A Trusted Subject Architecture for Multilevel Secure Object-Oriented Databases

Roshan K. Thomas and Ravi S. Sandhu

Abstract—In this paper, we address security in object-oriented database systems for multilevel secure environments. Such an environment consists of users cleared to various security levels, accessing information labeled with varying classifications. Our purpose is three-fold. First, we show how security can be naturally incorporated into the object model of computing so as to form a foundation for building multilevel secure object-oriented database management systems. Next, we show how such an abstract security model can be realized under a cost-effective, viable, and popular security architecture. Finally, we give security arguments based on trusted subjects and a formal proof to demonstrate the confidentiality of our architecture and approach.

A notable feature of our solution is the support for secure synchronous write-up operations. This is useful when low level users want to send information to higher level users. In the object-oriented context, this is naturally modeled and efficiently accomplished through write-up messages sent by low level subjects. However, such write-up messages can pose confidentiality leaks (through timing and signaling channels) if the timing of the receipt and processing of the messages is observable to lower level senders. Such covert channels are a formidable obstacle in building high-assurance secure systems. Further, solutions to problems such as these have been known to involve various tradeoffs between confidentiality, integrity, and performance. We present a concurrent computation model that closes such channels while preserving the conflicting goals of confidentiality, integrity, and performance. Finally, we give a confidentiality proof for a trusted subject architecture and implementation and demonstrate that the trusted subject (process) cannot leak information in violation of multilevel security.

Index Terms—Multilevel security, secure write-up, object-oriented databases, trusted subject architecture, covert channels, confidentiality proof.

1 Introduction

Over the past decade, object-oriented computing has become a very active field for research and development. We attribute this partly to the broad applicability of the object-oriented paradigm itself. It is thus perhaps inevitable that the field of information security also looks to this paradigm for fresh ideas and influences. Consider for example, the ability of the paradigm to model as objects the structure and behavior of real-world entities in an application domain. This makes it easier to specify, interpret, and implement security requirements and policies in terms of objects rather than more primitive computer-oriented abstractions and representations. Thus we have lately witnessed several research and development proposals for secure object-oriented databases—efforts that attempt to incorporate security features in object-oriented data models.

In this paper, we turn our attention to multilevel secure (mls) object-oriented database management systems [12], [13], [15], [21], [28]. A multilevel secure environment consists of users cleared to various security levels accessing information labeled with varying classifications. The clearances and classifications typically form a lattice structure with the high security levels lying towards the top of the lattice [23]. From a confidentiality viewpoint, information is allowed to flow only upwards in the lattice. An access control or security policy ensures this by governing the accesses to the information by the different users. The policy itself is enforced through appropriate mechanisms. In the object-oriented context, one way to accomplish this is to control the exchange of messages between objects at various levels.

The purpose of our discussions here is three-fold. First we wish to give a gentle introduction to multilevel security, multilevel architectures, and security in object-oriented systems. We present a message-based security model that can be used as a foundation in building secure object-oriented systems. Our second objective is to show how such an abstract security model can be realized using a viable and commercial architecture. Lastly we give security arguments and a formal proof to demonstrate the security of this architecture. This gives us the assurance that the architecture cannot be exploited to leak information in violation of the security policy.

A key feature of our solutions is the support for write-up operations. Write-up operations are initiated when low level information needs to be sent to higher levels. They are particularly useful in applications that call for active and reactive capabilities such as when events would have to be monitored and actions triggered when certain conditions become true. In secure active databases, the option of sending low level information to higher levels by read-down operations is not always feasible or efficient. A high level subject would have to continually keep polling the

- *R.K. Thomas is with Odyssey Research Associates, 301 Dates Dr., Ithaca, NY 14850. E-mail: rthomas@oracorp.com.*
- *R.S. Sandhu is with the Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030. E-mail: sandhu@isse.gmu.edu.*

Manuscript received Feb. 1, 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number K96003.

lower levels for updates of low level information. The polling window is often not easy to compute. As such, write-up remains the natural and efficient alternative.

Most commercial mls relational database systems do not support write-up operations. This can be attributed to the fact that arbitrary and primitive write-up operations can obliterate high-level data, thereby affecting its integrity. However, in the object-oriented context, write-up operations result in the invocation of appropriate methods in the high level target object. This, along with the properties of encapsulation and information hiding, provide defenses and ensure that the state of the object is modified only in predefined and controllable ways.

A formidable obstacle in building high-assurance multilevel secure systems is the presence of covert channels. Such a channel is a communication path that can be exploited by colluding users or processes to leak information. These channels arise due to processes sharing resources in a system. Security models and access control mechanisms often do not provide enough defenses against these channels. Unfortunately, in the object-oriented and message-based environments, an instance of this problem manifests in terms of signaling channels associated with synchronous write-up operations. A synchronous operation is one where the sender object's method is suspended until processing is completed in the higher level receiver object and a reply is returned back to the sender (mimicing remote procedure call semantics). Since a high level receiver object can modulate the time taken to complete its processing and return a reply, this time can be observed by the lower level sender. Such timings can be used to construct a pattern of signals which convey information from the high object to lower levels and pose confidentiality leaks which violate the security policy.

To address this covert channel, we have pursued a solution which calls for concurrent computations to service write-up operations. In other words whenever a write-up message is sent, the sender is allowed to continue and a new concurrent computation (process) is created to service the receipt of the message. Hence the sender object can no longer observe any timing delays and the channel is closed. It is important to note that in essence we now have an asynchronous operation; however we require that the net effect of the operation on objects be the same as that of a synchronous write-up.

The field of information security is concerned with three separate but interrelated goals, namely, confidentiality, integrity, and availability. We are thus interested in solutions that address the various dimensions of the confidentiality-integrity-availability security triad. The concurrent computation model mentioned above addresses the confidentiality aspect by closing the signaling channels associated with supporting synchronous write-up operations. However, the integrity objective will mandate that the concurrent computations not modify objects arbitrarily. These modifications should preserve the semantics of the originally intended synchronous execution. This will require scheduling and synchronization of the various concurrent processes. The availability objective mandates that computations do not starve and unnecessary delays in scheduling be minimized.

We elaborate on how our computing model can be implemented in a trusted subject architecture. This architecture along with two others, namely, the kernelized and replicated architectures, represent three well-known architectural approaches to the design of high-assurance multilevel secure systems. However, from a commercial standpoint, the trusted subject architecture has been the choice of database vendors. The implementation of our computation model in the kernelized and replicated architectures has been discussed elsewhere in the literature [26], [27].

The key component of our architecture is a trusted scheduler charged with scheduling the various concurrent processes created during the course of write-up processing. The scheduler itself is called a "multilevel process" as it deals with inputs and outputs at various security levels. As we shall discuss later, such a process is also considered to be a "trusted subject" and hence an architecture with such a subject is referred to as a trusted subject architecture. The term "trusted" is used to convey the fact that such a process by virtue of its ability to see and process information at multiple levels has to be exempted from access control restrictions, and as such, is trusted not to abuse this privilege and leak information.

To get a system certified for high-assurance environments, we have to prove, verify, and certify that a trusted subject does not leak information. We approach a confidentiality proof for our architecture using the theory of *noninterference* [6], [7]. Informally, we say a subject s_1 is noninterfering with a second subject s_2 if no action issued by the first can influence the future output of the system to s_2. We show how the scheduler cannot introduce interference across security levels. In other words, we show how the scheduling of high level processes cannot affect the scheduling of processes at lower levels. Our proof represents a concrete application of the noninterference approach to a real database problem.

The rest of this paper is organized as follows. We begin in Section 2 by covering some background to multilevel security and multilevel architectures. Section 3 is devoted to multilevel security in object-oriented systems and a discussion of our solutions to support secure write-up operations. In Section 4, we illustrate the trusted subject architecture and elaborate on how the scheduling scheme can be implemented within the architecture. In Section 5, we give a confidentiality proof for the trusted scheduler, and Section 6 concludes the paper.

2 Background

In this section, we give a brief introduction to multilevel security and multilevel architectures.

2.1 Multilevel Security

The notion of multilevel security for data confidentiality originated in the late 1960s when the U.S. Department of Defense wanted to protect classified information processed by computers. Environments and applications requiring multilevel security are characterized by users with more than one clearance level sharing data with more than one sensitivity level (classification). Access control policies and

mechanisms govern how the various subjects in a system are allowed to access shared data.

On closer examination, it becomes clear that the military security policy is indeed a special case of a more general lattice-based security policy [4], [23]. Every object in the system is assigned a security class, also known as a security label. Information is allowed to flow between two objects only if the policy allows information to flow between the corresponding classes. Given a set SC of security classes, we can formally define a binary can-flow relation $\rightarrow \subseteq SC \times SC$. It is also convenient to define the inverse of the can-flow relation called the dominates relation. We say $A \geq B$ (A dominates B) if and only if $B \rightarrow A$ (B can-flow to A). In a lattice-based approach to multilevel security, the security classes form a mathematical structure called a lattice. The security elements of the lattice are partially ordered under the can-flow ($\rightarrow$) relation.

Having introduced multilevel security, we now turn our attention to security models. The Bell and LaPadula security model (also called the BLP model) was the first to formally address multilevel security, and even today remains the de facto standard [1]. BLP distinguishes two types of subjects; untrusted and trusted. An *untrusted subject* is assigned a single security level (label). BLP characterizes and governs access control and information flow for untrusted subjects with the following two mandatory access control (MAC) rules (l denotes the label of the corresponding subject (s) or object (o)).

- **Simple Security Property.** Subject s can read object o only if $l(s) \geq l(o)$.
- **★-Property.** Subject s can write object o only if $l(s) \leq l(o)$.

The need for the simple-security rule is obvious; it prevents low level users (and subjects) from reading information stored at higher levels. It thus prevents "read-up" operations. This requirement parallels that of the paper world with documents and human beings (users). However, it turns out that disallowing read-up operations alone is not sufficient to prevent illegal information flows that violate the security policy. To illustrate, a high subject may read a file classified at high, and write a subset of its contents (or information derived from its contents) into a second file at a lower file. This would clearly violate the security policy as information is flowing downwards in the security lattice. Now in the paper (noncomputer) world, human users are trusted to not leak such information. However computer systems can be riddled with Trojan horses that are malicious pieces of software code for which one cannot associate the same notion of trust. Thus ★-property (pronounced star property) prevents such violations by disallowing write-down operations.

A major complication in implementing multilevel security is the existence of *trusted subjects*. This is because such subjects are exempt from mandatory access control, although we place some trust in them not to behave maliciously and leak information in violation of the security policy. The need for trusted subjects in computer systems can be attributed to the provision of many services that require access to data stored at different security levels. Examples include services from authentication and file servers. The processes providing these services cannot be bound by mandatory access control restrictions. Consequently, with the existence of trusted subjects, we need the assurance that such subjects do not leak information. In the latter part of this paper we develop a confidentiality proof for a trusted scheduler.

2.2 Multilevel Architectures

In this subsection, we briefly review three multilevel database management system (DBMS) architectures, namely, the kernelized, replicated, and trusted subject architectures. Of these, the first two comprise two of the three architectures identified by the Woods Hole study organized by the U.S. Air Force [3], and were seen as short-term solutions. Thus these architectures were motivated by the need to build multilevel secure DBMSs from existing untrusted DBMSs. The trusted subject architecture on the other hand, requires one to build a multilevel DBMS either from scratch or by modification of existing DBMS technology.

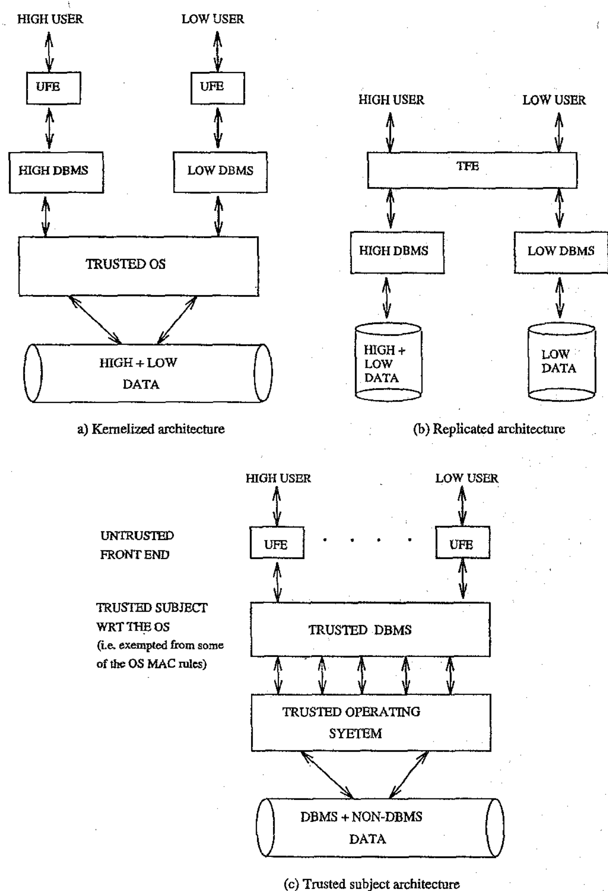

Fig. 1. Three multilevel system architectures.

The basic features of the kernelized architecture are shown in Fig. 1a. For ease of exposition, assume two security levels, high and low, where high dominates low. Most

noticeable is the fact that there exists an individual DBMS for every security level. A user cleared to level high, interacts with the high DBMS through an untrusted front end (UFE). Existing DBMSs can be directly incorporated with minimal modification since they have to manage only single-level data at their levels. These single-level DBMSs communicate with a trusted operating system that manages both high and low data storage. Although we have considered only two levels, it is important to note that this architecture carries over to arbitrary lattices.

The distinguishing feature of the replicated architecture shown in Fig. 1b is the existence of separate DBMSs for every level and the replication of low data at the higher level DBMSs. The high assurance of this architecture stems from the fact that the DBMSs are physically isolated and a subject cleared to a level, say *l*, is allowed to access only the database for level *l*, and can obtain all the data needed at the local DBMS location. In order to maintain the mutual consistency of replicated data, a trusted front end (TFE) is employed. The TFE is involved in propagating the various updates to the databases in accordance with some replica control scheme.

The trusted subject architecture shown in Fig. 1c differs from the kernelized one in that we no longer have single-level DBMSs for every level. Instead, there exists a single (trusted) DBMS that incorporates multilevel trusted subjects. Recall that such a subject is exempt from the mandatory access control rules enforced by the operating system. In this architecture, the trusted operating system is used to separate non-DBMS and DBMS data.

2.3 Trusted Subjects: Balancing Assurance, Cost, and Flexibility

As mentioned before, a trusted subject is allowed to access information at multiple security levels and thus cannot be bound by mandatory access control restrictions. As such it is trusted not to abuse this exemption. To verify that a system is secure, we need to develop assurance arguments and security proofs. Given that this may often be a challenging task, what then is the attraction of a trusted subject architecture? First, a trusted subject is able to have a global view of many system states. This simplifies the implementation of many algorithms, especially when a global snapshot is often required to make the next decision. Second, for most systems to be efficient and practical, a certain number of trusted system services have to be provided. Consider for example, a file server in a multilevel environment. Such a server will inevitably have to read and write files with different security labels, and would thus have to be trusted. The alternative would be to have a file server component for every security level, clearly not a very cost-effective solution. Other examples include login and authentication services. Third, there exist many applications in which certain users have to be occasionally given trusted privileges, e.g., system administrators have to be given certain privileges to do system maintenance.

In summary the trusted subject approach to architecturing a system currently represents the most viable tradeoff between assurance, cost, and flexibility. Consider the kernelized architecture in which all components and processes obey the security policy. As such, in theory, there is no need to develop rigorous security proofs. However, a look at the commercial multilevel database landscape will reveal that almost all commercial vendors have chosen the trusted subject architecture over the kernelized and replicated approaches [2], [11], [22], [24], [10]. In other words, even with the effort needed to prove that these systems are secure with respected to certain trusted subject exemptions, the trusted subject architecture appears to be most cost-effective and commercially viable at the present time. This impetus from industry has motivated us to explore this architecture for object-oriented databases.

3 Multilevel Security and Object-Oriented Systems

In this section, we discuss security in object-oriented systems and the complications that arise when write-up operations are supported.

3.1 Security Enforcement through Message Filtering

Central to the object-oriented model of computing is the notion that objects are encapsulated units of state and communicate with each other solely through messages. Thus a natural and intuitive way to enforce security is to control the exchange of messages. This is the basic idea behind the message-filter security model proposed in [12]. Attached to every object in the system is a classification. When a message is sent, the classifications of the sender and receiver objects are used to determine if an illegal information flow will take place. The message is delivered to the receiver only if the resulting information flow does not violate the security policy. It is important to note that there is no attempt to analyze the semantics (i.e., message type) or contents of the message itself. The advantages of this approach are many. It meshes well with the object-based model of computing, thus it has wide applicability in providing security for systems ranging from message and object-based operating systems to object-oriented databases. It is conceptually simple and elegant to enforce security by mediating messages at a central point in a system architecture.

Let us look at the message-filter model in more detail. The message filtering is accomplished through a message filter component. The filtering functions are illustrated graphically in Fig. 2. There are basically four cases of the filtering functions. In the first case, when a message is sent between objects at the same security level, both the message and the reply are allowed to pass through the filter. In the second case, when the objects are at incompatible levels, the message is intercepted before delivery to the receiver, and an innocuous NIL reply is returned by the filter. In the third case, involving a write-up message to the receiver at a higher level, the message is allowed to pass but the filter discards the actual reply and substitutes a NIL reply. The fourth case involves a read-down in which the message to the lower level receiver is allowed to pass and the reply is returned to the high level sender. However, the actual method invocation in the receiver

object is restricted from directly updating the state of the receiver object, or any other object at or below the security class of the receiver object, in order to prevent any potential write-down violations. It is important to note that this cannot cause a signaling channel as the receiver method cannot record any fact regarding its invocation. The actual implementation of restricted method invocations can be accomplished by using mandatory mechanisms and keeping track of the levels of method invocations. Due to space constraints we do not discuss it here, but details can be found in [26], [25]. In summary, these filtering functions implement the various mandatory access controls required in a multilevel environment.

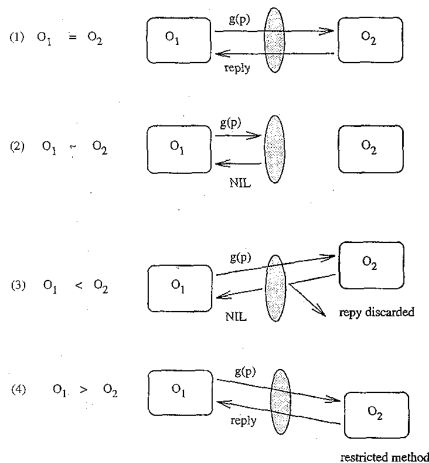

Fig. 2. Illustrating the message-filtering functions graphically.

3.2 Write-up Messages and Signaling Channels

The above filtering functions form an abstract security specification for enforcing multilevel security in object-oriented systems. However, when one elaborates such an abstract specification to an executable one, the potential for signaling channels surface. Consider, for example, case 3 of the filtering functions, where messages are sent upwards in security levels to implement write-up operations. Recall that mandatory security rules disallow read-up and write-down operations but place no restrictions on write-up operations. It turns out that the execution dynamics associated with processing such write-up messages has broad implications on confidentiality, integrity, and performance concerns.

Going back to the basics, let us see what happens when a message is sent to a higher security level for the purpose of initiating some write-up operation. Fig. 3 depicts case 3 of the filtering functions in more detail. Here a message g_1 is sent from a sender object O_1 to a receiver object O_2, with the receiver classified at a higher security level. With synchronous message passing, the sender method t_1 in object O_1 is effectively suspended once the message g_1 has been sent. The receipt of g_1 by O_2 will result in the invocation of a receiver method t_2. Eventually the invocation t_2 will terminate and return a reply which will resume the suspended sender.

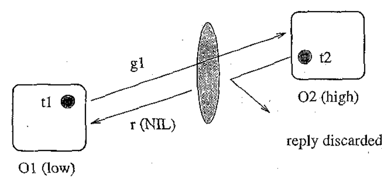

Fig. 3. A write-up message and its reply.

The problem with supporting synchronous message passing (as above) is that it is fundamentally insecure. This is because the low level sender object can deduce information about the high level receiver object's processing from the timing of the reply to the write-up message. A colluding high level object can return replies at specific times and induce a pattern which can be used to signal information to the low level sender. The solution we have pursued to address this problem is to return the reply independently of the receiver's processing and termination times. To be more precise, we return the NIL reply required by the filtering function instantaneously to the sender object when it issues a write-up message. This may result in concurrent computations as the sender and receiver methods may be executing at the same time.

The challenge now is to remain faithful to the originally intended synchronous semantics. Otherwise the integrity of applications will be affected, especially if concurrent computations modify objects arbitrarily. To see this, consider weekly payroll processing as illustrated in Fig. 4. With synchronous message passing and message filtering, the messages indicated by labeled arrows will be processed in the sequence a, b, c, d, e, f. On the other hand, instantaneous NIL replies and concurrency could lead to the sequence a, d, e, f, b, c. This means that the RESET-WEEKLY-HOURS message which resets the hours worked to zero will be received and processed by object WORK-INFO before the message GET-HOURS. Thus the message GET-HOURS will retrieve the reset hours as opposed to the actual accumulated hours, resulting in an erroneous calculation of the weekly pay. To avoid such integrity problems we need to provide appropriate synchronization schemes.

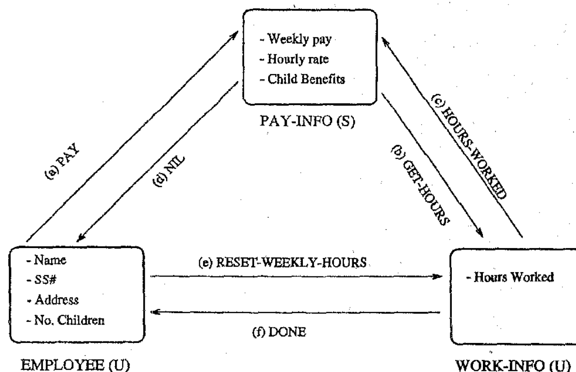

Fig. 4. Objects in a payroll database.

At this point, is is important to note that one can think of other obvious ways of returning independent replies, such as after random delays or constant intervals. However, in addition to the above mentioned integrity problems, these solutions result in unacceptable performance degradation, especially when senders are kept waiting longer for replies than they would have had to in a synchronous execution.

In general when there are cascaded write-up messages, the concurrent computations created to service the associated method invocations will fan out and form a tree such as that shown in Fig. 5. We refer to such a tree as a *session tree* as it encompasses all the write-up operations issued by user's session. In the tree in Fig. 5, a computation running at level unclassified (U) has issued three write-up operations by sending a message to an object at secret (S), a second message to an object at top-secret (TS), and a third message to an object at confidential (C). This has resulted in the creation of the corresponding three concurrent computations. The computation at level C in turn has issued other write-up operations resulting in the subtree rooted at node 4(C). When the computations in a session tree can guarantee that the reads and writes of the various methods will have the same effect as in a synchronous execution, we informally say that the session preserves *serial correctness*. A formal definition will be given later.

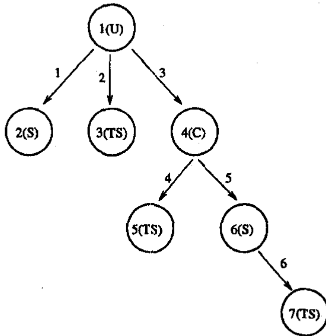

Fig. 5. A tree of concurrent computations.

Our approach to providing the necessary synchronization relies partly on scheduling the various computations in an order that does not violate certain requirements of a synchronous (serial) execution of the tree. The other half of the synchronization mechanism is a checkpointing and multiversioning scheme. Classical techniques such as those based on locking and semaphores cannot be used as they introduce signaling channels in a multilevel environment and thus are insecure. When a computation sends a write-up message, it checkpoints the state of its home object as well as other objects at the same security level, as new versions. Subsequent updates made by the computation to these objects are stored in new versions. These versions in turn will also be checkpointed when the next write-up message is issued. When a high level computation issues a read request to determine the state of a lower object object (this is allowed by mandatory access control rules as it is a read-down operation), the read is mapped to an appropriate version whose state reflects the state the high computation would have seen in a serial execution. Going back to the payroll example in Fig. 4, when the write-up message PAY is issued, the unclassified object WORK-INFO(U) will be checkpointed. The subsequent GET-HOURS message will retrieve this version which is unaffected by the RESET-WEEKLY-HOURS message.

3.3 Formal Definitions and Correctness Constraints

Before proceeding further, we formally define several notions related to serial correctness and session trees. In our further discussions we assume that the creation of a computation is accomplished through a fork call (as explained in detail in Section 4). However, the fact that a computation is forked (created) does not imply that it is executing, as this requires an explicit start call after the fork. Once started, a computation will eventually terminate and issue a terminate call.

DEFINITION 1. *We define a computation* `b` *to be to the right of a computation* `a`*, if neither* `b` *nor* `a` *is an ancestor of the other, and* `b` *is encountered later than* `a` *in a depth-first traversal of the corresponding session tree, starting at the root. Similarly, a computation to the left is one encountered earlier in a (left-to-right) depth-first traversal.*

DEFINITION 2. *We say a session preserves serial correctness if for any computation* `c` *in the session's computation tree, and running at level* `l`*, the following hold:*

1) `c` *does not see any updates (by reading-down) of lower level computations that are to its right, in the tree;*
2) *For any of* `c`*'s ancestor computations* `a`*, (i.e., any computation on the path from the root to* `c`*)* `c` *should see only the latest updates made by* `a` *just before* `a`*'s child (or* `c` *itself) on this path was forked.*
3) *For any level* `k` *that is not the level of an ancestor of* `c`*, and* $k \leq l$*,* `c` *should see the latest updates made by the rightmost terminated computations at level* `k` *that are still to the left of* `c`*.*

Given the above definitions, let us see the complications concurrency poses to the maintenance of serial correctness. If we were to execute the tree in Fig. 5 serially (i.e., in a synchronous fashion), the messages sent to higher level objects would be processed in the order given by the labels on the arrows. Note that this order can be derived by a left-to-right, depth-first traversal of the tree. If we follow this order, it is clear that any computation, say *c*, should not get ahead of earlier forked computations to its left. For example, under a serial execution of the tree of computations in Fig. 5, we would expect computation 2(S) and its descendants (if any) to terminate before computation 3(TS) to its right, is started. Computation 3(TS) should thus see all the latest updates by 2(S) and any of its descendants. Allowing arbitrary concurrency may not ensure this. Thus, there is a need to enforce some discipline on these concurrent computations by scheduling them in a manner that guarantees serial correctness.

To see the need for multiversioning, consider again this tree. With concurrent execution it is possible that computa-

tions 4(C) and 6(S) may terminate well ahead of 3(TS). Therefore, our synchronization schemes must ensure that computation 3 does not see any updates by computations 4 and 6, since 4 and 6 are to the right of 3. In other words, although 4(C) and 6(S) may terminate well ahead of 3(TS), our multiversioning scheme guarantees that a read-down request from 3(TS) will always read versions that existed before 4(C) and 6(S) were started.

The above example and discussion give us some insights into certain constraints that are sufficient to guarantee serial correctness and we state this below as a theorem. Due to space constraints a formal proof of this theorem is not given here but the interested reader is referred to [25].

THEOREM 1. *Correctness constraints 1, 2, and 3, given below, are sufficient to guarantee serial correctness of concurrent computations in a user session.*

Whenever a computation c *is started at a level* l,

- ***Correctness-constraint** 1: There cannot exist any earlier forked computation (i.e., to the left of* c*) at level* l*, that is pending execution;*
- ***Correctness-constraint** 2: All current as well as future executions of nonancestral computations that are to the left of* c*, should be at levels higher or incomparable to* l*;*
- ***Correctness-constraint** 3: At each level below* l*, the object versions read by* c *would have to be the latest ones created by the rightmost computation, say* k*, that is to the left of* c*. If* k *is an ancestor of* c*, then the latest version given to* c *is the one that was created by* k *just before* c *was forked.*

It follows from the above constraints that given any tree of forked computations, there can exist at most one running computation at a given level, at any given time. Also, a computation cannot be started until all earlier forked computations at dominated levels have terminated. In summary, the maintenance of serial correctness requires careful consideration on how computations are scheduled as well as on how versions are assigned to process read-down requests.

3.4 An Aggressive Scheduling Scheme

In the last subsection, we discussed the notion of serial correctness. In particular, we discussed how serial correctness mandates some scheduling discipline on the concurrent computations within a session tree. One can formulate a family of scheduling schemes where each scheme guarantees serial correctness by enforcing the various serial correctness constraints. These schemes differ in that they have varying performance characteristics. That is, the amount of delay a computation experiences after it is created and before it is allowed to start, varies from one scheme to the other. We informally characterize such delays into *necessary* and *unnecessary* delays. A necessary delay is one which a scheduling scheme imposes to prevent violation of serial correctness. On the other hand, a scheme induces an unnecessary delay if a computation is held up for reasons that have no bearing on serial correctness. Since serial correctness is an important requirement, we have no latitude in doing away with necessary delays. However, we can design scheduling schemes that vary in the amount of unnecessary delays induced on computations.

We now present an aggressive scheduling scheme. It never induces unnecessary delays and is thus optimal.[1] The aggressive scheme is governed by the following invariant.

Inv-aggressive: *A computation* c *is executing at a level* l *only if all computations that are to the left of* c *at levels l or lower, have terminated.*

To elaborate further, consider the trees in Fig. 6 that illustrate the different stages of the progressive execution of a sample session tree under the aggressive scheme. The session advances to the next stage as a result of the termination of an active (executing) computation in the tree. This next stage may see the release (startup) of one or more computations that were previously held up by the just-terminated computation. Thus, stage 2 (see Fig. 6(2)) has resulted from stage 1 due to the termination of computation 2(S). Stage 3 has resulted from stage 2 due to the termination of 4(C) which in turn has released 5(C) and 7(S) for execution, and so on.

This scheme is named "aggressive," because after every termination, the scheme aggressively attempts to recruit all possible queued computations for execution so long as this does not violate serial correctness. To ensure this, a check is always made to see if the startup of a computation would violate the above invariant.

In the next section we discuss the implementation of the aggressive scheduling scheme. In addition to the algorithms, we give an architecture that calls for a trusted scheduler. Such a scheduler always has a global snapshot of a session as it progresses to termination.

4 A Trusted Subject Architecture and Implementation

We now present the trusted subject architecture and the implementation of the aggressive scheduling scheme within this architecture.

4.1 The Architecture and Trusted Computing Base

The basic design of our secure architecture, illustrated in Fig. 7, is motivated by and built upon the architecture of existing object-oriented database systems such as ORION [14], IRIS [5], and GEMSTONE [17], [16]. Assuming a client-server model of computing, the server side of the 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 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.

1. To present the proof that this scheme is optimal would take us beyond the scope of this paper. The interested reader is referred to [25].

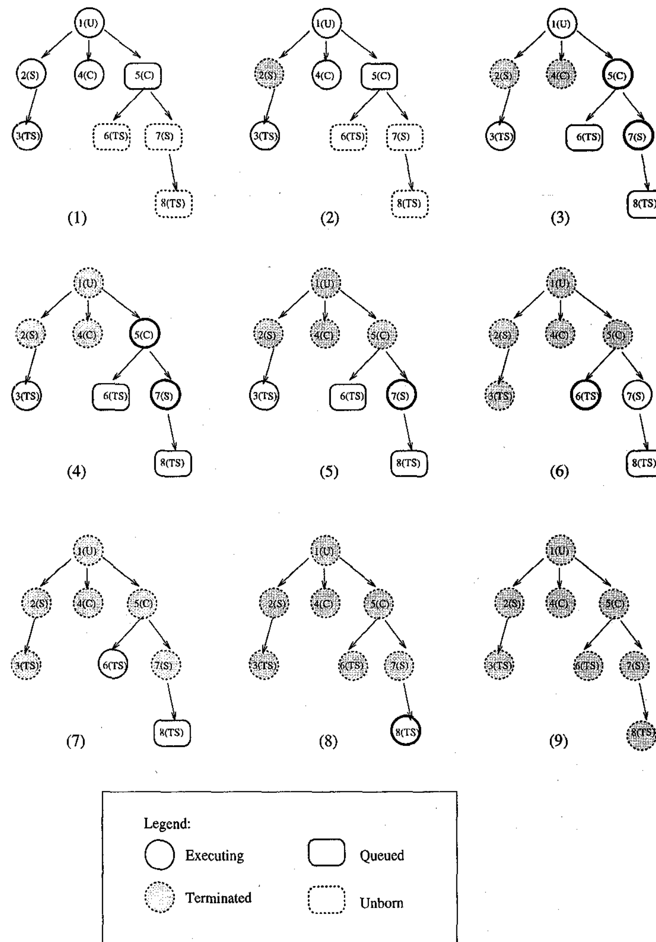

Fig. 6. Progressive execution under aggressive scheduling.

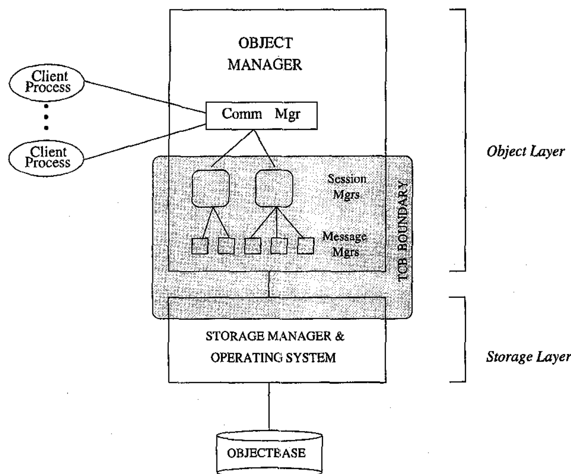

Fig. 7. A trusted subject architecture.

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.

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 trusted computing base (TCB). In this paper we assume that a small subset of the storage layer is trusted. This subset will provide operating system support including basic functions to manage stable storage in a secure fashion. As far as the object layer is concerned, only a small portion of the object layer needs to be within the TCB. This is in accord with an important design objective

for secure systems—keeping the size of the TCB to a minimum. In fact, the trusted functions are precisely those required to implement the message filtering functions. The trusted portion of the object layer consists of one or more *session manager* and *message manager* modules. The message manager and session manager modules collectively implement the message filtering algorithm.

A key aspect of our architecture is that the session manager process runs as a multilevel subject while the message managers are single-level subjects with respect to the Operating System TCB. As a multilevel process it is responsible for coordinating single-level (untrusted) message manager processes running at various security levels. The session manager is a long-lived process that is created when a session starts and is deleted only when the session eventually terminates. Typically, the start of a session is requested by a client process interacting with a communications manager or network listener module. A session manager may create several short-lived message-manager processes during a session. Whenever a write-up message is issued, a message manager process is created to service the request, and it implements the message filtering functions. Thus the message managers form the tree of concurrent computations that make up a session tree.

The interface between a message manager and its local session manager consists of *fork*, *terminate*, and *start* calls. A fork is issued by a message manager to request creation of a new (child) message manager. Such a newly created message manager may be started immediately if doing so would not violate any scheduling invariant. A message manager issues a terminate call to its session manager to signal termination. A start call is issued by a session manager to a message manager to initiate the execution of the message manager.

What is the motivation, if any, for the trusted subject architecture and implementation? The main advantage is the simplicity with which the scheduling algorithms can be implemented due to the availability of a trusted subject. A session manager always maintains a global snapshot of a session's tree of computations as they progress. With the help of such a global snapshot (view), a session manager is able to coordinate the various concurrent computations (message managers) and implement the scheduling algorithms.

The advantage of using a trusted subject for scheduling does come at a price. We now have to provide assurance that such a trusted subject cannot leak information. We later give a noninterference argument to demonstrate that although being exempt from mandatory access control rules, the session manager cannot leak information while coordinating various scheduling strategies.

4.2 Implementation of Scheduling Algorithms

Before discussing the algorithms in detail, we describe the data structures used by the session manager. Recall that our approach to synchronizing concurrent computations was based on multiversioning. Every version of an object is assigned a unique timestamp upon creation.

We also assume the existence of a forkstamping scheme such as that shown in Fig. 8. This scheme assigns unique forkstamps to individual message managers (computations) to reflect the serial order in a serial (synchronous) execution of the message managers. Briefly explained, every message manager except the root is assigned a unique forkstamp by the parent issuing the fork. Thus for the session tree in Fig. 8 the scheme starts by assigning an initial forkstamp of 0000 to the root message manager 1(U). Every subsequent and immediate child of the root is then given a forkstamp derived from this initial one by progressively incrementing the most significant (leftmost) digit by one. To generalize this scheme for the entire tree, we require that with increasing depth along any path in the tree, a less significant digit be incremented. In general for a security lattice with a longest maximal chain of n elements, we need to reserve $p * (n - 1)$ digits for the forkstamp. In a lattice with l levels, and c compartments, $n = l + c$. The value of p would depend on the maximum degree of a node in a computation tree. For example if we assume that any computation sends a maximum of 99 messages to higher levels, then setting $p = 2$ would be sufficient. Even with large lattices and a high number of messages sent to higher levels, these numbers are reasonable.

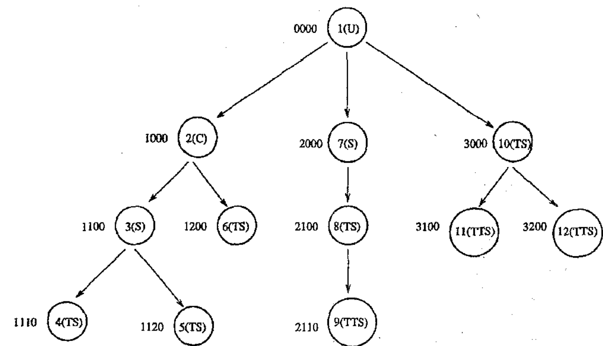

Fig. 8. Generation of forkstamps for a session's computation tree.

The session manager maintains the following data structure to keep track of initial versions.

- **Init-stamp:** This is a global table of timestamps 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 pertaining to 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:	active, terminated, queued
level:	the level of the message manager
local-stamp:	a local table of timestamp entries with an entry at each level dominated by the message manager and identifying the versions at each level that will be used to process read-down requests
forkstamp:	forkstamp issued by the parent message manager

parent:	pointer to parent message manager node
wstamp:	this is a timestamp entry indicating the next version that will be written by the message manager
object:	identification of receiving object
message:	message
p:	message parameters

A message manager's **local-stamp** vector is initialized in two phases, with the first one undertaken when a message manager is forked and the second one deferred until the message manager actually starts. For a message manager just forked, the first phase entries identify the versions to be read at the levels of ancestors, on the path from the root to itself. These first phase entries are actually obtained by a message manager from another vector that is passed along by its parent. Such a vector can be seen as one that is incrementally constructed along a path in the computation tree. To do this, every message manager is required to save the timestamps in the vector (astamps) obtained 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. Finally, in the second phase we obtain **local-stamp** entries for the levels that did not participate in phase one (this is done in the **start-trusted-agg** procedure of Fig. 11).

A high-level pseudocode specification of the session manager algorithms to implement the aggressive scheduling scheme is shown in Figs. 9, 10, and 11. 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 invariant **inv-aggressive** presented in the last section is never violated. For easy reference, we give the invariant below:

Inv-aggressive: *A computation* c *is executing at a level* l *only if all computations that are to the left of* c *at levels* l *or lower have terminated.*

```
Procedure fork-trusted-agg(level-parent,
level-create, forkstamp, astamps)
{
Let parent be the node issuing the fork;
Let child be a new message manager node;
Make child the rightmost child of parent;
child.level ← lub[parent.level, L(O2)];
child.forkstamp ← forkstamp;
%Begin phase 1 of acquiring local-stamp entries
For (every level l ≤ level-parent)
do
    initialize child.local-stamp table entries
    from astamps;
End-For
If in a depth-first traversal of the tree
starting at the leftmost path and
until child is traversed, there exists a
non-ancestor node, say n, with
{n.level ≤ child.level and n.status = active
or queued}
then child.status ← queued;
else
    start-trusted-agg(child);
end-if
}
end procedure fork-trusted-agg;
```

Fig. 9. Session manager algorithm for FORK.

```
Procedure terminate-trusted-agg(lmsgmgr, wstamp)
{
Let term be the node that terminated at level
lmsgmgr;
% Mark this node as terminated
term.status ← terminated;
% See if any queued nodes can be started
Initiate a depth-first traversal of the session
tree such that:
If for every leaf node, say leaf, that is
traversed to the right
of term such that leaf.level ≥ lmsgmgr,
there exists
no previously traversed non-ancestor node p
with {p.level ≤ leaf.level and
p.status = active or queued}
then
    start-trusted-agg(leaf);
end-if
}
end procedure terminate-trusted-agg;
```

Fig. 10. Session manager algorithm for TERMINATE.

```
Procedure start-trusted-agg(nn)
{
%Let node nn represent the message manager to
be started
%Complete phase 2 of acquiring local-stamp
entries
%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] ← n.wstamp;
end-if
%Update remaining local timestamp entries from
the Init-stamp 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] ← Init-stamp[l];
end-if
execute(nn);
}
end procedure start-trusted-agg;
```

Fig. 11. Session manager algorithm for START.

Whenever a fork request is received (see the procedure in Fig. 9), 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 procedure then records the forkstamp for the newly forked message manager that has been passed on by the parent, i.e., the message manager that generated the fork request. This is followed by the first phase of the initialization of the local-stamp entries. 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, active or queued, at the same or a lower level, the newly inserted node is queued and thus forced to wait. Otherwise it is started.

The processing of a terminate request begins by updating the status of the node to terminated (as shown in Fig. 10). We then check to see if this termination can release other queued up nodes. 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, these nodes have to be leaves in the tree). Thus a depth-first traversal of the tree is once again initiated. As in the fork case, a leaf node is allowed to execute only if required by the invariant.

Both the fork and terminate algorithms utilize a common Start procedure (shown in Fig. 11) by which message managers are started. This procedure is primarily concerned with completing the update of the local-stamp table entries of the node to be started. Recall from our previous discussion that the first phase of updating the local-stamp entries is achieved at fork time. The second phase is now accomplished from the following sources.

1) **Terminated left nodes:** For levels dominated by a node's level, and for which timestamps were not obtained from the ancestors, the start algorithm looks to the subtree of computations to the left of the node to be started. The timestamp 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.
2) **Init-stamp table:** If there are levels for which timestamps could not be obtained from phase 1 or from terminated left nodes, the algorithm then retrieves the timestamps from the global Init-stamp table maintained by the session manager. This is because objects at 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 timestamps in the Init-stamp table identify such versions.

Once all the local-stamp entries have been collected, the message manager is started (executed). 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 stored in the variable **wstamp** 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 A Noninterference Confidentiality Proof

In this section, we give a confidentiality proof for the trusted subject architecture. We begin by reviewing the noninterference approach to developing confidentiality proofs.

5.1 Confidentiality and Noninterference

The noninterference approach to proving systems secure was originally proposed by Gougen and Meseguer in [6]. Noninterference reasons about confidentiality from the input/output interactions a system has with its external environment. In a multilevel environment, we would consider as insecure any scenarios in which high level inputs influence future low level outputs. The elegance of the noninterference framework lies in its ability to abstract away unnecessary implementation details. Rather than focusing on mechanisms for enforcing confidentiality, it focuses on specifications that can be used to rule out nonsecure implementations.

The original formulation of noninterference in [6] was in the context of deterministic systems. Subsequently, a number of researchers have developed similar abstract models for nondeterministic systems [8], [18], [20]. In this paper, we limit our discussion to deterministic systems as the actions of our trusted subject are deterministic in nature.

Let us begin by considering the reception of inputs as well as the generation of outputs to be discrete events. As in [19], we call the events less than or equal to a level l as belonging to the *view* of that level, and all other events as *hidden* from l. The basic idea of noninterference can be stated as follows: A subject s_1 is said to be noninterfering with subject s_2 if no action issued by s_1 can influence the future output of the system to s_2. To prove that the entire system is noninterfering we must be able to make this claim hold up for all subjects, histories (sequences) of inputs, and outputs.

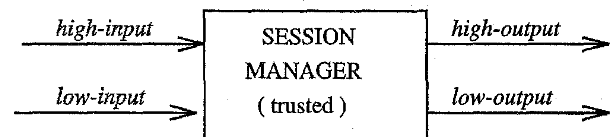

Fig. 12. The session manager as a black box with inputs and outputs.

Our task is to prove that the session manager, which is really a trusted scheduler and thus a trusted subject, is noninterfering. What does this mean? Recall that a session manager is responsible for the scheduling and management of the various concurrent computations (message managers) present in the associated session tree. These computations will be running at various security levels. We can now think of the session manager as a black box, as shown in Fig. 12, that interacts with its environment (consisting of computations at various security levels) by accepting inputs and producing outputs. The inputs and outputs are really the reception and generation of the following requests:

- **INPUT:** {fork, terminate}
- **OUTPUT:** {start}

In other words, when a running computation wishes to create a concurrent child computation, it sends a fork input request to the session manager. The session will then create a new child computation and update relevant data structures to keep track of it. If the computation can be immediately started, the session manager sends a start output mes-

sage to the process; otherwise the computation is queued for future execution. Such a queued computation will eventually be started when the session manager is notified of a subsequent termination. When a process is about to terminate it sends the session manager a terminate input request.

Noninterference in our context then amounts to demonstrating that the session manager, in the course of scheduling computations in a user session, will not introduce interference. Consider for example two security levels, low and high. Then this means that the reception of fork and terminate requests from high computations will not influence the output to low computations which manifest as start requests.

The most pertinent question at this point is rather obvious. How does one go about establishing noninterference? Given an input sequence, the most intuitive way to proceed would be to purge all hidden inputs and demonstrate that the events observed in the view for a lower level subject remains unchanged. Thus at the heart of the noninterference approach is the utilization of purge functions to systematically purge high inputs. For our architecture, we need to show that the purge (or pruning) of the session tree of high computations, will leave the outcome of the scheduling to low level computations, unaffected.

5.2 Definitions and Assumptions

Let us now make the above ideas more precise and concrete with the help of some definitions and formalisms. We also discuss some important assumptions.

DEFINITION 3. *We define an event as a triple (type, l, tstamp) where type ∈ {fork, term, start}, l is the level of the message manager (computation) from which the input originated or the level of the message manager to which an output is directed, and tstamp is a timestamp indicating the elapsed time since the occurrence of the last event at l.*

DEFINITION 4. *Given any security level, l, we define the events at less than or equal to l as belonging to a set called the view of l and all other events as belonging to a set called* hidden *from l.*

DEFINITION 5. *Given any security level, l, we define the subset of events in the view of l that are at levels strictly below l as belonging to the set* lower-view.

DEFINITION 6. *Given two event sequences β_1 and β_2, we say that they are l-equivalent (denoted as $\beta_1 \approx_l \beta_2$) if they contain the same events, in the same relative order, for levels l and lower (i.e., they contain the same values for the event triples and the events associated with these triples appear in the same relative order in both sequences).*

DEFINITION 7. *Given a sequence β, we define a purge function purge (β, l) as one that returns the sequence β but with all events of the form (type, l_e, tstamp) removed (purged) whenever $l \not\geq l_e$.*

Given any input sequence α_1, which when processed by the session manager produces an output sequence β_1 (denoted $\alpha_1 \longrightarrow \beta_1$), noninterference requires us to show the following: **If** for every level l, purge(α_1, l) $\longrightarrow \beta_2$, **then** $\beta_1 \approx_l \beta_2$.

Before proceeding on a formal proof that the session manager is noninterfering, we list our assumptions:

- **Input-totality.** If we view the session manager as a state machine, this assumption states that the session manager (or state machine) can accept inputs in any state. This ensures that the session manager is not conveying any information by accepting inputs.
- **Input-output atomicity.** This assumption requires the session manager to accept an input and produce the corresponding outputs, if any, atomically. In other words, in the interval between the acceptance of an input and the subsequent processing and generation of the corresponding outputs, the session manager cannot be interrupted, especially by other inputs.

The assumptions of input totality and input-output atomicity may seem at first to be irreconcilable. After all, if the session manager cannot be interrupted in the interval between the acceptance of an input and the production of the corresponding output, how could it be capable of accepting other inputs that come within such an interval? In other words, how could it remain input-total? Assume for a moment that inputs arrive at the session manager boundary synchronized with clock ticks that are a constant interval apart. It is important to note that such an interval can be chosen to deal with worst case arrival rates of inputs and to guarantee that no two inputs can arrive at the same tick. The session manager is required to accept an input at a clock tick, and produce all the corresponding outputs, before the next clock tick. In other words, at every clock tick, the session manager is ready to accept an input, and within clock ticks cannot be interrupted to accept other inputs.

In the above model, given an input, we require that the corresponding outputs be produced within the same interval. In other words, the outputs cannot spill over to time intervals between subsequent clock ticks. However, one needs to approach the implementation of this requirement with caution. In particular, the timing of the outputs within an interval should not be used to build a channel. Hence we require the scheduler to hold off all outputs until the expiration of the interval. Upon expiration, the outputs are delivered as a batch (unordered set) to a lower level subsystem or operating system.

The realization of the input-output atomicity assumption also requires that the tree data structure implementation utilized by the session manager be an "ideal" one. By ideal we mean that the elementary data structure operations such as the insertion and deletion of nodes in the tree are implemented in such a way that their timing cannot be exploited for covert timing channels. In particular, tree operations should be completed within a clock tick. For if this were not the case, a method in a high level object can maliciously cause the tree to grow to a considerable size by issuing write-up actions and causing a lot of nodes to be inserted into it. A low-level computation generating fork requests may now experience observable delays due to the increased time taken by the session manager to update and manage the tree.

A possible solution to deal with the above scenario would be for the TCB to do the tree operations at random

intervals. An approach that pursues a similar idea to address hardware timing channels is based on the technique of *fuzzy time* [9]. Fuzzy time techniques reduce the bandwidth of timing channels by adding noise to all sources of timing information and by ensuring that inputs and outputs are delivered at random intervals. We do not consider such solutions as they would take us beyond the scope of this paper.

5.3 The Confidentiality Proof

In approaching a confidentiality proof, we first make a crucial observation. If we look more closely at the processing of fork, term, and start requests, we see that it is a repetitive two-step processing cycle. More precisely, the cycle consists of accepting an input and generating the corresponding outputs, as shown in Fig. 12. What restrictions would confidentiality considerations impose on such a processing cycle? Confidentiality and noninterference requirements mandate that if an input is accepted at a level l, the corresponding outputs be generated at l or higher. To see this more clearly, consider only two security levels—low and high (low < high). A fork request issued by a process (computation) at low will form a low-input (low-fork) to the session manager. Since a fork is a request to create and start a process at a level higher than the requester, it follows from the semantics that such an input can generate only a high output (high-start) when the request is processed. The confidentiality and noninterference requirement is thus trivially satisfied. Now consider three levels—low, med, and high. The termination of a process at level med should result in the start-up of queued processes (if any) but only at levels med or high. If this were not the case, the process at level med can signal information to the subjects at low. The implication of this on scheduling is that a process could never be suspended or queued waiting for a higher level process to terminate.

From the above discussion, it should be clear that the two-step processing cycle invoked by the session manager upon accepting a single input in isolation is noninterfering. In particular, after the acceptance of an input, information flow occurs through the session manager only in an upwards direction in a lattice. But what about the case when the session manager accepts multiple inputs from different security levels? We now have to show that high inputs do not interfere with the outputs to low inputs for all sets of traces. We thus have to consider all possible interactions of high inputs and low inputs. We now state and prove this as a theorem.

THEOREM 2. *In scheduling various concurrent computations, the session manager process is noninterfering.*

PROOF.

The proof is by induction on the number of inputs in a session manager trace.

Basis: For the base case consider a trace with exactly one input. It follows from our earlier discussion that by accepting only one input at a single security level, say l, the corresponding outputs will be at l or higher. This does not influence the outputs at the lower-view of l and it follows that the session manager will be trivially noninterfering.

Inductive Step: For the induction hypothesis assume that for all traces with n or fewer inputs, the session manager is noninterfering. Consider any trace, say α, with $n + 1$ inputs which exhibits downward interference. Let δ be the $(n + 1)$th input in this string, as shown in Fig. 13. Let the outputs generated by the scheduler after the reception of the input δ, belong to the (possibly empty) set θ with the individual outputs in the set denoted as $\theta_1, \theta_2, \ldots, \theta_k$.

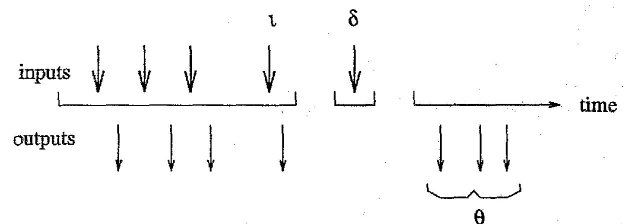

Fig. 13. Illustrating the noninterference proof.

Suppose the downward interference in α is observed at level τ. By this we mean that there is at least one input at τ, and the purge(α, τ) will cause some output(s) in the lower-view of τ to change. The output(s) that change must be in θ, otherwise interference with outputs prior to δ implies existence of a trace with n inputs that is interfering (contrary to induction hypothesis). From our earlier discussion on the two-step processing cycle of the session manager, the following observation should be clear. *Observation 1: The levels of the individual outputs in the set θ dominate the level of δ (i.e., $l(\theta_j) \geq l(\delta)$, $j = 1, \ldots k$).*

Consider all inputs preceding δ. Note that an input at a level can only interfere with outputs in the lower-view of the level. So let us pick the most recent input, say ι at level τ. We now show that ι cannot interfere with outputs in θ that are in the lower-view of τ. Since interference is observed at level τ in the outputs in θ, at least one output in θ is in the lower-view of τ. In conjunction with observation 1 this leads to another observation. *Observation 2: The level of the computation generating ι strictly dominates that of δ (i.e., $l(\iota) > l(\delta)$).* This observation, in turn, combined with our invariants and the requirements of serial correctness, lead to a very important third observation. *Observation 3: The node representing the computation that generated the input ι will be to the left of the node representing δ, in the session tree.* (If this were not the case, that is, if the low computation δ was to the left of the high computation, the high computation would not be executing due to serial correctness restrictions and thus cannot generate any high inputs.) As such the node of ι will always be traversed before that of δ, in a (left-to-right) depth-first search of the session tree.

Consider the interaction of ι and δ. There are two cases corresponding to δ being a fork or a term. If δ is a fork, the procedure **fork-trusted-agg** in Fig. 9 determines if

the forked computation generated by δ (referred to hereafter as *comp*δ) can be immediately started or if it has to be queued for future execution. This is done by initiating a depth-first traversal of the session tree starting at the leftmost path and ending at the node representing *comp*δ. In general, a computation f is denied immediate execution (startup) only if such a traversal encounters at least one nonterminated nonancestor computation at or below the level of f and to the left of f in the session tree. Hence, during the depth-first traversal, by observation 3 the computation associated with ι would be encountered before the computation generating δ. Now, if we purge the computation that generated the input ι, the outcome from the traversal of tree will be unaffected. In other words, if the computation associated with δ was denied execution, or allowed to start, this will continue to be the case after the purge of ι. So in this case ι does not interfere with θ.

For the second case δ is a term event. Next let us look at the procedure **terminate-trusted-agg** in Fig. 10 to see how terminate input requests are processed. We observe that when a computation, say t, terminates, a depth-first traversal of the session tree is initiated to identify potential leaf computations to the right of t that could be released for execution. A leaf computation in the tree is started only if there exists no previously traversed active or queued computation at or below the level of the leaf computation. Once again it follows from observation 3 that in a depth-first search, the computation generating ι will be encountered before the one generating δ. However, the computation generating ι is at a higher or incomparable level with respect to the computation generating δ, and thus the purge of the former will not affect the outcome of the traversal. Thus there is no interference as the output events in θ that are generated in response to δ would remain the same. More precisely, the output events in the lower-view of τ that are in θ, remain the same.

Thus far, we have shown that the input ι does not interfere with outputs in the set θ and in the lower-view of τ. We now construct a trace β that is identical to α upto but not including ι, followed by the inputs of α subsequent to ι at levels strictly below τ. The outputs of β will be identical to α up till ι. The outputs in β and α subsequent to ι can differ only at levels that are not strictly below τ (i.e., levels $\not\prec \tau$). In particular the output set θ' following δ may differ from θ only in outputs outside the lower-view of τ. Thus, outputs in the lower-view of τ remain unchanged in both sets θ and θ'. Since we just demonstrated that the input ι causes no interference, it follows that if the trace α with $n + 1$ inputs has interference at τ in θ, this interference must be also observed in β at τ in θ'. However, the trace β has n or fewer inputs, and cannot exhibit interference due to induction hypothesis. Hence, the induction step follows. □

6 Summary and Conclusions

The field of object-oriented database management systems continues to evolve as an important arena for research and development. The successful evolution of secure object-oriented databases depends to a great extent on the functionality, confidentiality, and integrity guarantees, as well as the commercial viability of various solutions that have been proposed.

In this paper, we have discussed a trusted subject multilevel architecture to implement synchronous abstract write-up operations free of signaling channels and an associated confidentiality proof to demonstrate the security of the architecture.

However, supporting write-up operations in object-oriented systems in inherently complicated by the fact that such operations are complex and, as such, take varying amounts of processing time. Thus, they are much more vulnerable to attacks that utilize timing information. Further, any solution to secure write-up in databases cannot address the confidentiality requirement in isolation. This is because integrity is a crucial aspect of database management systems. The utility of a database management system is largely dependent on maintaining data in accordance with certain integrity and correctness constraints. The solution we have given in this paper addresses the confidentiality issue by closing signaling channels and, in addition, provides the necessary synchronization to ensure integrity.

The trusted subject architecture mains the popular choice of many database vendors. The work reported here demonstrates that this commercially popular and viable architecture can be utilized to build multilevel secure, object-oriented database management systems that support high-assurance write-up operations. The confidentiality proof given in the paper gives us the assurance that the scheduler (trusted component) can be truly trusted not to leak information in a manner that violates mandatory security requirements. This also makes it easier to certify the architecture in environments that require high assurance.

There are several research issues that warrant further investigation. For example, how do we deal with failures and exceptions among the computations in a user session without compromising confidentiality in a multilevel environment? Also, how do we accomplish secure garbage collection? More study is needed to understand when versions created as a result of write-up operations can be safely purged. We hope to address these issues in the future.

ACKNOWLEDGMENTS

We wish to thank Dr. John Mclean of the U.S. Naval Research Laboratories for an afternoon discussion on how to approach the confidentiality proof. The work of both authors was partially supported by the U.S. National Security Agency through Contract No. MDA904-92-C-5140.

REFERENCES

[1] D.E. Bell and L.J. LaPadula, "Secure computer systems: Unified exposition and multics interpretation," Technical Report MTR-2997. Belford, Mass.: Mitre Corp., Mar. 1976.

[2] "Trusted Oracle 7 technical overview," ORACLE White Paper, technical report. Redwood Shores, Calif.: Oracle Corp., 1993.

[3] "Multilevel data management security," Nat'l Research Council Technical report, Committee on Multilevel Data Management Security, 1983.

[4] D. Denning, "A lattice model of secure information flow," *Comm. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[5] D. Fisherman, "IRIS: An object-oriented database management system," *ACM Trans. Office Information Systems*, vol. 5, no. 1, pp. 48–69, Jan. 1987.

[6] J.A. Goguen and J. Meseguer, "Security policies and security models," *Proc. IEEE Symp. Research in Security and Privacy*, IEEE, May 1982.

[7] J.A. Goguen and J. Meseguer, "Unwinding and inference control," *Proc. IEEE Symp. Research in Security and Privacy*, pp. 75–86, IEEE, May 1984.

[8] J. Gray, "Probabilistic interference," *Proc. IEEE Symp. Security and Privacy*, May 1990.

[9] W.M. Hu, "Reducing timing channels with fuzzy time," *Proc. IEEE Symp. Research in Security and Privacy*. IEEE, May 1991.

[10] *Building Applications for Secure SQL Server.* Emeryville, Calif.: Sybase Inc., Sept. 1993.

[11] "Technical summary: Open INGRES/enhanced security," *Proc. Workshop Research Progress in MLS Relational Database Systems*. Mt. Desert Island, Maine: Technical Cooperation Program, XTP-1, 1994.

[12] S. Jajodia and B. Kogan, "Integrating an object-oriented data model with multilevel security," *Proc. IEEE Symp. Research in Security and Privacy*, IEEE, May 1990.

[13] T.F. Keefe, W.T. Tsai, and M.B. Thuraisingham, "A multilevel security model for object-oriented system," *Proc. 11th Nat'l Computer Security Conf.*, pp. 1–9, Oct. 1988.

[14] W. Kim et al., "Integrating an object-oriented programming system with a database system," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 142–152, 1988.

[15] T.F. Lunt, "Multilevel security for object-oriented database system," *Database Security, III: Status and Prospects*, D.L. Spooner and C. Landwehr, eds., pp. 199–209. Amsterdam: North-Holland 1990.

[16] D. Maier et al., "Development of an object-oriented DBMS," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 472–482. New York: ACM, Sept. 1986.

[17] D. Maier and J. Stein, "Development and implementation of an object-oriented DBMS," B. Shriver and P. Wegner, eds., *Research Directions in Object-Oriented Programming*, pp. 355–392. Cambridge, Mass.: MIT Press, 1987.

[18] D. McCullough, "Specifications for multilevel security and a hook-up property," *Proc. IEEE Symp. Security and Privacy*, May 1987.

[19] D. McCullough, "A hookup theorem for multilevel security," *IEEE Trans. Software Engineering*, vol. 16, pp. 563–568, June 1990.

[20] J. McLean, "Security models and information flow," *IEEE Symp. Security and Privacy*, pp. 180–187, Oakland, Calif., May 1990.

[21] J.K. Millen and T.F. Lunt, "Security for object-oriented database systems," *Proc. IEEE Symp. Security and Privacy*, pp. 260–272, May 1992.

[22] J.P. O'Connor, "Trusted RUBIX technical overview," *Proc. Workshop on Research Progress in MLS Relational Database Systems*. Mt. Desert Island, Maine: Technical Cooperation Program, XTP-1, 1994.

[23] R.S. Sandhu, "Lattice-based access control models," *Computer*, vol. 26, no. 11, pp. 9–19, Nov. 1993.

[24] "SYBASE secure SQL server," technical paper series. Emeryville, Calif.: Sybase Inc., 1994.

[25] R.K. Thomas, "Supporting secure and efficient write-up in high-assurance multilevel object-based computing," PhD thesis, George Mason Univ., Fairfax, Va., Aug. 1994.

[26] R.K. Thomas and R.S. Sandhu, "A kernelized architecture for multilevel secure object-oriented databases supporting write-up," *J. Computer Security*, vol. 2, no. 3, pp. 231–275, 1994.

[27] R.K. Thomas and R.S. Sandhu, "Supporting object-based high-assurance write-up in multilevel databases for the replicated architecture," *Proc. European Symp. Research in Computer Security (ESORICS '94)*, Brighton, England, pp. 403–428, Nov. 1994.

[28] M.B. Thuraisingham, "A multilevel secure object-oriented data model," *Proc. 12th Nat'l Computer Security Conf.*, pp. 579–590, Oct. 1989.

Roshan K. Thomas holds bachelor's and master's degrees in computer science, and earned a PhD in information technology, from George Mason University, Fairfax, Virginia, in August 1994. He has been employed as a computer scientist at Odyssey Research Associates (ORA), since September 1994. During his tenure as a doctoral student, he was affiliated with the Center for Secure Information Systems (CSIS) at George Mason University and pursued active research in computer and information systems security under the direction of Professor Ravi Sandhu. Dr. Thomas' interests include models, mechanisms, and architectures for access control and secure distributed computing, multilevel secure high-assurance databases, and security in object-oriented systems.

Dr. Thomas is a member of the IFIP 11.3 Working Group on Database Security, and has been a reviewer for the *Journal of Computer Security*, *IEEE Transactions on Knowledge and Data Engineering*, and the annual IFIP WG 11.3 Database security workshops. He has presented his research work at major security conferences and has published papers in journals and conference proceedings, including the *Journal of Computer Security*, IEEE Computer Security Foundations Workshop, the National Computer Security Conference, and the European Symposium on Research in Computer Security.

Ravi S. Sandhu received PhD and MS degrees from Rutgers University, and BTech and MTech degrees from the Indian Institute of Technology, Bombay and Delhi, respectively. He is full professor and associate chair of the Information and Software Systems Engineering Department at George Mason University, Fairfax, Virginia; director of the Laboratory for Information Security Technology at GMU; and a member of the senior staff (part-time) at SETA Corporation, McLean, Virginia. He earlier served on the Computer and Information Science faculty at Ohio State University, Columbus, Ohio. His principal research and teaching interests are in information and systems security. He teaches several popular graduate-level security courses at GMU and has lectured all over the world on this topic.

Dr. Sandhu has published more than 80 technical papers on computer security in refereed journals, conference proceedings, and books. He has served on numerous program and conference committees for security-related conferences, and as program chair and general chair on several occasions. He is a founder and currently the general co-chair of the ACM Conference on Computer and Communications Security. Dr. Sandhu has served as a security consultant to several organizations, including the U.S. Internal Revenue Service, the Institute for Defense Analysis, and Odyssey Research Associates. He is currently chair of ACM's Special Interest Group on Security Audit and Control (SIGSAC).