11

Secure Locking Protocols for Multilevel Database Management Systems

Sushil Jajodia[1], *Luigi V. Mancini*[2] *and Indrajit Ray*[1]
[1]*George Mason University,*
Center for Secure Information Systems and Information and Software Systems Engineering Department, Fairfax, VA 22030-4444, USA. {jajodia,iray}@isse.gmu.edu
[2]*Università La Sapienza di Roma,*
Dipartimento di Scienze dell'Informazione, Via Salaria 113, 00100 Roma, Italy. mancini@dsi.uniroma1.it

Abstract

While there are several secure concurrency control protocols for multilevel database management systems, most of them employ timestamp ordering or multiple versions of data or a hybrid protocol that utilizes both. The only known secure locking protocol that maintains single version data and can guarantee serializability, immediately aborts a higher level transaction whenever any of its locks at the lower levels is broken.

In this paper, we offer two secure locking protocols. The first protocol produces pairwise serializable histories. The second protocol generates serializable histories if the security levels form a total order; however, in general, when the security levels form a partial order, it generates MLS-serializable histories, a notion of correctness that we introduce. The proposed protocols maintains single version data and require only the lock manager to be trusted; a higher level transaction can continue its execution and commit successfully even if some of its locks at the lower levels are broken. Rather than immediately aborting the high transaction when any of its low lock is broken, our protocols wait until such time as executing a high level action will actually create a cycle in the serialization graph, not merely whenever there is the possibility of a cycle being formed. These protocols work by a method of "painting" certain transactions and the data items accessed by these transactions and by detecting a cycle at the moment it is imminent in the serialization graph.

P. Samarati et al. (eds.), *Database Security*

Keywords
Database management, transaction processing, concurrency control, serializability, locking, multilevel security

1 INTRODUCTION

The problem of secure concurrency control makes transaction management in multilevel secure (MLS) database systems more complex than in traditional databases. In MLS databases, the data and user processes are classified into different security levels, and access to a data item by a process is governed by the following mandatory access rules: A transaction T can write to a data item x only if x is at the same security level as that of T; T can read x only if x is at a security level lower than or equal to that of T. Moreover, MLS databases must also prevent indirect information leakage through covert channels. The latter imposes serious restrictions on conventional concurrency control algorithms: A lower level transaction cannot be prevented from accessing a data item because a higher level transaction is already accessing it in a conflicting mode because doing so opens up a covert channel between the high and low security classes.*

Secure concurrency control has been studied by researchers in the context of multilevel database systems. Reed and Kanodia (1979) use the notions of eventcounts and sequencers to solve the secure readers-writers problem. Lamport (1977) and Schaefer (1974) offer a similar solution using version numbers. However, as shown in (Ammann & Jajodia 1992), none of these solutions generate serializable histories when applied to transactions. Moreover, these solutions suffer from the problem of starvation, i.e., transactions that are reading lower level data items may be subject to indefinite delays.

Other algorithms have been proposed that employ timestamp ordering or multiversion data or both. Ammann & Jajodia (1992) give two timestamp based algorithms on single version data that yield serializable histories. Keefe & Tsai (1990) propose a scheduler based on multiple versions of data and a priority queue of transactions according to their access classes. A third work by Ammann, Jaeckle & Jajodia (1995) proposes a concurrency control protocol using two snapshots of the database in addition to the most recently committed version, i.e. three copies of the database. This protocol can be naturally implemented using timestamp ordering to control the transactions executing at a given security level, although other scheduling algorithms can also be used. Other works, including Jajodia & Kogan (1990), Ammann & Jajodia (1994), Kang & Keefe (1995), and Ammann, Jajodia & Frankl (1996) are based on the subtle properties of the underlying database system architecture.

Although locking protocols have been found to be not only easy to implement but also efficient for transaction processing in conventional database systems, there are not many lock based secure concurrency control protocols. An exception is the set of *orange locking* protocols (McDermott & Jajodia 1993) that provide covert channel free concurrency control of database transactions. These protocols do not use multiversion data and can be implemented using single level untrusted schedulers. However, as we show here, except for the optimistic orange locking protocol with the assumption that a high transaction is always aborted whenever its low lock is broken, the other variations cannot guarantee the serializability of multilevel histories.

*Throughout this paper, we use the terms ***high*** and ***low*** to refer to two security levels such that the former is strictly higher than the latter in the partial order.

In this paper, we propose two locking protocols for secure concurrency control that maintain single version data and require only the lock manager to be trusted.† Rather than immediately aborting the high transaction when its low lock is broken, these algorithms wait until the last possible moment; they wait until such time as executing a high level action will actually create a cycle in the serialization graph and not whenever there is the possibility of a cycle being formed. This is achieved by a method of "painting" certain transactions and the data items they access and by detecting a cycle at the moment it is imminent in the serialization graph. The first algorithm guarantee pairwise serializability, a notion of correctness introduced in (Jajodia & Atluri 1992). The second algorithm guarantees serializability when the security levels of transactions and data items form a total order. As we discuss below, if the security levels form a partial order, such delayed abort may not be always possible without opening up a covert channel between transactions at incomparable levels. We present a new notion of correctness, MLS-serializability, and show that the second protocol guarantees MLS-serializable histories for partial orders.

This paper is organized as follows. Section 2 introduces the basic definitions and gives an example to motivate the coloring schemes we use. In section 3, we give our first protocol that generates pairwise serializability. In section 4, we present our notion of MLS-serializability, followed by a protocol that yields MLS-serializability. The rest of the paper deals with the second protocol. Section 5 discusses some issues relevant to its implementation. In section 6, we compare it with different orange locking algorithms. Section 7 gives a formal proof of its correctness. Finally section 8 concludes the paper.

2 SECURITY MODEL AND MOTIVATION FOR THE COLORING SCHEME

The multilevel secure system consists of a set D of data items; a set T of transactions (subjects) which manipulate these data items; and a partial order S of security levels, whose elements are ordered by the dominance relation $\preceq$. If two security levels s_i and s_j are ordered such that $s_i \preceq s_j$, then s_j ***dominates*** s_i. A security level s_i is said to be ***strictly dominated*** by a security level s_j, denoted as $s_i \prec s_j$, if $s_i \preceq s_j$ and $i \neq j$. Each data item from the set D and every transaction from the set T is assigned a fixed security level by a mapping L.

In order for a transaction T_i to access a data item x, the following two *necessary* conditions must be satisfied:

1. T_i is allowed a read access to data item x only if $L(x) \preceq L(\mathrm{T}_i)$.
2. T_i is allowed a write access to the data item x only if $L(x) = L(\mathrm{T}_i)$.

Note that the second constraint is the restricted version of the $\star$–property which allows transactions to write to higher levels (Denning 1982); the restricted version is desirable in databases for integrity reasons.

The simplest locking protocol on single version data that guarantees serializable histories and is secure at the same time, aborts a higher level transaction whenever one of the transaction's

†The whole body of a standard Lock Manager, written with all the requisite defensive programming, exception handlers, optimizations, deadlock detectors, etc. comes to about a thousand lines of actual code (see for example (Gray & Reuter 1993)) and, therefore, is easily verifiable.

High T_1: $r_1[x]$ $w_1[z]$ c_1

Low T_2 : $w_2[x]$ c_2

Figure 1 A serializable history rejected by the simplest secure algorithm

lower level read locks is broken by a lower level transaction. However, this simple algorithm is too pessimistic; it rejects even simple serializable histories like the one shown in figure 1 where the only dependency is $T_1 \rightarrow T_2$.

The reason why the simplest algorithm is too pessimistic is because the algorithm assumes that whenever there is a possibility of violation of the two-phase locking rule, a cycle will occur in the serialization graph. However, as figure 1 shows this is not always the case.

3 AN ALGORITHM THAT GUARANTEES PAIRWISE SERIALIZABILITY

The important observation about the history in figure 1 in particular, and histories in general, in which the low level read lock of a high level transaction T_1 is broken by a low level transaction T_2, is that T_2 and any other transaction T_k that reads data items that are written by T_2 or writes data items that are read by T_2, must serialize after T_1. This observation motivates us to present a simple algorithm based on a scheme of coloring transactions like T_2 and T_k and data items they access with an after-T_1 color (signifying that they must serialize after T_1). The data items are colored with after-T_1 color in order to pass on the transitive dependency to subsequent transactions. If T_1 ever reads or writes an after-T_1 data item, it indicates a cycle in the serialization graph and consequently T_1 is aborted at that time. This algorithm uses two colors for transactions - colorless and an "after" color - and three colors for data items - colorless, an "after" color and a "read-after" color. A transaction T_i becomes after-T_j if T_i is painted with an after-T_j color; a data item x becomes after-T_i or read-after-T_i if it is painted with an after-T_i or read-after-T_i color respectively. Moreover, a transaction or a data item can be painted with more than one color. Suppose a transaction T_i is painted with colors after-T_j, after-T_k and after-T_l. Then the transaction is considered to have turned after-T_j, after-T_k and after-T_l. Same for data items. The algorithm is summarized below:

1. Initially transactions and data items are painted colorless.
2. If a transaction T_j writes a data item x on which a higher level transaction T_i has a read lock, T_j becomes an after-T_i transaction and x an after-T_i data item.
3. If an after-T_i transaction T_j reads a data item z, z becomes a read-after-T_i data item; if T_j writes a data item y, y becomes an after-T_i data item.

High T_1 :	$r_1[x]$				$r_1[z]\ c_1$
Mid T_2 :		$r_2[y]$		$w_2[x]\ c_2$	
Low T_3 :			$w_3[y]\ w_3[z]\ c_3$		

Figure 2 A nonserializable history accepted by simple coloring scheme

4. Any transaction T_k that reads an after-T_i data item becomes after-T_i. If transaction T_k reads a read-after-T_i data item, there is no change in color of either the transaction or the data item.
5. Any transaction T_k that writes a read-after-T_i data item or an after-T_i data item becomes after-T_i.
6. Data items which have been read or written by T_j before T_j turned after-T_i, also turn read-after-T_i or after-T_i, respectively.
7. If at any point T_i tries to read or write a data item that is after-T_i, T_i is aborted.

It is easy to see that this algorithm guarantees pairwise serializability, but not serializability. Pair-wise serializability (Jajodia & Atluri 1992) requires that for any pair of security levels the sub-history restricted to those levels is serializable. We omit a proof due to lack of space.

To see why this algorithm does not guarantee serializability, consider the history shown in figure 2 where Low $\prec$ Mid $\prec$ High. Although this history is non-serializable, the coloring scheme just described does not reject this history. T_3 breaks the low read lock of T_2 first and is colored after-T_2; y is also colored after-T_2 at this time. T_3 then writes z; thus z is colored after-T_2. T_3 then commits. When T_2 breaks the low read lock of T_1, T_2 is colored after-T_1, and both x and y are colored after-T_1. Thus at this time we have the two edges $T_2 \rightarrow T_3$ and $T_1 \rightarrow T_2$. By serialization theory we should have the path $T_1 \rightarrow T_2 \rightarrow T_3$. To do this however, T_2 has to pass on the after-T_1 color from itself to all transactions which are after-T_2 - viz., T_3 in this case. The algorithm just presented does not guarantee the transitivity of the "after" color: It fails to color the data item z after-T_1. As a result the cycle in the history cannot be detected by the algorithm.

To overcome this difficulty, our second protocol uses a third color, the "before" color, to paint transactions T_1 and T_2, to indicate that they are before T_3 in the serialization order. Consequently T_1 will know that T_3 is after-T_1 (we will paint T_1 as before-T_3); if at any time T_1 becomes after-T_3, T_1 is aborted.

In the rest of this paper, we deal only with the second protocol.

4 MLS-SERIALIZIBILITY AND AN ALGORITHM THAT GUARANTEES MLS-SERIALIZIBILITY

Before we give our second protocol, we introduce a new correctness criterion called MLS-serializability.

Definition 1 *An history* H *is* MLS-serializable *if for any transaction T_i, the serialization graph* SG(H) *does not contain a cycle such that T_i is in the cycle and all other transactions in the cycle are at levels dominated by the level of T_i.*

Clearly if we assume that the security levels form a total order, then any MLS-serializable history is also serializable. We will give an example below to show that MLS-serializability is weaker than serializability in general. MLS-serializability seems useful if we do not allow database integrity constraints to span security levels.

We now describe our secure locking protocol with the coloring algorithm. We require a transaction to obtain a lock on a data item in the appropriate mode from the lock manager before accessing the data item. The locking used by a transaction is *strict* on all data items that are at the same level as that of the transaction; i.e., a transaction T_i releases all its locks on data items at security level $L(T_i)$ together, when T_i terminates (see Bernstein, Hadzilacos & Goodman 1987).

When reading a data item x at a lower level, a transaction T_i must acquire a read lock on x. However, if a transaction T_j requests a write lock on x while T_i has a read lock on x, the lock manager takes the read lock away from T_i and grants a write lock to T_j immediately.

Rather than notifying T_i to abort at this point, the lock manager simply starts to keep track of all the data items y that are accessed by T_j. To accomplish this, the lock manager "paints" transaction T_i with a *before-T_j* color, transaction T_j with an *after-T_i* color, any data item z read by T_j with a *read-after-T_i* color, and any data item y written by T_j with an *after-T_i* color. The after color of transaction T_j is propagated in an iterative manner to any transaction that follows T_j and executes an operation that conflicts directly or indirectly with some operation of T_j; the before color of transaction T_i is propagated to all active transactions that are before T_i in the serialization order, in a recursive manner. The following rules are used by the lock manager for coloring transactions after-T_i, before-T_j and data items read-after-T_i or after-T_i:

1. If a transaction T_j writes a data item x on which a higher level transaction T_i has a read lock, T_i is painted with the color before-T_j and T_j is painted with the color after-T_i. The data item x is also painted with after-T_i.
2. If a transaction T_j that is colored after-T_i reads a data item z, z is painted read-after-T_i; if T_j writes a data item y, y is painted after-T_i.
3. When T_j turns after-T_i, T_j inherits all the after-colors of T_i, i.e., if T_i is painted with (say) some after-T_m color, then T_j is also painted with the after-T_m color.
4. When T_i turns before-T_j, T_i inherits all the before-colors of T_j. Further the before-colors of T_j are recursively propagated from T_i to any transaction T_k that is already colored before-T_i, from T_k to transactions T_l that are colored before-T_k and so on.
5. Any transaction T_k that reads an after-T_i data item becomes after-T_i. If T_k reads a read-after-T_i data item, T_k does not change color.
6. Any transaction T_k that writes either a read-after-T_i data item or an after-T_i data item, becomes after-T_i.
7. Once a transaction T_k turns after-T_i, any data items which have been read or written by T_k before it turned after-T_i, turns read-after-T_i or after-T_i, respectively.

If at any point a transaction T_i is colored after-T_k and before-T_k for some transaction T_k, it signifies a cycle in the serialization graph. The lock manager at this point selects a suitable victim T_j (i may equal j) on the cycle such that $L(T_j)$ dominates the level of every other transactions in the cycle and informs T_j to abort thus removing the cycle from the history. If there does not exist such a T_j, the lock manager does not take any action. (Note that in this case the lock manager allows the cycle to remain in the history which nonetheless will still be MLS-serializable. We discuss this further below.)

Figure 3 gives the algorithm for the Trusted Lock Manager module. The Lock Manager is responsible for coloring the data items and the transactions in an appropriate manner. The coloring is done at the time a transactions requests a lock on some data item.

The algorithm works as follows: When a transaction requests a lock to the Lock Manager, the latter first verifies if the lock request violates the security policy, i.e., a write lock cannot be requested on a data item x by a transaction T_j if $L(T_j) \neq L(x)$ and a read lock cannot be requested by a transaction T_j on data item y if $L(T_j) \prec L(y)$. Once the Lock Manager is satisfied that the lock request does not violate the security policy, the Lock Manager tries to satisfy the lock request.

If the requested lock by T_j on x is a write lock, the lock manager first checks if there is a read lock already acquired on x by some T_i such that $L(T_j) \prec L(T_i)$. If there is such a read lock on x, the lock manager paints T_j with an after-T_i color by inserting transaction T_i in After-Set(T_j). During this time if the data item x is colored by some after-T_m or read-after-T_n colors, T_j acquires those colors of x too (i.e. the transactions T_m, T_n are entered in After-Set(T_j)). Next the recursive procedure Propagate-Before-Color() is invoked with the parameters T_j and T_j. The procedure starts by marking T_j as visited and then checks for transactions in After-Set(T_j). T_i is one such transaction in the After-Set(T_j). T_i is not yet marked as visited; as a result the procedure recursively calls itself with parameter T_i and T_j. During this pass T_i is marked as visited. For simplicity let us assume that After-Set(T_i) is empty and T_i is active. Then Before-Set(T_i) is set to the union of Before-Set(T_i) and Before-Set(T_j). Thus T_i is colored before-T_j by inserting T_j in the Before-Set(T_i). If there are other transactions in Before-Set(T_j) those transactions get inserted in Before-Set(T_i).

If After-Set(T_i) is not empty, for all active $T_k \in$ After-Set(T_i), the transactions in Before-Set(T_j) are inserted in Before-Set(T_k). Then this process is repeated for transactions in the After-Set(T_k) and so on till there are no more active transactions to be considered. The intuitive reason behind this recursive before color propagation is that if T_j becomes after some active transaction T_k, T_k should be colored before-T_j, even if there is no direct dependency between T_j and T_k.

Once this "before" color propagation is over the Lock Manager checks if for any of the transactions T_k (including T_j) whose Before-Set was just updated, the transaction T_k is colored both before-T_l as well as after-T_l for some T_l. If this is the case it implies that this transaction T_k is involved in a cycle in the serialization graph and the Lock Manager aborts T_k. Note that this check for transaction T_k is performed from the highest security level going down; this ensures that the highest transaction involved in a cycle is aborted. This strategy ensures that if a high level transaction and a low level transaction are involved in a cycle, the low level transaction is never aborted because of the high level transaction. Sacrificing the high level transaction prevents potential covert channels.

If T_j is not aborted by the above step, the Lock Manager updates the color of the data item x with the after colors of T_j. It also updates the after colors of all data items T_j has written and the read-after colors of all data items T_j has read, with the after colors of T_j. Finally it grants the write lock to T_j.

If there is no read lock on x by some higher level T_i, the Lock Manager finds out if there is any conflicting lock on x by a transaction T_k at the same level as T_j. If there are none, the write

```
TrustedLockManager( )
% This algorithm uses three colors for data items: after, read-after and colorless and three colors
% for transactions: before, after and colorless.
% The Lock Manager maintains two sets of colors for each Tj - the After-Set(Tj)
% and the Before-Set(Tj). Every transaction Tj is colored before-Tj when it is submitted.
% Ti ∈ After-Set(Tj) implies Tj is colored after-Ti. Similar for Ti ∈ Before-Set(Tj).
% The lock manager also maintains two sets of colors for each data item x -
% the After-Color(x) and Read-After-Color(x). Tj ∈ After-Color(x) implies x
% is colored after-Tj. Similar for Read-After-Color(x).

procedure Propagate-Before-Color(Tm,Tn)
% This procedure recursively propagates the before colors of Tn
% to any active Tl ∈ After-Set(Tm)
begin
    mark Tm as visited;
    for all Tk ∈ After-Set(Tm)
        if Tk is not marked as visited, then
            Propagate-Before-Color(Tk,Tn)
        if Tk is active then
            Before-Set(Tk) ← Before-Set(Tk) ∪ Before-Set(Tn)
    endfor
end

repeat
receive (TM,Tj,op,x);
case op do
    Write-Lock:
        If L(TM)≠L(Tj)≠L(x) then
        send (TM,Tj,LockIllegal);
    Read-Lock
        If L(TM)≠L(Tj) OR L(Tj)≺L(x)
        send (TM,Tj,LockIllegal);
endcase
case op do
    Write-Lock:
        if (there is a read lock that is already set on x by some Ti) and L(Tj)≺L(Ti) then
            After-Set(Tj) ← After-Set(Tj) ∪ After-Color(x) ∪ Read-After-Color(x) ∪ Ti;
            Propagate-Before-Color(Tj,Tj);
            Let Sbefore be the set of transactions whose before colors have been
            updated in the previous step, sorted in descending security level
            for each Tk ∈ {Sbefore ∪ Tj} do
              if (After-Set(Tk) ∩ Before-Set(Tk) ≠ ∅) ∧ (∀ Tn ∈ {{Sbefore ∪ Tj}-Tk}, L(Tn) ⪯ L(Tk)) then
                    abort Tk ;
                    remove Tk from all the color sets ;
                    if Tk = Tj then send(TM,Tj-aborted); return endif ;
                endif ;
            After-Color(x) ← After-Color(x) ∪ After-Set(Tj) ;
            for all the data items y which have been read previously by Tj do
                Read-After-Color(y) ← Read-After-Color(y) ∪ After-Set(Tj);
            for all the data items y which have been written previously by Tj do
                After-Color(y) ← After-Color(y) ∪ After-Set(Tj);
```

Figure 3 Trusted Lock Manager Module (continued)

```
        setLock(Tj,x,Write-Lock); send(TM,Tj,LockOK)
    elseif (there is no conflicting lock already set on x) then
        Old-Set(Tj) ← After-Set(Tj)
        After-Set(Tj) ← After-Set(Tj) ∪ After-Color(x) ∪ Read-After-Color(x) ;
        if After-Set(Tj) ≠ Old-Set(Tj) then
            Propagate-Before-Color(Tj,Tj)
        Let S_before be the set of transactions whose before colors have been
        updated in the previous step, sorted in descending security levels
        for each Tk ∈ {S_before ∪ Tj} do
          if (After-Set(Tk) ∩ Before-Set(Tk) ≠ ∅) ∧ (∀ Tn ∈ {{S_before ∪ Tj}-Tk}, L(Tn) ⪯ L(Tk)) then
                abort Tk ;
                remove Tk from all the color sets ;
                if Tk = Tj then send(TM,Tj-aborted)
                    return
                endif ;
            endif ;
        After-Color(x) ← After-Color(x) ∪ After-Set(Tj) ;
        for all the data items y which have been read previously by Tj do
            Read-After-Color(y) ← Read-After-Color(y) ∪ After-Set(Tj) ;
        for all the data items y which have been written previously by Tj do
            After-Color(y) ← After-Color(y) ∪ After-Set(Tj) ;
        setLock(Tj,x,Write-Lock); send(TM,Tj,LockOK)
    else delay(Tj);
Read-Lock:
    if there is no conflicting locks already set on x then
        Old-Set(Tj) ← After-Set(Tj)
        After-Set(Tj) ← After-Set(Tj) ∪ After-Color(x);
        if After-Set(Tj) ≠ Old-Set(Tj) then
            Propagate-Before-Color(Tj,Tj)
        Let S_before be the set of transactions whose before colors have been
        updated in the previous step, sorted in descending security levels
        for each Tk ∈ {S_before ∪ Tj} do
          if (After-Set(Tk) ∩ Before-Set(Tk) ≠ ∅) ∧ (∀ Tn ∈ {{S_before ∪ Tj}-Tk}, L(Tn) ⪯ L(Tk)) then
                abort Tk ;
                remove Tk from all the color sets ;
                if Tk = Tj then send(TM,Tj-aborted)
                    return
                endif;
            endif;
        Read-After-Color(x) ← Read-After-Color(x) ∪ After-Set(Tj);
        for all the data items y which have been read previously by Tj do
            Read-After-Color(y) ← Read-After-Color(y) ∪ After-Set(Tj);
        for all the data items y which have been written previously by Tj do
            After-Color(y) ← After-Color(y) ∪ After-Set(Tj);
        setLock(Tj,x,Read-Lock); send(TM,Tj,LockOK)
    else delay(Tj);
Unlock:
    release(Tj,x); send(TM,Tj,UnlockOK);
    awake transactions that are no more conflicting, if any;
endcase
forever
```

Figure 3 Trusted Lock Manager Module

receive(TM,T_i,op,x) :	receives a lock or unlock request op from the transaction manager TM on behalf of the transaction T_i on data item x
send(TM,T_i,msg) :	send the message msg pertinent to transaction T_i to the transaction manager TM for T_i)
setLock(T_i,x,ltype) :	sets the lock of type ltype on data item x, requested by transaction T_i
release(T_i,x) :	release the lock held by T_j on data item x
delay(T_i) :	puts the transaction T_i in a wait queue for a lock

Figure 4 Functions Invoked by Trusted Lock Manager

High T_1 :	$r_1[x]$				$r_1[z]$		
Mid T_2 :		$r_2[y]$					$w_2[x]$
Low T_3 :			$w_3[y]$	$w_3[z]$	c_3		

Figure 5 An example showing why T_1 must commit after T_2

lock should be granted. Before actually granting the lock, the Lock Manager updates the after colors of T_j with the after color or read-after color of x. This is because the data item x may already be after-T_k or read-after-T_k for some T_k and the transaction T_j by writing x, gets colored after-T_k. If T_j does get colored after-T_k (owing to accessing a colored x), the transaction T_k gets colored before-T_j. T_k inherits all the before-colors of T_j and this is propagated recursively to all transactions T_m that are before-T_k. As before, if some transaction gets colored both before-T_n as well as after-T_n, that transaction is aborted at this time. This includes T_j. Next the after color of x is updated with the after colors of T_j and finally the lock is granted.

If there is a conflicting lock, the transaction T_j is delayed.

For read lock requests, the Lock Manager proceeds as in the case of write lock requests. However, the lock manager has to check only for conflicting locks; there is no need for the Lock Manager to check for higher level transactions with low read locks on x. Also the set Read-after-color(x) is updated in this case.

When the transaction T_j requests the Lock Manager to release a lock on x, the Lock Manager, after verifying that the request does not violate the security policy, releases the lock. Next it selects a transaction that is waiting for a lock on x to be granted and performs the lock request operation for that transaction.

Note that along with the Trusted Lock Manager, there is another trusted component in the system which coordinates the lock requests by transactions in a strict 2PL manner and which ensures that when a transaction T_k tries to commit, if T_k is after-T_i for some T_i such that $L(T_i) \prec L(T_k)$ or there is some T_j such that T_k is before-T_j and $L(T_j) \prec L(T_k)$ then the commit of T_k is delayed till after T_i and T_j terminate. The reason this is done is to avoid possible covert channels as exemplified by the history shown in figure 5.

T_1:	$r_1[a]$				$r_1[d]$
T_2:			$r_2[b]$ $r_2[c]$		
T_3:		$w_3[a]$ $w_3[b]$ c_3			
T_4:				$w_4[c]$ $w_4[d]$ c_4	

Figure 6 A history that is nonserializable. but MLS-serializable

The history in figure 5 is not serializable as we have the cycle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$. If we allow T_1 to commit after executing $r_1[z]$ but before $w_2[x]$ is executed, then to prevent non-serializability we will have to abort T_2 when it executes $w_2[x]$. However this opens up a covert channel from level High to level Mid. To prevent this we cannot abort T_2.

To address this problem, our protocol does not allow T_1 to commit so long as T_2 is active and aborts transactions from higher security levels to lower security levels (in this order). Data item z is already colored after-T_2 by virtue of its being written by T_3 (which is after-T_2). Thus T_1 is colored after-T_2 when T_1 reads z. At this stage the commit of T_1 is delayed till T_2 commits. When $w_2[x]$ is executed T_1 is colored before-T_2. Since After-Set(T_1) $\cap$ Before-Set(T_1) $\neq \emptyset$, we abort T_1 and not T_2.

We next give an example, taken from (Sankarachary 1996), to show under what circumstances this protocol fails to yield serializable histories: Suppose that there are four transactions $T_1, \ldots,$ T_4 such that $L(T_4) \prec L(T_3)$, $L(T_3) \prec L(T_1)$, $L(T_3) \prec L(T_2)$, and $L(T_1)$ and $L(T_2)$ are incomparable. Data items a and b are at the same level as $L(T_3)$ and data items c and d are at the same level as $L(T_4)$.

Consider now the history shown in figure 6. We do not abort T_1 when its read lock on data item a is broken by T_3's write operation; neither do we abort T_2 when its read lock on c is broken by T_4. Instead we postpone the abort of T_1 or T_2 till such point as a cycle is imminent in the serialization graph, i.e., till the execution of $r_1[d]$ by T_1. Although our algorithm detects the existence of the cycle in the serialization graph, it still does not abort T_1 because doing so will open up a covert channel (T_1 is aborted due to T_2's read operations) between $L(T_1)$ and $L(T_2)$. Note however that this history is MLS-serializable.

5 IMPLEMENTATION ISSUES

Our protocol can be implemented within a Trusted Lock Manager. A simple implementation is as follows: The Lock Manager maintains a table, called the ***data status*** table, the number of columns in which equals the number of database items, and the number of rows equals the number of active transactions. Each cell in the table contains two bits and indicates the three colors of a data item with respect to transaction T_i, viz., colorless (00), read-after-T_i (10) and after-T_i (11).

When a new transaction T_j arrives, a row corresponding to T_j is added to the table and all its entries are initialized to 00. Whenever a data item x turns read-after-T_j, the cell in the jth-row and xth-column is set to 10 and when x turns after-T_j, the cell is set to 11.

The Lock Manager also maintains two sets associated with each transaction T_j - the Before-Set(T_j) and the After-Set(T_j). Initially After-Set(T_j) is empty and the transaction identifier T_j is inserted in Before-Set(T_j). When transaction T_j becomes after-T_i, T_i is added to After-Set(T_j). When T_j becomes before-T_k for some transaction T_k, T_k is inserted in Before-Set(T_j).

The data status table as well as each of the sets Before-Set(T_j) and After-Set(T_j) reside in the trusted part of the lock manager and are not accessible to any transaction or other untrusted components; hence these cannot be exploited as covert channels.

As and when transactions add on new colors, the various transaction identifiers are inserted in the sets. Also the cells in the data status table are set from one bit pattern to another.

The jth row in the data status table and the Before-Set(T_j) and After-Set(T_j) for a transaction T_j can be garbage collected in the following cases: (1) If there is no active transaction T_i such that T_i is colored before-T_j or after-T_j. (2) If transaction T_j is aborted. These conditions guarantee that the protocol does not miss out any dependency in which T_j played a part along with any currently active transaction.

6 COMPARISON WITH RELATED WORKS

We now show how our protocol compares with the orange locking protocols given in (McDermott & Jajodia 1993).

6.1 Optimistic Orange Locking

In the optimistic orange locking protocol (OOL), transactions are serialized at each level by two phase locking. A high transaction T_j sets read locks on low data items in order to read the data. If a low transaction T_i then tries to set a write lock on any of these data items, T_i's write lock request is immediately granted and T_j's low read lock is converted to an orange lock. The high transaction T_j is aborted if any of its low read locks is converted to an orange lock before T_j performs the first unlock operation.

OOL is more conservative than our protocol, as illustrated by the next example.

Example 1 Consider the history shown in figure 7. Transactions T_1 and T_3 are high transactions, while T_2 is a low transaction; y and p are low data items, while x, z, q, l and t are high data items. The operations of the transactions and the order in which these are submitted are shown in the figure. Under OOL, when T_2 writes to p, the read lock by T_1 on p is converted to an orange lock. Since this occurs before the first unlock operation of T_1 (which can occur only after $r_1[t]$), OOL aborts the transaction T_1, even though no cycle is formed in the serialization graph.

With our protocol, T_2 becomes after-T_1 when it writes to p. The data item p also turns after-T_1. T_1 is colored before-T_1 and before-T_2. When T_3 reads p, T_3 becomes after-T_1. T_1 becomes before-T_3. The data item l becomes after-T_1 when T_3 writes l. When T_1 reads t, it does not read any after-T_1 or after-T_2 or after-T_3 value and hence T_1 is not aborted. □

High: $r_1[y]\ r_1[p]\ r_1[x]\ w_1[z]\ w_1[q]$ $\quad\quad$ $r_3[p]\ w_3[l]\ c_3\ r_1[t]\ c_1$

Low: $\quad\quad$ $w_2[p]\ c_2$

Figure 7 A serializable history rejected by the optimistic orange locking protocol, but accepted by our protocol

6.2 Conservative Orange Locking

The conservative orange locking protocol (COL) tries to improve upon OOL by not aborting the high transaction as soon as a conflicting lock is requested by a low transaction; instead the orange locks are used to identify the low transaction from which the high transaction can safely read.

Briefly, COL assumes that a high transaction T_i predeclares the set E_i of lower level data items that it wants to read as well as the set W_i of data items that it wants to write. The execution of a transaction T_i proceeds in two phases. In the first phase, T_i tries to read the set of lower level data items into a local workspace. It begins by marking as empty the local workspace reserved for each element of E_i. While some element x is still marked as unread, T_i submits read-down operations for those unread data items. If a read lock can be placed on x, it is read into the local workspace. If no read lock can be placed, then T_i waits. When all the lower level data has been read into the local workspace, T_i is said to reach its ***home-free point.*** If before T_i reaches its home free point, a lower level transaction T_j acquires a write lock on a data item y already read by T_i, the read lock by T_i on y is converted into an orange lock and y is marked as unread in T_i's local workspace. T_i is then placed on a queue Q_j associated with T_j, so that T_i can read y from T_j, after the latter commits. When T_i reaches its home free point, either all the elements of E_i have been read locked and read into T_i's local workspace or orange locked and read into the local workspace. After that, T_i follows two phase locking and reads and writes data items at its own security level.

Example 2 In this example, there are three transactions: T_1 and T_3 of level high and T_2 of level low, as shown in figure 8. Data items x, y and z are low level data items, while t is a high level data item. T_1 reads x, y and z and writes t; T_3 reads z and writes t; T_2 writes to y and z. As in the previous example, each transaction reaches its home free point after it has read all its lower level data.

This history is accepted by COL scheduler, although it has a cycle. T_1 manages to read-lock all low data items and reach its home free point before T_2 acquires write locks on data items y and z. T_2 does not "override" any of the low read lock of T_1 and, thus, none of the low-read locks of T_1 gets converted to orange locks. The history, nontheless, has a cycle because COL fails to ensure the two-phase nature of all transactions in the system.

Note that this history is correctly rejected by our protocol as follows: When T_2 writes y, T_2 becomes after-T_1, y becomes after-T_1; T_1 is already before-T_1 and becomes before-T_2. T_2 writes to z; thus z becomes an after-T_1 value. T_3 reads z; thus T_3 becomes after-T_1; T_1 becomes before-T_3. When T_3 writes t, t becomes an after-T_1 value. When T_1 tries to acquire a write lock on t, T_1 becomes after-T_1; the Lock Manager detects that the intersection of Before-Set(T_1) and

High T_1: $r_1[x]\ r_1[y]\ r_1[z]\ HFP_1\ w_1[t]\ c_1$
High T_3: $r_3[z]\ HFP_3\ w_3[t]\ c_3$
Low T_2: $HFP_2\ w_2[y]\ w_2[z]\ c_2$

HFP_i = Home free point of transaction T_i

High: $r_1[x]\ r_1[y]\ r_1[z]\ HFP_1$ $r_3[z]\ HFP_3\ w_3[t]\ c_3\ w_1[t]\ c_1$

Low: $w_2[y]\ w_2[z]\ c_2$

Figure 8 A nonserializable history accepted by conservative orange locking

After-Set(T_1) is non-empty. Hence the protocol aborts T_1, i.e., rejects the history shown in figure 8. □

6.3 Reset Orange Locking

The Reset Orange Locking (ROL) protocol is very similar to COL. In ROL, when a low-read lock of a higher level transaction T_i is overwritten by a low level transaction T_j, the corresponding low data item x is orange locked and marked unread in T_i's local workspace. However, unlike in the COL protocol, T_i is not queued up in T_j's queue Q_j to read x from Q_j. Instead T_i at some later time asks the scheduler to re-acquire the low read lock. T_i's read request is queued waiting for a chance to read according to the normal rules of two-phase locking. The read may have to wait for other writes besides T_j's. Further, if another low transaction T_k tries to write lock the data item x after T_i has reacquired the low read lock, T_k overrides T_i's low read lock.

T_i reaches its home free point when it has read-locked all low data and read them into its local workspace or orange locked all low data and read them into its local workspace. Once T_i reaches the home free point it releases the locks on the read-down data items and performs the rest of its processing using conventional strict two-phase locking.

It is clear that in the ROL protocol a high level transaction is not two phase; consequently, as in COL, there is no guarantee that histories produced by the ROL scheduler are serializable.

7 CORRECTNESS OF THE ALGORITHM

We assume that the reader is familiar with serializability theory as explicated in (Bernstein et al. 1987) and adopt the terminology and notation contained therein.

Our protocol requires each transaction to lock a data item in an appropriate mode before accessing it and eventually unlocks it before completing (well-formed property). This is expressed by the following property:

Property 1 *Let $o_i[x]$ denote either a read or a write operation on data item x by transaction T_i, $ol_i[x]$ denotes the locking operation (i.e. read or write lock) on x and $u_i[x]$ denote the corresponding unlock operation. Given a history H, if $o_i[x] \in H$, then both $ol_i[x]$, $u_i[x] \in H$ and $ol_i[x] <_H o_i[x] <_H u_i[x]$.*

The locking used by a transaction is strict on all data items that are at the same level as that of the transaction; i.e. a transaction T_i releases all its locks on data items at security level $L(T_i)$ only after executing a commit or an abort. This property is expressed as follows:

Property 2 *For any pair of data items x and y accessed by a transaction T_i such that $L(T_i) = L(x) = L(y)$, if $ol_i[x]$ and $u_i[y]$ exists in H and either c_i or a_i exists in H, then either $ol_i[x] <_H c_i <_H u_i[y]$ or $ol_i[x] <_H a_i <_H u_i[y]$.*

The serialization graph $SG(H)$ for history H is defined as a directed graph in which (1) Each committed transaction in H is a node in $SG(H)$, and (2) There is a directed edge $T_i \rightarrow T_j$ in $SG(H)$ whenever H contains an operation in T_i that precedes and conflicts with an operation in T_j.

We distinguish between two different kinds of edges in the serialization graph $SG(H)$, viz., $\overset{a}{\rightarrow}$, and $\overset{u}{\rightarrow}$.

Definition 2 *Let H be a history over $\{T_1, \dots T_i, \dots T_j, \dots T_n\}$.*

1. *If there is an operation $o_i[x] \in T_i$ that precedes and conflicts with an operation $o_j[x] \in T_j$, and transaction T_i is colored before-T_j, and transaction T_j is colored after-T_i, then the directed edge $T_i \overset{a}{\rightarrow} T_j$ is in $SG(H)$.*
2. *If there is an operation $o_i[x] \in T_i$ that precedes and conflicts with an operation $o_j[x] \in T_j$, and T_i unlocks some data items before T_j locks them in history H, then the directed edge $T_i \overset{u}{\rightarrow} T_j$ is in $SG(H)$.*

Note that all edges $T_i \rightarrow T_j$ in the serialization graph for a history H can be labeled either with $T_i \overset{a}{\rightarrow} T_j$ or $T_i \overset{u}{\rightarrow} T_j$.

Lemma 1 *Let $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ be any path in $SG(H)$ and let T_1 be the last transaction to commit among $\{T_1, \dots, T_n\}$. Then there exists $T' \in \{T_1, \dots, T_n\}$ such that T_1 is before-T' and T_n is after-T'.*

Proof. Proof is by induction on n, the number of transactions in the path. First, we show that the Lemma is true for n=2. Let $T_1 \rightarrow T_2$ be in $SG(H)$. Since T_1 commits last, it can only be the case that T_1 had a read lock on some data item x that was broken by a write lock from T_2; otherwise T_1 would violate the strict 2PL protocol. Then by definition, T_2 is colored after-T_1 and T_1 is colored before-T_1 and the edge is of type $T_1 \overset{a}{\rightarrow} T_2$. Hence $T' = T_1$ satisfies the lemma.

Let us assume that the lemma holds for paths with n transactions. By the inductive hypothesis, given any path $T_1 \rightarrow \dots \rightarrow T_n$ on which T_1 is the last transaction to commit, there exists $T' \in \{T_1, \dots, T_n\}$ such that T_1 is before-T' and T_n is after-T'.

Consider a path consisting of n+1 transactions, and in particular consider the type of the edge $T_n \rightarrow T_{n+1}$. Either $T_n \xrightarrow{a} T_{n+1}$ or $T_n \xrightarrow{u} T_{n+1}$.

Let us first consider the case $T_n \xrightarrow{a} T_{n+1}$. Since T_n is after-T$'$ (by the inductive hypothesis), there must be at least one operation $o_n[x]$ in T_n such that $o_n[x]$ reads or writes an after-T$'$ data item x; moreover after $o_n[x]$ is executed, any data item read or written by T_n is colored read-after-T_n or after-T_n respectively.

Since $T_n \xrightarrow{a} T_{n+1}$ in SG(H), there must be at least a read operation $r_n[y]$ in T_n such that the read lock of T_n is broken by T_{n+1}, and T_{n+1} and y turn after-T_n.

Now there are two cases: (a) $o_n[x] <_H r_n[y]$ or (b) $r_n[y] <_H o_n[x]$. We consider each of these in turn.

If case (a) is true, T_n turns after-T$'$ before T_{n+1} turns after-T_n. Once T_n is colored after-T$'$ any data item read by T_n is colored read-after-T$'$. When T_{n+1} writes data item y, T_{n+1} is colored after-T_n and after-T$'$ as well. Hence, the Lemma holds since T_1 is before-T$'$ and T_{n+1} is after-T$'$.

If case (b) holds, T_{n+1} turns after-T_n before T_n turns after-T$'$. When T_n turns after-T$'$ it propagates recursively the color before-T_n to T$'$. And also to T_1 since T_1 was already colored before-T$'$. Hence, the Lemma holds since T_1 is before-T_n and T_{n+1} is after-T_n.

Let us next consider the case $T_n \xrightarrow{u} T_{n+1}$. T_n commits first, otherwise T_n would violate the strict 2PL protocol . As there is a dependency between T_n and T_{n+1}, it must be the case that there is some o_n[x] that preceeds and conflicts with some o_{n+1}[x]. If o_n[x] is r_n[x], then data item x will be colored read-after-T$'$. In this case o_{n+1}[x] has to be a w_{n+1}[x] and T_{n+1} becomes after-T$'$. If o_n[x] is a w_n[x], o_{n+1}[x] can be either r_{n+1}[x] or w_{n+1}[x].In either case x is after-T$'$ and hence T_{n+1} is also after-T$'$. Hence, the Lemma holds since T_1 is before-T$'$ and T_{n+1} is after-T$'$. □

Theorem 1 *Any history generated by our protocol is MLS-serializable.*

Proof. Assume SG(H) contains the cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in $SG(H)$ such that the security levels $L(T_1), \ldots, L(T_n)$ are totally ordered. There must be some transaction T_i on the cycle at the security level L(T_i) such that L(T_i) dominates the security levels of all the other transactions in the cycle. Then according to our protocol, T_i is the transaction to commit last compared with all the other transactions participating in the cycle. Consequently the cycle can be re-written as: $T_i \rightarrow \ldots \rightarrow T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \ldots \rightarrow T_i$

By lemma 1, it follows that there exists $T' \in \{T_1, \ldots, T_n\}$ such that T_i is colored before-T' and T_i is colored after-T'. But in such a case T_i should have been aborted by our protocol and the cycle could not have resulted. This is a contradiction. □

Corollary 1 *Suppose that the set S of security levels forms a total order. Then any history generated by our protocol is serializable.*

Proof. Follows immediately from Theorem 1. □

8 CONCLUSIONS

In this paper, we have described two lock based concurrency control algorithm for multilevel secure transactions. Both protocol use single version data and are based on a method of "painting" transactions and data items to prevent certain cycles. These algorithms are secure because they do not require a lower level transaction to wait or abort because a higher level transaction is accessing the same data in conflicting mode and, moreover, the second protocol does not abort a transaction resulting from an action of a transaction at an incomparable level.

ACKNOWLEDGEMENTS

The work of Sushil Jajodia was partially supported by National Science Foundation under grants IRI-9303416, IRI-9633541 and INT-9412507 and by National Security Agency under grants MDA904-96-1-0103 and MDA904-96-1-0104. The work of Luigi V. Mancini was partially supported by the Italian M.U.R.S.T. The work of Indrajit Ray was partially supported by National Science Foundation under grant IRI-9303416

REFERENCES

Ammann, P., Jaeckle, F. & Jajodia, S. (1995), 'Concurrency control in secure multi-level databases via a two-snapshot algorithm', *Journal of Computer Security* **3**(3), 87–113.

Ammann, P. & Jajodia, S. (1992), A timestamp ordering algorithm for secure, single-version multilevel databases, *in* C. E. Landwehr, ed., 'Database Security, V: Status and Prospects', North-Holland, Amsterdam, pp. 191–202.

Ammann, P. & Jajodia, S. (1994), Planar lattice security structures for multilevel replicated databases, *in* T. F. Keefe & C. E. Landwehr, eds, 'Database Security VII: Status and Prospects', North-Holland, Amsterdam, pp. 125–134.

Ammann, P., Jajodia, S. & Frankl, P. (1996), 'Globally consistent event ordering in one-directional distributed environments', *IEEE Trans. on Parallel and Distributed Systems* **7**(6), 665–670.

Bernstein, P. A., Hadzilacos, V. & Goodman, N. (1987), *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading.

Denning, D. E. (1982), *Cryptography and Data Security*, Addison-Wesley, Reading.

Gray, J. & Reuter, A. (1993), *Transaction Processing: Concept and Techniques*, Morgan Kaufmann, San Mateo, CA.

Jajodia, S. & Atluri, V. (1992), Alternative correctness criteria for concurrent execution of transactions in multilevel secure database systems, *in* 'Proc. IEEE Symposium on Security and Privacy', Oakland, CA, pp. 216–224.

Jajodia, S. & Kogan, B. (1990), Transaction processing in multilevel secure databases using replicated architecture, *in* 'Proc. IEEE Symp. on Research in Security and Privacy', Oakland, CA, pp. 369–383.

Kang, I. E. & Keefe, T. F. (1995), 'Transaction management for multilevel secure replicated databases', *Journal of Computer Security* **3**, 115–145.

Keefe, T. F. & Tsai, W. T. (1990), Multiversion concurrency control for multilevel secure database systems, *in* 'Proc. IEEE Symp. on Research in Security and Privacy', Oakland, California, pp. 369–383.

Lamport, L. (1977), 'Concurrent reading and writing', *Comm. ACM* **20**(11), 806–811.

McDermott, J. & Jajodia, S. (1993), Orange locking: Channel-free database concurrency control via locking, *in* B. Thuraisingham & C. Landwehr, eds, 'Database Security, VI: Status and Prospects', North-Holland, Amsterdam, pp. 267–284.

Reed, D. P. & Kanodia, R. K. (1979), 'Synchronization with eventcounts and sequencers', *Comm. ACM* **22**(5), 115–123.

Sankarachary, K. B. (1996), Concurrency control in multilevel secure database management system, based on serialization graph testing, Master's thesis, Pennsylvania State University.

Schaefer, M. (1974), Quasi-synchronization of readers and writers in a secure multi-level environment, Technical Report TM-5407/003, System Development Corporation.

BIOGRAPHY

Sushil Jajodia is Director of Center for Secure Information Systems and Professor of Information and Software Systems Engineering at the George Mason University, Fairfax, Virginia. His research interests include information security, temporal databases, and replicated databases. He has published more than 150 technical papers in the refereed journals and conference proceedings and has edited or coedited ten books, including *Multimedia Database Systems: Issues and Research Directions*, Springer-Verlag Artificial Intelligence Series (1996), *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press (1995), and *Temporal Databases: Theory, Design, and Implementation,* Benjamin/Cummings (1993). He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of Information Security. The URL for his web page is http://www.isse.gmu.edu/~csis/faculty/jajodia.html.

Luigi Mancini received the Laurea degree in Computer Science from the University of Pisa, Italy, in 1983, and the PhD degree in Computer Science from the University of Newcastle upon Tyne, Great Britain, in 1989. From 1989 to 1992, he was an Assistant Professor at the Dipartimento di Informatica of the University of Pisa. From 1992 to 1996 he was an Associate Professor of the Dipartimento di Informatica e Scienze dell'Informazione of the University of Genoa. Since 1996 he has been an Associate Professor of the Dipartimento di Scienze dell'Informazione of the University "La Sapienza" of Rome. His research interests include distributed algorithms and systems, transaction processing systems, and computer and information security.

Indrajit Ray is a Ph.D. student at the Center for Secure Information Systems within the School of Information Technology at George Mason University, Fairfax, Virginia. He had his Master of Engineering in Computer Science and Engineering from Jadavpur University, India in 1991 and Bachelor of Engineering in Computer Science and Technology from B.E. College, Calcutta University, India in 1988. Mr. Ray's research interests include distributed database systems, transaction processing and information systems security.