CONCURRENCY CONTROL IN MULTILEVEL-SECURE DATABASES BASED ON REPLICATED ARCHITECTURE*

Boris Kogan†
Sushil Jajodia†

Department of Information Systems and Systems Engineering
George Mason University
4400 University Drive
Fairfax, VA 22030-4444

ABSTRACT

In a multilevel secure database management system based on the *replicated* architecture, there is a separate database management system to manage data at or below each security level, and lower level data are replicated in all databases containing higher level data. In this paper, we address the open issue of concurrency control in such a system. We give a secure protocol that guarantees one-copy serializability of concurrent transaction executions and can be implemented in such a way that the size of the trusted code (including the code required for concurrency and recovery) is small.

1. Introduction

A majority of databases in the Department of Defense (DoD) and the intelligence community contain data that are classified to have different security levels. All database users are also assigned security clearances, and it is the responsibility of a *secure* database management system (DBMS) to assure that all users will have access to only those data for which they have been granted a clearance. Although most commercial DBMS's provide some form of data security, they do so by controlling modes of access privileges of users to data. (See, for example, [13].) These, so-called *discretionary* access controls, do not provide adequate mechanisms for preventing unauthorized disclosure of information. For example, they cannot prevent *Trojan horse*† attacks. Therefore, commercial DBMS's are not suitable for use in military systems. Military systems require additional mechanisms for enforcing *mandatory* (or *nondiscretionary*) access controls‡ that are based on security levels and clearances *plus* an assurance that the system is free of covert channels.§ As a consequence, databases containing sensitive data are operated in a *system-high* mode: every user is cleared to see all the information stored in the database. Not only is this approach expensive (since background investigations are quite costly), but it increases security risk also (since the number of users having the highest possible clearance becomes needlessly large).

Motivated by these concerns, U. S. Air Force organized a study group which convened in July, 1982 at Woods Hole, Massachusetts to examine multilevel security in the database context. Their report, commonly known as the Woods Hole Report [1], considered different architectures for building secure multilevel DBMS's from existing untrusted DBMS's. Under the assumption that a user always sees a logically integrated multilevel database, the study derived three interesting architectures, depending on how the multilevel data are physically stored [1, pp. 21-22].

* This work was supported by the U. S. Air Force, Rome Air Development Center through subcontract # RI-64155X of prime contract # F30602-88-D-0028, Task B-9-3622 with University of Dayton.

† Also with the Center of Excellence for Command, Control, Communications, and Intelligence at George Mason University. The Center receives general support via grants from the Virginia Center for Innovative Technology, MITRE, the Defense Communications Agency, and PRC/ATI.

† A Trojan horse is a malicious piece of code which is hidden within a program and leaks information to unauthorized users. As an example, an innocuous-looking sort routine may have hidden in it a Trojan horse such that whenever a user invokes the sort routine, in addition to accessing the user's file to be sorted, it accesses other files belonging to this user and copies them into files belonging to some unauthorized user.

‡ Mandatory access controls provide protection against Trojan horses. See Section 2.

§ Covert channels are paths not normally meant for information flow that could nevertheless be used to leak information from high to low subjects.

In the first architecture, called the *kernelized*[+] DBMS, the multilevel database is partitioned into single-level databases which are then stored separately (See Figure 1.)

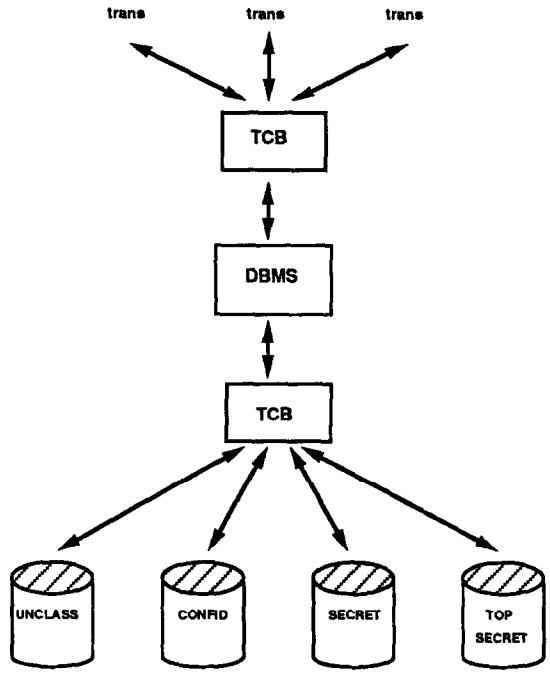

Figure 1

The second architecture, which we refer to as the *replicated*[*] DBMS, uses a separate database management system to manage data at or below each security level, a database at a security class contains all information at its class and below, and therefore, lower level data are replicated in all databases containing higher level data. (See Figure 2.) The third architecture is based on the integrity lock technology, and therefore, appropriately named *spray paint* DBMS or *integrity lock* DBMS

The study found problems with each of the three architectures. (See Section 3 for a more detailed discussion.) According to the study, in a replicated DBMS, any changes in a low database cannot be propagated to the higher databases by the untrusted low database since this could corrupt the higher databases.[§] Similarly, any changes to the low data in a high database cannot be propagated to the lower databases by the untrusted high database since otherwise we have an illegal information transfer from high to lower databases. Therefore, the report concludes, the changes must be propagated across various databases by a trusted synchronization mechanism, however, the code to implement such a mechanism would be large, and therefore, difficult to verify[†]

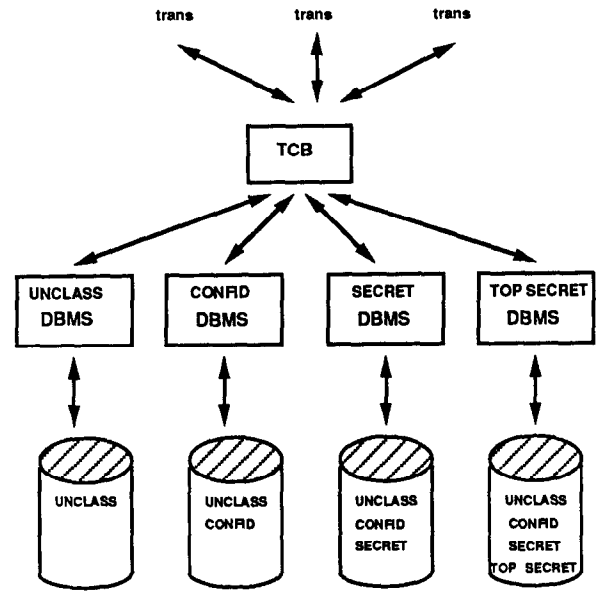

Figure 2

In this paper, we produce evidence to the contrary by giving a concurrency control algorithm for a system which is based on the replicated architecture Our protocol guarantees one-copy serializability of concurrent transaction executions; it is secure since the information always flows in one direction–from databases at lower security classes to databases at higher security classes, and it can be implemented in such a way that the size of the trusted code (including the code required for concurrency and recovery) is small.

The remainder of the paper is organized as follows. We begin in Section 2 with a brief tutorial on the military security concepts. In Section 3, we again discuss the three secure DBMS architectures, point out the problems with the kernelized and integrity lock architectures, and give a more detailed, but informal description of a secure DBMS based on the replicated architecture In Section 4, we present an abstract model that isolates the transaction processing aspects of the replicated DBMS architecture and defines what it means for such a system to function correctly. In Section 5, we give two simple algorithms that at the first glance seem to work and show why they do not work. In Section 6, we describe our protocol for transac-

[+] Perhaps *partitioned* DBMS would be more appropriate in this paper

[*] Called *distributed* DBMS in [1]

[§] This is an integrity flaw, not a security one While integrity flaw is undesirable, it is an unavoidable consequence of any architecture that relies on untrusted DBMS's

[†] The DoD Trusted Computer System Evaluation Criteria [2] requires a formal top-level specification of the system and a demonstration that the implemented system is consistent with the specification Unfortunately, it is impractical to perform the latter task rigorously for large programs

tion processing and prove that it is correct We also consider the issue of recovery and show how the database can be brought to a consistent state after failures. The conclusion is given in the last section.

2. Basic Concepts of Multilevel Security

Below we give a brief description of the relevant security concepts. For a more detailed discussion, we refer the reader to [6] or [21]

The way in which a secure DBMS controls access to data is known as the system's *security policy* In the context of military databases, a secure DBMS must enforce a suitable interpretation of the DoD security policy (mandatory access controls), and a well-accepted interpretation is based on the Bell-LaPadula model [3] This model is stated in terms of *subjects* and *objects*. An object is understood to be a data file, record, or a field within a record. A subject is an active process that can request access to objects. Every object is assigned a classification, and every subject a clearance Classifications and clearances are collectively referred to as *security classes* (or levels) A security class consists of two components. a hierarchical component (usually, *Top Secret*, *Secret*, *Confidential*, and *Unclassified*, in this order) together with a set (possibly empty) of non-hierarchical categories (e.g., NATO or Nuclear) Security classes are partially ordered as follows: Given two security classes C_1 and C_2, $C_1 \geq C_2$ iff the hierarchical component of C_1 is greater than or equal to that of C_2 and the categories in C_1 contain those in C_2. Since the set inclusion is not a total order, neither is $\geq$.

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

The Bell-LaPadula model imposes the following restrictions on all data accesses:

(a) *The Simple Security Property.* A subject is allowed a read access to an object only if the former's clearance is identical to or higher (in the partial order) than the latter's classification.

(b) *The *-Property* (pronounced "the star property"). A subject is allowed a write access to an object only if the former's clearance is identical to or lower than the latter's classification.

The above two restrictions are intended to ensure that there is no flow of information from high objects to low subjects. For otherwise (since subjects can represent users[†]) a breach of security occurs wherein a user gets access to information for which he has not been cleared.

Since the above restrictions are mandatory and enforced automatically, the system checks security classes of all reads and writes, which provides protection from Trojan horses.

As an example, a subject with a *Secret* clearance, can read *Secret*, *Classified* and *Unclassified* items, while it can write to *Secret* and *Top Secret* items.

It turns out that the system may not be secure even if it always enforces the two Bell-LaPadula restrictions correctly A secure system must guard against not only the direct revelation of data but also violations that produce illegal information flows through indirect means. *Covert* channels fall into that category of violations. They provide indirect means by which information at high security classes can be passed down to subjects at lower security classes. To illustrate, suppose the database uses a two-phase commit protocol to commit a transaction. Further, suppose that a certain transaction requires a *ready-to-commit* response from both a *Secret* as well as an *Unclassified* process to commit it; otherwise the transaction is aborted. From a purely database perspective, there does not appear to be a problem, but from a security viewpoint, this is sufficient to compromise security. Since the *Secret* process can send one bit of information by either agreeing or not agreeing to commit a transaction, both *Secret* and *Unclassified* processes may cooperate to compromise security as follows. The *Unclassified* process generates a number of transactions; it always agrees to commit a transaction, but the *Secret* process by selectively causing transaction aborts can establish a covert channel to the *Unclassified* process. (See [20] for additional examples.)

There is one other aspect of secure DBMS's which we need to mention. To meet DoD requirements [2], it must be possible to demonstrate that the DBMS is secure To this end, the secure DBMS designers follow the concept of a *trusted computing base*[‡] (TCB), an approach conceived by Schell, which is responsible for all security relevant actions of the system. (See [21, p.254]) TCB mediates all database accesses and cannot be bypassed, it is small enough and simple enough so it can be verified that it works correctly; and it is isolated from the rest of the system so it is tamperproof.

3. Three Architectures Revisited

As we mentioned in the introduction, the Woods Hole Report identified three interesting architectures for building secure multilevel DBMS's from existing untrusted DBMS's, depending on how the multilevel data is physically stored [1, pp. 21-22]. In the kernelized DBMS, the multilevel database is partitioned into single-level databases that are stored separately. (See Figure 1.) The replicated DBMS uses a separate database management system to manage data at or below each security level; a database at a security class contains all information at its class and below. (See Figure 2.) The third architecture is based on the integrity lock technology.

Since the Woods Hole Report raised serious concerns about the replicated architecture and it turned out

[†] It is a mistake, however, to completely identify subjects with users While the users may be trusted not to divulge knowingly classified data to uncleared users, subjects (processes) running on behalf of the users cannot be trusted to enforce security properly For example, subjects may contain Trojan horses

[‡] Also known as *security kernel* or *reference monitor*

that the integrity lock architecture is vulnerable to Trojan horse attacks (see, for example, [15, pp 65-66]), most of the research in secure databases has been focused on the kernelized architecture [7, 13, 14, 17, 23] As has been pointed out in [1, 8, 9, 10], systems based on the kernelized architecture may suffer from poor performance There are several reasons for this. First, partitioned approach works well when database queries involve data only at a single level, however, any query involving multilevel data must be decomposed into single-level queries and answers from each single-level database must be buffered and then combined to satisfy the query This means that satisfying multilevel queries involves taking repeated natural joins of relations, and it is well-known that natural join is an expensive operation. Second, the usual concurrency control algorithms are no longer applicable in a kernelized DBMS (for example, locking cannot be used since a high transaction wishing to read a low data item cannot keep a low transaction from writing that item by placing a read lock on it), and a scheme which uses a simple counter must be used instead (see [14, pp 114-125] or [24, pp 120-121]). We believe that this simple counter scheme will impose an intolerable overhead and most likely will not be appropriate for enforcing mutual exclusion in a database system.† Finally, kernelized DBMS's require that indices must be stored as multiple files, which may also have an adverse impact on the performance [9, 10]

In view of these performance concerns, there has been a reexamination of replicated architecture. Froscher and Meadows in [8] propose using a general-purpose parallel machine (BBN's Butterfly) to build a secure DBMS which has the potential for substantial performance improvement relative to the kernelized architecture (See Figure 2.) There is a trusted front-end (TCB) which mediates all accesses to the database A user wishing to access the database requests a session at a certain security class. If the user is cleared to that class, the TCB assigns the user to the appropriate database, and any user queries and updates are subsequently directed to that single-class database When a user wishes to update the database, the TCB automatically assigns the security class of the user's session to the update Since the lower data is replicated at higher levels, the TCB broadcasts the update to all databases whose security levels dominate the security level of the update Exactly how this broadcast is accomplished in a secure fashion is left as an open problem.

It is easy to see that the replicated architecture in Figure 2 addresses the security concerns specified in the Woods Hole report Since a user's query and updates are always assigned to a database at a single security class, all covert channels associated with multilevel queries are eliminated. Since there are no mandatory security considerations which need to be taken into account, the challenge is to come up with a secure mechanism to propagate updates across various databases that not only preserves database consistency, but does not have to be trusted at the same time

The Secure Distributed Data Management (SDDM) system [17] and the Secure Distributed DBMS (SD-DBMS) [23] developed by the Unisys Defense System also allowed for the possibility of a complete replication of lower data at higher levels. Unfortunately, their synchronization algorithm, which is crucial to the replicated architecture, is incorrect The algorithm they use is based on a primary copy scheme Once an update is successfully performed on the primary copy, any changes are then propagated to appropriate copies. In this paper, we show that this propagation must occur only in a particular order; otherwise, the correctness of the concurrent executions of transactions cannot be guaranteed. (See Section 5.2.)

In this paper, we give a transaction processing algorithm in a system based on the replicated architecture in Figure 2. Our protocol has many desirable properties. We show that our protocol guarantees one-copy serializable executions of concurrent transactions. This means that an execution of transactions in the replicated database is equivalent to a serial execution of the same transactions on a *single-copy* database. Second, our algorithm is secure in the sense that the information flow always occurs in one direction – from lower databases to higher databases. Finally, we rely for the most part on the recovery manager of each single-level database for the purpose of global recovery. The motivation behind this objective is that we wish to keep the amount of trusted code within the TCB as small as possible. The code for recovery in a DBMS, on the other hand, is usually quite large, and moreover this code is very hard to write [5] As an example, according to [12], about 10% of the System R code is devoted to recovery,† and writing code for a recoverable action is 30% harder and requires 20% more code than for a nonrecoverable action.

4. The Model

4.1. Security Model.

The system consists of a set D of *objects* (data elements), a set T of *subjects* (transactions), and a partially ordered set S of *security classes* with ordering relation $<$ A class S_i is said to be *dominated* by another class S_j (denoted $S_i \leq S_j$) if $i = j$ or $S_i < S_j$.

There is a mapping L from $D \cup T$ to S, i.e., for every $x \in D$, $L(x) \in S$, and for every $T_i \in T$, $L(T_i) \in S$ In other words, every data element as well as every transaction has a security class associated with them.

The system is considered *secure* only if the following two conditions are satisfied at all times.

(1) Transaction T_i is not allowed to read data element x unless $L(T_i) \geq L(x)$.

† Database folklore suggests that 20% of the data items are accessed 80% of the time Since certain items are likely to be very heavily accessed, the likelihood of transactions having to abort and retry the read accesses becomes large, and in the worst case, the number of times transactions may have to abort and retry has no upper bound, creating a *livelock* situation [4]

† The figure for IMS is even larger [5]

(2) Transaction T_i is not allowed to write data element x unless $L(x) = L(T_i)$

Intuitively, adherence to the above two conditions guarantees that there is no information transfer from objects (i.e., data elements) with higher security levels to subjects (i.e., transactions) with lower levels.

Property (2) is a restricted version of the *-property The original *- property [3] allows T_i to write x when $L(T_i) \leq L(x)$ In the database context, however, a write-up, i.e., the situation when $L(T_i) < L(x)$, is undesirable for integrity reasons. Namely, if the update that T_i intends to make to x would violate an integrity constraint, T_i should normally be aborted. However, if aborts for the reason of integrity violations on high data were allowed, then an untrusted transaction manager would be in a position to open up a covert channel to low transactions by making selective aborts. Thus, out of security considerations, such aborts should be prohibited.

4.2. Replicated Architecture Model

Our model of the replicated architecture for multilevel secure database systems is illustrated in Figure 3 It consists of a set C of *containers*, one for each security class,† i.e., $card(C) = card(S)$‡ $(C = \{C_1, C_2, ., C_N\})$. We generalize the definition of L to make it a mapping from $D \cup T \cup C$ to S Then, for all $C_i \in C$, $L(C_i) \in S$ and $L(C_i) = L(C_j)$ iff $i = j$

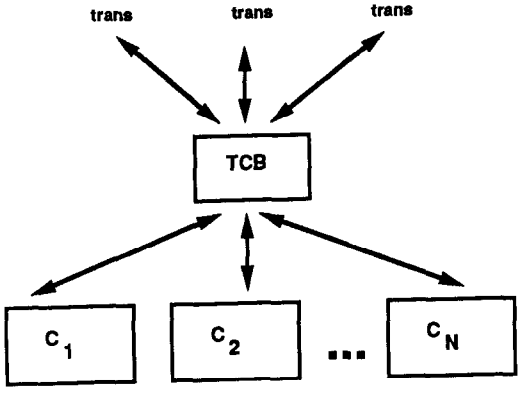

Figure 3

Each container is essentially a separate (non-trusted) database that stores a copy of all the data whose class is dominated by that of the database, i.e., for all $x \in D$, a copy of x is contained in C_i iff $L(x) \leq L(C_i)$ The copy of x in C_i is denoted x_i

The Trusted Computer Base (TCB) is a trusted transaction interface for the entire system. The functionality of the TCB is defined in the next subsection.

† This is not impractical since most defense applications need to support only two or three security classes [17]

‡ For a set X, $card(X)$ denotes its cardinality

4.3. The Transaction Model

We define a database *transaction* as a sequence of atomic operations on data elements. An operation on a data element is either a Read (returns the value of the element) or a Write (updates the element with a specified new value) For simplicity, we assume that every transaction can read and write any data element at most once (Note that this is not really a limiting assumption.) Then, for any transaction T_i and data element x, $r_i[x]$ denotes a Read executed by T_i on x. Similarly, $w_i[x]$ denotes a Write executed by T_i on x In general, a transaction does not have to be a totally ordered sequence. When two operations are not ordered relative to each other, it means that they can be executed in any order A Read and a Write on the same data element, however, always must be ordered.

Definition 1. A *transaction* T_i is a partially ordered set with ordering relation $<_i$ such that

(1) $T_i \subseteq \{r_i[x] \; x \in D\} \cup \{w_i[x] \cdot x \in D\}$,

(2) for all $x \in D$, if $r_i[x] \in T_i$ and $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$. □

To model the distribution of updates produced by a given transaction to other containers, we need some more definitions.

Definition 2. A set $U_i = \{w_i[x]. \; w_i[x] \in T_i\}$ is called an *update projection* of transaction T_i T_i is called a *parent transaction* of U_i. □

Definition 3. An *update report* R_{ij} on transaction T_i by container C_j is an ordered pair (U_i, C_j). □

Update projections and update reports are used to propagate updates among containers. Having executed transaction T_i, container C_j constructs an update projection U_i and sends the update report $R_{ij} = (U_i, C_j)$ to the TCB The TCB subsequently distributes U_i to all containers that are permitted to see it, i.e., all containers C_k with $L(C_k) > L(C_j)$. Each U_i is executed at a C_k just like a regular transaction.

We will see that the exact order in which the distribution occurs is crucial to the correctness of the protocol and will be described in the next section.

In the rest of this paper, when no ambiguity can arise, we will refer to the set of transactions and update projections executed at a given container, collectively as transactions.

4.4. Correctness Criteria for Replicated Architecture

We begin by outlining the serializability theory for replicated data. Our discussion is adapted from [4]

In what follows we will use the notation q_i to denote an operation of a transaction T_i, i.e., q_i is either r_i or w_i

To execute a transaction on replicated data, we must first translate every operation on a data element into an operation on one (in the case of a Read) or several (in the case of a Write) operations on copies of that element. Let h be a *translation* function, i.e., let $h(r_i[x]) = \{r_i[x_j]\}$ and $h(w_i[x]) = \{w_i[x_{j_1}], ., w_i[x_{j_n}]\}$, where $x_j, x_{j_1}, ., x_{j_n}$ are copies of x Thus, the translation of a given operation on a

data element is a set of operations on copies (a singleton in the case of a Read) To simplify notations, let $h(T_i)$ denote the union of translations of all the operations in T_i, i e, $h(T_i) = \bigcup_{o_i[x] \in T_i} h(o_i[x])$ Also, let $h(T) = \bigcup_{T_i \in T} h(T_i)$

Definition 4. Two operations $o_i[x_k]$ and $o_j[x_k]$ *conflict* with each other if they are on the same copy of a data element (x_k) and at least one of them is a Write □

Definition 5. A *replicated data history* H over a set of transactions T is a partial order with ordering relation $<_H$ such that

(1) $H = h(T)$,

(2) for any $T_i \in T$ and all x, y such that $r_i[x], w_i[y] \in T_i$, if $r_i[x] <_i w_i[y]$ and $r_i[x_j], w_i[y_k] \in H$, then $r_i[x_j] <_H w_i[y_k]$, and

(3) all pairs of conflicting operations in H are ordered by $<_H$ □

Informally, a (replicated data) history is a partially ordered set of operations on copies that is consistent with the order of operations within transactions (condition (2)) In addition, since a history is intended to model a particular execution of a set of transactions on replicated data, it should order all conflicting operations (condition (3))

Definition 6. Given a replicated data history H over a set of transactions T and transactions $T_i, T_j \in T$, we say that T_i *reads-x-from* T_j if $w_j[x_m] \in H$, $r_i[x_m] \in H$, $w_j[x_m] <_H r_i[x_m]$, and there exists no k such that $w_j[x_m] <_H w_k[x_m] <_H r_i[x_m]$. □

Definition 7. Two replicated data histories over transactions $T_0, T_1, \ldots, T_m$ are said to be *equivalent* if they have the same read-from's, i. e., if T_j reads-x-from T_i in one history, then this relationship holds in the other history as well. □

Definition 8. A *serial* history H is a totally ordered replicated data history such that for every pair of transactions T_i and T_j in H, either all of T_i's operations precede all of T_j's or vice versa. □

Definition 9. A serial history H is a *one-copy* serial history if for all i, j, and x in H, if T_j reads-x-from T_i, then T_i is the last transaction before T_j to write into any copy of x. □

Definition 10. We say that a replicated data history is *one-copy serializable* if it is equivalent to a one-copy serial history □

Thus, in our model, an execution of transactions is *correct* if it is equivalent to a serial execution of the same transactions on a single copy database. Therefore, one-copy serializable histories hide all aspects of data replication from user transactions and give transactions one-copy view of the database

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

Definition 11. A *serialization graph* for history H is a directed graph $G(H)$ whose nodes are transactions in T A (directed) edge $T_i \rightarrow T_j$ is in $G(H)$ iff for some x in D and integer k, $o_i[x] \in T_i$, $o_j[x] \in T_j$, $o_i[x_k] \in H$, $o_j[x_k] \in H$, $o_i[x_k] <_H o_j[x_k]$, and $o_i[x_k]$ conflicts with $o_j[x_k]$ □

Informally, an edge $T_i \rightarrow T_j$ is included in $G(H)$ if T_i and T_j have conflicting operations such that the operation of T_i precedes that of T_j in history H

Let v and w be nodes of a a directed graph G If there is a path from v to w we denote this fact by $v \rightarrow\rightarrow w$.

Definition 12. A *replicated data serialization graph* (RDSG) for history H is a directed graph whose nodes are transactions in T and whose (directed) edges constitute a superset of edges of $G(H)$ such that for all $x \in D$ the following conditions hold:

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

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

Note that according to Definition 12, an RDSG (unlike its non-replicated counterpart) is not necessarily unique for a given history H [4]

Theorem 1 [4]. Let H be a replicated data history. If H has an acyclic RDSG, then H is one-copy serializable. □

To specify the proposed protocol in Section 4, we need one final definition.

Definition 13. Let V_i denote a transaction or update projection executed at container C_i The *serialization order* on V_i's is a binary relation O_i defined as follows. $(V_i, V_k) \in O_i$ iff $o_i[x] \in V_i$ conflicts with $o_k[x] \in V_k$ for some x and $o_i[x_j] <_H o_k[x_j]$, i.e., V_i and V_k have conflicting operations such that the operation of V_i is scheduled at C_k ahead of that of V_k □

Obviously, for all correct local schedulers, the serialization order must be a partial order

5. Two Simple Approaches That Do Not Work

In this section, we give two simple algorithms and show that they do not work, albeit for different reasons. The first algorithm always produces one-copy serializable histories; however, it is not secure since it uses a commit algorithm for update propagation, which introduces a covert channel. Although secure, the second algorithm, which has been used in [17,23], does not always produce one-copy serializable executions, hence is incorrect.

5.1. A Solution Using Two-Phase Commit

One way to achieve one-copy serializability is by making the transaction executions atomic across security classes using a protocol similar to the distributed two-phase commit protocol [11] as follows. For each transaction T_i, the TCB becomes the transaction *coordinator* and all containers C_j with $L(C_j) \geq L(T_i)$ act as the *participants* Let P denote the set $\{C_j : L(C_j) > L(T_i)\}$ When the TCB receives a transaction T_i with $L(T_i) = S_i$, it sends T_i to the container C_i such that $L(C_i) = S_i$ If C_i can successfully

execute T_i, it sends to the TCB the update projection U_i; however, C_i does not commit the parent transaction T_i at this time. Upon receiving the update projection U_i from C_i, the TCB initiates the first phase of the protocol by sending U_i to all participants in P. Upon receipt of U_i, a participant C_j executes U_i like a local transaction, but does not commit it. If C_j can successfully execute U_i, C_j sends the update report $R_{ij} = (U_i, C_j)$ to the TCB. If the TCB receives an update report from all the participants, it instructs all participants in P together with container C_i to commit the transaction, otherwise, it instructs everyone to abort the transaction.

Although it insures one-copy serializability, this solution has a serious flaw in that it introduces a covert channel as follows [22]. A high-level participant by not sending an update report can cause abort messages to be sent to all other participants. A high-level participant is thus capable of sending information to lower-level nodes by selectively causing transaction aborts.

5.2. A "Primary Copy" Solution

Another possible strategy might be to identify for each security class S_i, the container C_i as the *primary* container and to designate all other containers C_j with $L(C_j) > S_i$ as *secondary* containers. A transaction T_i is directed to the primary container at the security class of T_i. If T_i is executed (and committed) successfully, the primary container forwards any updates (the update projection) to the TCB, which in turn propagates these updates to all secondary containers.

This strategy does not preserve one-copy serializability as the following example shows.

Example. Suppose that the database system consists of three security classes S_1 = (Confidential, {Conventional}), S_2 = (Secret, {Conventional}), and S_3 = (Secret, {Nuclear, Conventional}). Thus $S_1 \leq S_2 \leq S_3$. Furthermore, suppose that the database consists of two elements x and y with $L(x) = S_1$ and $L(y) = S_2$. Now, consider the following transactions.

$$T_1 = w_1[x],\ L(T_1) = S_1$$

$$T_2 = r_2[x]\, w_2[y],\ L(T_2) = S_2$$

$$T_3 = r_3[x]\, r_3[y],\ L(T_3) = S_3$$

Suppose now that the transaction T_1 is executed first at C_1. Then at C_2, the update projection U_1 is executed followed by T_2, while at C_3, U_2 is executed first followed by T_3 and then U_1 (so the update dispatches by the TCB occur in the order U_1 to C_2, U_2 to C_3, and U_1 to C_3). Thus, the replicated data history is given as follows.

At C_1 $\quad w_1[x_1]$

At C_2 $\quad w_1[x_2]\, r_2[x_2]\, w_2[y_2]$

At C_3 $\quad w_2[y_3]\, r_3[x_3]\, r_3[y_3]\, w_1[x_3]$

Since the serialization graph given in Figure 4 has a cycle, the replicated data serialization graph has a cycle as well, and so the history given above is not one-copy serializable.

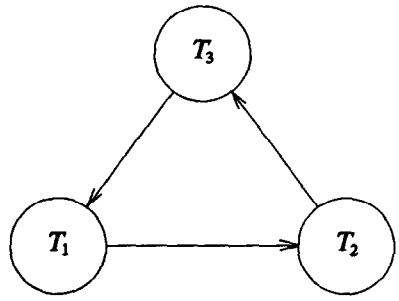

Figure 4

6. Protocol for Transaction Processing

In a nutshell, our transaction processing protocol works as follows. When a new transaction is submitted to TCB, the latter sends it for execution to the container of the same security class as that of the transaction. Upon executing (and committing) the transaction, the container returns to TCB a list of the resulting updates (the update projection), which TCB subsequently distributes to all other containers that are allowed to see them, i.e., the containers of higher security classes. An update report is sent by a container to the TCB after the corresponding transaction commits. The update report contains, in addition to the update projection, the name of the container from which the report originates. Thus the mutual consistency of copies of the same data is preserved. To guarantee that correctness of concurrent transaction execution on replicated data is also preserved, the propagation must occur precisely in a specified manner detailed later in this section in Protocol A.

There are two important things which must be kept in mind. First, an update projection whose parent transaction was executed at container C_i can be executed at container C_j much like a regular transaction but with one difference. Namely, the effect of a transaction execution on the database depends on the transaction itself (i.e., on its operations and their order) and on the state in which it finds the database (i.e., the values it reads). The effect of an update projection execution, on the other hand, does not depend on the current state of the database, but rather on the state of the database as seen by the corresponding parent transaction. Thus an update projection can only be defined (and executed) after its parent transaction was executed.

Second, our solution does not utilize a commit protocol when executing a transaction and its update projections, i.e., in our protocol the parent transaction commits without waiting for its projections to commit. In fact, projections are dispatched to TCB for propagation to other containers only *after* the parent transaction commits. Com-

mitting transactions independently at a single container is possible because there should be no reason to permanently abort an update projection once its parent transaction has successfully committed. That is so because a permanent abort (as opposed to an abort out of concurrency control considerations, say in timestamp ordering protocol, when the projection may be resubmitted at a later time) is usually caused by either a run-time error or violation of integrity constraints. However, since executing an update projection involves no computations, a run-time error cannot occur. Furthermore, since the parent transaction has committed, it must have been considered safe from the integrity point of view, and since the update projection only mimics its parent transaction's update activity by writing the same values into copies of corresponding data elements, the projection cannot violate any integrity constraints either

The transaction processing protocol is specified below.

Protocol A.

At all $C_i \in C$

(A1) Each transaction T_k and each update projection U_j received from TCB are submitted to the local scheduler Actions on every element x specified by the transaction (or update projection) are translated into the corresponding actions on copy x_i.

(A2) For all U_j and U_k such that U_j is received at C_i before U_k and has an operation conflicting with one of the U_k's operations (which can occur only if $L(T_j) = L(T_k)$), U_j is serialized by the local scheduler before U_k

(A3) For each committed transaction T_j, an update report R_{ji} is sent to TCB Similarly, for each committed update projection U_k (executed at C_i), an update report $R_{ki} = (U_k, C_i)$ is sent to TCB The update reports are dispatched in the order consistent with the serialization order of locally executed transactions and update projections. That is, if V_j is serialized before V_k, then R_{ji} is sent to TCB before R_{ki}. If T_j and T_k are not ordered by the scheduler with respect to each other (note that this can occur only if they do not have any conflicting operations), then T_j and T_k can be sent to TCB in an arbitrary order

At TCB

(A4) Each transaction T_j submitted to TCB is sent for execution to container C_i such that $L(T_j) = L(C_i)$

(A5) A queue Q of update reports is maintained by TCB Each newly received report R_{ij} is appended to the end of Q. Whenever TCB is ready to send an update projection to a container, the first update report

$R_{kl} = (U_k, C_i)$

is removed from Q and U_k is sent to C_m such that $L(C_m)$ immediately follows $L(C_i)$ in a fixed topological order imposed on the security class lattice. That is, C_m is such that $L(C_m)$ is the lowest numbered (in the topological order) class that strictly dominates $L(C_i)$. If such a container cannot be found, then R_{kl} is discarded.

Protocol A describes the functionality of TCB as well as the interface between TCB and the individual containers. Note that the implementation of part A3 of the protocol is not obvious. Therefore, we should briefly discuss the issue of implementation. Since most of the commonly used schedulers are based either on two-phase locking [4] or timestamp ordering [4], we limit our discussion to these two basic techniques.

Schedulers can easily produce an explicit serialization order needed in part A3 For instance, schedulers based on timestamp ordering order transactions according to their (unique) initiation times, while two-phase locking schedulers order transactions by the time of the first lock release. However, the complication arises when the order of transaction commit times is not the same as their serialization order That leads to the question of how to decide when to send a given update report to TCB Because of the possible discrepancy between the commit-time order and the serialization order, we cannot very well dispatch an update report as soon as its parent transaction commits (for there may be a still active transaction that will be put before the committed one in the final serialization order) A simple method to resolve this difficulty would be, in the case of a timestamp ordering, to postpone sending the update report for a committed transaction while there are still active transactions at that container which have lower timestamps. Similarly, in the case of a scheduler based on two-phase locking, the update report is being withheld until there are no more active transactions with earlier first lock-release times than the one that just committed.

Part A5 is crucial to the protocol, for it specifies the order in which update projections are directed to the containers that are supposed to see them. Note that TCB does not ship any given update projection to all the relevant containers as soon as it receives the first update report that contains the projection. Instead, it selects the next container to receive the projection based on the structure of the security class lattice. Upon executing the projection, that container submits to TCB a report consisting of the same projection and the container's identity (see part A3). Only after that (and after the report advances through Q) does TCB determine the next destination for the projection in question, and so on.

Thus, there is constant passing back and forth of the update projection between TCB and containers until all the containers that are supposed to get this projection have a chance to execute it. Notice, however, that the information always flows from lower to higher databases, never from higher to lower ones.

It is possible to have optimizations of the protocol, which will reduce message traffic between TCB and containers and also, in many cases, make the update projections available to containers more promptly The details are discussed in [18]

We note that the propagation discipline used in part A5 is similar to the techniques used in *Balanun* data networks [19], where the objective is to balance concurrency-control requirements against availability requirements for distributed databases with data replication in the face of communication failures. One notable difference is that in our architecture for security there is a centralized module – TCB – that controls the propagation.

6.1. Recovery Under Our Protocol

In this subsection, we show what TCB must do in the event of failures. For recovery purposes, the only thing that the TCB needs to do is to ensure that the queue Q consisting of all update reports can be reconstructed after a failure This can be accomplished very simply as follows. Every time the TCB receives an update report from a container, a log record containing this fact is written onto the stable storage In the event of failures, the TCB can use these records to reconstruct the required queue Q.

From time to time, the TCB can take a *checkpoint* for the purpose of limiting the number of log records which must be scanned during a restart. Since the TCB no longer needs to keep track of an update projection originated at a container C_i once this projection has been seen by all containers C_j such that $L(C_j)$ strictly dominates $L(C_i)$, the checkpoint record consists of the addresses on the stable storage of all update projections which have not been seen by all required containers. In this way, the TCB can reduce the size of log records which must be searched during a recovery.

Since the TCB has very limited responsibility, the code to accomplish this will be small and hence, easy to verify correct

6.2. Correctness of the Protocol.

In this subsection, we argue briefly that our protocol is correct since it always yields one-copy serializable histories. In order to do this, we will need the following two lemmas. We omit their proofs in this paper; the details are available from the authors [16]

Lemma 1. Let H be a replicated data history over a transaction set T produced by protocol A. Then $G(H)$ is an RDSG for H □

Lemma 2. Let H be a replicated data history on a transaction set T produced by protocol A. Let $C_q \in C$ be such that $L(C_i) < L(C_q)$ for all $C_i \in C(r \neq q)$, and $L(T_i) < L(C_q)$ for all T_i (if such C_q does not exist, we can always conceptually add it to C for the purposes of the proof) Suppose that there are no transactions of class $L(C_q)$ in T. Let $\ll$ be a binary relation defined on T as follows. $T_i \ll T_j$ iff U_i is received at C_q before U_j. Then $T_i \ll T_j$ for all T_i and T_j such that $T_i \rightarrow T_j$ is an edge in $G(H)$ □

Using these lemmas, we can prove that our algorithm is correct.

Theorem 2. Any history H produced by protocol A is one-copy serializable

Proof. To prove the theorem, we demonstrate that $RG(H)$ is acyclic (see Theorem 1) Assume to the contrary and let $T_1 \rightarrow\rightarrow T_k \rightarrow T_1$ be a cycle in $RG(H)$ It is easy to see that the relation $\ll$ is a total order In particular, it is transitive. Therefore, by Lemma 2, $T_1 \ll T_1$ But $\ll$ is also irreflexive. Hence, we arrive at a contradiction and have to conclude that $RG(H)$ is acyclic. □

7. Conclusion

In recent years, there have been many efforts to build secure relational database management systems that are capable of protecting data with a variety of classification levels from users with different clearances. Since there were many security related obstacles which had to be overcome, the issue of performance in these systems has been relegated to a secondary role. However, much progress has been made in recent years, and we expect that such systems will start appearing in the market place in a few years. The users are sure to demand acceptable performance of these systems, making the issue of performance most relevant The performance issue is especially important in secure DBMS's since many of them will be used in military applications having real-time performance requirements.

It is our view that replicated DBMS's offer a potential for substantial performance improvement over the kernelized systems. However, to make this architecture work, it is crucial to have a secure synchronization mechanism which not only preserves database consistency, but does not have to be trusted at the same time. Our transaction management algorithm meets both these goals. We have shown that it yields one-copy serializable histories, hiding all aspects of data replication from transactions, and therefore, the replicated DBMS behaves like a single copy database. Our protocol is secure since information always flows only in one direction–from lower to higher databases. Finally, we have shown that our scheme can be implemented in such a way that the TCB can rely for the most part on the individual databases for the concurrency and recovery Since the TCB has very limited responsibilities, the code needed to accomplish them will be very small and would not be difficult to verify correct

Acknowledgement

We are indebted to Joe Giordano for his support and encouragement which made this work possible We also wish to thank Judy Froscher and Catherine Meadows for their helpful comments.

References

1. "Multilevel Data Management Security," Committee on Multilevel Data Management Security, Air Force Studies Board, National Research Council,

Washington, DC, 1983

2. "Department of Defense Trusted Computer System Evaluation Criteria," Department of Defense, National Computer Security Center, December 1985

3 D E. Bell and L. J. LaPadula, "Secure computer systems. Unified exposition and multics interpretation," The Mitre Corp., March 1976

4 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, 1987

5 C. J Date, *An Introduction to Database Systems, Volume II*, Addison-Wesley, Reading, 1983

6 Dorothy E. Denning, *Cryptography and Data Security*, Addison-Wesley, Reading, Mass., 1982.

7 Dorothy E. Denning, Teresa F Lunt , Roger R. Schell, William R. Shockley, and Mark Heckman, "The SeaView security model," *Proc. Symp. on Security and Privacy*, pp. 218-233, April 1988.

8 Judith N Froscher and Catherine Meadows, "Achieving a trusted database management system using parallelism," in *Database Security, II. Status and Prospects*, ed. Carl E. Landwehr, pp 151-160, North-Holland, Amsterdam, 1989

9 Cristi Garvey, Thomas Hinke, Nancy Jensen, Jane Solomon, and Amy Wu, "A layered TCB implementation versus the Hinke-Schaefer approach," *Proc. 3rd IFIP WG 11 3 Workshop on Database Security*, September 1989

10 Richard Graubart, "A comparison of three secure DBMS architectures," *Proc. 3rd IFIP WG 11 3 Workshop on Database Security*, September 1989

11. Jim Gray, "Notes on data base operating systems," in *Lecture Notes in Computer Science, Volume 60*, pp 394-481, Springer-Verlag, New York, 1978.

12. Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The recovery manager of the System R database manager," *ACM Computing Surveys*, vol. 13, no 2, pp 223-242, June 1981.

13 M. J. Grohn, "A model of a protected data management system," I P Sharp Assoc. Ltd., June 1976.

14 Thomas H. Hinke and Marvin Schaefer, "Secure Database Management System," RADC-TR-75-266, Final Technical Report, System Development Corporation, November 1975

15 Thomas H. Hinke, "DBMS Technology vs. Threats," in *Database Security Status and Prospects*, ed. Carl E. Landwehr, pp 57-87, North-Holland, Amsterdam, 1988.

16 Sushil Jajodia and Boris Kogan, "Transaction processing in multilevel-secure databases using replicated architecture," Technical Report, Center of Excellence for Command, Control, Communications, and Intelligence, George Mason University, October 1989.

17. Catherine D Jensen, Robert M. Kiel, and Richard D. Verjinski, "SDDM. A prototype of a distributed architecture for database security," *Proc. 5th IEEE Int'l. Conf on Data Engineering*, pp 356-364, 1987.

18. Boris Kogan and Hector Garcia-Molina.

19. Boris Kogan and Hector Garcia-Molina, "Update Propagation in Bakunin Data Networks," *Proc. 6th ACM Symp on Principles of Distributed Computing*, pp 13-26, August 1987.

20. B. W Lampson, "A note on the confinement problem," *Comm. of ACM*, vol. 16, no 10, pp 613-615, October 1973.

21. Carl E. Landwehr, "Formal models for computer security," *ACM Computing Surveys*, vol. 13, no 3, pp 247-278, September 1981.

22. Glen H. MacEwen, "Effects of distributed system technology on database security A survey," in *Database Security Status and Prospects*, ed. Carl E. Landwehr, pp 253-261, North-Holland, Amsterdam, 1988.

23. James P. O'Connor and James W Gray, III, "A distributed architecture for multilevel database security," *Proc. 11th National Computer Security Conference*, pp. 179-187, October 1988.

24 David P. Reed and Rajendra K. Kanodia, "Synchronization with eventcounts and sequencers," *Comm. of the ACM*, vol. 22, no 2, pp. 115-123, February 1979.