential computation, concurrently. In other words, a computation in a sender object is no longer blocked waiting for a reply, but rather proceeds concurrently with the corresponding computation invoked in a receiver object.

In our further discussions, we deliberately use the term *(downward) signaling channel* rather than covert channel. A downward signaling channel is a means of downward information flow which is inherent in the data model and will therefore occur in *every* implementation of the model. A covert channel on the other hand is a property of a specific implementation and not a property of the data model. In other words, even if the data model is free of downward signaling channels, a specific implementation may well contain covert channels due to implementation quirks.

An architecture that was investigated earlier in [8] called for a trusted subject (session manager) to manage and coordinate the concurrent computations initiated by a user session. The session manager has to be trusted so that it can deal with multilevel computations. In this paper we give an alternate architectural and implementation framework that eliminates the need for such trusted subjects. It turns out however, that concurrent computations are still needed. This is because to execute computations sequentially, we need to observe the termination times of higher level receiver computations and return replies in order to resume lower level blocked computations. However only a multi-level trusted subject that is exempted from the $\star$ property can do this.

Thus in an architecture without trusted subjects, the only feasible implementation scheme is to return replies independent of the termination of receiver computations. This can be accomplished by returning the NIL reply either after some constant time interval that represents an upper bound for completion times, or after some random delay, or instantaneously when messages are sent. The first option incurs a heavy performance penalty whenever computations wait unnecessarily. The second option may incur the same performance penalty as the first, and further require synchronization of concurrent computations. The third option while requiring synchronization for concurrent computations, eliminates the need for sender computations to unnecessarily wait for time quantums to expire.

In the revised architectural framework, the management and coordination of concurrent computations is no longer centralized, but rather achieved in a distributed (and secure) fashion. This is because no system component has a global view of the concurrent computations as they progress. The new framework offers obvious advantages. First, it eliminates the need for operating system support for trusted subjects, and thus security comes for free. This clearly makes security arguments easier for our implementation. Secondly, the implementation allows us to meet the conflicting goals of integrity and performance (without compromising security in the process).

The rest of this paper is organized as follows: Section 2 gives some background to the message filter model and its evolution. Section 3 presents a reworked architecture without trusted subjects, and further describes how concurrent computations can be coordinated in a distributed fashion. Section 4 discusses some informal proofs and section 5 concludes the paper.

2 BACKGROUND TO THE MESSAGE FILTER MODEL

In this section we give some background to the message filter model and the original implementation of the model with trusted subjects. Our presentation is limited to those aspects relevant to the understanding of the results in this paper. For more comprehensive details on the motivation and evolution of the model the reader should see [3, 8, 9].

2.1 The Message Filter Specification

Objects and messages constitute the main entities in the message filter model. Messages are assumed, and required to be, the only means by which objects can communicate and exchange information. Thus the core 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). We emphasize that a reply (NIL or other) must always be returned to prevent the sender of a message from blocking indefinitely.

Figure 1 illustrates the message filtering graphically. The full algorithmic specification is given in figure 2 (in this and other algorithms, % is a delimeter for comments). In case (1), the sender and receiver are at the same security level and the message g_1 and its reply

Figure 1: Illustrating message filtering

are allowed to pass. In case (2) the levels are incomparable and thus the filter blocks the message from getting to the receiver object and further injects a NIL reply. Case (3) involves a receiver at a higher level than the sender. The message is allowed to pass but the filter discards the actual reply and substitutes a NIL instead. In case (4) the receiver object is at a lower level than the sender and the filter allows both the message and the reply to pass unaltered.

In cases (1), (3), and (4) the method in the receiver object is invoked at a security level given by the variable *rlevel*. The *rlevel* needs to be computed for each receiver method invocation and it is in turn derived from the *rlevel* of the method invocation in the corresponding sender object. The intuitive significance of *rlevel* is that it keeps track of the least upper bound of all objects encountered in a chain of method invocations, going back to the root of the chain. This is required to implement the notion of restricted method invocations so as to prevent write-dowm violations. To be more precise, we say that a method invocation t_i has a *restricted status* if $rlevel(t_i) > \mathrm{L}(o_i)$. The application of restricted invocations is explained below.

The cases (1) through (4) that we have seen so far deal with abstract messages. However abstract messages will eventually result in the invocation of primitive messages. These include READ, WRITE and CREATE [2]. READ operations always succeed while WRITE and CREATE succeed only if the status of the method invoking the operation is unrestricted. Thus if a message is sent to a receiver object at a lower level (as in case (4)), the resulting method invocation will always be restricted and the corresponding primitive WRITE operation will not succeed. This will ensure that a write-down violation will not

[2]The DELETE operation has not been directly incorporated into the model. It can be viewed as a particularly drastic form of WRITE.

```
% let $g_1 = (h_1, (p_1, \ldots, p_k), r)$ be the message sent from $o_1$ to $o_2$
% let $h_1$ be the message name, $p_1, \ldots, p_k$ be the parameters in the message, $r$ the return value
if $o_1 \neq o_2 \vee h_1 \notin$ {READ, WRITE, CREATE} then case
% i.e., $g_1$ is a non-primitive message
(1)  $L(o_1) = L(o_2)$ :    % let $g_1$ pass, let reply pass
                            invoke $t_2$ with $rlevel(t_2) \leftarrow rlevel(t_1)$;
                            $r \leftarrow$ reply from $t_2$; return $r$ to $t_1$;
(2)  $L(o_1) <> L(o_2)$ :   % block $g_1$, inject NIL reply
                            $r \leftarrow$ NIL; return $r$ to $t_1$;
(3)  $L(o_1) < L(o_2)$ :    % let $g_1$ pass, inject NIL reply, ignore actual reply
                            $r \leftarrow$ NIL; return $r$ to $t_1$;
                            invoke $t_2$ with $rlevel(t_2) \leftarrow \mathrm{lub}[L(o_2), rlevel(t_1)]$;
                            % where lub denotes least upper bound
                            discard reply from $t_2$;
(4)  $L(o_1) > L(o_2)$ :    % let $g_1$ pass, let reply pass
                            invoke $t_2$ with $rlevel(t_2) \leftarrow rlevel(t_1)$;
                            $r \leftarrow$ reply from $t_2$; return $r$ to $t_1$;
end case;
if $o_1 = o_2 \wedge h_1 \in$ {READ, WRITE, CREATE} then case
% i.e., $g_1$ is a primitive message
% let $v_i$ be the value that is to be bound to attribute $a_i$
(5)  $g_1 = (\mathrm{READ}, (a_j), r)$ : % allow unconditionally
                            $r \leftarrow$ value of $a_j$; return $r$ to $t_1$;
(6)  $g_1 = (\mathrm{WRITE}, (a_j, v_j), r)$ : % allow if status of $t_1$ is unrestricted
                            if $rlevel(t_1) = L(o_1)$
                              then [$a_j \leftarrow v_j$; $r \leftarrow$ SUCCESS]
                              else $r \leftarrow$ FAILURE;
                            return $r$ to $t_1$;
(7)  $g_1 = (\mathrm{CREATE}, (v_1, \ldots, v_k, S_j), r)$ : % allow if status of $t_1$ is unrestricted relative to $S_j$
                            if $rlevel(t_1) \leq S_j$
                              then [CREATE $i$ with values $v_1, \ldots, v_k$ and $L(i) \leftarrow S_j$; $r \leftarrow i$]
                              else $r \leftarrow$ FAILURE;
                            return $r$ to $t_1$;
end case;
```

Figure 2: Message filtering algorithm

occur. Finally, the CREATE operation allows the creation of a new object at or above the *rlevel* of the method invoking the CREATE. The creation of objects lower than *rlevel* is again prevented by restricted invocations.

2.2 Implementation with Trusted Subjects

In our earlier work, we have presented the complications that arise due to downward signaling channels in object-oriented computations [8, 9]. Let us review these briefly. Whenever messages are sent to objects at higher levels, the receiver method should not be able to modulate the timing of the NIL reply. Hence we have no choice but to return the NIL reply immediately, resume execution of the suspended sender, and further execute the receiver object's method concurrently.

Thus the message filter specification calls for an underlying asynchronous implementation/execution model. This could lead to a tree of concurrent computations (methods) as shown in figure 3. Each computation is executed by a separate *message manager* process that implements the message filtering function (in our discussions we often use the terms computations, methods, and message managers interchangeably). Such a tree represents computations forked by a single user (at a single security level) within a single user session. However, each message manager may be executing at a different security level and we thus have a single user but a multilevel tree of computations.

A key feature of an architecture investigated earlier (see figure 5) was the use of a *session manager* process to act as a trusted subject in order to manage and coordinate such a tree. A session manager has to be a trusted subject as it is dealing with computations at different security levels and thus needs to bypass the usual mandatory access controls (particularly the $\star$ property) in a Bell-LaPadula framework. A session manager always maintains a global snapshot of the entire tree of computations as it progresses.

Although conceptually a message sent to a higher level object results in the immediate fork of a new concurrent message manager, the session manager limits the actual degree of concurrency by scheduling computations in a secure and correct manner. Figure 4 illustrates the overall strategy used by the session manager in scheduling these concurrent computations. It utilizes the following invariant 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. In other words, some forked computations may be temporarily queued for execution.

The derivation of this invariant is actually motivated by the dual requirements of correctness and security. To see this, we observe that if security were our only objective, we could allow maximum concurrency by enabling computations to unconditionally proceed. However, ensuring correctness (equivalence to the intended logically sequential execution) would then be difficult, if not impossible. Thus the "only if" part of the above invariant is

Figure 3: A tree of concurrent message managers

Figure 4: Progressive execution of figure 3

Figure 5: A kernelized architecture with trusted subjects

required for correctness. To do this we have to ensure that all writes performed by earlier forked computations at or below the level of a computation say n, are made visible to n (in accordance with sequential precedence). Thus by the time n starts, all these earlier forked computations should have terminated.

The "if" part of the invariant is an artifact of our algorithm and intuitively maximizes the degree of concurrency (as computations are not unnecessarily help up). In fact, we conjecture that there are many algorithms allowing varying degrees of concurrency.

We illustrate one such algorithm that allows the least concurrency (but guarantees correctness). The basic idea is to follow a level by level scheduling strategy. Thus given a finite set of actions at multiple levels, we first schedule and execute (to completion) all the lowest level actions in the security lattice (one at a time, of course). This is followed by actions at the next higher levels and we continue in this fashion until those at the highest level in the lattice are scheduled last. It follows that whenever there are actions at incomparable levels, they will be executed concurrently (to avoid a sideways signaling channel). For example, given the security lattice in figure 6 we would first schedule and execute all the actions of message managers running at the lowest level unclassified. Upon completion, we would then execute concurrently the actions at the incomparable levels (S{A}) and (S{B}). Finally when all actions at both these levels have completed, those at the highest level (TS{A,B}) are scheduled. Our focus in this paper will be on implementing this simpler level by level scheduling strategy without the use of session managers as trusted subjects.

Now back to our original invariant. The progressive execution of the tree in figure

Figure 6: Level by level scheduling in a simple lattice

3 as governed by this "if and only if" invariant is shown in figure 4. 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. However message manager 3 is queued up. Our invariant guarantees that message manager 3 remains queued (suspended) at least till such time as message manager 2 (and any future children at level top secret) terminate. This action is necessary so that the writes by message manager 2 and its children are made visible to the top secret message manager 3, due to sequential precedence. Message manager 4 at level confidential is allowed to execute immediately on being forked, since all active as well as future message managers to the left of it will be at levels higher than confidential. We also notice that the termination of message manager 2 results in the execution of message managers 3 and 6. In essence, our invariant guarantees that the execution of a lower level message manager is never delayed due to an earlier forked and executing message manager at a higher level.

2.3 Serial Correctness

As mentioned before, one of the key issues to be tackled with these concurrent computations is related to providing synchronization and ensuring *serial correctness*. We have to ensure that the concurrent execution guarantees the same result (object states) as in the original logically executed sequential (single) computation. A multiversioning scheme for this purpose was presented in [9]. The scheme guarantees that high level methods would read down object states at lower levels that were equivalent to the one in the sequential execution. To enable this, versions identifying lower object states that existed at the time a (higher) message manager was forked are maintained. When the forked high level message manager becomes active, it has the necessary timestamps (in a **local-stamp** table) identifying all versions of objects at lower levels that it will need to process read-down requests. These entries are never modified after a message manager starts. A write stamp

(WStamp) at the level of the message manager identifies the next version that will be written at its level. On start, a message manager increments this timestamp entry unconditionally before the first write/update operation and subsequently increments it after every fork request made to the session manager.

3 IMPLEMENTATION WITHOUT TRUSTED SUBJECTS

Having given a background to the trusted subject architecture and implementation framework, we now address the issue of implementing the message filter model without trusted subjects. Thus we can no longer rely on the central control and coordination (of the concurrent computations generated by a user session) that was provided by a session manager. Rather, these computations would have to be managed in a distributed fashion. This also follows from the fact that no system component would at any time ever have a global snapshot of the set of computations in progress. In light of this, is distributed management possible? We assert (and later demonstrate) the feasibility of this based on the following observations:

- The decision to start or queue a computation can only be affected by other computations at a lower level. In other words, one only needs to look down to determine this and thus will not violate mandatory security.
- The termination of a computation (message manager) can result in the subsequent start-up of other computations only at higher levels. This can be accomplished by sending messages upwards and without violating mandatory security or introducing downward signaling channels.

3.1 The Revised Architecture

Figure 7 illustrates the reworked architecture without session managers acting as trusted multilevel subjects. In comparison to the trusted subject architecture in figure 5, a multilevel session manager is now replaced by single-level *level managers*. A level manager is responsible for scheduling and coordinating all computations forked at its level. Message managers are short-lived as they are created and terminated dynamically as the need arises with fork requests. On the other hand, this architecture calls for the existence of a long-lived level manager at every security level for which a computation can be potentially forked.

As can be seen in figure 7, this architecture is a layered one and consists of a storage and an object layer. The security perimeter of the object layer consists of the following primitive operations: SEND, QUIT, READ, WRITE, and CREATE. The READ, WRITE, and CREATE are related to to the primitive messages discussed earlier. The SEND and QUIT are system primitives used by message managers (methods) to send (non-primitive) messages and replies. The message manager algorithms for these are shown in figure 8. In

Figure 7: A kernelized architecture without trusted subjects

our original architecture with trusted subjects, the interface between a message manager and its session manager consisted 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. In our revised architecture without trusted subjects, these calls form the interface between a message manager and its local level manager.

3.2 Achieving Distributed Coordination

We now address the issue of coordinating a tree of concurrent computations to enable overall progress. We begin by describing a few data structures.

Every message manager maintains the following information:

Local-stamp:	a vector of timestamps to process read down requests;
Fork-stamp:	a stamp identifying the message manager's fork order;
WStamp:	the write stamp for versions written by the message manager;

Every level manager maintains the following data structure:

Current-WStamp:	the current timestamp given to objects written;
Queue:	a queue of message managers waiting to be activated;
Fork-history:	a list of ordered pairs (fork-stamp, WStamp);

```
procedure SEND($g_1, o_1, o_2$)
% let $g_1 = (h_1, (p_1, \ldots, p_k), r)$ be the message sent from $o_1$ to $o_2$ where
% where $h_1$ is the message name, $p_1, \ldots, p_k$ are message parameters, and $r$ is the return value
% let $p$ be the parameter set $p_1, \ldots, p_k$ and let lmsgmgr be the level of the message manager
if $o_1 \neq o_2 \vee h_1 \notin$ {READ, WRITE, CREATE} then case % i.e., $g_1$ is non-primitive
(1)  $L(o_1) = L(o_2)$ :   push-stack($p$); $t_2 \leftarrow$ select method for $o_2$ based on $h_1$; execute $t_2$;
(2)  $L(o_1) \sim L(o_2)$ :   write-stack(NIL); resume;
(3)  $L(o_1) < L(o_2)$ :   append-local-stamp-vector(rstamps, WStamp);
                           FORK(lmsgmgr, lub[lmsgmgr, $L(o_2)$], fork-stamp, rstamps);
                           WStamp $\leftarrow$ WStamp + 1; write-stack(NIL); resume;
(4)  $L(o_1) > L(o_2)$ :   push-stack($p$); $t_2 \leftarrow$ select method for $o_2$ based on $g_1$; execute $t_2$;
end case;
if $o_1 = o_2 \wedge h_1 \in$ {READ, WRITE, CREATE} then case % i.e., $g_1$ is a primitive message
(5)  $h_1$ = READ :        if $L(o_1)$ = lmsgmgr then v $\leftarrow$ WStamp else v $\leftarrow$ local-stamp ($L(o_1)$);
                           read $o_1$ with max{version: version $\leq$ v};
(6)  $h_1$ = WRITE :       write $o_1$ with version $\leftarrow$ WStamp;
% Let $o$ be the object-identifier of the new object created at level $S_j$
(7)  $h_1$ = CREATE : create $o$ with $L(o) \leftarrow S_j$ and version $\leftarrow$WStamp; write-stack($o$);
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 8: Message manager algorithms for SEND and QUIT

Our focus in the remaining sections of the paper will be on algorithms to achieve the simpler level-by-level scheduling strategy. Although this scheduling approach is not optimal in terms of the degree of concurrency allowed, it gives valuable insights into how a centralized coordination task can be carried out in a distributed, correct, and secure manner in the multilevel context.

3.2.1 Maintaining Global Serial Order

As mentioned earlier, one of the difficulties in achieving distributed coordination can be attributed to the lack of a global view of the computations as they progress. In particular, without a global data structure such as a tree we would not know the relative order in which message managers are forked (in a sequential execution) by a user session.

Figure 9: Hierachical generation of fork-stamps

Knowledge of such ordering is crucial in maintaining serial correctness.

To elaborate on the above, consider the tree in figure 9. With a level-by level scheduling strategy, the fork of message manager 10(TS) by 1(U) will be queued up at the top secret level manager before the fork of 6(TS) by 2(C) (as 2(C) will not be started until 1(U) terminates). However 10(TS) should be dequeued and executed only after the termination of 8(TS) and message manager 8(TS) in turn should be executed only after the termination of 6(TS). This is required to guarantee correctness as the updates of 10(TS) at level top secret should not be visible to either 6(TS) or 8(TS) as they are both to the left of it. On the other hand, we want to make the updates of 6(TS) visible to 8(TS) and the updates of both message managers 6(TS) and 8(TS) in turn visible to 10(TS). Thus there is a need to capture the information that 10(TS) is to the right of 8(TS) and 8(TS) is itself to the right of 6(TS). One could be tempted to obtain the above ordering information by reading off a system low real-time clock, every time a message manager is forked. However, as shown above, forks are not always generated (in real time) in the order consistent with a sequential execution and thus a solution with real-time clocks will not work.

Our proposal here is to derive such an ordering from a hierarchical scheme to generate fork-stamps. Every message manager on being forked is assigned a unique fork-stamp by the parent issuing the fork. The actual fork-stamps generated for the tree in the above example is also shown in figure 9. A set of four digits are used for this example with fork requests at levels U, C, S, TS, and TTS. The root message manager 1(U) at level unclassified (U) is given an initial fork-stamp of 0000. It is then required to increment the most significant (leftmost) digit for every fork request issued. Thus 1(U) assigns the fork-stamps 1000, 2000, and 3000 to its children 2(C), 7(S), and 10(TS) respectively. Now a message manager at confidential such as 2(C) is required to increment the second most significant digit of its fork-stamp for every subsequent child it forks. In other words, with increasing levels, a less significant digit is incremented. Hence the top secret message manager in our example will be required to increment the least significant (rightmost) digit (as shown by the fork-stamps assigned to the children of 10(TS)). Thus a message manager in the subtree rooted at 2(C) will always have smaller fork-stamp than one in the subtree rooted at either 7(S) or 10(TS).

In light of our earlier discussion of this example, we now see that 6(TS) will indeed have a lower fork-stamp than 8(TS), and 8(TS) in turn will have one lower than 10(TS). We are thus able to implicitly capture the fork order in these fork-stamps. Finally, we note that an appropriate number of digits can be allocated for generating fork-stamps so as to account for the maximum degree of a node as well as the maximum depth of a tree of computations.

3.2.2 The Processing of FORK and TERMINATE requests

A tree of message managers (computations) advances to completion through the generation of FORK and TERMINATE events. Every FORK results in the creation of a new message manager and every TERMINATE could release one or more message managers for execution and subsequent completion. We now discuss the algorithms to coordinate FORK AND TERMINATE events. In our exposition, we will highlight how schemes to guarantee serial correctness are incorporated in these algorithms through the use of multi-versioning techniques.

The algorithm to process FORK requests is shown in figure 10. On receiving a FORK request from a lower level, a level manager creates a new message manager process (computation) and records the fork-stamp (passed on by the parent issuing the FORK) in the message manager's data structure. Now comes the task of initializing the message manager's **local-stamp** table entries. The timestamps for these entries are actually acquired in two phases. The first phase is at FORK time and the second phase is deferred until the message manager starts.

Consider for the moment the first phase. During this phase, a message manager's **local-stamp** entries are initialized for all the levels of its ancestors on the path from the root to itself. The timestamps for these entries are obtained from a vector of timestamps passed on by the parent issuing the FORK request. In fact, every message manager is required to save the timestamps in the vector (rstamps) it receives from its parent and on issuing a FORK, to reconstruct a new vector to give to its child. This newly constructed vector will contain the timestamps from the old vector appended with the write stamp (WStamp) at the level of the issuing message manager (which identifies the latest versions written before the FORK was issued). Obviously, the number of entries in such a vector that is incrementally constructed increases with the number of direct ancestors and the number of security levels (i.e., with the depth of the computation tree).

To see why the timestamps acquired in the first phase preserve serial correctness, consider the path from 1(U) to 5(TS) in the tree in figure 4 (a). In a sequential execution,

```
**Procedure FORK**(*level-parent, level-create, fork-stamp, rstamps*)
{
%*Let level-create be the level of the local message manager*
Create a new message manager *mm* at level *level-create*;

%*Record the fork-stamp passed on by the parent*
mm.fork-stamp ← fork-stamp

%*Begin phase 1 of acquiring local-stamp entries*
For (every level $l \leq$ level-parent)
do
    initialize mm.local-stamp table entries from rstamps;
End-For

%*This is a priority queue maintained in fork-stamp order*
enqueue(mm);
}
end procedure FORK;
```

Figure 10: Level manager algorithm for FORK processing

```
Procedure TERMINATE(lmsgmgr, WStamp)
{
%Let tt be the message manager that just terminated at level lmsgmgr
%Let lm be the level manager at level lmsgmgr

%Update local current write stamp from tt
lm.current-wstamp ← tt.WStamp

%Update local fork-history with the fork-stamp and WStamp of tt
Append-fork-history(fork-stamp, WStamp);

If queue is not empty
then
    dequeue(queue, mm);
    start(mm);
Else
    Send a WAKE-UP message to all immediate higher level managers;
End-If
}
end procedure TERMINATE;
```

Figure 11: Level manager algorithm for TERMINATE processing

when 5(TS) is forked the ancestor message managers 4(C) and 1(U) will be blocked (waiting to resume execution). To be more precise, when 4(C) was forked its parent 1(U) was blocked and when 5(TS) was forked 4(C) in turn was blocked. Hence the versions that will be read by 4(C) at the lower level unclassified will be the ones that existed at the time 1(U) was blocked. In a similar fashion, the versions read by 5(TS) will be those that existed at the time 4(C) was blocked. Also, the versions read by 5(TS) at level unclassified will be the same as those read by 4(C) since 1(U) remains blocked until both 5(TS) and 4(C) terminate. Now the timestamps identifying these versions at levels unclassified and confidential are precisely those passed along to 5(TS) when it was forked. Thus equivalence (in read down operations) to a logically sequential execution is achieved. Now for an intermediate level such as secret which is not the level of any of the direct ancestors of 5(TS), the initialization of the secret **local-stamp** entry would have to be delayed until 5(TS) is actually started (for execution). This is thus accomplished only in phase 2. The timestamp for such an entry will identify the latest version written by the last forked message manager at level secret (if any), that is to the left of 5(TS) in the computation tree. If no message manager was ever forked for the user session at secret, the timestamp for the initial version that existed at the start of the session is used.

Upon completing the first phase of initializing the **local-stamp** entries, the level manager proceeds to queue up the newly created message manager in its local queue. It is important to note that with our level-by-level scheduling strategy, we unconditionally queue up fork requests. This differs from the strategy governed by the "if and only if" invariant presented earlier, where a forked message manager may be immediately started under certain circumstances (thus allowing more concurrency).

The processing of TERMINATE requests is shown in figure 11. When a message manager terminates, the write stamp (WStamp) identifying the latest versions written at its level is recorded by the local level manager. Next the level manager's **fork-history** data structure is appended with the ordered pair (fork-stamp, WStamp). This captures the fact that a certain message manager was forked in the order given by fork-stamp, and terminated writing versions with timestamp WStamp. Such a history is needed to implement the multiversioning scheme and to guarantee serial correctness. Finally, a level manager dequeues and starts the next message manager (if any) from the local queue. If the queue is found to be empty, a WAKE-UP message is sent to all immediately higher level managers in the security lattice. The receipt of this WAKE-UP message could potentially initiate the execution of queued up message managers at these levels. The processing of these WAKE-UP messages is described in the next subsection.

3.2.3 WAKE-UP Messages and Level by Level Activation

We now describe the semantics of processing WAKE-UP messages and how this achieves the level-by-level activation of computations (see figure 12). When a WAKE-UP message is received, a level manager has to determine if it can release for execution, computations queued up at its level. It can do this only if all activity at all lower levels in the security lattice have ceased. A message manager can be certain of this only when it has received a WAKE-UP message from all immediate lower levels in the security lattice. In other

```
Procedure WAKE-UP
{
%Proceed if the necessary condition has been met
If a WAKE-UP message has been received from all lower levels
then
    If the queue is not empty
    then
      DEQUEUE(queue, nn);
      START(nn);
    else
      Send a WAKE-UP message to all immediate higher levels;
    End-If
End-If
}
end procedure WAKE-UP;
```

Figure 12: Level manager algorithm for processing WAKE-UP messages

```
Procedure START(nn)
{
%Let nn represent the message manager to be started
%Let lm represent the level manager managing nn

%Complete phase 2 of acquiring local-stamp entries
For (every level l lower than the level of nn for which no timestamp has
been obtained so far)
do
    nn.local-stamp[l] ← mm.fork-stamp;
    where mm is the message manager entry in the fork-history at level l
    with max{fork-stamp: fork-stamp < nn.fork-stamp}
End-For

%Update the write stamp (WStamp) from the level manager
WStamp ← lm.Current-WStamp + 1;

%Begin execution of the message manager nn
execute(nn);
}
end procedure START;
```

Figure 13: Level manager algorithm for START

Figure 14: A lattice with WAKE-UP forwarding through empty levels

words, this is a necessary (and sufficient) condition for releasing computations at any level. Once this condition is met, a level manager sorts its queue of pending message managers by ascending order of fork-stamps. This is necessary to ensure that message managers are activated in the same order as in a logically sequential execution. After the sort is finished, the message manager at the head of the queue is dequeued and started. Subsequent termination of this and other message managers will cause the queue to be emptied in due time.

If on receiving a WAKE-UP message from all immediate lower levels, a level manager finds its queue to be empty, it simply propagates (or feeds forward) a WAKE-UP message to all immediate higher levels in the lattice (see figure 14 for an illustration). It can do this because it is certain that the queue will remain empty for the rest of the duration of the user session as no more FORK requests will be forthcoming from lower levels.

As illustrated so far, TERMINATE and WAKE-UP requests potentially result in the release and start of queued up message managers (computations). Once dequeued, a common START procedure (see figure 13) is used to complete the second-phase (alluded to earlier) of the task of initializing the **local-stamp** entries. Now for all lower levels for which no entries were obtained at fork time, the level manager examines the fork histories. The level manager does this to determine the versions written by the last forked computations that terminated before the fork of the message manager that is to be started. This can be accomplished by comparing the fork-stamps at lower levels to the fork-stamp of the message manager to be started. At each level, the largest such stamp that is less than the stamp of the message manager to be started is picked, and the associated version/timestamp is read. Once the second phase is completed, the level manager provides the **Current-WStamp** value at its level incremented by one, to the message manager. This will enable the just started message manager to write the correct versions of objects at its level. It is important to note the need for incrementing this value by one, for otherwise older versions will initially be overwritten (and this would violate serial correctness).

4 DISCUSSION

Having discussed a level-by-level scheduling strategy, we now briefly and informally argue proofs of correctness, termination, and security. Introducing formal machinery to do this is beyond the scope of this paper and unnecessary as the arguments are simple and straightforward.

- **Correctness.** As in the proof sketches given in [9] we argue serial correctness by showing that the versions read (down) by a message manager are the same as in a sequential execution, and that write operations at its level occur in the same relative order. In phase 1 of the protocol to obtain these timestamps, we have seen how these timestamps identify versions written by blocked ancestors. Since in a sequential execution the ancestors are always blocked due to a running child message manager, the equivalence of these versions follow. In phase 2, forkstamps are used to identify the latest versions written by earlier forked terminated message managers (that are not direct ancestors) at lower levels. Again equivalence follows from the fact that in a sequential execution all lower level message managers that are not direct ancestors of a starting message manager would have terminated. Finally, at every level the message managers are executed in ascending fork-stamp order. Thus the relative order of write operations would be the same in a history generated by our level-by-level scheduling strategy when compared to a second history generated by the logically equivalent sequential execution.

- **Termination.** The proof that with a level-by-level scheduling and execution strategy the entire set of computations will eventually terminate, can be argued from the following: (1) Once a message manager starts, it runs uninterrupted to completion (although the forks it issues may be accumulated for later scheduling). Thus the time needed to empty the queue at any level is bounded; (2) A WAKE-UP message is sent only when the local queue at a level is empty and hence the receipt of a WAKE-UP message is a guarantee that all the computations at the lower sender's level have terminated; (3) There exists no cyclical wait-for relations for WAKE-UP messages among level managers in a security lattice.

- **Security.** This follows from the fact that all subjects are indeed single-level and mandatory access control is never bypassed in our architecture. Thus the potential for a multi-level trusted subject to open up signaling chanels is also eliminated.

As mentioned before, we have opted to present the simpler level-by-level scheduling strategy in this paper. But, what does it take to implement the more optimal scheduling strategy governed by the "if and only if" invariant? A FORK request may now immediately result in the start of a new computation. The major complication arises in determining when to release computations at a higher level. When a computation terminates at a level, say l, we may have to send WAKE-UP messages to higher levels even when there are pending forked computations at l (to allow maximum concurrency). We are currently developing the algorithms formally to achieve this within the architectural framework of single level message managers coordinated by single-level level managers.

5 CONCLUSION

In this paper we have reworked a kernelized architecture for implementing the message filter model so as to eliminate the need for trusted subjects. The centralized management of computations now had to be done in a distributed fashion. The new architecture is in line with the true spirit of kernelized approaches to providing security, and would make the message filter model more acceptable to commercial implementation efforts. Having laid the above groundwork, we will be looking into issues involved in supporting multiple user sessions. In particular, we will be investigating the impact of concurrent computations from multiple users on concurrency control and transaction management schemes. Addressing and solving these issues would be critical to the evolution of the message filter model as a full fledged solution for multilevel secure object-oriented databases.

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] R.S. Sandhu, R. Thomas, and S. Jajodia. Supporting timing-channel free computations in multilevel secure object-oriented databases. *Proc. of the IFIP 11.3 Workshop on Database Security,* Sheperdstown, West Virginia, November 1991.

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