A Multilevel Transaction Problem for Multilevel Secure Database Systems and its Solution for the Replicated Architecture

Oliver Costich

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

John McDermott

Naval Research Laboratory
Code 5542
Washington, D.C. 20375

1. Introduction

A user of a database management system has an intuitive idea of a transaction as a sequence of database commands that he or she submits. The user expects this sequence of commands to be executed in the order of submission, without interference from other database commands submitted by other users. Techniques for doing this while concurrently supporting multiple database users are well known for conventional (i.e., not multilevel) database systems [2]. Transaction management theory for conventional database systems is not only mature, but useful in practice. The corresponding theory for multilevel secure database systems is still developing but some progress has been made [3,5,6,7,8,9,10].

In this paper we attempt to make further progress along a different dimension of the problem. Most of the transaction management theory for multilevel secure database systems has been developed for transactions that act within a single security class. In this paper, we look at transactions that act across security classes, that is, the transaction is a multilevel sequence of database commands, which more closely resemble user expectations. We then give an algorithm for controlling concurrent execution of these transactions on a particular mulitilevel secure database architecture.

2. The Problem

Human users of trusted database systems expect to execute what are effectively multilevel transactions. A human user can log on at several security levels to accomplish what he or she considers a single transaction. Users that do this expect the system to execute the single level pieces of these multilevel transactions in particular way, i.e., in the order they were submitted. In other words, the users expect the single multilevel process to be executed atomically across security levels. Most commonly accepted notions

of correctness and the attendant protocols for transaction processing in multilevel multiuser database systems fail to satisfy this expectation of the users [3,5,6,7,8].

A user may log on at a low security level and update some information in the database, and then subsequently log on at a higher level and attempt to use that information in a high level transaction. More abstractly, the user wants to write a value to a low security level data item and subsequently read it again from a higher level process. The user then expects that the value read will be the value he or she just wrote at the lower level, but this may not be the case unless both the read and write operations are implemented within a single atomic process.

In conventional database system environments, the expected behavior is ensured by including the two operations in the same transaction, or simply using the value without rereading it. Unfortunately, in the multilevel security environment, where transactions have been treated as single level subjects, the global transaction must be decomposed into multiple single level ones.

If the usual criteria of correctness for transaction processing (usually a serializability condition [2]) are then applied to these resulting single level transactions, the results may be contrary to the expectations of the user who submitted them. Without somehow enforcing the implicit ordering among the conflicting single level components, the scheduling process may reorder these components in another way. In the preceding example, the scheduler may delay the single level transaction that writes the newest value to the low level data item until after executing the higher security single level transaction, without violating the typical serializability requirement. Moreover, given two (or more) such global, multilevel transactions, the scheduler may interleave the single level transactions so that the multilevel transactions do not read the data items consistently, again still satisfying the serializability condition on the single level transactions, though not for the multilevel ones as expected by the user. The user expects the multilevel transaction to satisfy a serializability condition over all levels of the transaction.

An example should clarify the discussion. Suppose we have a multilevel secure database system that manages information about the status of all the nuclear warheads in the world. On edatabase in the system has information obtained by a special reconnaissance satellite. This satellite has a sensor that can pinpoint the location of nuclear warheads even when they are shielded, etc. The sensor has a very narrow aperture and scans regions of the earth using programmed patterns. There are two kinds of information in the satellite database: the ground track or location of the satellite, denoted by x, and the orientation of the satellite's sensor, denoted by y. The location of the satellite is unclassified, since it cannot be concealed.

Consider the problem of entering two successive reports from the satellite. We will simply number the reports One and Two. The value of y stored in the database is a function of x, that is y(x). Each report is entered by a sequence of multilevel secure database operations:

```
write(x);
change_session_level(S);
read(x);
write(y);
```

If the report is entered as single-level transactions, one possible result is:

```
    One                 Two
write(x);
                    write(x);
                    change_to_level(S);
                    read(x);
change_to_level(S);
read(x);
                    write(y);
write(y);
```

Since other transactions and users could access the database at any point during these updates, there are several views of the information that could be read. The database users need and expect the database to present a state containing $x_1, y_1(x_1)$ when report One has finished and a state containing $x_2, y_2(x_2)$ when report Two has finished. Instead, after report One has finished we have a state with $x_2, y_1(x_2)$. This view does not represent any real search performed by the satellite.

Speaking more theoretically, if we view this activity as two multilevel transactions, the schedule above is not serializable. If it is viewed as four single level transactions, then the schedule is serializable (as can be verified by examining the serialization graphs for the two cases).

The problem, then, lies in reconciling security considerations with the notion of transaction as it is commonly used in the database world. Transactions are atomic, in the sense that the transaction either executes completely and the results are made permanent, or it aborts and has no effect on the system. Beyond this, transactions are independent in the sense that no transaction communicates with any other transaction. This requirement is the source of the problem to be addressed in this paper. When a transaction that is inherently multilevel is decomposed into single level ones, any dependencies that may have existed between the levels of the original transaction are lost.

The next sections of this paper contain a discussion of how one might approach this problem and what approach we have chosen to make the problem tractable. To do this we will look at a restricted set of multilevel transactions and a notion of correctness for concurrent execution of them. We then turn our attention to providing a solution to this problem, a concurrency control algorithm for multilevel secure databases with replicated architecture.

3. Approaching the Problem

An obvious, but obviously unacceptable, solution is to trust the entire system. Practically, for a solution to be acceptable, the amount of trusted software that would have to be developed and evaluated should be minimal. In other words, a solution is viable only if a large proportion of the activities required can be carried out by untrusted processes.

If multilevel transactions are permitted to read and write data at all levels arbitrarily (at or below the clearance level of the user running the transaction), it is extremely difficult to eliminate the overt channel caused by a high level subject simply reading high level data and writing it into a low level data item. Because of the trust placed in the users, this behavior would be permitted in the paper world. Accomplishing the same thing in an automated system requires that the system be trusted from the security kernel up to the user interface, i.e., virtually the whole system would have to be trusted.

These considerations argue for some restrictions on the transactions that will be permitted. The difficulty related above results from allowing data to be written to a lower level after data has been accessed at a higher level. Disallowing these "writes-down" eliminates this difficulty. In other words, we permit only those multilevel transactions that, once having accessed data at a high level, can no longer write data at any lower level. This restriction assures that no information can flow from high levels to lower levels via operations of a transaction. It does not preclude covert channels arising from concurrent execution of transactions. One consequence of this restriction is that these multilevel transactions can be parsed into a sequence of single level subtransactions that can be executed in non-descending order of their security classes.

It is also necessary to adopt an appropriate concept of correctness for processing of these transactions. While the specific definition of correctness depends on the underlying architecture of the database system, there appears to be no reason to abandon serializability, as applied to the entire multilevel transaction, as the criterion. The ideas of this section will be presented more formally, and with further explanation, in our presentation of the model, below.

4. The Model

The security model used here is based on the fundamental access control restrictions of Bell and LaPadula [1], as applied to the replicated architecture. The notation for the security and the replicated architecture database model is adapted from that in Jajodia and Kogan [5], and that for the basic concepts of transaction processing from Bernstein, et al, [2].

4.1. Security Model

The database system (**DBS**) consists of a finite set $\boldsymbol{D}$ of *data items* that are objects of the trusted system, and a finite set $\boldsymbol{T}$ of *single-level transactions*, which act on behalf of users at a single security level, and are subjects of the trusted system. Admissible multilevel transactions, which will be more clearly specified below, will be constructed from the elements of $\boldsymbol{T}$. There is a lattice **SC** of *security classes*, (**SC**,<). If security classes **u** and **v** are in **SC** then **u** *dominates* **v** if $\mathbf{v} \leq \mathbf{u}$. There is also a *labeling function* **L** that assigns unique security classes to data items and transactions:

$$\mathbf{L}: \boldsymbol{D} \cup \boldsymbol{T} \rightarrow \mathbf{SC}$$

The notion of security here only encompasses mandatory access control requirements. Discretionary access control issues are not discussed. The mandatory access control requirements are:

(1) If single-level transaction **T** reads data item **x** then $\mathbf{L(T) \geq L(x)}$.

(2) If single-level transaction **T** writes data item **x** then $\mathbf{L(T) = L(x)}$.

Single-level transaction will be defined more carefully later. For now, one can think of a transaction executed by a single-level subject. Enforcement of these two conditions guarantees that information concerning high security level data items cannot flow to lower security level transactions (and users). The second condition is more restrictive than the usual ★-property in that **T** cannot write data item **x** if $\mathbf{L(T) < L(x)}$, i.e., no write-ups are permitted. In [5], it is argued that write-ups are undesirable in trusted database systems for integrity reasons and may permit covert channels. In any case, many transactions that write up in security level can be treated as a simple special instance of the multilevel transactions to be defined later in this paper.

4.2. DBS Architecture

The model for the replicated architecture used here is adapted from that described in [4]. The replicated database architecture is obtained by including a collection of single-level, untrusted database systems in the security model, one for each security class. That is, a set $\{C_v \mid v \in \mathbf{SC}\}$ of *back-end databases* is added. The system also contains a *front-end* processor, the trusted front end (**TFE**), which mediates the access of subjects (transactions, both single and multilevel) to objects (data items) in the back-end databases. The **TFE** will contain the trusted computing base (TCB), but not all of the **TFE** need be trusted. In particular, much of the scheduling mechanisms for the algorithm will be in the untrusted portion of the **TFE**.

Each database $\mathbf{C_v}$ contains copies of all data items in all databases whose security level is dominated by **v**. The copy of data item **x** in the database $\mathbf{C_u}$ is denoted by $\mathbf{x_u}$. Alternatively, if $\mathbf{L(x)=u}$ so $\mathbf{x} \in \mathbf{C_u}$, there is a copy of **x** in each database whose security level dominates **u**.

4.3. Transaction Model and Concepts

A database transaction is the execution of a set of atomic operations on the data items of the database. The operations permitted on the data items are Read(x), which returns the value stored by the data item, and Write(x), which changes the value of the data item to a specified value. Other transaction operations such as Start, Commit, and Abort, while significant for the control of transaction processing [2], need not be made explicit for communicating this algorithm. In fact, only committed transactions will be considered in defining the algorithm. If **T** is a transaction, then a Read(x) operation by **T** is denoted $\mathbf{r_T[x]}$, and a Write(x) operation by $\mathbf{w_T[x]}$.

Definition A *transaction* **T** is a totally ordered set with ordering relation $<_T$ where $\mathbf{T} \subseteq \{\mathbf{r[x]} \mid \mathbf{x} \in \boldsymbol{D}\} \cup \{\mathbf{w[x]} \mid \mathbf{x} \in \boldsymbol{D}\}$.

The definition requires that operations of a transaction be linearly ordered. We note here that the weaker definition of [2], which requires only a partial ordering could be used at some increase in the complexity of other definitions to accommodate multiple reads of the same data item. We will avoid this complexity by the assumption above. We say two operations, from perhaps different transactions, *conflict* if they operate on the same data item and at least one of them is a Write.

A single-level transaction is distinguished from a multilevel transaction by the fact that the latter mimics the behavior of an individual user who can log onto a multilevel system at any security level dominated by his clearance level. Thus a multilevel transaction is a transaction in which each operation is intended to be executed at a particular security level. Unless a security level is associated with an operation as part of the definition of the transaction, ambiguity in what is intended can occur. The operations of the multilevel transaction should be executed from the same security level as a user would enter them.

An operation of such a transaction can be viewed as a pair (**o[x]**,**a**), where **o[x]** is an operation on a data item and **a** is the security level from which the operation is to be executed. The security policy requires that **a** dominate **L**(**x**) in the security lattice when **o** is a Read operation, and be the same as **L**(**x**) when **o** is a Write operation. For simplicity, we write (**o[x]**,**a**) as $\mathbf{o_a[x]}$ and refer to $\mathbf{o_a[x]}$ as a multilevel operation.

We rely on context to discriminate between $\mathbf{o_a[x]}$, which is an operation on **x** specified by the user on a logical data item in a hypothetical single-copy database, and $\mathbf{o_a[x_u]}$, which is the corresponding operation performed by the replicated system on a particular copy $\mathbf{x_u}$ of **x** in $\mathbf{C_u}$.

The concept of multilevel transaction needs to be reformulated in terms of multilevel operations.

Definition A Definition A *multilevel transaction* **T** is a totally ordered set with ordering relation $<_T$ where $\mathbf{T} \subseteq \{\mathbf{r_a[x]} \mid \mathbf{x} \in \boldsymbol{D} \text{ and } \mathbf{a} \in \mathbf{SC}\} \cup \{\mathbf{w_a[x]} \mid \mathbf{x} \in \boldsymbol{D} \text{ and } \mathbf{a} \in \mathbf{SC}\}$. In addition, if $\mathbf{r_a[x]} \in \mathbf{T}$ then $\mathbf{L(x)} \leq \mathbf{a}$, and if $\mathbf{w_a[x]} \in \mathbf{T}$ then $\mathbf{L(x)} = \mathbf{a}$.

Notice that if we have a multilevel transaction for which all operations are executed from the same security level, then we have a conventional single-level transaction. We can now specify more formally the class of multilevel transactions that are of interest in this paper.

Definition A multilevel transaction **T** is *admissible* if for all **x**,**y** in $\boldsymbol{D}$, $\mathbf{o_a[x]} <_T \mathbf{w_b[y]}$ implies $\mathbf{b} \not< \mathbf{a}$.

A consequence of this restriction is that an admissible multilevel transaction **T** is equivalent to the sequence $\mathbf{T_a T_b} \cdots \mathbf{T_z}$ where each $\mathbf{T_u}$ is a single-level transaction executing at security level **u**, which consists of all the operations of the form $\mathbf{o_u[x]}$. This follows from the fact that two transactions are equivalent if one can be transformed into the other by interchanging adjacent pairs of non-conflicting operations [10]. If $\mathbf{T_a}$ contains the operation **o[x]**, we write $\mathbf{o_{Ta}}$ to distinguish that operation.

Definition A multilevel transaction **T** is in *canonical form* if $\mathbf{T} = \mathbf{T_a T_b} \cdots \mathbf{T_z}$ where each $\mathbf{T_u}$ consists of all operations of the form $\mathbf{o_u[x]}$, $\mathbf{x} \in \boldsymbol{D}$. Moreover, if $\mathbf{T_u}$ precedes $\mathbf{T_v}$ in this decomposition, then $\mathbf{v} \not< \mathbf{u}$.

It is standard to assume that transactions read or write each data item at most once. We can see from our earlier example that this assumption is inappropriate for multilevel transactions with respect to read operations. However it is reasonable to enforce this condition on the single-level components, the $\mathbf{T_u}$. This is not a severe restriction for the same reasons usually given for ordinary transactions [2].

Operations of several transactions can be commingled to permit concurrent execution. Execution of a set of transactions is represented as follows.

Definition A *complete multilevel history* **H** over a set of multilevel transactions {**S**, **T**, - - - , **Z**} is a partial order with ordering relation $<_H$ where

(1) $\mathbf{H} \supseteq \mathbf{S} \cup \mathbf{T} \cup \text{ - - - } \cup \mathbf{Z}$

(2) $<_H \supseteq <_S \cup <_T \cup \text{ - - - } \cup <_Z$

(3) If **p**, **q** are operations of **H** which conflict, then either $\mathbf{p} <_H \mathbf{q}$, or $\mathbf{q} <_H \mathbf{p}$.

Since only complete histories will be considered, they will be referred to simply as *histories*. Moreover, all histories are multilevel histories.

The preceding view of transaction processing is the view of the user, to whom replicas or versions of data items are transparent. 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, which must deal with copies of data items.

From the system's point of view, when a set of transactions is executed by a replicated **DBS**, an operation in a transaction must be translated into the equivalent operation on some or all of the copies of the data item. A *translation function* **h** performs the mapping. For a Read(x), **h** determines the copy of **x** to be read,and for a Write(x), **h** determines what copies of **x** are to be updated. In the case at hand, $\mathbf{h}(\mathbf{r}_{Ta}[\mathbf{x}]) = \{\mathbf{r}_{Ta}[\mathbf{x}_a]\}$, and $\mathbf{h}(\mathbf{w}_{Ta}[\mathbf{x}]) = \{\mathbf{w}_{Ta}[\mathbf{x}_v] \mid \mathbf{v} \geq \mathbf{a} \text{ in } \mathbf{SC}\}$. That is, logical Read operations are translated to Read operations on the actual copy of the data item at the security level of the operation, but logical Write operations must be translated to Write operations on all actual copies at higher security levels as well. Notice that although a multilevel transaction can read a data item more than once, since any given read operation is associated with a security class, it translates to a read operation on a single copy in the replicated database. In the case of write operations, updates must be propagated to all back-end databases at higher security levels.

The idea of a *replicated data history* is needed to represent the actions of the translated transactions on the replicated data. In the replicated architecture, two operations on data items *conflict* if they operate on the same copy of the data item and at least one of them is a Write. In the following definition, **o[x]** represents either a Read(x) or a Write(x) operation.

Definition A *multilevel replicated data history* **H** over a set of multilevel transactions ***T***={**S**, **T**, - - -,**Z**} is a partial order with ordering relation $<_H$ such that

(1) $\mathbf{H}=\mathbf{h}(\mathbf{S}) \cup \mathbf{h}(\mathbf{T}) \cup \; - - - \cup \mathbf{h}(\mathbf{Z})$

(2) If $\mathbf{r}_{Va}[x] <_V \mathbf{o}_{Vb}[y]$ in transaction $\mathbf{V}$, then $\mathbf{h}(\mathbf{r}_{Va}[x]) <_H \mathbf{p}$ for all $\mathbf{p} \in \mathbf{h}(\mathbf{o}_{Vb}[y])$

(3) If $\mathbf{w}_{Va}[x] <_V \mathbf{o}_{Vb}[x]$ in $\mathbf{V}$, then $\mathbf{w}_{Va}[x_u] <_H \mathbf{o}_{Vb}[y_u]$ for all $\mathbf{u} \in \mathbf{SC}$ such that $\mathbf{u} \geq \mathbf{a}$ and $\mathbf{u} \geq \mathbf{b}$

(4) If $\mathbf{p}, \mathbf{q} \in \mathbf{H}$ and they conflict, then either $\mathbf{p} < \mathbf{q}$ or $\mathbf{q} < \mathbf{p}$

(5) If $\mathbf{w}_{Va}[x] < \mathbf{r}_{Vb}[x]$, then $\mathbf{w}_{Va}[x_b]$ is in $\mathbf{h}(\mathbf{w}_{Va}[x])$

Replicated data histories represent the execution of a set of transactions as seen by the entire **MLS-DBS** rather than as seen by the user, to whom copies of data items are transparent. Notice that such histories preserve the orderings stipulated by the transactions (conditions (2) and (3)), and that this together with (5) ensures that if a transaction writes into a data item before it reads it, then it must subsequently read the value that it has written.

The ideas of *reads-from* and *final write* are essential to understanding the relationships among histories over the same set of transactions. These may be defined for one-copy or replicated data histories.

Definition Let **H** be a multilevel history (one-copy or replicated data) over a set of transactions ***T***.

(1) Transaction **T** *reads-***x***-from* **S** in **H** if $\mathbf{w}_S[x] <_H \mathbf{r}_T[x]$ and there is no transaction $\mathbf{V} \in \mathbf{T}$ for which $\mathbf{w}_S[x] <_H \mathbf{w}_V[x] <_H \mathbf{r}_T[x]$

(2) $\mathbf{w}_T[x]$ is a *final write* of **x** in **H** if there is no $\mathbf{V} \in \boldsymbol{T}$ for which $\mathbf{w}_T[x] <_H \mathbf{w}_V[x]$

This definition clearly makes sense for one-copy histories, and does also for replicated data histories if applied to a single copy of a data item. That is, "**T** reads-$\mathbf{x}_u$-from **S**" and "$\mathbf{w}_V[x_u]$ is a final write of $\mathbf{x}_u$" are meaningful. These ideas are used to define the notion of equivalent histories.

Definition Let **H** and **G** be multilevel histories of the same type (one-copy or replicated data) over the same set of transactions. **H** and **G** are *view equivalent* if they have precisely the same *reads from* relationships and the same *final writes.*

Correct execution of a set of transactions should appear to the user as if the transactions were executed one at a time in some order. This concept of correctness is formalized as follows.

Definition A multilevel history **H** is *serial* if for every pair of multilevel transactions **S**, **T** of **H**, either all operations of **S** appear before all those of **T**, or vice versa. A history **H** is *serializable* if it is view equivalent to a serial history.

Our algorithm for processing multilevel transactions on a replicated architecture database gives rise to a multilevel replicated data history. This history will represent a "correct" execution of the transactions if it appears to the user that the transactions executed serially on a one-copy database. Therefore a notion of equivalence between a one-copy history, the user's view, and a replicated data history, the system's view, is necessary. This requires the following definitions.

Definition If **H** is a multilevel replicated data history, say **T** *reads*-**x**-*from* **S** in **H** if

(1) For some $\mathbf{a} \in \mathbf{SC}$, $\mathbf{w}_{\mathbf{Sa}}[\mathbf{x}] \in \mathbf{S}$

(2) For all $\mathbf{b} \in \mathbf{SC}$, $\mathbf{b} \geq \mathbf{a}$, if $\mathbf{r}_{\mathbf{Tb}}[\mathbf{x}] \in \mathbf{T}$, then **T** *reads*-$\mathbf{x}_{\mathbf{b}}$-*from* **S**

Definition Let **H** and $\mathbf{H}_{\mathbf{1C}}$ be multilevel replicated data and one-copy histories, respectively, over the same set of transactions ***T***. **H** and $\mathbf{H}_{\mathbf{1C}}$ are *equivalent* if

(1) **H** and $\mathbf{H}_{\mathbf{1C}}$ have the same reads-from relationships, i.e., **T** reads-**x**-from **S** in **H** if and only if **T** reads-**x**-from **S** in $\mathbf{H}_{\mathbf{1C}}$.

(2) For each final write $\mathbf{w}_{\mathbf{Ta}}[\mathbf{x}]$ in $\mathbf{H}_{\mathbf{1C}}$, $\mathbf{w}_{\mathbf{Ta}}[\mathbf{x}_{\mathbf{u}}]$ is a final write in **H** for some $\mathbf{u} \in \mathbf{SC}$.

Definition A multilevel replicated data history **H** is *one-copy serializable* (**ML-1SR**) if it is equivalent to some one-copy serial multilevel history.

ML-1SR is the criterion for "correctness" thatwill be applied to algorithms for transaction processing in a multilevel replicated architecture database. The algorithm to be described herein yields multilevel replicated data histories thatare **ML-1SR**.

To specify the algorithm, we need a few additional concepts. First, we define the *update projection* $\mathbf{U}_{\mathbf{T}}$ of a transaction **T** to be $\{\mathbf{w}_{\mathbf{T}}[\mathbf{x}] \mid \mathbf{w}_{\mathbf{T}}[\mathbf{x}] \in \mathbf{T}\}$. If **T** is a read-only transaction, a dummy update projection is created. If $\mathbf{T}=\mathbf{T}_{\mathbf{a}}\mathbf{T}_{\mathbf{b}} \cdots \mathbf{T}_{\mathbf{z}}$, then $\mathbf{U}_{\mathbf{T}}=\mathbf{U}_{\mathbf{Ta}}\mathbf{U}_{\mathbf{Tb}} \cdots \mathbf{U}_{\mathbf{Tz}}$. For each transaction $\mathbf{T}_{\mathbf{a}}$, $\mathbf{U}_{\mathbf{Ta}}$ can be regarded as a single-level transaction that must be executed at each database $\mathbf{C}_{\mathbf{u}}$ for which $\mathbf{u} > \mathbf{a}$, to propagate the updates generated by $\mathbf{T}_{\mathbf{a}}$. In particular, each transaction $\mathbf{T}_{\mathbf{a}}$ on the replicated database can be decomposed into a primary transaction, which is also denoted $\mathbf{T}_{\mathbf{a}}$, which acts on $\mathbf{C}_{\mathbf{a}}$, and its update projection $\mathbf{U}_{\mathbf{Ta}}$, which acts on $\mathbf{C}_{\mathbf{v}}$ for $\mathbf{v} > \mathbf{a}$. It is often useful to think of a primary component or an update projection as a one-copy transaction acting on a single back-end database.

Since any two multilevel transactions **S** and **T** eventually execute operations, either primary or updates, at a common security level (all transactions execute their updates at the highest security class) then there is a lowest point in **SC** where this occurs. This *collision point* is determined as follows. Without loss of generality, we can assume that any multilevel transaction has a single level component whose security class is the greatest lower bound of the security classes of its other single level components. Let **s** and **t** be the minimum security classes of **S** and **T** respectively. Then the *collision point* of **S** and **T** is **max**{**s**,**t**}. The significance of the collision point of two transactions, **S** and **T** is that it determines the first point in the execution of database operations where the (serialization) order of the two transactions becomes relevant. Below the collision point, the relative orders of executing the operations of the single-level components of **S** and **T** are irrelevant with respect to serializability. The idea of collision point will become clearer when the algorithm is described.

5. Description of the Algorithm

The problem is to define a protocol for executing the primary transactions and the update projections in the "correct" way. As previously mentioned, a protocol will be considered "correct" if the resulting replicated data history is **ML-1SR**.

In the following, the symbol < will be used for all order relations (on the security lattice, transactions and histories). The intended order relation will be clear from the context in which it is used. The notation $\mathbf{T}_a$ will be used for both the transaction on the one-copy database and its primary component on the back-end database, with the context again the arbiter.

We give an overview of the algorithm before describing it in detail. The user who wishes to execute a multilevel transaction logs onto the system at a security level at or below that of any operation of the transaction (the greatest lower bound of the levels of the operations, say), and submits it to the **TFE**. The **TFE** puts the transaction in canonical form if necessary, and rejects it if it is not admissible. It verifies that the user's clearance dominates the security levels of all the operations. A trusted process at the log-in level distributes the single-level components of the transaction by writing down to the appropriate levels. A component whose security level does not dominate that of any other component may be sent to the scheduler in the corresponding back-end database. Other components are held awaiting their dispatch to the back-end database.

As single-level components and update projections from lower level security levels are executed, the resulting serialization order from the back-end scheduler is maintained as a list in the **TFE** at the level of the back-end database. At the next higher security level, there is an untrusted process in the **TFE** that can view the lists at lower levels and determine when it is permissible to retrieve an update projection from these lower level lists for execution. When it is permissible, the update projections from all lower level

components of the same multilevel transaction are retrieved, and any current level component of it is appended. The entire package is sent to the scheduler in the back-end database. The process continues until the transaction percolates up to the highest security level. We now take a more detailed look at the algorithm.

The algorithm is a variant of the usual *primary site* algorithm for replicated database systems [2]. Each back-end $\mathbf{C_u}$ will have a scheduler $\mathbf{B_u}$ that produces view serializable schedules for the transactions executed there (primary components and update projections). In addition, $\mathbf{B_u}$ must preserve, in its serialization ordering, the order in which it receives conflicting update transactions. There are several types of schedulers that accomplish this, among which are variants of conservative two-phase locking and conservative timestamp ordering protocols [2]. $\mathbf{B_u}$ need not be trusted.

A transaction $\mathbf{T=T_aT_b - - -T_z}$ or an update projection $\mathbf{U=U_{Ta}U_{Tb} - - -U_{Tz}}$ can be considered a sequence of single-level transactions or update projections. Consequently, we can speak of prefixes or subsequences of them. We will use this framework in manipulating sequences of these entities to "grow" update projections as the transactions are executed through the security lattice. We will need the concept of a *shuffle* of two sequences, which is any sequence whose elements are exactly those of the original two sequences and contains each as a subsequence.

Given any level $\mathbf{m}$ in $\mathbf{SC}$, $\mathbf{P_m(U_T)}$ is the subsequence of $\mathbf{U_T}$ containing those single-level update projections in $\mathbf{U_T}$ whose security level is dominated by $\mathbf{m}$. ($\mathbf{P_m}$ can be defined similarly for transactions.) That is, $\mathbf{P_m(U_T)}$ represents the sequence of update projections in $\mathbf{U}$ that must be executed at $\mathbf{C_m}$ to maintain data consistency. It can be regarded as a single-level transaction at $\mathbf{C_m}$.

A list $\mathbf{Q_m}$ is associated with each back-end database $\mathbf{C_m}$. The purpose of $\mathbf{Q_m}$ is to maintain a list of the $\mathbf{P_m(U_T)}$ that have been committed at $\mathbf{C_m}$. The list is ordered by the serialization order of the execution of these transactions, which need not agree with the order in which transactions are actually executed or committed. The $\mathbf{Q_u}$ are used to make the correct order of execution at lower security level back-end databases available to those at higher security levels.

In addition, there is, for each $\mathbf{u \in S}$, an untrusted mechanism $\mathbf{R_u}$ that maintains $\mathbf{Q_u}$ and can read the contents of $\mathbf{Q_v}$ for all $\mathbf{v \leq u}$ and is considered part of the global scheduler. Beyond this, $\mathbf{R_u}$ can receive and hold for execution the single-level components of transactions initiated at security levels dominated by $\mathbf{u}$.

The actual location of the $\mathbf{Q_u}$ or $\mathbf{R_u}$ is not important for the correctness of the protocol, but since any access to them must be monitored by the TCB, it is most efficient for them to be within the untrusted part of the **TFE**.

Recall that **u** covers **v** in a lattice if **u>v** and there is no **w** for which **u>w>v**. The distribution and timing of the update projections is controlled by the untrusted process $\mathbf{R_v}$ for **v>u**. If **v** covers **u**, $\mathbf{R_v}$ can scan $\mathbf{Q_u}$, retrieve an update projection, and dispatch it to the scheduler at $\mathbf{C_v}$. The crucial part of the protocol is in specifying the rules for retrieval and manipulation of the $\mathbf{P_u(U_T)}$ by $\mathbf{R_v}$. If not done correctly, the resulting histories will not be **ML-1SR**.

Notice that each $\mathbf{C_u}$, in isolation, can be considered a one-copy database. Primary transactions with security level **u**, and update projections from transactions whose security level is dominated by **u** can be considered as transactions on this one-copy $\mathbf{C_u}$. Therefore, the ideas of scheduling, execution, and commitment can be applied to these transactions locally (at $\mathbf{C_u}$). The scheduler $\mathbf{B_u}$ can generate a serializable schedule for these transactions at $\mathbf{C_u}$ and, as they commit, place the update projections into $\mathbf{Q_u}$ in the order of an equivalent serial schedule. The reader is referred to [5] for methods of placing update projections into $\mathbf{Q_u}$ in serialization order when that order differs from the commit-time ordering.

The protocol processes transactions as follows.

At each back-end database $\mathbf{C_u}$:

I.1 Primary transactions and update projections are received from the **TFE** and submitted to the local scheduler. Actions on data items are translated into the correct actions on local copies.

I.2 As local transactions (primary transactions and update projections) are committed, a report of their commitment is sent to the **TFE**. These reports are sent in an order consistent with the serialization order determined by the local scheduler.

At the **TFE**:

II.1 For each transaction $\mathbf{T=T_aT_b}$ - - - $\mathbf{T_z}$ submitted to the **TFE**, the single-level component $\mathbf{T_l}$ is distributed to $\mathbf{R_l}$. $\mathbf{R_l}$ submits $\mathbf{T_l}$ to $\mathbf{C_l}$ immediately if **l** dominates no other component's security class. Otherwise $\mathbf{T_l}$ is held awaiting execution.

II.2 The $\mathbf{R_u}$ scan the lists $\mathbf{Q_v}$ for those **v** for which **u** covers **v**, looking for $\mathbf{P_u(U_T)}$ satisfying the following conditions:

a. $\mathbf{R_u}$ has already retrieved and processed (as described below) all $\mathbf{P_v(U_S)}$ that were serialized before $\mathbf{P_v(U_T)}$ by $\mathbf{B_v}$.

b. If **u** also covers **w**, and $\mathbf{P_w(U_T)}$ will eventually appear in $\mathbf{Q_w}$, then $\mathbf{P_w(U_T)}$ does appear in $\mathbf{Q_w}$.

When these conditions are satisfied, $\mathbf{R_u}$ retrieves the string $\mathbf{P_v(U_T)}$ for the **v** covered by **u**, and creates a shuffle of them, say $\mathbf{U_u}$. If transaction **T** has a single-level component at level **u** that is being held by $\mathbf{R_u}$, then it is appended to the shuffle forming $\mathbf{U_uT_u}$. The resulting string of update projections and (possibly) a primary component is submitted as a single transaction to the scheduler $\mathbf{B_u}$ at $\mathbf{C_u}$ for processing.

II.3 Whenever a commitment report for some $\mathbf{U_uT_u}$ is received from $\mathbf{C_u}$, it is added to the end of $\mathbf{Q_u}$ as $\mathbf{P_u(U_T)}$.

The crux of the algorithm is II.2 because it controls the order in which updates are distributed to each back-end by holding the submission of updates until preceding updates are submitted. The condition testing can be performed by untrusted mechanisms.

The condition III.2.a is clearly detectable by $\mathbf{R_u}$. The significance of checking this condition is to ensure that for transactions **S** and **T**, their relative order serialization order determined at their collision point is maintained at all higher security levels.

Condition III.2.b is detectable by $\mathbf{R_u}$ because if **u** covers both **v** and **w**, and $\mathbf{P_v(U_T)}$ appears in $\mathbf{Q_v}$, then $\mathbf{P_w(U_w)}$ will appear in $\mathbf{Q_w}$ if and only if **T** was initially submitted at a security level that is dominated by both **v** and **w**. The significance of checking this condition is to ensure that all conflict relationships between transactions that occur below security level **u** have been recognized.

The replicated architecture must exact the price of maintaining the consistency of the replicated data. In this algorithm, the price is paid in two ways. First, there is the overhead of maintaining the lists and holding components for future processing in the **TFE,** in terms of both the space required and the added processing. Second, because updates are delayed, and the delay increases as the distance from the security level where submitted to higher levels increases, transactions at higher security levels may read data that may not be current. However, this algorithm is much faster than using single level transactions with repeated log-ins, and it guarantees the atomicity of the user's process as a whole.

6. Proof of Correctness

We will give only a brief version of the proof here. A more formal proof parallels that given in [3], and those interested are referred to that paper. We need to show the multilevel replicated data history created by the algorithm is **ML-1SR**.

First, we need a candidate for an equivalent one-copy serial multilevel history. To specify this, let **m** be the maximal element of the lattice **SC**. Then for each multilevel transaction

$\mathbf{T}$, $\mathbf{P_m(U_T)}$ and $\mathbf{T}$ are equivalent as ordinary transactions on $\mathbf{C_m}$. The serialization order for the $\mathbf{P_m(U_T)}$ in $\mathbf{Q_m}$ defines the one-copy serial history for the multilevel transactions.

A key observation in seeing that the algorithm is correct is that for each pair of multilevel transactions, their relative serialization order will be established by the scheduler in the back-end database corresponding to their collision point. The algorithm then preserves that order at all higher level back-end databases. Below the collision point, the order of execution of the operations these two transactions is irrelevant since no ordering between them can be established there. It is relatively straightforward to see that the one-copy history is equivalent to the multilevel replicated data history, as follows.

Obviously, final writes are the same in the two histories. If $\mathbf{T}$ reads-$\mathbf{x}$-from $\mathbf{S}$ in the one-copy serial history, then there is an $\mathbf{a \in SC}$ for which $\mathbf{T}$ reads-$\mathbf{x_a}$-from $\mathbf{S}$ in the replicated data history. If $\mathbf{T}$ fails to read-$\mathbf{x}$-from $\mathbf{S}$ in the replicated data history, then there is a $\mathbf{b \in SC}$, $\mathbf{b \geq a}$, for which $\mathbf{r_{Tb}[x] \in T}$ and $\mathbf{T}$ does not read-$\mathbf{x_b}$-from $\mathbf{S}$. That is there is some multilevel transaction $\mathbf{V}$ that writes $\mathbf{x_b}$ between $\mathbf{S}$ writing it and $\mathbf{T}$ reading it. Then $\mathbf{V}$ conflicts with both $\mathbf{S}$ and $\mathbf{T}$ and would have to fall between them in the one-copy history, which contradicts the original assumption.

Conversely, if $\mathbf{T}$ reads-$\mathbf{x}$-from $\mathbf{S}$ in the multilevel replicated data history, then there is some point $\mathbf{a \in SC}$ for which $\mathbf{T}$ reads-$\mathbf{x_b}$-from $\mathbf{S}$ for all $\mathbf{b \geq a}$. This condition, in the one-copy history, precludes any transaction writing the data item $\mathbf{x}$ between $\mathbf{S}$ and $\mathbf{T}$. Thus $\mathbf{T}$ reads-$\mathbf{x}$-from $\mathbf{S}$ in the one-copy history.

In a general sense, it is not possible to improve on this algorithm. Once one-copy serializability is selected as the criterion for correctness, the preservation of the relative serialization orders at all security levels must be maintained. Establishing this order at a high security level would require that this information be made known to lower security levels, and thus create a potential covert channel. Therefore any algorithm for the replicated architecture of the primary site type that guarantees one-copy serializability must establish the serialization order between transactions at the lowest possible level and maintain it by propagating it up through the security lattice. The algorithm presented here demonstrates one way to do this by reading down to lower levels to learn the correct order. A technique that writes up could be used as effectively. A more optimistic approach might be to initially distribute a transaction to all appropriate back-end databases simultaneously and execute them. As the serialization order is propagated upward from lower security classes, transactions that were improperly ordered at higher levels would have to be rolled back and redone. Depending on the specific application and system there are many implementation variants on this theme, but the basic requirement to determine the order at the lowest level and sustain it remains.

We should point out that there are some variants of the "immediate write" class of algorithms thatavoid this upward propagation by executing write operations

simultaneously at all levels, but these have significantly more overhead. The concurrency control mechanisms of the back-end databases are disabled and is implemented in the **TFE** instead. Operations on data are performed one at a time and "simultaneously" on all the back-end databases.

7. Garbage Collection and Recovery

In implementing the protocol as described, the size of the lists, the $\mathbf{Q_u}$, can become arbitrarily large. This may waste storage as well as increase the execution time for scanning the $\mathbf{Q_u}$ by $\mathbf{R_u}$. To remedy this situation, some form of *garbage collection* should be included to maintain the $\mathbf{Q_u}$ at reasonable dimensions.

The security policy does not allow $\mathbf{R_u}$ to access information that would indicate whether a particular $\mathbf{P_u}(\mathbf{U_T})$ must be kept for future use by the protocol, as this depends on information known only at higher security levels. Therefore, garbage collection requires that trusted mechanisms be used. The earliest point at which a particular $\mathbf{P_u}(\mathbf{U_T})$ may be discarded from $\mathbf{Q_u}$ is when $\mathbf{U_i}$ has been retrieved from it (and dispatched to the appropriate back-end database) by all the $\mathbf{R_v}$ for which **v** dominates **u**.

Trusted components could be built to perform the deletions at this point, but would be inordinately complex for the task. A more likely approach would be to wait until a particular $\mathbf{P_m}(\mathbf{U_T})$ is inserted into $\mathbf{Q_m}$, where **m** is the maximum class in the security lattice. In fact, this may be the only reason for maintaining $\mathbf{Q_m}$ (other than to simplify the proof of correctness), since it is otherwise unnecessary. A relatively simple trusted mechanism could then remove $\mathbf{P_u}(\mathbf{U_T})$ from all of the relevant $\mathbf{Q_u}$. Such a mechanism can be invoked at regular intervals. Doing garbage collection at the checkpoints taken for recovery purposes may be sufficient.

As for recovery, the back-end databases have their own recovery managers, so that the only concern is the recovery of the contents of the dynamic data structures in the **TFE**. The recovery scheme to accomplish this is straightforward and quite similar to what is generally used for databases. A log is maintained in the stable storage corresponding to security level **u**. The log for security level **u** can be located on the same hardware as the back-end database $\mathbf{C_u}$ to maintain security of the recovery logs. Alternatively, a collection of single level logs could be maintained in the stable storage dedicated to the **TFE**. Whenever $\mathbf{R_u}$ receives an update report from a back-end database and adds it to $\mathbf{Q_u}$, or receives a component single-level transaction for later execution, a log entry is created and written to the log in stable storage. If the **TFE** should fail, the logs can be used to reconstruct the $\mathbf{Q_u}$ and the data structures of $\mathbf{R_u}$.

In addition, as for databases in general, a *checkpoint* can be taken, recording the state of the whole DBS. Performing the garbage collection function at this point and pruning the logs of any unnecessary entries reduces the amount of data that is stored.

After a crash, recovery is accomplished by restoring the state at the last checkpoint and using the logs to update the **DBS** to the point of failure. The proposed technique requires little trusted code.

8. Conclusion

The notions of "correctness" for transaction processing that are usually suggested for multiuser databases are not necessarily appropriate when these databases are also multilevel secure 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. 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.

In this paper we have proposed a definition of *multilevel transaction* for multilevel secure databases and defined a notion of correctness that is consistent with the traditional idea of correctness for replicated systems. To demonstrate the applicability of these ideas, an algorithm for correct transaction processing within this framework was presented for replicated architecture multilevel databases.

We chose to develop the algorithm for this architecture since we are actively involved in building a prototype of such a system. The problem for multilevel secure database systems based on the kernelized architecture, however, is no less interesting a research issue. An algorithm for this case, using a multiversion technique, will be the subject of future work. In addition, there is a need to extend these notions to a more general class of multilevel transaction.

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*, Sushil Jajodia and Carl Landwehr, editors, North-Holland, 1992.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

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

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