Maintaining Multilevel Transaction Atomicity in MLS Database Systems with Kernelized Architecture

Oliver Costich[*]
Sushil Jajodia

Center for Secure Information Systems, George Mason University, 4400 University Drive, Fairfax, Virginia 22030, USA

Abstract

In most models of trusted database systems, transactions are considered to be single-level subjects. As a consequence, users are denied the ability to execute some transactions which can be run on conventional (untrusted) database systems, namely those that perform functions that become inherently multilevel in the MLS environment. This paper introduces a notion of multilevel transaction and proceeds to an algorithm for their concurrent execution. The algorithm is proven to be correct in the sense that resulting schedules for executing the multilevel transactions is one-copy serializable.

1. INTRODUCTION

Most approaches to transaction processing for trusted database systems (**TDBS**) do something like the following. There is a set of *data items*, labeled with security classes from a lattice of security classes (or levels), which serve as the *objects* of

[*]The work of this author was supported in part by the Naval Research Laboratory under Contract N0001489-C-2389

the system. There is another set, of *transactions*, also labeled with security classes, in the role of the *subjects* of the system. A mandatory access control policy is adopted that enforces the Simple Security and ★-property of Bell and LaPadula [1]; namely subjects may write to objects only if the label of the subject is dominated by that of the object, and subjects may read from objects only if the label of the subject dominates that of the object. (Frequently the write condition is restricted to permit a subject to write to an object only if the labels are the same.) From the point of view of the security world, then, subjects and objects are the atomic units of interest. The approach described above enforces this point of view on the database system (and its users) as well.

On the other hand, in the database world, a different view of what constitutes atomicity prevails. Data items remain the elements of interest from the users' point of view. However, **DBS** users see two types of entities that operate on the data items. First there are *read* and *write operations* that are applied directly to data items. They also construct *transactions* as sequences of these operations that the users' expect to be executed atomically on the database. That is, a transaction is either executed completely and the resulting changes to the values of the data items made permanent, or the transaction has no effect at all. In addition, these transactions are independent in that there is no communication among transactions except through their effect on the values of data items. There is no external communication among transactions.

At first glance, the views of the security world and the database world seem in agreement. However, a conflict between them does exist. Implicit in the security view is that transactions have a unique security level. That is, subjects are single-level. From the database users' view, operations on data items are single-level, but requiring entire transactions to be so may be inadequate for many transactions that they may want to use. Examples may help.

A satellite uses sensors to collect sensitive information in its scanning range. That data, together with the position of the satellite, is used by an analytical process. The position data has security level **U** (unclassified) while the other data and the result of the analysis has security level **S** (secret). The security world would like to split this into two transactions; a **U** transaction that records the position data and an **S** transaction that then reads the position data, retrieves the other data, performs the analysis, and finally writes the result to the database. To do this, the user would have to log-on to the system at level **U** and submit the first transaction, then log-out and log-in at level **S**, where the second transaction would be submitted. The database world would like to do this using only a single transaction since it must be done as a single atomic action to insure that the result be correct. The two transactions approach embodies the following pitfall. Since the two transaction cannot communicate except through their action on

data items, there is no assurance that the second transaction will read the position data submitted by the first. Incorrect data could be read by the second transaction in two ways. The update to the position data may not have been made by the time the second transaction reads the position data item and so the old position would be used in the analysis. This can be overcome by the user waiting for a commitment message from the system before logging out from the **U** level. But there is another way that incorrect data can be read by the second transaction. Namely, newer position data is written by a different user's transaction before the correct data is read by the second transaction and again the analysis is incorrect. This cannot be corrected without the two transactions communicating in some external way.

The situation can get more complex, as shown in this second example. Suppose a company records the hours worked by each employee and computes the employees' salaries for the pay period. The hours data has security level **U** (unclassified) while the hourly rate data and the gross salary data have security level **S** (secret). The transaction to be performed first updates the hours worked from the time card, and then retrieves the hourly rate and computes the salary. Finally, the hours worked data item is reset to zero. The security world technique would require three transactions. The first updates hours worked. The second retrieves the rate and computes the salary. The third resets the hours worked to zero. Three distinct log-ins are required, and the first and second have the same problem as our previous example. Beyond that if the third were completed before the second transaction retrieved the hours data, the salary would be calculated incorrectly.

In the conventional (not trusted) database world, these problems would not exist, because the user could submit these combined actions as a single transaction. Ordinary concurrency control mechanisms (which enforce *serializability* of collections of transactions) would insure that correct values were read and written in the proper sequence.

It appears that some notion of multilevel transaction is required to resolve this dilemma. Previous work in this direction for a limited class of multilevel transactions and for replicated architecture multilevel database systems appears in [4]. Here we intend to extend the idea of multilevel transaction to a significantly larger class of multilevel transactions, one that encompasses virtually every situation that we can construct. We will formally define notions of multilevel transaction and the correctness of their execution (serializability) in centralized multilevel database systems with kernelized architecture. A scheduling algorithm will be presented and shown to be correct.

2. THE SECURITY MODEL

The architecture for the systems under consideration is based on one that is frequently proposed [7,8]. The security features are enforced by a security kernel, the trusted computing base of the trusted operating system, together with whatever additional trusted processes are necessary in the database application to enforce the overall system's security policy. The idea is to minimize the trusted processes required to do this.

The security policy for our system will be a variant of the mandatory access control policy of Bell and LaPadula [1]. There is a set **D** of data items of the database system that serve as the objects of the multilevel system. The subjects of the system, denoted **Sub**, are quite similar to the single-level transactions used in earlier work [3,6,7,8,9,10]. However our transactions will be more complex and will be formed by interleaving the subjects of the database system.

More formally, we limit operations on the data items. Only Reads, denoted **r[x]**, and Writes, denoted **w[x]** together with Aborts, denoted **a**, or commits, denoted **c** are considered. A subject of **Sub** is a sequence of Reads and Writes ending with either an Abort or a Commit (but not both). There is a lattice (**SC**,<) of security labels and a function L, mapping subjects and objects into security classes, i.e., $L: \mathbf{D} \cup \mathbf{Sub} \rightarrow \mathbf{SC}$. The security policy has two conditions:

(1) (Simple Security Property) If $\mathbf{T} \in \mathbf{Sub}$ and $\mathbf{r[x]} \in \mathbf{T}$, then $\mathbf{L(x)} \leq \mathbf{L(T)}$.

(2) (Restricted $\star$-Property) If $\mathbf{T} \in \mathbf{Sub}$ and $\mathbf{w[x]} \in \mathbf{T}$, then $\mathbf{L(x)} = \mathbf{L(T)}$.

That is, subjects can read from dominated security levels, but only write at their own security level. These are basically the mandatory access control policies of [1], slightly modified.

3. THE TRANSACTION MODEL

To define multilevel transaction, we need some preliminaries. A data item x can take on values from its domain, **dom(x)**. A state of the database is determined by assigning each **x** in **D** a value from its domain, i.e., the states are functions $\mathbf{f}: \mathbf{D} \rightarrow \cup \{\mathbf{dom(x)} \mid \mathbf{x} \in \mathbf{D} \text{ and } \mathbf{f(x)} \in \mathbf{dom(x)}\}$. **V** will denote the set of all such functions. Further, let $\mathbf{D}_{|\leq \mathbf{l}} = \{\mathbf{x} \in \mathbf{D} \mid \mathbf{L(x)} \leq \mathbf{l} \text{ and } \mathbf{l} \in \mathbf{SC}\}$ and let $\mathbf{V}_{|\leq \mathbf{l}}$ be obtained by restricting each $\mathbf{f} \in \mathbf{V}$ to $\mathbf{D}_{|\leq \mathbf{l}}$ (denoted $\mathbf{f}_{\leq \mathbf{l}}$). Notice that any action on the database defines a mapping of **V** to **V**. In particular, if **T** is a transaction and $\mathbf{f} \in \mathbf{V}$, then $(\mathbf{T}(\mathbf{f}))(\mathbf{x})$ is the value of **x** resulting from executing **T** on the database starting with the values

of the data items specified by **f**. $\mathbf{T}_{|\leq \mathbf{l}}$ will denote **T** restricted to $\mathbf{V}_{|\leq \mathbf{l}}$. Alternatively, $\mathbf{T}_{|\leq \mathbf{l}}$ is the transaction obtained by discarding the operations not dominated by **l** (and keeping the implied order).

Definition A *multilevel transaction* $\mathbf{T}_i$ is a sequence[**], ordered by $\leq_i$, of ordered pairs of the form $(\mathbf{o}_i,\mathbf{l})$ where $\mathbf{o}_i$ is one of $\mathbf{a}_i$, $\mathbf{c}_i$, $\mathbf{r}_i[\mathbf{x}]$, $\mathbf{w}_i[\mathbf{x}]$ for some $\mathbf{x}\in\mathbf{D}$ and $\mathbf{l}\in\mathbf{SC}$ that satisfies the following conditions:

(1) Either $\{(\mathbf{c}_i,\mathbf{l}) \mid \mathbf{l}\in\mathbf{SC}\}\subseteq\mathbf{T}_i$ or $\{(\mathbf{a}_i,\mathbf{l}) \mid \mathbf{l}\in\mathbf{SC}\}\subseteq\mathbf{T}_i$, but not both.

(2) Let $\mathbf{e}_i$ be either $\mathbf{a}_i$ or $\mathbf{c}_i$. Then for each $\mathbf{l}\in\mathbf{SC}$, $(\mathbf{o}_i,\mathbf{l})\leq_i(\mathbf{e}_i,\mathbf{l})$.

(3) For $\mathbf{f}\in\mathbf{V}$, we have $\mathbf{T}_{i|\leq \mathbf{l}}(\mathbf{f}_{\leq \mathbf{l}})=(\mathbf{T}_i(\mathbf{f}))_{|\leq \mathbf{l}}$ for each $\mathbf{l}\in\mathbf{SC}$.

The first condition requires a multilevel transaction to commit at each security level or abort at each security level. Security considerations alone would only require that no commits occur at security levels higher than one at which an abort had occurred, else lower level subtransactions would have to be rolled back to insure atomicity. Imposing this condition guarantees that this possibilty is avoided. We should point out that we are only accounting for aborts due to concurrency control considerations and not for those due to violations of integrity constraints, such as range constraints. Aborts for other reasons are problematic regardless of the concurrency control technique employed. The second condition forbids further operations to be done at a given security level after the commit or abort at that level.

The third condition is more difficult to explain. Notice that for a multilevel operation $(\mathbf{o}_i,\mathbf{l})$, $\mathbf{o}_i$ can only operate on a data item whose security level is dominated by **l** in **SC**. This means that operations in $\mathbf{T}_{|\leq \mathbf{l}}$ are only applied to data items at the level of **l** or below, and that $\mathbf{T}_{|\leq \mathbf{l}}(\mathbf{f}_{\leq \mathbf{l}})$ is the result of applying these operations to those data items. $(\mathbf{T}_i(\mathbf{f}))\big|_{\leq \mathbf{l}}$ is the result of executing **T** on the entire database and then looking only at the result on the data items with security level **l** or below. The equality in the condition says that the values of data items at level **l** and below which result from executing **T** depend only on the values of those data items when **T** was initiated, and not on the values of any

[**]We could use a definition of transaction based on partial orders, as in [2]. However the results are actually no more general but the definitions, the algorithm, and the proofs are more complicated. We will use sequences rather than partial orders throughout this paper since it simplifies the explication with no loss of generality.

higher level data items. Said differently, no information about the value of higher level data items can flow to lower level data items by virtue of running the transaction.

Consider our earlier examples in light of this definition. The first example becomes (**w[x],U**)(**c,U**)(**r[x],S**)(**r[y],S**)(**w[z],S**)(**c,S**) where **L(x)=U**, and the other data items have security level **S**. This clearly satisfies the conditions. Condition (3) is satisfied since the transaction never writes to a lower level data item after accessing a higher level one. Transactions of this form are treated in [4] for a different architecture.

Our second example becomes (omitting the commit operations for simplicity), (**w[x],U**)(**r[y],S**)(**r[y],S**)(**w[z],S**)(**w[x],U**). Unlike the prior example, this transaction writes a lower level data item after reading a higher level one. But since the second time **x** is written the value is always zero, the last condition is satisfied, and we have a legitimate multilevel transaction.

Whether the third condition is satisfied is not easily determined by the **TDBMS** itself. One way to resolve this difficulty is to limit transactions to those that satisfy a more restrictive, but more easily detectable form (as in [4], for example).

Another solution, which we believe is more likely, is to restrict multilevel transactions to predefined transactions that can be determined ahead of time and verified to satisfy this condition. Ad hoc user defined transactions would not be allowed because of the risk of violating the condition. Under this approach, the data items on which a transaction would operate would be known at the time the transaction is submitted to the system. The algorithm presented here relies on this assumption.

Operations of several transactions can be commingled so that concurrency of execution can be extended to sets of transactions, as reflected in the following.

Definition A *complete multilevel history* **H** over a set of multilevel transactions $\mathbf{T} = \{\mathbf{T_1}, \mathbf{T_2}, \text{ - - - }, \mathbf{T_n}\}$ is a sequence with ordering relation $<_H$ where

(1) There is a multilevel $\mathbf{T_0}$ that precedes all other transactions. $\mathbf{T_0}$ has operations $\{(\mathbf{w_0[x],l}) \mid \mathbf{x} \in \mathbf{D}\}$.

(2) $\mathbf{H} \supseteq \mathbf{T_0} \cup \mathbf{T_1} \cup \mathbf{T_2} \cup \text{ - - - } \cup \mathbf{T_n}$

(3) $<_H \supseteq <_0 \cup <_1 \cup <_2 \cup \text{ - - - } \cup <_n$

The first condition provides initial values of the data items, so a Read operation always succeeds some Write operation on the desired data item [11]. The second

requires the history to contain precisely the operations of the original transactions. The third condition provides that the ordering of operations within the history is consistent with that of each transaction.

Notice that our notion of multilevel transaction does not limit the number of Read or Write operations on a given data item within a transaction, or even within a given level of a transaction, in contrast to the usual practice in concurrency control theory [2,11]. Since it does not, a transaction may read or write the same data item several times. We will denote the n^{th} Write operation on **x** by $\mathbf{T_i}$ by $\mathbf{w_{i,n}[x]}$ when necessary to avoid confusion. This multiple write capability has led us to choose a multiversion approach to concurrency control. Multiversion systems create a new version of a data item each time it is written and maintain the old versions, which does not impose significant additional burden on the **DBMS**, since these versions must be maintained for purposes of recovery in any case.

The preceding definition of history represents the view of the user, to whom versions of data items are transparent. The users' view represents the logical order of the execution of operations as seen by the users. Histories of this type will be called *one-copy histories* when it is necessary to distinguish them from histories that represent the system's view of a transaction and that deal with multiple versions of data items. The representation of the system's view requires a different definition of history.

When a set of transactions is executed by a multiversion **DBMS**, an operation in a transaction must be translated into the equivalent operation on some version of the data item. A *translation function* **h** performs the mapping. For a Read, **h** determines the version of **x** to be read, i.e., $\mathbf{h}(\mathbf{r_i[x]},l))=(\mathbf{r_i[x_{i,n}]},\mathbf{l})$ where $\mathbf{x_{i,n}}$ is the n^{th} version of **x** written by $\mathbf{T_i}$. For a Write, **h**, determines what version of x will be created, i.e., $\mathbf{h}(\mathbf{w_i[x]},\mathbf{l})=(\mathbf{w_i[x_{i,n}]},\mathbf{l})$ if $\mathbf{w[x_i]}$ is the n^{th} Write operation on **x** in $\mathbf{T_i}$.

The concept of a *multiversion data history* is needed to represent the actions of the translated transactions on the multiversion data. Recall that $\mathbf{T_0}$ is an initializing transaction that writes initial values into every data item in D.

Definition A *multiversion data history* **H** over a set of multilevel transactions $\mathbf{T}=\{\mathbf{T_1}, \mathbf{T_2}, \text{ - - -}, \mathbf{T_n}\}$ is a linear order with ordering relation $<_H$ such that

(1) $\mathbf{H} \supseteq h(\mathbf{T_0}) \cup \mathbf{h}(\mathbf{T_1}) \cup \mathbf{h}(\mathbf{T_2}) \cup \text{ - - -} \cup \mathbf{h}(\mathbf{T_n})$

(2) If $(\mathbf{p_i},\mathbf{l})$, $(\mathbf{q_i},\mathbf{m})\in\mathbf{T_i}$ with $(\mathbf{p_i},\mathbf{l}) <_i(\mathbf{q_i},\mathbf{m})$ then $\mathbf{h}(\mathbf{p_i},\mathbf{l})<_H\mathbf{h}(\mathbf{q_i},\mathbf{m})$

(3) For all $l\in\mathbf{SC}$, all i>0, $(\mathbf{w_0[x_{0,1}]},\mathbf{l})<_H(\mathbf{r_{i,1}[x]},\mathbf{l})$ for all $\mathbf{x}\in\mathbf{D}$.

A multiversion data history represents the order in which operations are executed by the data manager of the **TDBMS** on the data items stored in the **TDBMS**.

The first condition says that the history contains the translations of the original transactions. The second condition insures that the order of operations within transactions is preserved. The third condition provides that at each security level, $\mathbf{T_0}$ initializes each data item before it is read by any of the original transactions.

Histories, one-copy or multiversion, are complete if they contain no operations from aborted transactions. Since we are primarily concerned with these kinds of histories, we will refer to them simply as histories. (As it turns out, our algorithm prevents aborts, so the notions are coincident in any case.) We now turn to defining a notion of correctness for execution of multilevel transactions.

A history, one-copy or multiversion, is *serial* if for every pair of transactions $\mathbf{T_i}$ and $\mathbf{T_j}$ that appear in **H**, either all of the operations of $\mathbf{T_i}$ precede those of $\mathbf{T_j}$ or vice versa. In one-copy histories, correctness is defined as being equivalent to a serial history, where equivalence is, in turn, defined in terms of reads-from relationships and final writes in the usual way [2]. It is well known in the theory of database concurrency control that the parallel notion of equivalence is insufficient for multiversion transactions [2] because Read operations may now read from different versions of a data item, and a transaction may read from the correct transaction but choose the wrong version. We will now make these ideas more formal.

Definition In one-copy histories, we say $(\mathbf{r_j[x],m})$ *reads-***x***-from* $(\mathbf{w_i[x],l})$ if $(\mathbf{w_i[x],l}) <_H (\mathbf{r_j[x],l})$ and there is no $(\mathbf{w_k[x],l})$ for which $(\mathbf{w_i[x],l}) <_H (\mathbf{w_k[x],l}) <_H (\mathbf{r_j[x],l})$. Notice that $\mathbf{m} \geq \mathbf{l}$ and that there is no requirement that i, j, and k be distinct. If $i \neq j$, we say $\mathbf{T_j}$ reads-**x**-from $\mathbf{T_i}$, and call it a *transaction reads-from.* If i=j, we call it a *reflexive reads-from.* We can extend these notions to multiversion histories by considering different versions of a data item **x** as distinct data items, as usual.

Definition Two histories, one-copy or multiversion are, *equivalent* if they have the same transaction and reflexive read-from relationships. In the one-copy case we also require that they have the same final writes (which we will not define as we will not use equivalence for this case).

As previously mentioned, it is inadequate to require that a multiversion history be equivalent a serial multiversion history for a history to represent a correct execution of the given transactions. Something more is required. We must require that, in addition to being equivalent to a serial history, that it be equivalent to a special class of serial history.

Definition A multiversion history **H** is *one-copy serial* if it is serial and satisfies

(1) If $\mathbf{T_j}$ *reads-***x***-from* $\mathbf{T_i}$ is a transactions reads-from then the first Read operation in $\mathbf{T_j}$ that reads any version of **x**, reads it from the version of **x** written by the last Write operation of $\mathbf{T_i}$ (note that this is well-defined since **H** is serial.)

(2) If $(\mathbf{r_i[x_{i,n}]},\mathbf{m})$ reads-**x**-from $(\mathbf{w_i[x_{i,p}]},\mathbf{l})$ is a reflexive reads-from, then p=n. That is, each Read of **x** in $\mathbf{T_i}$ reads-**x**-from the version produced by the immediately preceding Write of **x** in $\mathbf{T_i}$.

It is intuitively quite clear that one-copy serial histories are correct since they look just like the corresponding one-copy history except that the data items have versions and the Read operations are "correctly" matched to the right Write operations.

Definition A multiversion history is *one-copy serializable* (**1SR**) if it is equivalent to a one-copy serial multiversion history.

To show that this is an adequate criterion for correctness for (multilevel) multiversion histories, we state the following theorem without proof. A proof can be easily constructed from [2, Theorem 5.3]. Only minor changes are required are to account for reflexive reads-froms.

Theorem Let **H** be a multiversion history over a set of multilevel transactions $\mathbf{T}=\{\mathbf{T_1}, \mathbf{T_2}, \text{ - - - }, \mathbf{T_n}\}$. Then **H** is equivalent to a serial one-copy history over **T** if and only if **H** is **1SR**.

4. THE INFORMAL PRESENTATION OF THE ALGORITHM

What must the concurrency control process in our multilevel database system do? First, it must produce a **1SR** multiversion history. In addition, the way in which transactions and their operations are scheduled cannot result in the flow of high security level information to lower security level subjects. In particular, no lower security level transaction can be allowed to roll back because of the execution of a higher security level part of a transaction. We want to accomplish this with a minimum of trusted processes.

Definition Given a multilevel transaction T_i and $l \in SC$, the l-*projection* of $T_{i|=l}$ of T_i is $\{(o_i,l) \mid (o_i,l) \in T_i\}$ with the linear ordering inherited from T_i. We refer to these l-projections generally as *subtransactions.*

Definition The *write set* of $T_{i|=l}$ is $WS(T_{i|=l})=\{x \in D \mid (w[x],l) \in T_i\}$ and it's *read set* is $RS(T_{i|=l})=\{x \in D \mid (r[x],l) \in T_i\}$. Similarly, $WS(T_{i|\leq l})=\{x \in D \mid (w[x],m) \in T_i$ and $m \leq l\}$ and $RS(T_{i|\leq l})=\{x \in D \mid (r[x],m) \in T_i$ and $m \leq l\}$.

Notice that $T_{i|=l}$ is a subject of the trusted system as we have defined them, and that $T_{i|=l}$ also can be viewed as a single-level transaction with security level l. Every multilevel transaction naturally gives rise to a set of subtransactions. Notice that the definition of multilevel transaction guarantees that values written at one security level cannot depend on values read at higher security levels, even if the Read precedes the Write. This means that every multilevel transaction is equivalent to one that executes the subtransactions in an order consistent with the security lattice ordering (from low to high).

Our multiversion **TDBMS** will also have an untrusted strict multiversion timestamp order scheduler [2], P_l, for each $l \in SC$. These schedulers will be called *local schedulers* and will be used to schedule the subtransactions for their level, just as if they were single level transactions. Subtransactions, therefore, may commit (and, in theory, abort) and we refer to such as *local commits* (or *aborts*). If a subtransaction has begun execution but not locally committed, we say it is *active.*

There is also a *global scheduler* **Q**, which will manage subtransactions across security levels (between the local schedulers). **Q** will be largely untrusted, though a few trusted processes will reside there. **Q** will assign timestamps to transactions, compare read sets and write sets as necessary, and distribute subtransactions to the appropriate local schedulers.

Informally, the algorithm works as follows. When a multilevel transaction is received, a timestamp is assigned and Read and Write operations on each data item are indexed. I.e., the first Write of **x** is indexed by 1, the next is indexed by 2, and so on. Read operations receive the same index as the last preceding Write of the same data item, or 0 if there is none. The indices will allow reflexive reads-froms to find the correct version and also indicate which Read operations are involved in transaction reads-froms.

The multilevel transaction is then parsed into its subtransactions, which are distributed to the corresponding schedulers. The algorithm will execute the multilevel transaction by correctly executing the subtransactions and controlling the interleaving of subtransaction among the various transactions. The algorithm

must simultaneously insure that reads-froms are executed so as to generate a **1SR** history and yet not allow information to flow from high security levels to lower ones because of concurrency control mechanisms. (In this paper, we do not address covert channels that may arise for other reasons.)

We see two ways in which transaction processing might allow high level data to be transmitted to lower security levels. First, since multilevel transactions can have Write operations that execute after higher level data items have been read by the same transaction, one must be sure that any values written after reading higher level data items do not depend on the values read at the higher levels. This is precisely what is insured by the third condition of our definition of a multilevel transaction. Second, execution of transactions must be scheduled so that no rollback of lower level or noncomparable level subtransactions can result from the scheduling mechanisms for those at higher security levels. In particular, the concurrency control algorithm cannot allow a subtransaction to abort after another subtransaction of the same multilevel transaction has been executed at a lower level. Allowing this would require that the subtransactions at lower levels or noncomparable levels be rolled back (to satisfy the first condition of the definition of multilevel transaction), creating a covert channel.

The problem is avoided by the following technique. First, for a given multilevel transaction, subtransactions are executed in the order determined by the security lattice. That is, the subtransaction at a given security level cannot begin to execute its operations until all of the subtransactions at dominated security levels have locally committed. This guarantees that a reflexive read-from will always be able to find the correct version of the data item to be read. But it is not sufficient to insure correct schedules that avoid aborts.

Multiversion timestamp order schedulers require that each Write operation create a new version of the data item, and each Read operation will read the last version written by a committed transaction with an earlier timestamp (or from the last version written by the same transaction in the case of reflexive reads-froms). Aborts arise in such schedulers when a Write operation occurs after a Read operation on the same data item and the timestamp of the Read is later than that of the Write. That is, the Write operation has arrived too late to preserve the timestamp ordering. In such instances, the transaction requesting the Write is aborted because executing it would invalidate a Read operation that had already been performed. In our system, we cannot allow a subtransaction to locally abort if another subtransaction of the same multilevel transaction locally commits. Because the security classes form a lattice, and a transaction may have subtransactions at noncomparable security levels, to prevent covert channels we must insure that transactions never locally abort.

In other words, we must guarantee that if a subtransaction is going to write a data item, then no subtransaction with a later timestamp will ever want to read it. We must be sure that Read operations only occur after all subtransactions with later timestamps and that Write the same data item have been locally committed. Our security policy implies that it is sufficient that the local commitment criterion hold for subtransactions at security levels dominated by the level of the Read operation. The algorithm forces subtransactions to wait to start until its read set has a null intersection with the write sets of all subtractions that are active (not locally committed) at the same or lower security levels and have earlier timestamps. Notice that there is no reason to ever delay the execution of a lower level subtransaction because of a higher level subtransaction, since any relexive reads-froms can always locate the correct version of the required data item.

Finally, though these are really implementation details, we mention how one might start a multilevel transaction, though other scenarios are possible. The user would submit the transaction to **Q** by logging on the system at the least upper bound of the security classes of the operations of the transaction. If there is no operation of the transaction at the level of the greatest lower bound of the transaction's operations, then **Q** creates an artificial one, thereby reducing the amount of trusted code if the lowest levels of the transaction's operations are noncomparable, since then the indication that it is all right to start the transaction would be transmitted across noncomparable levels. The processing could then proceed as described above.

5. SPECIFICATION OF THE ALGORITHM

We need a few additional definitions before specifying the algorithm. We use $\mathbf{ts}(\mathbf{T}_i)$ to denote the timestamp assigned to the multilevel transaction $\mathbf{T}_i$. Similar notation is used for the timestamps of subtransactions and operations, which inherit them from their parent transactions. We denote by **lub(i)** the least upper bound of the security classes of the subtransactions of $\mathbf{T}_i$.

Definition If $\mathbf{T}_{i|=l}$ is a subtransaction, the *conflict set* of $\mathbf{T}_{i|=l}$, $\mathbf{CS}(\mathbf{T}_{i|=l})$, is $\cup\{\mathbf{WS}(\mathbf{T}_{j|\leq l}) \mid \mathbf{T}_{j|\leq l}$ is active$\}$.

$\mathbf{CS}(\mathbf{T}_{i|=l})$ is precisely the set of Write operations that have the potential to invalidate a Read operation of $\mathbf{T}_{i|=l}$, because subtransactions can only read data items at or below their own security level.

The algorithm processes transactions as follows.

At the global scheduler **Q**:

I.1 The initializing transaction $\mathbf{T_0}$ is received and assigned a timestamp.

I.2 As a transaction $\mathbf{T_i}$ begins to be received, it is assigned a timestamp and Read and Write indices are assigned.

a. ($\mathbf{w_i[x]}$,**l**) receives an index of 1 larger than the last preceding Write operation of $\mathbf{T_i}$ on **x** unless there are none, whence it receives an index of 1.

b. ($\mathbf{r_i[x]}$,**l**) receives an index equal to that of the last preceding Write operation of $\mathbf{T_i}$ on **x** unless there are none, whence it receives an index of 0.

I.3 After an operation ($\mathbf{o_i}$,**l**) is indexed, it is sent to a single-level subprocess $\mathbf{Q_l}$ of **Q**.

I.4 The $\mathbf{Q_l}$ construct the subtransactions $\mathbf{T_{i|=l}}$ for all relevant **l**, and form each $\mathbf{WS(T_{i|=l})}$ and $\mathbf{RS(T_{i|=l})}$.

I.5 The $\mathbf{Q_l}$ distribute the $\mathbf{T_{i|=l}}$, their read and write sets, and their index information and timestamps to the corresponding local scheduler $\mathbf{P_l}$.

I.6 Commit($\mathbf{T_i}$) messages are received from $\mathbf{P_{lub(i)}}$, and **Q** globally commits the multilevel transaction.

At each local scheduler $\mathbf{P_l}$:

II.1 $\mathbf{P_l}$ receives subtransaction packages for the $\mathbf{T_{i|=l}}$ from $\mathbf{Q_l}$ and determines $\mathbf{RS(T_{i|=l})} \cap \mathbf{CS(T_{i|=l})}$.

a. If $\mathbf{RS(T_{i|=l})} \cap \mathbf{CS(T_{i|=l})} = \varnothing$, then $\mathbf{P_l}$ initiates execution of $\mathbf{T_{i|=l}}$, provided that $\mathbf{T_{i|=m}}$ has been locally committed for all $\mathbf{m < l}$ in **SC**.

b. If $\mathbf{RS(T_{i|=l})} \cap \mathbf{CS(T_{i|=l})} \neq \varnothing$, then $\mathbf{P_l}$ queues $\mathbf{T_{i|=l}}$ for later execution.

II.2 As $\mathbf{P_l}$ executes $\mathbf{T_{i|=l}}$, it performs the operations according to the rules

a. For ($\mathbf{w_i[x]}$,**l**), a new version $\mathbf{x_{i,n}}$ is created where **n** is the index of the operation, i.e., ($\mathbf{w_i[x_{i,n}]}$,**l**) is done.

b. For $(\mathbf{r}_i[\mathbf{x}],\mathbf{l})$, if the index of the operation is $\mathbf{n}>0$, the value written by $(\mathbf{w}_i[\mathbf{x}_{i,n}],\mathbf{m})$ with index $\mathbf{n}$ is read. If the index is 0, $(\mathbf{r}_i[\mathbf{x}],\mathbf{l})$ reads from the last version of $\mathbf{x}$ written by a committed $\mathbf{T}_{j|=m}$ that has the largest timestamp $< \mathbf{ts}(\mathbf{T}_{j|=l})$.

c. When all operations of $\mathbf{T}_{i|=l}$ have been performed, $\mathbf{P}_l$ locally commits $\mathbf{T}_{i|=l}$.

II.3 For the $\mathbf{T}_{i|=l}$ that have been queued for later execution by II.1.b, $\mathbf{P}_l$ periodically checks whether $\mathbf{RS}(\mathbf{T}_{i|=l})\cap\mathbf{CS}(\mathbf{T}_{i|=l})=\varnothing$. If so then $\mathbf{P}_l$ initiates execution of $\mathbf{T}_{i|=l}$, provided that $\mathbf{T}_{i|=m}$ has been locally committed for all $\mathbf{m}<\mathbf{l}$ in $\mathbf{SC}$. (A new timestamp is not issued).

II.4 $\mathbf{P}_{\mathbf{lub(i)}}$ recognizes when every subtransaction of $\mathbf{T}_i$ has been locally committed, and then sends a Commit to $\mathbf{Q}$.

Steps I.1, I.2,and I.3 require some level of trust. These steps deal directly with the multilevel transaction, so will see operations at various security levels. The amount of trusted code needed to implement these is very small relative to what would be required to construct a complete trusted scheduler. The remainder of the algorithm can be done with untrusted processes.

Since the $\mathbf{Q}_l$ deal only with entities of a single security level, they may be untrusted (for the access control policy here), so I.4 and I.5 can be performed without trusted processes.

For I.6, notice that the global commitment of a multilevel transaction is a technical requirement only, used solely to let the "user" know that the work submitted is finished. The rest of the scheduling mechanism does not make use of this information, but relies only on the behavior of the local schedulers.

II.1 can be untrusted since the computation required relies solely on information available at $\mathbf{P}_m$ for $\mathbf{m}\leq\mathbf{l}$. The same observation applies to II.2 and II.3. II.4 clearly can be implemented untrusted.

6. VARIATIONS ON THE ALGORITHM

The version of the algorithm presented above is very pessimistic (conservative?) in that it waits to execute a subtransaction until it is absolutely certain that it can be executed without (locally) aborting or rolling back. Aside from minor changes that make the algorithm somewhat more optimistic, which may be

beneficial for some applications, we believe that the algorithm is intuitively as good as can be done for these kinds of transactions.

Given the nature of multilevel transactions, the conventional definitions of database theory, and the security policy, there seems to be limited choice in the approach to constructing a multiversion timestamp scheduler if trusted processes are to be minimized.

There appear to be three general statements that can be made about this family of algorithms. First, because of the security policy, it seems necessary that there be some way of determining when the subtransaction are finished, so that it is safe to use the values they produce. This is our notion of local commit or abort. Second, because of the required atomicity of database transactions, if one local commit occurs, then all local commits must occur. Third, the behavior of higher security level operations must conform to what is done at lower levels in order to obtain correct results and prevent covert channels. The variations of the algorithm arise by enforcing these three conditions in slightly different ways.

For reflexive reads-froms, a Read of a data item **x** cannot occur until the proper Write operation has been done, possibly by a lower security level subtransaction. We have chosen to make the higher security level subtransactions wait until the lower level ones from the same multilevel transaction have locally committed to insure that the correct version is available to be read. A more optimistic alternative would allow the higher level subtransaction to begin and progress until some Read operation could not be performed because the corresponding Write operation had not been completed. The higher level subtransaction could continue to attempt to read other data items until a Write operation is encountered (which could depend on a blocked Read operation for its value), at which point the whole subtransaction would be suspended. As the appropriate Write operations are done and the blocked Read operations completed, the subtransaction would resume. This technique requires significantly more overhead than the pessimistic approach, but may be warranted for some applications when it is unlikely that the reflexive reads-froms will result in waiting.

For transaction reads-froms, there are two variations, both more optimistic than the one presented. The first is similar to the technique for reflexive reads-froms. Subtransactions process their operations up to the point of reading data that might be invalidated by a Write operation at the same or lower security level, whence it is blocked until the lower level subtransaction from which it might read is locally committed. As before, such Read operations could continue until a Write operation is encountered, and then suspended. It could continue when the potentially offending lower level subtransaction is locally committed. The second variation is more optimistic in that it attempts to execute all subtransactions at the various security levels simultaneously. No initial determination of the read sets or write sets of potentially conflicting subtransactions is done. Rather, as

subtransactions are completed at lower security levels, the resulting read sets and write sets are compared with the results already obtained at the higher levels. If the higher level actions are inconsistent with those of the lower levels, they must be rolled back and redone, reading the correct versions and redoing the Write operations that depend on them. These roll backs may involve cascading of these reversals through all higher security levels. Whether either of these variants is more appropriate than the pessimistic approach depends on the frequency of the need to invoke suspensions or rollbacks.

7. PROOF OF CORRECTNESS OF THE ALGORITHM

We must show that the multiversion (multilevel) history produced by the algorithm is **1SR** by showing that it is equivalent to a one-copy serial history. To this end let **G** be the multiversion history produced by arranging the multilevel transactions in timestamp order with operation indices and versions ordered so that **G** is one-copy serial. That this is possible is trivial. Let **H** be the multiversion history produced by the algorithm. Clearly **H** satisfies the definition of a multiversion history. We have the following result.

Theorem Let $\mathbf{T}=\{\mathbf{T}_1,\mathbf{T}_2, \text{ - - -},\mathbf{T}_n\}$ be a set of multilevel transactions. If **H** is a multiversion history produced by the algorithm for **T**, then **H** is **1SR**.

Proof Let **G** be as defined above. It is obvious from the definition of **G** and the specification of the algorithm, that **G** and **H** have the same reflexive reads-from relationships. Thus it is sufficient to show that **G** and **H** have the same transaction reads-from relationships to prove the theorem.

First, suppose $\mathbf{T}_j$ reads-**x**-from $\mathbf{T}_i$ in **G** so there are $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})\in\mathbf{T}_i$, $(\mathbf{r}_j[\mathbf{x}],\mathbf{m})\in\mathbf{T}_j$ for which $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})<_G(\mathbf{r}_j[\mathbf{x}],\mathbf{m})$, and no other Write operation on **x** falls in between. Suppose now that $\mathbf{T}_j$ does not read-**x**-from $\mathbf{T}_i$ in **H**. Then there is a $(\mathbf{w}_k[\mathbf{x}],\mathbf{l})\in\mathbf{T}_k$ for which $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})<_H(\mathbf{w}_k[\mathbf{x}],\mathbf{l})<_H(\mathbf{r}_j[\mathbf{x}],\mathbf{m})$. Now if i=k, then $(\mathbf{r}_j[\mathbf{x}],\mathbf{m})$ would not have read **x** from the last Write of **x** by a committed subtransaction as required by the algorithm, so i, j, and k must be distinct. Since $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})<_H(\mathbf{w}_k[\mathbf{x}],\mathbf{l})$, the local scheduler $\mathbf{P}_l$ must have scheduled $(\mathbf{w}_k[\mathbf{x}],\mathbf{l})$ after $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})$, so $\mathbf{ts}(\mathbf{T}_i)=\mathbf{ts}(\mathbf{T}_{i|=l})<\mathbf{ts}(\mathbf{T}_{k|=l})=\mathbf{ts}(\mathbf{T}_k)$. Again, by the algorithm, $\mathbf{P}_m$ would not allow $\mathbf{T}_{j|=m}$ to begin until $\mathbf{T}_{k|=m}$ was locally committed, since it reads **x** after it, so $\mathbf{ts}(\mathbf{T}_k)=\mathbf{ts}(\mathbf{T}_{k|=l})<\mathbf{ts}(\mathbf{T}_{j|=m})=\mathbf{ts}(\mathbf{T}_k)$. Therefore $\mathbf{T}_k$ appears between $\mathbf{T}_i$ and $\mathbf{T}_j$ in **G**, contradicting that $\mathbf{T}_j$ reads-**x**-from $\mathbf{T}_i$ in **G**. Hence $\mathbf{T}_j$ does read-**x**-from $\mathbf{T}_i$ in **H**.

Conversely, suppose $\mathbf{T}_j$ reads-**x**-from $\mathbf{T}_i$ in **H** so there are $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})\in\mathbf{T}_i$, $(\mathbf{r}_j[\mathbf{x}],\mathbf{m})\in\mathbf{T}_j$) for which $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})<_H(\mathbf{r}_j[\mathbf{x}],\mathbf{m})$, and no other Write operation on **x** falls in between. Suppose now that $\mathbf{T}_j$ does not read-**x**-from $\mathbf{T}_i$ in **G**. Then there is a $(\mathbf{w}_k[\mathbf{x}],\mathbf{l})\in\mathbf{T}_k$ for which $(\mathbf{w}_i[\mathbf{x}],\mathbf{l})<_G(\mathbf{w}_k[\mathbf{x}],\mathbf{l})<_G(\mathbf{r}_j[\mathbf{x}],\mathbf{m})$. If i=k, **G** would not be one-copy serial. As before, i ,j, and k are all distinct, and

$ts(T_i)=ts(T_{i|=l})<ts(T_{k|=l})=ts(T_k)<ts(T_{j|=m})=ts(T_k)$ because **G** is one-copy serial in timestamp order. Since $T_{k|=l}$ writes **x** and has an earlier timestamp than $T_{j|=m}$ but later than that of $T_{i|=l}$, the algorithm would have finished $T_{k|=l}$ before starting $T_{j|=m}$, contradicting that T_j reads-**x**-from T_i in **H**. Thus T_j does read-**x**-from T_i in **G**.

We conclude that **G** and **H** have the same reads-from relationships and so are equivalent. Therefore **H** is **1SR**. ▌

8. CONCLUSION

The notions of atomicity for transaction processing that are usually suggested for databases are not easily reconcilable with those of multilevel secure systems. This is extremely problematic for multilevel secure database systems. Users' expectations may not be met if what the user considers a single transaction is decomposed into a sequence of single-level transactions that are then treated as non-communicating entities by the system's concurrency control mechanisms. Further, it is incumbent upon those who develop multilevel secure database systems to ensure that the users' needs and expectations are met to avoid misunderstandings about the system's functionality. To this end, we have proposed the idea of multilevel transactions to resolve these difficulties. In cases where this is not an acceptable solution, system-high systems may be a solution, or developing completely trusted database systems, though this would be a significantly more costly route.

In this paper we have defined *multilevel transaction* for multilevel secure databases and defined a notion of correctness that is consistent with the traditional idea of correctness for database systems. To demonstrate the applicability of these ideas, an algorithm for correct transaction processing within this framework was presented for a multiversion architecture multilevel database. Very few trusted processes are needed to implement the algorithm, which greatly reduces the time and cost needed to develop a system using the algorithm.

We chose to develop the algorithm for the kernelized architecture since it has been the one of most interest to the database security community. The problem for multilevel secure database systems based on the replicated architecture [5], however, is no less interesting a research (and application) issue. An algorithm for this case, based on the correctness criterion for transaction processing in replicated database systems, will be the subject of future work.

9. REFERENCES

1. D.E. Bell and L.J. LaPadula, "Secure Computer Systems:Unified Exposition and Multics Interpretation," The Mitre Corp., March 1976.

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

3. Oliver Costich, "Transaction Processing Using an Untrusted Scheduler in a Multilevel Database with Replicated Architecture", in *Database Security V:Status and Prospects*, ed. Sushil Jajodia and Carl Landwehr, North-Holland, 1992.

4. Oliver Costich and John McDermott, "A Multilevel Transaction Problem for Multilevel Secure Database Systems and its Solution for the Replicated Architecture", *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA May 1992.

5. Judith N. Froscher and Catherine Meadows, "Achieving a Trusted Database Management System using Parallelism," in *Database Security II:Status and Prospects*, ed. Carl Landwehr, pp.151-160, North-Holland, 1989.

6. Sushil Jajodia and Boris Kogan, "Transaction Processing in Multilevel-Secure Databases using Replicated Architecture" in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 360-368, Oakland, CA May 1990.

7. T.F. Keefe and W.T. Tsai, "Multiversion Concurrency Control for Multilevel Secure Database Systems" in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 369-383, Oakland, CA May 1990.

8. William T. Maimone and Ira B. Greenberg, "Single-Level Multiversion Schedulers for Multilevel Secure Database Systems" in *Proceedings of the Sixth Annual Computer Security Applications Conference*, pp.137-147, Tucson, AZ December 1990.

9. John McDermott, Sushil Jajodia, and Ravi Sandhu, "A Single Level Scheduler for the Replicated Architecture for Multilevel-Secure Databases" in *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX December 1991

10. Richard C. O`Brien, J.T. Haigh, and D.J. Thomsen, "Trusted Consistency Policy" Rome Air Development Center Technical Report RADC-TR-90-387, December 1990.

11. C.H. Papadimitriou, *The Theory of Concurrency Control*, Computer Science Press, 1986.