A Two Snapshot Algorithm For Concurrency Control In Multi-Level Secure Databases

Paul Ammann Frank Jaeckle Sushil Jajodia

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

Abstract

We offer a concurrency control algorithm for replicated, secure, multi-level databases. In secure databases, single copy techniques cannot avoid indirect channels without subjecting high level transactions to starvation due to malicious low level processes. However, multi-version and replicated databases can avoid starvation problems without introducing indirect channels by maintaining stable copies of old low level data values for use by high level transactions. The algorithm presented here improves on two comparable techniques, a direct multi-version approach of Keefe and Tsai [10] and full replication scheme of Jajodia and Kogan [9]. In the latter, each security level has a container that holds a copy of all lower level data. This paper shows that only a constant number of old copies - two, as it turns out - need be maintained. We argue correctness of our algorithm and demonstrate that the algorithm is free of indirect channels and starvation.

1. Introduction

In a multi-level secure database, the issue of correctness is compounded by indirect channels and starvation. If transaction histories are not serializable, then the scheduling algorithm is incorrect and therefore uninteresting. If high-level transactions can signal low-level transactions, perhaps through clever manipulation of the scheduler or through a Trojan horse in the database software, then the system is insecure. Finally, if low-level transactions can effectively block high-level transactions from making progress, the system is subject to starvation. In this paper we present a replicated concurrency control algorithm that addresses each of these three concerns.

1.1. Overview of the 2-Snapshot Algorithm

The algorithm presented here uses copies or replicates of data elements that are collected into *snapshots*. A snapshot is a complete and consistent copy of a database from which data values can be read, but to which updates are not made, except when subsequent snapshots are taken. There are a variety of standard techniques for constructing snapshots [5].

The fully enumerated name for the algorithm in this paper is "A Two-Snapshot Multi-Level Secure Concurrency Control Algorithm". We use an abbreviated term, the 2-Snapshot Algorithm. As the name suggests, the 2-Snapshot Algorithm requires two snapshots of the database at each security level. In addition, there is a full working database at each security level. Since there are no transactions that "read down" to the highest level, the algorithm does not keep snapshots of the highest level database(s).

High-level transactions access snapshots of low-level data instead of accessing low-level data directly. Periodically, new snapshots are taken at specified security levels and high-level transactions are methodically given access to the new snapshots. During a brief transition period, certain transactions access the old snapshots, while other transactions access new snapshots. Eventually transactions no longer access a given snapshot and the snapshot is discarded. When all old snapshots have been discarded, the set of new snapshots assumes the role held by the discarded old snapshots, and the algorithm repeats. A number of issues, such as how many snapshots are needed, when and how to take the snapshots, how to grant access to the snapshots, and how often the algorithm repeats, require attention. Before elaborating these issues, we discuss two comparable algorithms.

0-8186-2825-1 /92 $3.00 © 1992 IEEE

1.2. Comparison To Other Algorithms

In [10], Keefe and Tsai present a multi-version timestamp algorithm for concurrency control in multi-level secure databases. The 2-Snapshot Algorithm, although slightly different in applicability, improves on the solution in [10] in three respects. First, in the Keefe and Tsai solution the number of old versions of a data element that must remain directly available to high transactions is unbounded; in the 2-Snapshot Algorithm, all high transactions access one of two copies of a low-level data element.† Second, the Keefe and Tsai solution explicitly relies on timestamp ordering, and the 2-Snapshot Algorithm is more general. Although the 2-Snapshot Algorithm is naturally implemented with timestamp ordering, other scheduling algorithms, such as two-phase locking, can be used for transactions executing at a given security level. Finally, if low-level schedulers in the Keefe and Tsai solution divulge actual timestamp values to other low-level processes, perhaps through a Trojan Horse, then a signaling channel is available. We note in passing that this third problem is not inherent to the general scheme proposed in [10] and can be addressed with a more sophisticated timestamp generation scheme, such as the one in [3].

In [9], Jajodia and Kogan present a replicated data concurrency control algorithm. (Costich presents a related algorithm in [6].) Each security level has associated with it a container that holds the working database at that level and also a copy of each lower level database. Updates from lower levels are propagated through the containers by an algorithm that guarantees one-copy serializable transaction histories. Depending upon the lattice, the 2-Snapshot Algorithm may require fewer copies of the low-level databases. The more levels the lattice contains, the greater the savings in space. For example, in a lattice with N strictly hierarchical levels, [9] requires $N(N-1)/2$ copies of various databases; the 2-Snapshot Algorithm requires $2(N-1)$. The 2-Snapshot Algorithm may also use less space than a fully replicated architecture on lattices with a small number of levels. For example, consider a lattice in which N incomparable high security classes access a single low database. The 2-Snapshot Algorithm requires 2 copies of the low database; the fully replicated architecture requires N copies.

† It is important to note that there is *not* a bound on the number of old values of a data element that the 2-Snapshot Algorithm must maintain. There is only a bound on the number of values to which access is granted. There is a crucial practical difference between the two criteria.

Fig. 1 shows three related architectures for a standard security lattice of Unclassified (U), Confidential (C),

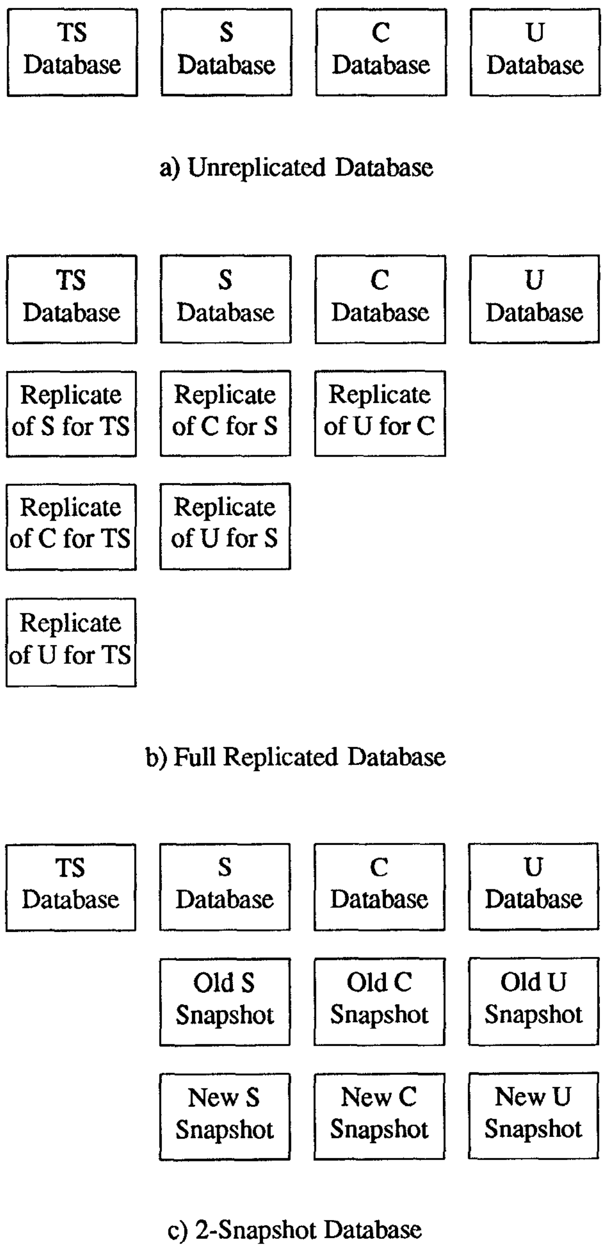

Fig. 1: Replication Comparison In 3 Architectures

Secret (S), and Top Secret (TS). Fig. 1(a) shows a simple non-replicated database such as one might find in a kernelized architecture [2]. Such an architecture is known to face either starvation problems or indirect channels. Fig. 1(b) shows a standard replicated database [2] in which each security level has a private copy of each lower level database. Fig. 1(c) shows the 2-Snapshot solution in which a high transaction accesses either an old or a new snapshot of a given low-level database. Fig. 1 illustrates the general organization of non-replication, full replication, and the intermediate 2-Snapshot replication.

1.3. Definitions

We adopt the standard lattice model [1, 4, 7] for specifying the set S of security classes. There is a set **T** of subjects (transactions) and a set **D** of objects (data elements). Each element of **D** and each transaction in **T** is associated with exactly one security class in S. The mapping L describes this association. If d is a data element in **D** and $L(d)=S_i$, then we say that the level of d is S_i. Similarly, if T is a transaction in **T** and $L(T)=S_i$, we say that the level of T is S_i.

There is a partial order on the security classes; we use the symbols $>$ and $\geq$ to denote this ordering. A security class S_i dominates security class S_j if the relation $S_i \geq S_j$ holds. Strict domination is indicated by $S_i > S_j$. We consider the relations $>$ and $\geq$ to be transitive.

We consider a system secure if

(1) Transaction T cannot read data element d unless $L(T) \geq L(d)$.

(2) Transaction T cannot write data element d unless $L(T) = L(d)$.

Note that the second constraint, which allows a transaction to write only at its level, is a restricted version of the *-property, which allows transactions to write up to higher levels. In the database context, the constrained version is desirable for integrity reasons.

It is not sufficient to prohibit high-level transactions from directly writing low-level data. In addition, we must be concerned with indirect violations, including covert and signaling channels.† For example, it is not permissible for a high transaction to lock a low data item for the duration of the high transaction, as would be required in a standard two-phase locking protocol. The difficulty is that a low-level transaction could repeatedly query the data item to determine whether or not it is locked. By choosing either to lock or not to lock the data item, the high transaction can signal the low transaction. Such channels are well known and have been extensively reviewed [7]. The 2-Snapshot Algorithm is designed to avoid signaling channels.

We define a *transaction* as a sequence of atomic operations on data elements. An operation on a data element $x \in D$ is either a *read*, denoted $r_i[x]$, or a *write*, denoted $w_i[x]$. For the bulk of this paper, we are not concerned with recovery issues, and so we ignore the commit and abort operations, c_i and a_i.

We adopt the serializability theory for replicated data from [5]. We summarize definitions informally where possible; the reader is referred to [5] for a complete formalism. We use the notation o_i (or p_i) to denote an operation of a transaction T_i, i.e., o_i is either r_i or w_i.

To execute a transaction on replicated data, we must first translate every operation on a data element into an operation on one (in the case of a read) or several (in the case of a write) operations on copies of that element. Let h be a *translation* function, i.e., let $h(r_i[x]) = \{r_i[x_j]\}$ and $h(w_i[x]) = \{w_i[x_{j_1}], \ldots, w_i[x_{j_r}]\}$, where $x_j, x_{j_1}, \ldots, x_{j_r}$ are copies of x. Thus, the translation of a given operation on a data element is a set of operations on copies (a singleton in the case of a read). To simplify notations, let $h(T_i)$ denote the union of translations of all the operations in T_i, i.e., $h(T_i) = \bigcup_{o_i[x] \in T_i} h(o_i[x])$. Also, let $h(T) = \bigcup_{T_i \in T} h(T_i)$. Two operations $o_i[x_k]$ and $o_j[x_k]$ *conflict* with each other if they are on the same copy of a data element (x_k) and at least one of them is a write.

A *replicated data history H* over a set of transactions T is a partial order with ordering relation $<_H$ that is consistent with the order of operations within transactions and that orders all conflicting operations.

Given a replicated data history H over a set of transactions T and transactions $T_i, T_j \in T$, we say that T_i *reads-x-from* T_j if $w_j[x_m] \in H$, $r_i[x_m] \in H$, $w_j[x_m] <_H r_i[x_m]$, and there exists no k such that $w_j[x_m] <_H w_k[x_m] <_H r_i[x_m]$. Two replicated data histories over transactions $T_0, T_1, \ldots, T_m$ are said to be *equivalent* if they have the same read-from's; i. e., if T_j reads-x-from T_i in one history, then this relationship holds in the other history as well.

† We distinguish signaling channels from covert channels as follows. A signaling channel is a means of information flow inherent in the basic algorithm or protocol, and hence appears in every implementation. A covert channel is a property of a specific implementation, and not the general algorithm or protocol. Thus a covert channel may be present in a given implementation even if the basic algorithm is free of signaling channels.

A *serial* history H is a totally ordered replicated data history such that for every pair of transactions T_i and T_j in T, either all of T_i*'s* operations precede all of T_j*'s* in H or vice versa. We say that a replicated data history is *one-copy serializable* if it is equivalent to a serial execution in a one-copy database. An execution of transactions is *correct* if its replicated data history is one-copy serializable. One-copy serializable histories hide all aspects of data replication from user's transactions and give transactions a one-copy view of the database.

To test for one-copy serializability, one usually makes use of a *replicated data serialization graph*, defined as follows.

Definition 1. A *serialization graph* for history H is a directed graph $G(H)$ whose nodes are transactions in T. A (directed) edge $T_i \rightarrow T_j$ is in $G(H)$ iff for some x in D and integer k, $o_i[x_k] \in H$, $p_j[x_k] \in H$, $o_i[x_k] <_H p_j[x_k]$, and $o_i[x_k]$ conflicts with $p_j[x_k]$. □

Informally, an edge $T_i \rightarrow T_j$ is included in $G(H)$ if T_i and T_j have conflicting operations such that the operation of T_i precedes that of T_j in history H. Let v and w be nodes of a a directed graph G. If there is a path from v to w we denote this fact by $v \rightarrow w$.

Definition 2. A *replicated data serialization graph* (RDSG) for history H is a directed graph whose nodes are transactions in T and whose (directed) edges constitute a superset of edges of $G(H)$ such that for all $x \in D$ the following conditions hold:

(1) if $w_i[x] \in T_i$ and $w_j[x] \in T_j$, then either $T_i \rightarrow T_j$ or $T_j \rightarrow T_i$;

(2) if T_j reads-x-from T_i, $w_k[x] \in T_k$ for some k $(k \neq i,\ k \neq j)$, and $T_i \rightarrow T_k$, then $T_j \rightarrow T_k$. □

There is the following correspondence between replicate data serialization graphs and one-copy serializable histories [5]: If a replicated data history H has an acyclic RDSG, then H is one-copy serializable.

We now supply the formalism for snapshots. We consider there to be an unbounded sequence of snapshots, even though the algorithm only requires that at most two snapshots be physically represented and available for access at any one time. Suppose that T_i writes x. We define the translation function applied to $w_i[x]$ as

$$\{ w_i[x_0] \} \cup \{ w_i[x_k], w_i[x_{k+1}], w_i[x_{k+2}], \cdots \}$$

where 0 is the index reserved to denote the working database at a given security level and k is index of the snapshot in which the effects of T_i are first reflected.[†]

We constrain the sequence of snapshots to be monotonic, in the sense that if the effects of a transaction appear in a given snapshot, they continue to appear in subsequent snapshots until such time as the values are overwritten. We model monotonicity as follows. Suppose that T_i and T_j both write x, that the effects of T_i are first reflected in snapshot k, that the effects of T_j are first reflected in snapshot l, and that $l>k$. Then in all replicated data histories that we allow, $w_i[x_l] <_H w_j[x_l]$. Informally, if T_i and T_j update x, and T_i updates x in a snapshot that T_j never writes, then T_i must serialize before T_j. Thus in all snapshots which T_i and T_j both update x, T_i must update x before T_j.

For subsequent discussions we require the following lemma.

Lemma 1. Let transaction T_i read x, transaction T_j write x, and T_i read an older snapshot of x than T_j writes. Let k be the snapshot T_i reads, and l be the first snapshot T_j writes. Formally,

$$r_i[x_k]$$
$$\forall\, m \bullet m \geq l \Rightarrow w_j[x_m]$$
$$l>k$$

Then $T_i \rightarrow T_j$ in the corresponding RDSG.

Proof. There exists some transaction T_0 from which T_i reads x, and we thus have

$$T_0 \rightarrow T_i$$

It is necessarily the case that T_0 writes x_k, and, since $l>k$, T_0 also writes x_l. T_0 and T_j conflict since both write x_l. The monotonicity property requires

$$T_0 \rightarrow T_j.$$

By part 2 of definition 2 above, we have $T_i \rightarrow T_j$. □

Lemma 1 is repeatedly employed in the examples below to argue dependencies in replicated data serialization graphs.

1.4. Outline Of Remainder Of Paper

The organization of the remainder of the paper is as follows. In section 2, we explain why a single snapshot is inadequate. In section 3, we derive the conditions required for snapshot creation. In section 4, we obtain the rules for granting high-level transactions access to new low-level snapshots, and we present the 2-Snapshot

[†] There is a practical difficulty in that the value of k is not known at the time T_i writes x_0, and so a possibly unbounded set of values for x must be maintained to update the physical snapshots. Although this point requires attention, it is not a serious difficulty.

Algorithm. In section 5 we discuss concurrency control at a given security level. In sections 6 we develop a rationale for why the algorithm is correct. Section 7 concludes the paper.

2. The Two Snapshot Framework

In objection to the 2-Snapshot Algorithm, it can be noted that storage space is required for each snapshot. In this section we address the question of why the algorithm is based on two snapshots instead of one. In addition, there are periods during which concurrently executing high-level transactions read different snapshots of low-level data. The schedulers must ensure that the resulting histories are correct. Certainly space requirements would be smaller and concurrency control would be simpler in a 1-Snapshot Algorithm.

Unfortunately, as the example below demonstrates, a straightforward implementation of a single snapshot scheme leads directly to an unpleasant choice between unserializable histories, signaling channels, starvation, or low concurrency among transactions. The first three alternatives are unacceptable, and the last alternative requires the abortion of certain transactions† that are executing at the time a switch to a new snapshot is made. In this context, we regard such a radical solution as uninteresting and do not address it further.

Notation For The Examples

For the examples given below, we consider a multi-level secure database with the three hierarchical security classes of unclassified (*U*), confidential (*C*), and secret (*S*). The names and classifications of the data items involved are given in the format of $d : L$ where d is the name of the data item and L is the classification level. Subscripts on d have the following interpretations: The subscript 0 is used to indicate that the copy of d being accessed is in the working database. The subscript *old* indicates that the copy of d is in what is currently regarded as the old snapshot. Similarly, the subscript *new* indicates that the copy of d is in what is currently regarded as the new snapshot. Note that *new* and *old* are indices, and that it is always the case that $new - old = 1$. Finally, the subscript k is used to indicate that the copy of d being accessed (written) is in some unspecified future snapshot. In the examples, it is always the case that $k > new$, and so we may apply Lemma 1 to argue the dependency $T_i \rightarrow T_j$ where T_i reads x_{new} and the first snapshot of x that T_j writes is x_k. Transactions are indicated by $T_i[L]$, where T_i is the transaction name and L is the level at which the transaction executes. Operations in transactions follow the format defined in Section 1. Histories are denoted H_i, and replicated data histories are denoted RDH_i. Transaction names in histories are abbreviations for the entire sequence of operations in the transaction.

The special operation $Snap_L$, where L is a classification level, represents the creation of a new snapshot of the database at level L. $Snap_L$ is an abbreviation for the sequences of writes to the new snapshot as well as the corresponding writes to all subsequent snapshots. The writes in the sequence correspond to those L level transactions that are associated with the new snapshot. It is important to note that, in two snapshot databases, the creation of a new snapshot at level L does not necessarily imply that transactions above level L immediately begin to use the new snapshot. Quite to the contrary, the correctness of the complete algorithm depends upon carefully controlling access by high transactions to new snapshots.

Updates To A 1-Snapshot Database

Clearly, the snapshots must be updated in some fashion or else high-level transactions can never access new low-level information. Suppose that a new snapshot of the U database is created (*i.e.* the event $Snap_U$ occurs). Since there is only a single snapshot, transactions at the C and S level necessarily access the new snapshot. Example 1 illustrates problems that arise in this scenario.

Example 1 - Updates In A 1-Snapshot Database

Data

$x : U$
$y : C$

Transactions

$T_1[U]$: $w_1[x]$
$T_2[C]$: $r_2[x]$ $w_2[y]$
$T_3[S]$: $r_3[x]$ $r_3[y]$

Histories

H_1: T_1 $r_2[x]$ $Snap_U$ T_3 $w_2[y]$
H_2: T_1 $r_2[x]$ $Snap_U$ $w_2[y]$ T_3

† It turns out that transactions that do not read data from lower levels need not be aborted; however, as will be shown in the next section, in this case care is still required as to which transactions may be projected into the new snapshot. A correct, but low concurrency, algorithm is

1) Delay all new transactions
2) Abort all active transactions that miss a given deadline
3) Produce a new snapshot at each level of the database
4) Execute delayed and new transactions (steady state)
5) Repeat

Replicated Data Histories

RDH_1:

$w_1[x_0]\ r_2[x_{old}]\ w_1[x_{new}]\ r_3[x_{new}]\ r_3[y_{new}]\ w_2[y_0]$

RDH_2:

$w_1[x_0]\ r_2[x_{old}]\ w_1[x_{new}]\ w_2[y_0]\ r_3[x_{new}]\ r_3[y_{new}]$

RDSG Dependencies

$T_1 \rightarrow T_3$ because T_3 reads x_{new} from T_1

$T_3 \rightarrow T_2$ since T_3 reads y_{new} and T_2 first writes y_k, $k>new$ †

$T_2 \rightarrow T_1$ since T_2 reads x_{old} and T_1 first writes x_{new}, $new>old$

Cycles in RDH_1 and RDH_2

$T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

Discussion

Both H_1 and H_2 are included to illustrate that the timing of T_3 does not affect the serialization problem, which is that the RDSG's for both RDH_1 and RDH_2 are cyclic. Suppose we attempt to avoid the cycle by aborting the transaction that would complete the cycle if it were allowed to commit. As H_1 shows, the transaction T_2 could be the victim. If T_2 is aborted due to the detection of a cycle, then a signaling channel from S to C has been introduced. Thus an optimistic solution that allows T_3-type transactions to proceed is not acceptable. Suppose we attempt to avoid the cycle by delaying transactions at the S level. If T_3 is delayed until after the T_2 write of y_k, then C level transactions may starve S level transactions that read from both the U and C snapshots. Thus a pessimistic solution that forces T_3-type transactions to delay is also not acceptable. □

Updates To A 2-Snapshot Database

In a single snapshot database, the only correct solution that does not suffer from either signaling channels or starvation is to delay and/or abort transactions each time a snapshot is introduced. The two snapshot database offers an alternate choice. All active and certain new transactions can simply continue to use old snapshots. Other new transactions can use the new snapshots. Transactions can gradually be weaned from the old snapshots, and eventually the old snapshots may be discarded. Before we describe the requirements that drive the process of constructing, introducing, and discarding snapshots, let us revisit Example 1 to see how a two snapshot approach improves the amount of concurrency without introducing undesirable side effects.

† Recall that the T_2 write of y_k occurs during some subsequent $Snap_C$ operation. The dependency follows from Lemma 1.

In Example 1, it is transaction T_3 that is the real culprit, since T_3 simultaneously needs to serialize before T_2 and after T_1. As discussed in Example 1, if T_3 is to serialize after T_2, then T_3 is subject to delay, and, in the single snapshot scenario, starvation. In the two snapshot approach, we need not delay T_3. Instead we may force T_3 to serialize prior to both T_1 and T_2 by requiring T_3 to read x_{old}. The desired (acyclic) replicated data histories are

RDH_3:

$w_1[x_0]\ r_2[x_{old}]\ w_1[x_{new}]\ r_3[x_{old}]\ r_3[y_{old}]\ w_2[y_0]$

RDH_4:

$w_1[x_0]\ r_2[x_{old}]\ w_1[x_{new}]\ w_2[y_0]\ r_3[x_{old}]\ r_3[y_{old}]$

Note that the new U snapshot can be used by new C-level transactions without difficulty. For example, a new transaction T_4 at the C level is allowed to access the new U-level snapshot. It turns out that transactions at the S level must continue to use the old U-level snapshot until several further conditions are met. We derive such conditions in section 4. However, we first turn to the question of what is permitted to be in a snapshot.

3. Snapshot Creation Requirements

We now turn to the question of what requirements a snapshot must satisfy, either by itself or in relation to other snapshots, so that serialization conflicts or other problems do not occur. The actual construction and representation of a snapshot is beyond the scope of this paper. Standard techniques apply [5].

Committed Transactions Requirement

We do not allow a snapshot to reflect any writes made by uncommitted transactions. If such writes were reflected in a snapshot, and the transaction in question was subsequently aborted, the snapshot would be invalid. High level transactions that referenced the ultimately invalid snapshot would either be forced to delay or participate in cascading aborts. Thus if uncommitted values were permitted in the snapshot, low level transactions could maliciously starve high level transactions. Since starvation is unacceptable, we do not allow the effects of uncommitted transactions to be reflected in a snapshot.

Serialization Prefix Requirement

Simply requiring a snapshot to reflect only committed transactions is insufficient, as Example 2 demonstrates.

Example 2 - Serialization Prefix Requirement

Data

$x, y : U$

Transactions

$T_1[U]$: $w_1[x]$ $r_1[y]$ c_1 [†]

$T_2[U]$: $w_2[y]$ c_2

$T_3[C]$: $r_3[x]$ $r_3[y]$ c_3

Histories

H_1: $w_1[x]$ $r_1[y]$ T_2 c_1 T_3

H_2: $w_1[x]$ $r_1[y]$ T_2 $Snap_U$ c_1 T_3

Replicated Data Histories

RDH_1:

$w_1[x_0]\, r_1[y_0]\, w_2[y_0]\, c_2 \quad c_1\, r_3[x_{old}]\, r_3[y_{old}]\, c_3$

RDH_2:

$w_1[x_0]\, r_1[y_0]\, w_2[y_0]\, c_2\, w_2[y_{new}]\, c_1\, r_3[x_{new}]\, r_3[y_{new}]\, c_3$

Dependencies in RDH_1

$T_1 \rightarrow T_2$ since T_1 reads y_0 before T_2 writes y_0

$T_3 \rightarrow T_1$ since T_3 reads x_{old} and T_1 first writes x_k, $k > old$

$T_3 \rightarrow T_2$ since T_3 reads y_{old} and T_2 first writes y_l, $l > old$

Dependencies in RDH_2

$T_1 \rightarrow T_2$ since T_1 reads y_0 before T_2 writes y_0

$T_3 \rightarrow T_1$ since T_3 reads x_{new} and T_1 first writes x_k, $k > new$

$T_2 \rightarrow T_3$ since T_3 reads y_{new} from T_2

Cycle in RDH_2

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$

Discussion

Note that H_1 is recoverable and serializable. There is no reason not to permit it, although it is noteworthy that in this execution the order in which the transactions commit differs from the order in which the transactions serialize. However, RDH_2 illustrates a serialization conflict. The problem stems from the timing of $Snap_U$, which contains effects from T_2 but not from T_1, even though T_1 must serialize before T_2. □

Example 2 demonstrates that it is insufficient for snapshots to merely reflect the effects of committed transactions. Snapshots also must not reflect the effects of any transaction that must follow an uncommitted transaction in any serialization order. In other words, a snapshot must reflect a committed prefix of some serialization order.

This requirement is not difficult achieve in practice in that most schedulers can easily produce an explicit serialization order [9]. Timestamp ordering schedulers serialize transactions according to the explicit timestamp. Two phase locking schedulers serialize transactions by the timing of the first lock release. Thus if the scheduler at a given level is either timestamp ordering or two phase locking, it is possible to determine which transactions may be reflected in a given snapshot.

Preconditions For Snapshot Creation

The previous two requirements on snapshot creation, namely that snapshots reflect committed transactions and that the snapshots reflect a prefix of a serialization order, are static requirements in the sense that they can be satisfied for any state of the database. These requirements can be satisfied independent of which snapshots current transactions access. However, there is an additional requirement that must be satisfied prior to snapshot construction. The requirement is illustrated in Example 3 below.

Example 3 - Precondition On Creating A Snapshot

Data

$x : U$

$y : C$

Transactions

$T_1[U]$: $w_1[x]$

$T_2[C]$: $r_2[x]$ $w_2[y]$

$T_3[S]$: $r_3[x]$ $r_3[y]$

Histories

H_1: T_1 $r_2[x]$ $Snap_U$ $Snap_C$ $w_2[y]$ T_3

H_2: T_1 $r_2[x]$ $Snap_C$ $w_2[y]$ $Snap_U$ T_3

Replicated Data Histories

RDH_1:

$w_1[x_0]\, r_2[x_{old}]\, w_1[x_{new}]\, w_2[y_0]\, r_3[x_{new}]\, r_3[y_{new}]$

RDH_2:

$w_1[x_0]\, r_2[x_{old}]\, w_2[y_0]\, w_1[x_{new}]\, r_3[x_{new}]\, r_3[y_{new}]$

Dependencies in RDH_1 and RDH_2

$T_1 \rightarrow T_3$ because T_3 reads x_{new} from T_1

$T_3 \rightarrow T_2$ since T_3 reads y_{new} and T_2 first writes y_k, $k > new$

$T_2 \rightarrow T_1$ since T_2 reads x_{old} and T_1 first writes x_{new}, $new > old$

[†] Note that commits are explicitly identified in this example only. The scheduler at a given security level determines whether a transaction may commit based only on the operations carried out on the working database at that level or on snapshots at lower levels. Thus, transactions commit *before* the corresponding updates to the (new) snapshot at the transaction's own level occur. There are two justification for allowing this. From a practical perspective, there is no integrity difficulty with allowing the transaction to commit. From a theoretical perspective, we may consider that the transaction completes its updates to the appropriate snapshot before the snapshot in question physically exists.

Cycles in H_1 and H_2

$T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

Discussion

Both histories have the same serialization conflict. It is the intent of this example to illustrate the case in which we wish for T_3 to access *new* low level data. Since there must be some point where transactions at the S level can read updated low data items, we are not content to simply serialize T_3 before T_1 and T_2 by forcing T_3 to access x_{old} and y_{old}. There is nothing suspicious about either the content or timing of T_3, and so the difficulty must lie in the construction of the new snapshots. Indeed, the problem arises because $Snap_C$ occurs while there is an active transaction at the C level, namely T_2, that is accessing the *old* U level snapshot. To avoid serialization problems in this example, $Snap_C$ must occur after T_2 has committed so that $Snap_C$ may reflect the effects of T_2. Indeed, $Snap_C$ *must* reflect the effects of T_2 for the snapshot to be usable. □

The conclusion from example 3 is that a *new* snapshot may not be constructed for the database at level L while there are active transactions at level L that access *old* snapshots at levels dominated by L. In general, this implies that there is an order in which snapshots must be generated. New snapshots are generated first at the lowest security level and progressively later for higher security levels. There is a delay involved in snapshot generation while transactions that accessed old data are permitted to terminate.

The delay requirement on the construction of new snapshots brings out an underlying assumption in the algorithm. The assumption is that transactions are not permitted to execute for arbitrary periods of time.[†] In particular, we assume that if a transaction executes longer than a certain time period then we are free to abort that transaction. Note that this assumption does not introduce a signaling channel, since the decision to abort a transaction is not dependent upon activity at a higher level, but is instead dependent on a fixed deadline.

The fixed period for which transactions are allowed to execute determines the rate at which low level data percolates up to high level transactions. Snapshots at successive levels are separated in time by one period, since we are required to wait precisely one period for termination of transactions that begin just before a snapshot is taken. Thus if a security lattice is N levels deep, it is necessary to wait N periods for an update at the lowest level to become readable at the highest level.

[†] Transactions at the lowest level(s) are exempt from the requirement to terminate by a set deadline. However, low level transactions that run for long periods can force high level transactions to read arbitrarily old low data values.

4. Switching New Transactions To New Snapshots

In this section we consider various strategies for deciding which set of snapshots new transactions at a given level may begin to use. By means of counterexamples, we show that there is only one workable strategy, namely for transactions at level L to view either all old snapshots at lower levels or all new. Combinations of old and new snapshots lead to serialization problems.

To see one such problem, consider Example 4 below, in which an S level transaction attempts to read from the new U snapshot and the old C snapshot simultaneously.

Example 4 - Snapshot Selection Counterexample

Data

$x : U$

$y : C$

Transactions

$T_1[U]$: $w_1[x]$

$T_2[C]$: $r_2[x]$ $w_2[y]$

$T_3[S]$: $r_3[x]$ $r_3[y]$

History

H_1: T_1 T_2 $Snap_U$ T_3

Replicated Data History

RDH_1:

$w_1[x_0]$ $r_2[x_{old}]$ $w_2[y_0]$ $w_1[x_{new}]$ $r_3[x_{new}]$ $r_3[y_{old}]$

Dependencies in RDH_1

$T_1 \rightarrow T_3$ because T_3 reads x_{new} from T_1

$T_3 \rightarrow T_2$ since T_3 reads y_{old} and T_2 first writes y_k, $k{>}old$

$T_2 \rightarrow T_1$ since T_2 reads x_{old} and T_1 first writes x_{new}, $new{>}old$

Cycle

$T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

Discussion

This execution results in a serialization conflict due to the fact that T_3 read from a new snapshot at the U level but an old snapshot at the C level. □

Consider Example 5 below, in which an S level transaction attempts to read from the old U snapshot and the new C snapshot simultaneously.

Example 5 - Snapshot Selection Counterexample

Data
$x : U$
$y : C$

Transactions
$T_1[U]$: $w_1[x]$
$T_2[C]$: $r_2[x]$ $w_2[y]$
$T_3[S]$: $r_3[x]$ $r_3[y]$

History
H_1: T_1 $Snap_U$ T_2 $Snap_C$ T_3
where T_2 is assumed to use the snapshot created by $Snap_U$, but T_3 is assumed to use the old snapshot at the U level.

Replicated Data History
RDH_1:
$w_1[x_0]$ $w_1[x_{new}]$ $r_2[x_{new}]$ $w_2[y_0]$ $w_2[y_{new}]$ $r_3[x_{old}]$ $r_3[y_{new}]$

Dependencies in RDH_1
$T_1 \rightarrow T_2$ because T_2 reads x_{new} from T_1.
$T_2 \rightarrow T_3$ because T_3 reads y_{new} from T_2.
$T_3 \rightarrow T_1$ since T_3 reads x_{old} and T_1 first writes x_{new}, $new > old$

Cycle
$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$

Discussion
This execution results in a serialization conflict due to the fact that T_3 uses a new snapshot at the C level but an old snapshot at the U level. □

Examples 4 and 5 illustrate that mixing and matching old and new snapshots from different levels leads directly to serialization problems. High-level transaction must view lower levels as either all old or all new. The switching algorithm outlined below reflects this requirement.

Switching algorithm

We suppose that we have N security levels. We describe the algorithm for a linear ordering of levels; however, the basic algorithm may be modified for any security lattice.

Initial State - Create an initial snapshot at each level.

```
While (Normal Operation)
  For i := 1 .. N-1
    1: Create A Snapshot At Level i.
    2: Let New Level i+1 Transactions
       Use New Snapshots At Levels 1..i
       /* Old Level i+1 Transactions Continue
       To Use Old Snapshots At Levels 1..i */
    3: Delay One Period
       /* Allow Old Level i+1
       Transactions To Finish */
    4: Abort Active Level i+1 Transactions
       Using Old Snapshots
       /* Deadline Has Expired */
  End For
  Discard Old Snapshots
End While
```

To adapt the algorithm for arbitrary lattices, it is necessary for the switching to take place in parallel across the lattice.[†] The case where a given security level dominates multiple other security levels does not pose a problem, since synchronization responds to what may be viewed as an external clock rather than to specific events at a given level.

This paper is mostly concerned about concealing the actions of high transactions from low transactions. In some cases it may be desirable to hide the very existence of certain security levels. As the algorithm is formulated, the number of dominating levels can be deduced from the number of periods that pass between snapshots. Each additional period reveals the existence of another dominating security class. However, the algorithm can be modified to meet the more stringent goal of hiding the existence of high security levels. The trick is to make the period at a given level half the length of a period at the next lower level. Under these circumstances, additional security levels can be squeezed into the lattice without affecting the schedule for taking snapshots at lower levels. Of course, the technique forces higher level transactions to obey deadlines that grow exponentially shorter in the number of levels.

5. Concurrency Control In Each Working Database

At each level L there exists a working database that serializes transactions at level L. We permit standard, untrusted schedulers to manage these databases.

[†] In fact the algorithm can be adapted to partial orders, of which lattices are a special case.

However, the schedulers must be modified to determine which snapshot should be accessed when a transaction reads a low level value. To minimize the amount of trusted code, we assume that the schedule sends a request to a trusted monitor which controls read access to each snapshot. (Updates to the snapshot need not be done by trusted code). As was explained in the previous section, the mapping to new or old snapshots is determined by whether a transaction begins before or after access to the new snapshot is permitted.

It is also necessary for scheduler to resolve serialization conflicts caused by access to the low level snapshots. In particular, we require that all transactions that begin before access to the new snapshot is granted be serializable before any transaction that begins after access to the new snapshot is granted. By way of illustration, we describe a timestamp oriented scheme to manage this process. Note that there are no security considerations that need be taken into account when designing this part of the 2-Snapshot Algorithm.

One possible algorithm is

(1) Assign each transaction a generation number based on the current generation when the transaction starts.† The current generation number is incremented each time access is given to new low level snapshots.

(2) Attach to each data value item two values, the generation number of the latest transaction to read the data value and the generation number of the latest transaction to write the data value. Note that these records need only be kept for data values at the security level of the scheduler.

(3) Abort any (old generation) transaction that attempts to access a data element "too late", *i.e.* after a new generation transaction has performed a conflicting operation on the data element.

Note that if the scheduler at a given level is a timestamp ordering scheduler, the algorithm above is just a subset of the timestamp ordering scheduling that is performed on all operations at that level. Thus timestamp ordering schedulers are a natural fit for the 2-Snapshot Algorithm. However, the use of timestamp ordering is not strictly necessary; other schedulers may also be used.

Finally, we are in a position to describe what we mean by a 2-Snapshot Algorithm. We define a 2-Snapshot Algorithm to be the combination of

† The notion of a "generation" is formalized in the next section.

(1) A snapshot generation algorithm.

(2) A snapshot switching algorithm.

(3) The modifications to the scheduling algorithms at each level that ensure that transactions accessing old snapshots serialize before transactions accessing new snapshots.

6. Algorithm Analysis

Prior sections have motivated the construction of the 2-Snapshot Algorithm with (counter) examples that led to constraints that necessarily must be satisfied if non-serializability, signaling channels, and starvation are simultaneously to be avoided. Such a presentation gives an explicit view of the requirements that guide the derivation of the algorithm. However, satisfaction of necessary conditions does not guarantee sufficiency.

We now turn to the task of showing that the 2-Snapshot Algorithm has serializable transaction histories, does not have signaling channels, and does not suffer from starvation. That the latter two proof obligations are met can be ascertained by inspection. The schedulers at a given level are unaware of the actions at higher levels, and thus the scheduling decisions of whether to abort or delay a transaction cannot be used by high processes to signal low processes. With respect to starvation, snapshots are stable objects that high transactions can access. Malicious low processes cannot interfere with access to the snapshots, nor can they interfere with snapshot construction.†

Thus the interesting proof obligation is to show that transactions histories are serializable. To discharge this obligation, we first give an informal illustration of why the 2-Snapshot Algorithm is correct. A formal proof is given in [8].

If we begin in a state that satisfies certain properties (namely that transaction histories can be serialized in a specified way), and show that eventually the 2-Snapshot Algorithm returns us to a state that enjoys the same properties, then we have a strong basis for arguing correctness. Specifically we employ the notion of a generation to describe the transactions that execute during this transition period and show that all transactions in a given generation may be serialized after the prior generation of transactions and before any subsequent generation of transactions. Within a given generation, we show that

† The reader is reminded that if transactions at the lowest level(s) are not subject to deadlines then high level transactions may be forced to use arbitrarily stale low data values.

the algorithm yields a serialization of transaction histories.

We define a *generation* to be a partition on the set of transactions at a given security level as follows. Each time access to a new set of snapshots is made available to new transactions at a given security level, we mark the event as the end of the old generation at that security level. Transactions that begin before the snapshot event have access to the old snapshot and belong to the old generation; transactions that begin after the snapshot event have access to the new snapshot and belong to some newer generation. As a special case, we define an old generation at the lowest security (which, of course, does not access any lower level snapshots) level each time a snapshot at that level is taken.

Since the 2-Snapshot Algorithm gives access to new sets of snapshots to each security level in turn (before repeating), the generation numbers of transactions at different security levels differs by at most 1. When a transaction is executing, we do not necessarily know to what generation the transaction will eventually belong, and thus we use the symbol G_L^i to denote the ith or later generation at security level L. When a transaction is complete, we are certain as to its generation, and we use the symbol $\hat{G}_L^i$ to denote the ith generation at security level L at a time when transactions in that generation are no longer executing. Such a generation is referred to as being "dead". Dead generations are useful in the correctness arguments below.

An Illustration

Consider transactions at the three security levels, U, C, and S. We begin in the state where generation 0 (and later) transactions are executing and end in a state where generation 1 (and later) transactions are executing.

C transactions access the snapshot U_{old}, and S transactions access the snapshots U_{old} and C_{old}. In the absence of updates, it is the case that all S transactions may be serialized prior to all C transactions, which may in turn be serialized before all U transactions. Let us overload the notation for a generation to mean not just a set of transactions, but also the replicated data serialization graph for transactions in that set. The *abstract dependency graph*† for all generation 0 transactions may be represented by

$$G_S^0 \rightarrow G_C^0 \rightarrow G_U^0.$$

If the serialization graph at each level is acyclic, a task which is the responsibility of the scheduler at each level, then the entire replicate data serialization graph is also acyclic, a condition that is captured by the abstract dependency graph shown above.

Now, let us take a snapshot at the U level, but, for the moment, let us delay access to this snapshot by new C level transactions. The taking of the snapshot defines the separation between old and new U level generations. Further, due to the property that the snapshot reflect only a prefix of the serialization order at the U level, it is the case that no transaction in the G_U^1 generation need serialize before on any transaction in G_U^0. Thus we may draw the new abstract dependency graph as

$$G_S^0 \rightarrow G_C^0 \rightarrow G_U^0 \rightarrow \qquad G_U^1.$$

All active transactions (perhaps in addition to some committed transactions) at the U level belong to the G_U^1 (or later) generation. The G_U^0 generation is dead, and we may rewrite the abstract dependency graph:

$$G_S^0 \rightarrow G_C^0 \rightarrow \hat{G}_U^0 \rightarrow \qquad G_U^1.$$

The property that $\hat{G}_U^0$ is dead is vital at this stage, because it is now safe for new C level transactions to serialize after transactions in $\hat{G}_U^0$ without the possibility of introducing an indirect channel based on the actions of U transactions. In particular, the 2-Snapshot Algorithm allows new C level transactions to access the new U snapshot, which leads to the following abstract dependency graph:

$$G_S^0 \rightarrow G_C^0 \rightarrow \hat{G}_U^0 \rightarrow \qquad G_C^1 \rightarrow G_U^1.$$

Note that transactions in the G_C^0 generation are still alive; such transactions have a complete period during which to finish their activity. As discussed in Section 5, the scheduler at the C level ensures that C level transaction from the two generations are serialized according to the abstract dependency graph shown above.

Eventually, C transactions in the G_C^0 generation either finish or are aborted for executing too long. As a result, the abstract dependency graph evolves to:

$$G_S^0 \rightarrow \hat{G}_C^0 \rightarrow \hat{G}_U^0 \rightarrow \qquad G_C^1 \rightarrow G_U^1.$$

Since the $\hat{G}_C^0$ generation is dead, it is now safe to take a snapshot at the C level. Taking the C snapshot partitions the G_C^1 transactions into two sets, namely those that

† An abstract dependency graph, arcs in which are denoted by $G_A^i \rightarrow G_B^i$, reflects the informal notion that G_A^i transactions *may* correctly serialize before G_B^i transactions. As such, $\rightarrow$ is a transitive relation. More precisely, $\rightarrow$ denotes the fact that if we were to explicitly insert the dependencies between the transactions in G_A^i and the transactions in G_B^i, it would be the case that all dependency arcs (if any) would point from some transaction in G_A^i to some transaction in G_B^i. Since we wish only to capture the notion that one set of transactions may serialize before another set, the dependency between specific transactions may be abstracted out.

serialize before the new C snapshot and those that serialize after. Those that serialize before the snapshot are, by definition, members of a dead set of transactions. We represent the generation 1 C transactions that serialize before the C snapshot by $\hat{G}_C^{1'}$. The abstract dependency graph becomes:

$$G_S^0 \rightarrow \hat{G}_C^0 \rightarrow \hat{G}_U^0 \rightarrow \hat{G}_C^{1'} \rightarrow G_C^1 \rightarrow G_U^1.$$

Since there are no active transactions in $\hat{G}_C^0$ or $\hat{G}_C^{1'}$, new S level transactions may immediately begin to use the new C and U snapshots without the possibility of introducing an indirect channel based on the actions of C transactions. The abstract dependency graph becomes:

$$G_S^0 \rightarrow \hat{G}_C^0 \rightarrow \hat{G}_U^0 \rightarrow \hat{G}_C^{1'} \rightarrow G_S^1 \rightarrow G_C^1 \rightarrow G_U^1.$$

Eventually transactions in G_S^0 complete, the GSO generation dies, and we obtain the abstract dependency graph:

$$\hat{G}_S^0 \rightarrow \hat{G}_C^0 \rightarrow \hat{G}_U^0 \rightarrow \hat{G}_C^{1'} \rightarrow G_S^1 \rightarrow G_C^1 \rightarrow G_U^1.$$

As in the initial state, it is the case that in the absence of updates, all active transactions can be serialized according to:

$$G_S^1 \rightarrow G_C^1 \rightarrow G_U^1$$

and thus the above argument may be applied again from the beginning.

7. Conclusions

In this paper we have presented the 2-Snapshot Algorithm, an algorithm for concurrency control in multilevel secure replicated databases. The 2-Snapshot algorithm improves on related techniques in [9, 10] and has the following general characteristics:

(1) In the 2-Snapshot Algorithm, a high transactions need only access one of two copies of a low level data element.

(2) The 2-Snapshot Algorithm may be implemented with a variety of scheduling disciplines at each security level; the algorithm is not restricted to timestamp ordering.

(3) In the 2-Snapshot Algorithm, the scheduler at any given security level need not be trusted.

An argument was given as to why the replicated data transaction histories allowed by the algorithm are one-copy serializable. A formal proof of one-copy serializability is given in [8]. The informal argument introduced the notion of an abstract dependency graph, which is a new tool for arguing serializability. Abstract dependency graphs can be used to express serialization dependencies between sets of transactions without reference to specific transactions in each set.

References

[1] "Trusted Computer System Evaluation Criteria", DoD Computer Security Center, CSC-STD-001-83, 1983.

[2] "Multilevel Data Management Security", Committee on Multilevel Data Management Security, Air Force Studies Board, National Research Council, Washington, DC, 1983.

[3] P. Ammann, S. Jajodia, "A Timestamp Ordering Algorithm For Secure Single-Version, Multi-Level Databases", *Proceedings of the 5th IFIP WG 11.3 Working Conference On Database Security*", Shepherdstown, WV, November, 1991.

[4] D.E. Bell, L.J. LaPadula, "Secure Computer Systems: Unified Exposition and Multics Interpretation", The Mitre Corporation, March, 1976.

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

[6] O. Costich, "Transaction Processing Using An Untrusted Scheduler In A Multilevel Database With Replicated Architecture", *Proceedings 5th IFIP WG 11.3 Working Conference On Database Security*", Shepherdstown, WV, November, 1991.

[7] D.E. Denning, *Cryptography and Data Security*, Addison-Wesley, Reading, MA, 1982.

[8] F. Jaeckle, "A Proof of the 2-Snapshot Algorithm", Draft GMU Technical Report, November, 1991.

[9] S. Jajodia, B. Kogan, "Data Replication And Multi-Level Secure Transaction Processing", *Proceedings of the 1990 IEEE Symposium Research in Security and Privacy*, Oakland, CA, May, 1990.

[10] T.F. Keefe, W.T. Tsai, "Multiversion Concurrency Control for Multilevel Secure Database Systems", *Proceedings of the 1990 IEEE Symposium Research in Security and Privacy*, Oakland, CA, May, 1990.