12

A Locking Protocol for MLS Databases Providing Support For Long Transactions

Shankar Pal[1] (pal@cse.psu.edu)
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802.

ABSTRACT

This paper presents a locking protocol for multilevel secure databases which produces schedules that are 1–copy serializable and satisfy the MLS property. It executes read downs on an old database snapshot; transactions accessing objects at their own level manipulate the most recently committed database, and are synchronized using weak and strong locks, somewhat like strict 2PL. At regular intervals, known as the version period, the old snapshot is upgraded to the most recently committed database version.

The protocol supports long read–only transactions that perform their read downs during a single version period but continue to execute same–level reads thereafter. The protocol also supports long update transactions that perform a batch of read downs close to their commit point, and commit during the same version period.

1 Introduction

A *long transaction* accesses a large number of objects in the database and executes for a long period of time [9]. These transactions contain predominantly read operations, and many of them are off–line. Examples of such transactions are generating monthly account statements at a bank, browsing through a large file, and processing purchase orders [16].

The long duration of these transactions causes severe performance bottlenecks. This is all the more serious in multilevel secure databases, as the prevention of signaling channels limits the size of the transactions that can successfully complete.

Many concurrency control protocols have been proposed in the literature for multilevel secure databases. These protocols use a variety of techniques and ensure read consistency or one–copy serializability [5]. They are secure in the sense of the MLS property [7, 10].

Orange locking [13] and optimistic protocol [6] abort a transaction T which reads down an object (i.e., reads a value at a strictly dominated level) that is subsequently overwritten before T's completion. Although this strategy avoids signaling channels, the cost is the possible starvation and the greater likelihood of aborting high level readers. Furthermore, orange locking restricts all read downs to be executed at the beginning of a transaction. Thus, these protocols allow a transaction to execute a few read downs within a very short period of time.

Two single version timestamp ordering (TO) protocols have been proposed in [2], which delay read downs or transaction commits by an arbitrary amount of time to synchronize with low level writers. Such delays may cause later operations from high level transactions to become tardy.

[1]Current address: Microsoft Corp., One Microsoft Way, Redmond, WA 98052.

D. L. Spooner et al. (eds.), *Database Security IX*

Multiversion TO based methods [11, 12] avoid high–reader–low–writer conflicts by executing read downs on old versions of objects. Multiple versions have the overhead of extra storage space and the extra cost of accessing the old versions.

In [1], a timestamp–oriented protocol has been proposed that uses two snapshots of the database which are used by read downs. In addition, a working database is manipulated by committed updates and same–level reads (i.e., read operations from transactions at the same level as the accessed objects). In this protocol, transactions that read down must commit within a deadline which grows exponentially shorter at higher security levels.

The above–mentioned protocols are not particularly suitable for the execution of long transactions. For example, suppose a long transaction T needs to read an object x at a strictly dominated level. Using a single–version TO based protocol, if x is updated frequently during T's lifetime, then T's access to x may be tardy in every execution of T; as a result, T may never be able to commit. The multiversion TO based methods potentially require an unbounded number of versions in the presence of long transactions, which is infeasible.

In this paper, we propose a locking protocol for multilevel secure databases which requires an old snapshot and the most recently committed version of the database. Implementation of locking protocols is well understood and efficient for nonsecure databases; we expect similar convenience and efficiency in secure databases. Having a limited number of versions requires less storage and less access time, and is desirable.

Read downs are executed on the old snapshot, while transactions manipulating objects at their own level execute on the most recently committed version and are synchronized somewhat like strict 2PL [5] but using weak and strong locks. (We choose strict 2PL for the convenience of our presentation, although we can very easily incorporate more sophisticated algorithms, e.g., altruistic locking [16].) At regular intervals, known as the *version period*, the snapshot is upgraded to the most recently committed version, so that read downs can observe the updates accumulated during the previous version period.

The protocol allows long read–only transactions to commit whenever their read downs are executed during a single version period; the transactions may have any number of same–level reads before and after the version period of their read downs. The protocol also supports the execution of long update transactions that have a group of read downs just before their commit point, or none at all.

In [3], the authors have proposed an extension of their two–snapshot algorithm to support the execution of long read–only transactions. A long transaction T reads older snapshots even at its own level; this sets a deadline by which T must complete, failing which T is aborted. Furthermore, they assume that data accesses are executed within a kernel, although they do not provide any details of how the implementation could be done. By contrast, the protocol proposed in this paper allows a long read–only transaction to continue reading values from its own level, under certain conditions, for as long as it needs to; this paper also presents an untrusted snapshot maintenance scheme, based on which the concurrency control protocol is designed. What we present in this paper are "core" ideas that can be easily extended to algorithms utilizing multiple snapshots or variable snapshot lifetimes.

The locking protocol presented in this paper is an extension of the one proposed in [14]. The protocol in [14] requires a transaction T to complete *all* its reads, following its first read down, within the same version period as T's first read down. In this paper, we discuss techniques to overcome this limitation: the same–level reads can occur in arbitrarily later

version periods as long as all of T's read downs are performed during a single version period. This improvement benefits read–only transactions in particular, especially the long ones.

The proposed protocol can be implemented using an untrusted scheduler at each security level. Transactions are assumed to predeclare their read sets at their own level (i.e., the collection of objects they intend to read at their own level). The schedules produced are one–copy serializable and strict, and satisfy the MLS property.

The rest of the paper is organized as follows. In Section 2, we introduce the theoretical background and our terminology. In Section 3, we provide the motivation behind the design of the proposed locking protocol. In Section 4, we present our locking protocol, describe its features, and explain a snapshot maintenance scheme. We present the proof of serializability and an informal proof of the MLS property in Section 5. We then conclude the paper with a discussion in Section 6.

2 Basic Concepts

2.1 Multiversion Serializability

We introduce some concepts of multiversion serializability as discussed in [5]. We consider only read and write operations on the objects in the database. A read operation from transaction T_k on a version x_i of an object x is denoted as $r_k[x_i]$, while a write operation from T_k that creates the version x_k is denoted as $w_k[x_k]$.

Let S be any multiversion schedule. The committed projection of S, denoted $\mathrm{C}(S)$, is the subsequence of operations in S from committed transactions. A transaction T_i is said to precede another transaction T_j in S if all operations of T_i occur before the first operation of T_j in S. If T_i precedes T_j in S, then they do not execute concurrently in S.

A multiversion schedule S is said to be *serial* if for every pair of transactions T_i and T_j in S, either T_i precedes T_j, or vice versa. A serial multiversion schedule S is *one–copy serial* if for every object x and for every transaction T_i that reads x, either T_i reads x from itself, or T_i reads x from the last transaction preceding T_i in S which wrote into any version of x. A multiversion schedule S is said to be *one–copy serializable* if there exists a one–copy serial schedule which is equivalent to (i.e., contains the same operations as) $\mathrm{C}(S)$.

We define a *version order* for a multiversion schedule S and each object x as a total order on the committed versions of x. The initial version of each object is denoted with subscript zero. We choose a particular version order in which version x_i precedes version x_j, denoted as $x_i \prec x_j$, if both T_i and T_j are committed in S and x_i is created before x_j. A *version order for* S is the union of the version orders for S corresponding to all the objects.

A *multiversion serialization graph* $\mathrm{MVSG}(S, \prec)$ corresponding to a multiversion schedule S and the version order $\prec$ for S has a node for each committed transaction in S, and two types of edges inserted as follows:

- A conflict edge exists in $\mathrm{MVSG}(S, \prec)$ from transaction T_i to T_j if $\mathrm{C}(S)$ contains the operations $w_i[x_i]$ and $r_j[x_i]$, in that order, for some object x.
- If $\mathrm{C}(S)$ contains operations $r_k[x_j]$ and $w_i[x_i]$, where i, j and k are distinct and x is some object, then a version order edge exists in $\mathrm{MVSG}(S, \prec)$ from T_i to T_j if $x_i \prec x_j$, otherwise a version order edge from T_k to T_i occurs in $\mathrm{MVSG}(S, \prec)$.

Whether or not a multiversion schedule S is one–copy serializable can be determined by inspecting MVSG(S, $\prec$). This is stated in the following theorem found in [5].

Theorem 2.1: [Bernstein et al.] A multiversion schedule S is one–copy serializable if and only if MVSG(S, $\prec$) is acyclic.

In the protocol proposed in this paper, transactions may become deadlocked. If a transaction is blocked for a timeout period, it is aborted. This method of deadlock detection, although imprecise, has the advantage of simplicity and efficiency. For accurate deadlock detection, we maintain a waits–for graph at each security level to record transaction blocking resulting from concurrency control. The *waits–for graph* (WFG) for a schedule S is a digraph containing a node for each active transaction in S and edges inserted as follows: an edge from T_i to T_j in the graph indicates that T_i is waiting for a lock to be released by T_j. A cycle in WFG indicates the presence of a deadlock; periodically, the WFG is tested for cycles, and cycles are broken by aborting transactions.

2.2 Deferred Update

In the deferred update recovery mechanism [8], each transaction T has an *intentions list* where it records its update operations. T is started with an empty intentions list.

The shared database contains only committed values. When T executes $w[x]$, the update is appended to T's intentions list, instead of applying it immediately to the shared database. If T executes $r[x]$ and x has been updated by T, then x's value is returned from T's intentions list. Otherwise, the value of x is obtained directly from the shared database.

When T commits, its intentions list is replayed on the shared database. If T aborts instead, then T's intentions list is simply discarded.

2.3 Security Model

We consider a security model containing n security levels $L_1, L_2, \ldots, L_n$ forming a lattice, in which the strictly dominates relation defines the partial order "$<$". A transaction belongs to a single security class which is that of the subject issuing it. The objects in the database are at the sensitivity levels $L_1, L_2, \ldots, L_n$. The security level of a transaction T is denoted by level(T). Similarly, the sensitivity level of an object x is denoted by level(x). Accesses to objects are controlled by the mandatory security policies expressed as follows [4]:

- *Simple Security Property:* A transaction T at level L_i is allowed to read the value of an object x at level L_j if and only if $L_j \leq L_i$.
- *(Restricted) Star Property:* A transaction T at level L_i is allowed to write into an object x at level L_j if and only if $L_j = L_i$.

The theory of *noninterference* [7] can be used to analyze and reveal covert channels in a deterministic system. A subject $\mathcal{S}_1$ is said to be noninterfering with another subject $\mathcal{S}_2$ if the execution of $\mathcal{S}_2$ does not depend upon that of $\mathcal{S}_1$. The MLS property [7] states that if subject $\mathcal{S}_1$ is at a higher security level than subject $\mathcal{S}_2$, then $\mathcal{S}_1$ does not interfere with the

execution of $\mathcal{S}_2$. Therefore, a concurrency control protocol satisfies the MLS property if in every execution allowed by the protocol, $\mathcal{S}_2$ executes the same way as when all events of $\mathcal{S}_1$ are removed from the execution.

3 Motivation of the Protocol

In this section, we motivate the design of the proposed locking protocol. We assume the following. The database consists of an old snapshot, on which read downs are executed, and the most recently committed version, which is manipulated by committed updates and same–level reads. The latter version accumulates all updates during a version period.

The version period is advanced at regular intervals to make recent updates available to the strictly dominating levels. At the beginning of a new version period, the most recently committed database is copied into the old snapshot.

Each security level has a scheduler that implements a locking based concurrency control protocol; the scheduler is aware of the concurrency control only at its own level.

3.1 Read Downs

In such systems, all read downs from a transaction must complete in a single version period [1, 14], as the next example shows.

Example 3.1: Consider a database containing two objects, x and y, both at the sensitivity level L_1. These objects are accessed by transactions T_1 and T_2 at security levels L_1 and L_2 respectively, where $L_1 < L_2$[2]. Furthermore, T_2 reads both x and y while T_1 updates them both. Consider the schedule S_1 below (we do not show commit operations in schedules for convenience).

T_1 : $w_1[x]\ w_1[y]\ c_1$
T_2 : $r_2[x]\ r_2[y]\ c_2$
S_1 : $r_2[x_0]\ w_1[x_1]\ w_1[y_1]\ ||\ r_2[y_1]$

In S_1, T_2 reads x_0 from the old snapshot during the initial version period. In the same version period, T_1 updates the most recently committed versions of x and y, and commits. The updated values become available to the strictly dominating levels during the next version period. After the version period advance (denoted by the double vertical bars "||" in S_1), T_2 reads down the value y_1 and commits.

The schedule S_1 is not one–copy serializable since MVSG(S_1, $\prec$) contains a cycle. There is a version order edge from T_2 to T_1 on x and a conflict edge from T_1 to T_2 on y. □

In the schedule S_1, the L_2–level scheduler is unaware of the conflict and version order dependencies due to data access at lower security levels. Although the scheduler at each level guarantees the serializability of the local schedule, the overall schedule is nonserializable.

[2]Although we assume a linear lattice in the examples, the protocol proposed in this paper holds for a general lattice.

Nonserializable executions of the type S_1 can be prevented by ensuring that all read downs of a transaction are performed during a single version period, so that read downs execute on the same database snapshot. In this example, transaction T_2 should be restricted to reading either the initial snapshot or the snapshot in the new version period, but not both.

3.2 Same–Level Reads Following Read Down

The same–level reads of a transaction T, *following* T's first read down, should also complete in the same version period as T's first read down [14]. However, we can eliminate this restriction as discussed below, giving a transaction a larger window for the execution of its same–level reads.

Example 3.2: Two objects, x and z, are at sensitivity levels L_1 and L_2 respectively, where $L_1 < L_2$. These objects are manipulated by four transactions, T_1, T_2, T_3 and T_4. T_2 and T_4 are at the security level L_2, T_1 is at level L_1, while T_3 is at level L_3, where $L_1 < L_2 < L_3$. Consider the schedule S_2 shown below.

T_1 : $w_1[x]\ c_1$
T_2 : $r_2[x]\ r_2[z]\ c_2$
T_3 : $r_3[x]\ r_3[z]\ c_3$
T_4 : $w_4[z]\ c_4$
S_2 : $r_2[x_0]\ w_1[x_1]\ ||\ r_3[x_1]\ r_3[z_0]\ w_4[z_4]\ r_2[z_4]$

In the first version period, T_2 reads the value x_0 from the old snapshot, while T_1 updates the most recently committed version of x and commits. After the version period advance (indicated by the double vertical bars "||"), the old snapshot contains $\{x_1, z_0\}$. T_3 (at level L_3) then reads down x_1 and z_0, and commits. T_4 (at level L_2) now updates z and commits. Thereafter, T_2 reads the value z_4 and commits.

Schedule S_2 is not one–copy serializable since MVSG(S_2, $\prec$) contains a cycle. There is a version order edge from T_2 to T_1 on x, a conflict edge from T_1 to T_3 on x, a version order edge from T_3 to T_4 on z, and a conflict edge from T_4 to T_2 on z. □

Schedule S_2 is a very troublesome case. The L_2–level scheduler is unaware of the conflict and version order edges due to the highest level transaction T_3. Furthermore, it finds T_4's update innocuous, since $w_4[z]$ comes in before T_2 has submitted its request $r_2[z]$. When T_4 commits, its effect becomes immediately visible to T_2. The L_2–level scheduler cannot recognize T_2's conflict with T_4 on z.

Schedules of type S_2 must be prevented before a transaction can be allowed to perform same–level reads in a later version period than its read downs. The proposed protocol uses weak and strong locks to detect the conflict between T_4 and T_2; it blocks T_4 until T_2 completes.

Long read–only transactions in particular benefit from executing same–level reads in arbitrarily later version periods than their read downs. To derive the maximum benefit, a read–only transaction should execute all its read downs in a batch during a single version period (*batched read downs*). Then the read–only transaction can have a large number of same–level reads both before and after its batched read downs, and still complete successfully.

3.3 Commit of Update Transactions

At the beginning of a new version period, the most recently committed database should be copied into the old snapshot. Even when this copying can be done atomically and instantly, there are certain serialization problems, as the example below illustrates [1, 14].

Example 3.3: Consider a database containing two objects, x and z, at sensitivity levels L_1 and L_2 respectively, where $L_1 < L_2$. Three transactions, T_1, T_2 and T_3 execute in the system yielding the schedule S_3 shown below. T_1, T_2 and T_3 are at security levels L_1, L_2 and L_3 respectively, where $L_1 < L_2 < L_3$.

T_1 : $w_1[x]\ c_1$
T_2 : $r_2[x]\ w_2[z]\ c_2$
T_3 : $r_3[z]\ r_3[x]\ c_2$
S_3 : $w_1[x_1]\ r_2[x_0]\ ||\ r_3[z_0]\ r_3[x_1]\ w_2[z_2]$

T_1 updates the most recently committed version of x and commits. During the same version period, T_2 reads down the value x_0 from the old snapshot. Thereafter, a version period advance takes place (indicated by the double vertical bars "||" in S_3), and the old snapshot for the new version period becomes $\{x_1, z_0\}$. In the new version period, T_3 reads both z and x from the old snapshot and commits. Finally, T_2 updates z and commits.

There is a version order edge from T_2 to T_1 on x, a conflict edge from T_1 to T_3 on x, and a version order edge from T_3 to T_2 on z. Thus, MVSG(S_3, $\prec$) contains a cycle, and the schedule S_3 is not one–copy serializable. □

The problem with schedule S_3 is that T_2 reads an earlier snapshot than T_3 but creates a later version of an object than the one that T_3 reads. This sets up conflict and version order dependencies that the L_2–level scheduler is unable to detect.

Such nonserializable executions can be avoided by restricting an update transaction to commit during the same version period as its first read down. However, if a transaction does not read any lower level data, then there is no restriction on its commit.

Our assumption — that all updates of a transaction T are installed in the database instantly — is impossible to realize. During T's commit, especially near the beginning of the next version period, there may be a version period advance when some but not all of T's updates have been installed in the most recently committed database. A transaction at a strictly dominating level may observe the partial effects of T in the new version period, as the following example shows, with the result that the schedule may not be 1-copy serializable [14].

Example 3.4: Consider two objects, x and y, both at sensitivity level L_1, and the schedule S_4 shown below. Transactions T_1 and T_2 are at security levels L_1 and L_2 respectively, where $L_1 < L_2$. Transaction T_1 updates both x and y, while T_2 reads them both.

T_1 : $w_1[x]\ w_1[y]\ c_1$
T_2 : $r_2[y]\ r_2[x]\ c_2$
S_4 : $w_1[x_1]\ ||\ r_2[y_0]\ w_1[y_1]\ r_2[x_1]$

During the commit of T_1, there is a version period advance (denoted by the double vertical bars "||" in S_4) after installing $w_1[x_1]$ but before installing $w_1[y_1]$ in the most recently committed database; the database snapshot for the new version period therefore becomes $\{x_1, y_0\}$. Both the read downs from T_2 execute on this snapshot in the new version period.

In MVSG(S_4,$\prec$), there is a conflict edge from T_1 to T_2 on x, and a version order edge from T_2 to T_1 on y. Owing to this cycle, the schedule S_4 is not 1-copy serializable. □

Clearly, the problem illustrated in S_4 can be avoided if T_1's updates are *all* installed in the database during the *same* version period. In that case, T_2 would observe either the initial values x_0 and y_0, or all the updates made by T_1, but not the partial effects of T_1. This principle can be encapsulated as an *atomic commitment protocol* which ensures that a transaction commit is performed atomically despite version period advances.

The atomic commitment protocol must guarantee that when the commit of a transaction T crosses a version period boundary, the objects updated by T are unavailable to the strictly dominating levels for the duration of the commit. The details of one such protocol is presented as part of the locking protocol in Section 4.

4 The Two–Version Locking Protocol

4.1 Data Structures

A *version period number generator* maintains a counter VN which is initialized to zero to reflect the initial version period. To indicate a version period advance, the generator increments the value of VN, so that successive version period numbers are 0, 1, 2, 3, ...

The scheduler at level L_i is assumed to be untrusted and is denoted as Sch(i), $1 \leq i \leq n$. The data structures maintained by Sch(i) are also indicated with the same index. Thus, the waits–for graph at level L_i is written as WFG(i), $1 \leq i \leq n$.

Sch(i), $1 \leq i \leq n$, is allowed to read the existing value of VN but cannot update VN. Therefore, accesses to VN obey the simple security and the restricted star properties, with the version period number generator at the minimum level in the security lattice.

4.2 Versioning

We maintain two committed versions of each object in the database. A *version directory* contains an entry for each object x; this entry consists of two records, ECV(x) ("earlier committed version") and MRCV(x) ("most recently committed version"). Each of these records contains a pointer field and a version period number field. The directory entries may be stored in a single table or may be partitioned among the various security levels.

The pointer field within MRCV(x), denoted by MRCV(x).ptr, points to the latest committed version of x in the database. The pointer field within ECV(x), written ECV(x).ptr, points to an earlier committed version of x, which ideally belongs to the old snapshot appropriate for the existing version period. The respective version period number fields, indicated by MRCV(x).vn and ECV(x).vn, are described below.

Request	Weak Lock	Strong Read	Strong Write
Weak Lock	Y	Y	Y
Strong Read	Y	Y	N
Strong Write	C	N	N

Figure 1: The compatibility matrix for weak and strong locks. "Y" indicates compatibility; "N," incompatibility. The condition "C" is — incompatible if the weak lock holder has read down during an earlier version period, otherwise compatible.

4.3 Version Period Numbers in Directory Entries

When a version period advance takes place, it is imprudent to copy the entire most recently committed database into the earlier committed version. Lazy update of the earlier committed version of each object x is clearly more efficient.

The earlier committed version of x should be updated when the most recently committed version of x is about to be overwritten *for the first time* in a new version period. At other times, ECV(x).ptr may lag behind the snapshot of x appropriate for the new version period.

To indicate when ECV(x).ptr should be updated, the version period numbers in the directory entry for x bear a certain relationship with VN. The field MRCV(x).vn contains the value VN+1 (i.e., one greater than the value of the counter maintained by the version period number generator) whenever ECV(x).ptr points to the old snapshot of x appropriate for the version period VN. If MRCV(x).vn is less than or equal to VN at the time of overwriting the most recently committed version of x, then the value being overwritten is copied into the earlier committed version of x by saving MRCV(x).ptr in ECV(x).ptr, storing VN in ECV(x).vn, and updating MRCV(x).vn to ECV(x).vn+1.

Similarly, the field ECV(x).vn contains the value VN whenever ECV(x).ptr points to the old snapshot of x appropriate for the version period VN. Otherwise, ECV(x).vn has a smaller value than VN. Thus, for a read down on x, whenever ECV(x).vn is equal to VN, the earlier committed version of x should be retrieved; otherwise, the most recently committed version of x belongs to the old snapshot for the version period VN, and should be retrieved.

In this scheme, a read down retrieves the values of ECV(x).vn and either ECV(x).ptr or MRCV(x).ptr, while a transaction commit updates them. This high reader–low writer synchronization can be easily achieved using eventcounts and sequencers [15].

4.4 Weak and Strong Locks

Each object in the database supports two types of locks, *weak* and *strong*. Strong locks allow two lock modes, *read* and *write*. The lock compatibility matrix is shown in Figure 1.

When a transaction T starts, it acquires a weak lock on each object x at level(T) it intends to read; no locks are acquired at the lower security levels at any time. This does not preclude other transactions from obtaining weak or strong locks on those objects: weak locks are merely used to specify T's intention to read the objects sometime in the future.

When T reads x, its weak lock is converted to a strong *read* lock provided there is no

- **To schedule $r[x]$ from transaction T, where level(T) = level(x) = L_i:**
  WL=strong *write* lock holders on x besides T

```
If (WL is empty)
  Convert T's weak lock on x to strong read lock;
  If (T has updated x)
    Return x's value from T's intentions list;
  Else
    Return *MRCV(x).ptr;
Else /* block the transaction */
  Insert waits-for edge in WFG(i) from T to
  every transaction in WL;
```

- **To schedule $w[x]$ from transaction T, where level(T) = level(x) = L_i:**
  RDL = transactions other than T that hold a weak lock on x and have read down during an earlier version period than VN
  RWL=strong lock holders on x other than T

```
If (RDL ∪ RWL is empty)
  Insert strong write lock on x for T;
  Append w[x] to T's intentions list;
Else /* block the transaction */
  Insert waits-for edge in WFG(i) from T to
  every transaction in RDL ∪ RWL;
```

- **To schedule $r[x]$ from transaction T, where level(T) = L_i > level(x), and FVN_T is the version period of T's first read down ($FVN_T = \infty$ if T has not read down):**

```
While (ECV(x).vn < VN and x is latched)
    Skip;
If (ECV(x).vn < VN)
  p = MRCV(x).ptr;
Else
  p = ECV(x).ptr;
If (FVN_T < VN)
  Abort T;
Else {
  FVN_T = VN;
  Return *p;
}
```

- **To abort a transaction T at level L_i:**

```
Discard T's intentions list;
Release locks held by T;
Garbage collect T and all incident edges
from WFG(i);
Schedule transactions blocked by T;
```

Figure 2: The locking protocol for each security level L_i. All transactions at level L_i submit their operations to Sch(i). An explanation of the protocol is given in Section 4.

lock conflict at x according to the matrix of Figure 1. Otherwise, T is blocked until the lock conversion is allowed.

Transaction T is also blocked when it tries to acquire a strong *write* lock on x but another transaction T' holds a strong lock on x, or T' holds a weak lock on x and has executed a read down during an earlier version period. This avoids the serializability problem illustrated in Example 3.2; T remains blocked until T' completes and releases its lock on x.

4.5 The Locking Protocol

The locking protocol for each level L_i is presented in Figure 2, while an atomic commitment protocol [14] is shown in Figure 3. An explanation of these protocols is given below.

Each transaction T is started with an empty intentions list and a weak lock on each object x in T's read set at level(T). All L_i–level transactions submit their concurrency control requests to Sch(i). The read downs are performed on the old snapshot appropriate for the version period of execution. The accesses to objects at level L_i are synchronized

somewhat like strict 2PL on the most recently committed version, but using weak and strong locks; the recovery mechanism employed is deferred update. When a transaction commits, its intentions list is replayed on the most recently committed database.

4.5.1 Data Access at Same Level

When a transaction T tries to read the value of an object x at its own level L_i, Sch(i) checks whether or not T's weak lock can be converted to a strong *read* lock on x. If a different transaction T' holds a strong *write* lock on x, then T is blocked. Furthermore, a waits–for edge is inserted in WFG(i) from T to every such T'.

If no transaction blocks T, then T's weak lock is converted to a strong *read* lock. The value of x is retrieved from T's intentions list if T has updated x, otherwise the most recently committed version of x is read.

If T tries to update x, Sch(i) blocks T under the following conditions: (a) another transaction T' holds a strong *read* or *write* lock on x, or (b) T' holds a weak lock on x and has read down during an earlier version period than VN (see also subsection 4.5.3). In both cases, a waits–for edge is inserted in WFG(i) from T to every transaction T' that blocks T. Otherwise, Sch(i) grants T a strong *write* lock on x. Also, the $w[x]$ operation of T is appended to T's intentions list. This update is installed in the most recently committed version of x when T commits.

The code for $w[x]$ is sensitive to the value of VN, which may change; this critical section can be implemented in a straightforward way using eventcounts and sequencers [15].

4.5.2 Transaction Commit

When an update transaction T at level L_i commits, its intentions list is replayed on the most recently committed database using an atomic commitment protocol (Figure 3).

Let FVN_T ("first version number of T") be the version period of T's first read down. For each object x in the intentions list of T, Sch(i) acquires a latch (i.e., a short duration lock) on x for the duration of T's commit. If Sch(i) can successfully latch *all* the objects in T's intentions list during FVN_T, then T is allowed to commit. Sch(i) then visits each latched object, installing T's update and unlatching the object. However, if not all the latches can be acquired during FVN_T, then Sch(i) releases the latches it acquired for T, aborts T, and discards T's intentions list. In both cases, the locks held by T (on objects at its own security level) are released, and the node for T, together with all incident edges, are reclaimed from WFG(i). This enables the operations waiting for T's completion to be scheduled. Since at most one transaction can update an object at a time, there is no contention for latches.

This scheme lets a transaction T commit if all its latches are acquired during the version period of T's first read down; there is no need to actually install T's updates during FVN_T. A version period advance in between acquiring the latches and installing the updates requires synchronization with read downs, and is explained in subsection 4.5.3.

Before overwriting the most recently committed version of an object x, Sch(i) checks whether MRCV(x).vn is less than or equal to VN. If this condition is true, then the version being overwritten belongs to the old snapshot. Consequently, Sch(i) updates the earlier

To commit a transaction T at level L_i by replaying T's intentions list (IL) on the most recently committed database, FVN_T being the version period number of T's read downs ($\text{FVN}_T = \infty$ if T did not read down)

```
If (IL is not empty) {
  For each object x ∈ IL
      If (MRCV(x).vn ≤ VN) { /* Update earlier committed version of x if necessary */
        ECV(x).ptr = MRCV(x).ptr; ECV(x).vn = VN; MRCV(x).vn = ECV(x).vn + 1;
      }
  For each object x ∈ IL
      Latch x;
  If (FVN_T ≥ VN)
    For each object x ∈ IL {
        MRCV(x).ptr = pointer to version of x created by T; Unlatch x;
    }
  Else /* Commit cannot be permitted, so T must be aborted */
    For each object x ∈ IL: Unlatch x;
} Discard T's intentions list;
Release locks held by T;
Garbage collect T and all incident edges from WFG(i);
Schedule transactions blocked by T;
```

Figure 3: Atomic commitment protocol for each security level L_i used in the locking protocol. All transactions at level L_i are committed by Sch(i). An explanation is given in Section 4.5.2.

committed version of x by copying MRCV(x).ptr into ECV(x).ptr, storing VN in ECV(x).vn, and updating MRCV(x).vn to ECV(x).vn+1.

In spite of version period advances, when there are no updates of x, ECV(x) is not updated. This contributes to the efficiency of the proposed version maintenance scheme.

4.5.3 Read Downs

A transaction T at level L_i submits its read down requests to the scheduler at its own level. Sch(i) should access the old snapshot appropriate for the current version period.

Suppose T requests read access to an object x at level L_j, where $L_j < L_i$. Sch(i) checks whether ECV(x).vn is equal to VN. If it is, then the earlier committed version of x is retrieved. Otherwise, the most recently committed version of x belongs to the old snapshot appropriate for the version period VN, and is retrieved. No lock is set on x for a read down.

If T has read down during an earlier version period, i.e., $\text{FVN}_T < \text{VN}$, then T is aborted, as it has read down during two different version periods. Otherwise, FVN_T is set to VN, and the retrieved value of x is returned to T.

If Sch(j) is simultaneously installing a new value of x and there is a version period advance while x is latched, then the read down should retrieve the value being installed by Sch(j). Consequently, the read down spin locks until x is unlatched or a later update of x makes ECV(x).ptr point to the snapshot version of x appropriate for the existing version period.

This technique is effective since the spin lock lasts only for the duration of the commit.

The spin lock synchronizes read downs with transaction commits, but does not result in the starvation of high transactions. Suppose a read down spin locks on the condition (ECV(x).vn < VN and x is latched). When x is unlatched, the read down can proceed with reading the object's value. However, another transaction may try to commit at x before the read down can proceed any further. In that case, the atomic commitment protocol would copy the most recently committed version of x into the earlier committed version, since there has been a version period advance in the meantime. It would also store the current version period number VN in ECV(x).vn, so that the spin lock is broken, and the read down can execute.

4.5.4 Deadlocks

A transaction T at security level L_i is blocked if (a) T cannot acquire a strong *write* lock on an object x because another L_i level transaction T' holds a weak lock on x and has read down during an earlier version period, or (b) T has a strong lock conflict at an object at level L_i. In both cases, the blocking transaction is at the same security level as T. Thus, WFG(i) contains nodes only for the active transactions at level L_i, which may be deadlocked. Periodically, Sch(i) checks WFG(i) for cycles, and breaks them by aborting transactions.

5 Proofs of Serializability and Security

5.1 Serializability

Theorem 5.1: All schedules produced by the locking protocol of Section 4 are one–copy serializable.

Proof: We show by contradiction that for any schedule S produced by the locking protocol, MVSG(S, $\prec$) is acyclic.

If possible, let MVSG(S, $\prec$) contain a cycle of the form $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n \geq 2$ and the symbol $\rightarrow$ stands for a conflict or a version order edge. The nodes in MVSG(S, $\prec$) correspond to the committed transactions in S.

A version order edge exists in MVSG(S, $\prec$) from T_i to T_{i+1} if (a) $r_i[x_k]$ and $w_{i+1}[x_{i+1}]$ occur in C(S) and $x_k \prec x_{i+1}$, or (b) $w_i[x_i]$ and $r_k[x_{i+1}]$ occur in C(S) since $x_i \prec x_{i+1}$.

If in addition level(T_i) < level(T_{i+1}), then there is at least one version period advance between $w_i[x_i]$ and $r_k[x_{i+1}]$ in case (b) (case (a) is not allowed by our security model). If level(T_i) > level(T_{i+1}), then $r_i[x_k]$ occurs in an earlier or the same version period as $w_{i+1}[x_{i+1}]$ in case (a) (case (b) is not permitted by our security model). However, if level(T_i) = level(T_{i+1}), both (a) and (b) are possible; moreover, T_i's access to x must occur in an earlier or the same version period as those by T_{i+1} in case (a) and by T_k in case (b).

A conflict edge is inserted in MVSG(S, $\prec$) from T_i to T_{i+1} if $w_i[x_i]$ is followed by a same–level read or a read down $r_{i+1}[x_i]$ in C(S). In the former case, the read can occur in the same or a later version period than the write, while in the latter case, the read down certainly occurs in a later version period.

The edge $\rightarrow$ in the cycle is consistent with the version order, i.e., the edge $T_i \rightarrow T_j$, $i \neq j$, is inserted in the cycle when T_i manipulates an earlier or the same version of an object as T_j. Without loss of generality, we assume that $T_1 \rightarrow T_2$ is the first edge inserted in the cycle in the earliest version period.

If level(T_i) = level(T_{i+1}), $1 \leq i \leq n-1$, then none of T_1, T_2, T_3, ..., T_n read any values from lower security levels. In this case, the scheduler actions at each level are just like strict 2PL. Therefore, the committed projection of the schedule at each level is conflict serializable, and hence the overall schedule is one–copy serializable.

Now suppose that transactions *do* read down, so that not all the transactions in the cycle are at the same security level. In the cycle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, for every edge $T_i \rightarrow T_{i+1}$ such that level(T_i) < level(T_{i+1}), there is at least one version period advance. Therefore, T_1 is active during at least two version periods.

$T_1 \rightarrow T_2$ must be a version order edge with level(T_1) > level(T_2), since T_1 is active in a later version period. Thus, T_1 executes a read down in the earliest version period.

If level(T_1) < level(T_n), then T_1 would have to update an object and commit in a later version period than its first read down. This is prohibited by the proposed locking protocol.

If level(T_1) = level(T_n) then either T_1 would have to update an object and commit in a later version period than its first read down, or T_n would have to acquire a strong *write* lock on any object x in spite of T_1 holding a weak lock on x and having read down during an earlier version period. Neither of these cases are permitted by the proposed locking protocol.

Finally, if level(T_n) < level(T_1), then T_1 would have to perform read downs in different version periods, which is not allowed by our concurrency control protocol.

Therefore, there cannot be an edge from T_n to T_1, and MVSG(S, $\prec$) cannot contain a cycle. Hence, the schedules produced by the locking protocol are one–copy serializable. □

5.2 Security

We informally argue that the proposed locking protocol satisfies the MLS property [7].

The lock table at level L_i, the waits–for graph WFG(i), and FVN$_T$ for an L_i–level transaction T are all internal to Sch(i), as they are used for synchronizing only the transactions at level L_i. Since these data structures are not shared by multiple levels, there cannot be any covert channels resulting from accesses to them.

The version directory accesses in response to a read down or the commit of a transaction are noninterfering. Consider an object x at sensitivity level L_i. Sch(i) is allowed to read and write MRCV(x), and write but not read ECV(x). Sch(k) at a strictly dominating level is allowed to read ECV(x) and MRCV(x), but cannot update either of them. Furthermore, Sch(i) can latch and unlatch x, while Sch(k) can only examine whether or not the latch is set, but cannot latch or unlatch x.

For a same–level read, Sch(i) inspects MRCV(x) and sets a strong *read* lock on x. On the other hand, for a read down on x issued from level L_k, Sch(k) inspects both ECV(x) and MRCV(x) in the worst case. Neither of these operations update the version directory entry for x, while the strong *read* lock is internal to Sch(i). Therefore, a read down and a same–level read do not give rise to any covert channels.

Two concurrent read downs are noninterfering as well. The corresponding schedulers access the same data, and neither update any fields or set latches.

From the events discussed above, it follows that Sch(i) is unaware of the events occurring in the system owing to the activities at the dominating levels. Therefore, it executes the same way regardless of the events occurring at the higher levels. Hence, the proposed locking protocol satisfies the MLS property.

6 Discussion

The locking protocol proposed in this paper allows an update transaction T to commit as long as T's commit occurs in the same version period as T's first read down. The resulting execution is one–copy serializable and satisfies the MLS property. Thus, the protocol supports the execution of those long update transactions which perform a batch of read downs close to their commit points, and commit during the same version period.

The restriction on a read–only transaction is that it must complete all its read downs during a single version period. Provided a long read–only transaction satisfies this condition, it can continue to read data from the its own security level for as long as it needs to, either before or after the version period of its read downs.

The version maintenance scheme discussed in this paper requires between 1 and 2 times the storage of the most recently committed version of the database. The actual factor depends upon the workload: if updates are uniformly distributed across the database and the update rate is high, then the factor is closer to 2; when updates are infrequent or restricted to a small part of the database, the factor is closer to 1. During a read down, if the earlier committed version of the accessed object is found to be out of date, then it can be garbage collected, reducing the storage overhead for version maintenance. Of course, some care must be exercised to ensure that garbage collection does not give rise to a signaling channel.

The version period is an important parameter which determines the window during which a transaction can commit. A long version period gives the transactions a large window for completing read downs and committing. On the other hand, a short version period gives transactions a small window in which they can commit, but read downs retrieve more recent values. The version period is therefore a tunable parameter that determines the number and the type of long transactions that can commit in the system.

The length of the version period depends upon both the enterprise and the applications. If objects in the database are updated infrequently, the version period can be relatively large. Furthermore, in the example of generating monthly bank statements, the old snapshot needs to be updated only at the end of each day. A full day serves as a natural choice for the version period for this application, although account updates may take place any time.

References

[1] P. Ammann, F. Jaeckle, and S. Jajodia. A Two–Snapshot Algorithm for Concurrency Control In Multi–Level Secure Databases. In *Proceedings of 1992 IEEE Symposium on Research in Security and Privacy*, pages 204–215, Oakland, CA, May 1992.

[2] P. Ammann and S. Jajodia. A Timestamp Ordering Algorithm for Secure, Single–Version Multilevel Databases. In C. E. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 191–202. North–Holland, Amsterdam, 1992.

[3] P. Ammann and S. Jajodia. An Efficient Multiversion Algorithm for Secure Servicing of Transaction Reads. In *Proceedings of Second ACM Conference on Computer and Communications Security*, pages 118–125, November 1994.

[4] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretations. Technical Report MTR–2997, MITRE Corporation, March 1976.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison–Wesley, Reading, MA, 1987.

[6] A. R. Downing, I. B. Greenberg, and T. F. Lunt. Issues in Distributed Database Security. In *Proceedings of Fifth Annual Computer Security Applications Conference*, pages 196–203, Tucson, AZ, December 1989.

[7] J. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–22, 1982.

[8] J. Gray. *Operating Systems - An Advanced Course*, volume 60 of *Lecture Notes on Computer Science*, chapter Notes on Data Base Operating Systems. Springer–Verlag, 1978. R. Bayer, R. Graham and G. Seegmuller (eds.).

[9] J. N. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, Cannes, France, August 1981.

[10] J. T. Haigh and W. D. Young. Extending the Noninterference Version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE–13(2):141–150, February 1987.

[11] T. F. Keefe and W. T. Tsai. A Multiversion Transaction Scheduler for Centralized Multilevel Secure Database Systems. Computer Science and Engineering Department Tech. Report CSE–94–001, The Pennsylvania State University, June 1994.

[12] W. T. Maimone and I. B. Greenberg. Single–Level Multiversion Schedulers for Multilevel Secure Database Systems. In *Proceedings of Sixth Annual Computer Security Applications Conference*, pages 137–147, Tucson, AZ, December 1990.

[13] J. McDermott and S. Jajodia. Orange Locking: Channel–Free Database Concurrency Control Via Locking. In *Proceedings of IFIP Sixth Working Conference on Database Security*, pages 271–288, Vancouver, British Columbia, August 1992.

[14] S. Pal. A Locking Protocol for Multilevel Secure Databases Using Two Committed Versions. In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 197–210, Gaithersburg, MD, June 1995.

[15] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[16] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic Locking. *ACM Transactions on Database Systems*, 19(1):117–165, March 1994.