An Efficient Multiversion Algorithm For Secure Servicing Of Transaction Reads

Paul Ammann * Sushil Jajodia †

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

Abstract

We propose an efficient multiversion algorithm for servicing read requests in secure multilevel databases. Rather than keep an arbitrary number of versions of a datum, as standard multiversion algorithms do, the algorithm presented here maintains only a small fixed number of versions – up to three – for a modified datum. Each version corresponds to the state of the datum at the end of an externally defined version period. The algorithm avoids both covert channels and starvation of high transactions, and applies to security structures that are arbitrary partial orders. The algorithm also offers long-read transactions at any security level conflict-free access to a consistent, though slightly dated, view of any authorized portion of the database. We derive constraints sufficient to guarantee one-copy serializability of executions histories, and then exhibit an algorithm that satisfies these constraints.

1 Introduction

We address the problem of concurrency control in secure multilevel databases. In a secure multilevel database, a request to read a datum is satisfied if the *simple security* property is met, and a request to write a datum is satisfied if the **-property* is met [BL76, Den82, DoD85]. These rules may be summarized as stating that processes cannot read from higher security classes or write to lower security classes. In addition to enforcing the simple security and *-property on direct access, secure multilevel databases must also prevent indirect access or covert channels. For this reason concurrency control algorithms for conventional databases are inadequate. Accordingly, a variety of algorithms [AJJ92, Cos92, CK93, CM92, FM89, Jae92, JK90, KK92, KT90, KJ90, McD93, MJS91] and architectures [Com83, KJ90] specifically tailored for secure multilevel databases have been developed. In order to satisfy the security constraints, each of the current solutions sacrifices one or more of the following desirable properties: (i) efficient storage, (ii) minimization of the trusted computing base (or TCB), (iii) non-starvation of high transactions, or (iv) applicability to arbitrary security structures. We propose a new algorithm – a periodic multiversion algorithm, show that it achieves the security objectives, and argue that it retains a better set of desirable properties than comparable algorithms.

For the *kernelized* architecture [Com83, KJ90], both single version [AJ93] and multiversion [KT90] algorithms have been developed. Unfortunately, to avoid covert channels, all single version algorithms subject high transactions to starvation. To avoid covert channels, the multiversion algorithm in [KT90] requires that all transactions be serialized in a global queue that is managed by the TCB. In addition, there is no bound on the number of versions of a datum that might be required.

At each security class, the standard *replicated* architecture [Com83, KJ90], to which most multilevel algorithms pertain, requires the complete replication of the databases at all dominated security classes. For large

*This work was supported in part by the National Science Foundation under grant CCR-9202270.

†This work was supported in part by the National Science Foundation under grant IRI-9303416.

CCS '94- 11/94 Fairfax Va., USA
© 1994 ACM 0-89791-732-4/94/0011..$3.50

databases or for security lattices with more than a very few nodes, this can pose a prohibitive cost. In addition, propagation of updates from low classes to high classes is a technically challenging problem, and many of the algorithms are only correct for a subset of the desirable security structures. In particular, some algorithms restrict security structures to be total orders, and others restrict security structures to be *crown-free* [1] [AJF93]. Those algorithms for the standard replicated architecture that allow security structures to be lattices or arbitrary partial orders *must* manage at least some of the concurrency control within the TCB.

An alternate replication architecture, in which snapshots of low databases are maintained for read access by high transactions, is described in [AJJ92, Jae92]. Concurrency control for this architecture is achieved with a two-snapshot algorithm. Instead of accessing low data directly, high transactions are instead given access to stable snapshots of low data, and snapshots are regenerated on a periodic basis. Although the two-snapshot algorithm applies to general partial orders, it still requires on the order of three times the storage of a single version database, even if most low data values do not change from one snapshot to the next. In addition, the two-snapshot algorithm imposes an upper bound on the time during which high transactions can execute; any transaction that overruns the resulting deadlines must be aborted.

Finally, none of the existing concurrency control algorithms for secure, multilevel databases offer support for long read-only transactions. It is well known that a mixture of long read-only transactions and update transactions causes performance problems for conventional database concurrency control algorithms [PMC+92]. For example, with conventional two-phase locking, a long read-only transaction can block many update transactions as it acquires the necessary read-locks on data. The problem is simply exacerbated in concurrency control algorithms for secure multilevel databases.

In this paper, we propose a secure, periodic, multiversion concurrency control algorithm suitable for a kernelized architecture in which:

- Storage is used efficiently. Specifically, only versions of a modified datum are maintained, and of these, at most three, including the current version, are ever simultaneously maintained.
- No part of the concurrency control algorithm is managed inside the TCB. (A TCB is still required; however, serializability is achieved without recourse to the TCB).
- The algorithm applies to arbitrary security structures, *i.e.* any partial order.
- High transactions are not subject to starvation.
- Long read-only transactions can execute without interfering with update transactions.

To achieve these goals, the algorithm imposes the following constraints. First, the algorithm sets a limit on transaction duration; a transaction that overruns its deadline is aborted. Second high transactions read slightly out-of-date versions of low data; however, most other concurrency control algorithms for multilevel databases share this trait. Third, long read-only transactions that require conflict-free access also read slightly out-of-date versions of the data. However, read-only transactions that do not require conflict-free access can read current versions of the data. We argue that these constraints are justified by the benefits enumerated above.

1.1 Other Related Work

Aside from the work on secure multilevel databases cited above, there are two additional areas of relevant work.

First, the handling of long read-only transactions has received considerable attention in conventional databases. One algorithm for supporting consistent reads of entire databases that employs two-phase locking is the on-the-fly algorithm of Pu [Pu86].[2]

Multiversion algorithms are also suitable for servicing long reads, and they are better suited to long read-only transactions that only access a subset of the database. In a multiversion algorithm, each write of a datum produces a new version, and long reads can be satisfied by accessing old versions. In standard multiversion algorithms [BHG87], there is no bound on the number of versions of a datum that need be maintained, and for some algorithms, *e.g.* multiversion two-phase locking, there are still conflicts between read locks and certify locks. For extremely long read transactions or extremely large data, the lack of a bound on the number of versions can be a significant implementation problem. Mohan *et al.* proposed an multiversion algorithm that limits the maximum number of versions that need be maintained to a small fixed value [MPL92]. Since the current paper builds on the work of Mohan *et al.*, we describe the algorithm in more detail next.

[1] Unfortunately, lattices, which are natural security structures for many applications [Den82], are not a subset of crown-free partial orders.

[2] In Pu's algorithm, the global read transaction locks and releases each datum incrementally and colors a datum black if the global read has accessed the datum and white otherwise. Each update transaction is also colored based on the color of data the transaction attempts to write. A color test aborts gray transactions, which are transactions that attempt to write both black and white data. It turns out that the color test in [Pu86] is not quite strong enough. However, the algorithm can be repaired; a correct color and shade test is given in [Mav93].

Transactions in [MPL92] are classified into two types, which we will call *update* transactions (called type Tu transactions in [MPL92]) and *long-read* transactions (called type Tr transactions in [MPL92]). Update transactions access the current versions of a datum via a two-phase locking protocol. Long-read transactions do *not* acquire locks; instead they access old versions of a datum. Thus, they enjoy conflict-free access to data in exchange for tolerating a slightly out-of-date, although consistent, view of the database. Long-read transactions are prohibited from updating any values. It is important to note that if a read-only transaction wishes to read the current version, it may run as an update transaction.

Rather than creating a version upon each write of a datum, the scheme in [MPL92] only maintains versions corresponding to the end of externally defined *version periods*. Even if many transactions write a datum x during a given version period n, only the last value written is recorded in the version n instance of x. If no transaction writes x during version period $n+1$, no version n instance of x need be separately maintained since the values of version n and version $n+1$ are identical.

Any number of versions can be maintained, but to make the scheme workable, two, plus the current version, is the minimum number. A zero version scheme corresponds to a database with no multiversioning. Maintaining exactly one extra version is not satisfactory in that active long-read transactions must be aborted at the switch of each version period. Maintaining two extra versions is satisfactory [MPL92]; additional versions allow long-read transactions additional time to complete.

The other area of relevant work is that on hierarchically decomposed databases [HC86]. Such databases share the same structure and (non)interference properties as secure multilevel databases, although the motivation differs. In an hierarchical database, data is partitioned into classes. Each transaction executes at a particular class, and the access characteristics of a transaction are restricted by analogs of the simple security and *-properties. One useful interpretation of the partitions is that data at one class may be considered as 'raw' with respect to dominating classes and 'derived' with respect to dominated classes. The motivation for the access constraints on transactions is that transactions manipulating derived data should not modify raw data, nor should they interfere with transactions that are maintaining the raw data. The algorithm developed in this paper for secure multilevel databases can be applied directly to hierarchically decomposed databases.

1.2 Outline Of Paper

The remainder of the paper is organized as follows. In section 2, we give definitions and an architecture to support a secure periodic multiversion algorithm. In section 3 we develop the constraints on periodic multiversion algorithms that are sufficient to simultaneously satisfy noninterference between classes and serializability of execution histories. In section 4, we present an algorithm that satisfies the constraints derived in section 3, allows transactions timely access to data at dominated classes, and permits transactions to smoothly execute during version switches. Section 5 summarizes our results.

2 A Secure Multiversion Architecture

First, we present definitions suitable for defining a secure multilevel architecture. Next, we describe the general architecture for maintaining periodically generated versions at different classes.

2.1 Definitions

We consider a finite set of transactions, denoted $\mathbf{T} = \{T_1, T_2, ...\}$. Each transaction accesses (reads or writes) data in a finite set $\mathbf{D} = \{x_1, x_2, ...\}$. Each transaction and datum is associated with one of a finite number of classes in the set $\mathbf{S} = \{S_1, S_2, ...\}$. The function $L : \mathbf{T} \cup \mathbf{D} \rightarrow \mathbf{S}$ denotes this mapping; in other words, if $L(T_a) = S_i$, then S_i is the class of transaction T_a. Similarly, if $L(x) = S_i$, then S_i is the class of datum x. As a notational convention, we use the subscripts i, j and k to index classes in $\mathbf{S}$, and the subscripts a, b and c to index transactions in $\mathbf{T}$.

Classes are related by a dominance relation, denoted $> : \mathbf{S} \leftrightarrow \mathbf{S}$. If $S_i > S_j$, then S_i is said to dominate S_j. We write $S_i \geq S_j$ to include the case that i might equal j, and we write $S_j < S_i$ as another way of expressing $S_i > S_j$.

It is useful to know how far 'up' in the dominance relation a given class S_i is. To this end, define the grade function, $g : \mathbf{S} \rightarrow N$, to be a function from the set of classes to the natural numbers. Consider a directed graph whose nodes are $\mathbf{S}$ and whose edges are the transitive reduction of the dominance relation, $>$. For $S_i \in \mathbf{S}$, let $g(S_i)$ equal the length of the longest path starting at S_i.

2.2 Servicing Reads With Old Versions

In the versioning scheme proposed by Mohan *et al* [MPL92], correctness follows directly from the algorithm description. Serializability of execution histories with respect to update transactions is ensured by a two-phase locking protocol. Serializability of execution histories with respect to long-read transactions is also ensured since long-read transactions always view the data-

base state as it appears at the end of some version period, and the end of each version period is guaranteed to be consistent.

Developing a versioning scheme for a secure multilevel database or a hierarchically decomposed database requires a more careful approach to correctness. The main reason is that serializability of update transactions cannot be ensured simply with two-phase locking or some other conventional concurrency control mechanism. Attempting to ensure serializability with a conventional mechanism results in a covert channel or starvation in the case of a secure multilevel database and interference or starvation in the case of a hierarchically decomposed database.

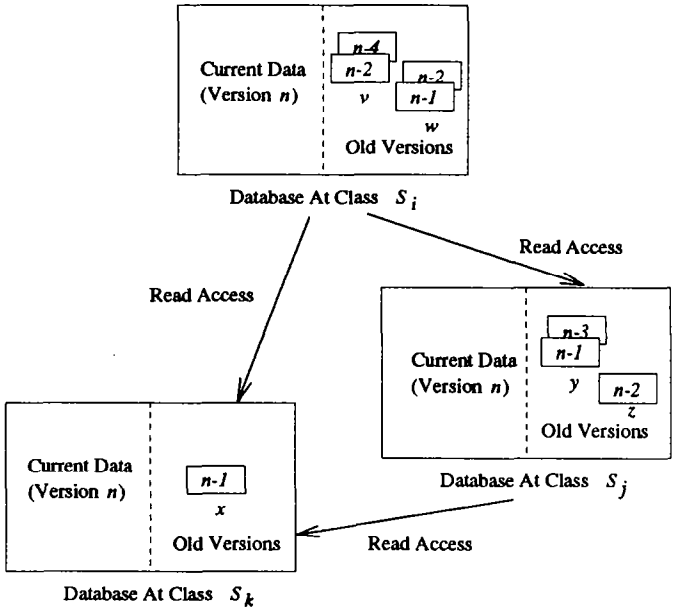

Figure 1: Servicing Reads With Old Versions

For example, suppose a transaction at a dominating class obtains a read lock for the current version of a datum. A transaction at the dominated class can detect the presence or absence of the read lock by attempting to obtain a write lock, thus creating a covert channel. On the other hand, if the read lock is simply overridden by the write lock, the dominating transaction loses it guarantee that the value obtained is consistent, and must retry. Repeated unsuccessful retries can lead to starvation of the dominating transaction. Even the usual multiversion approach of creating a new version does not work in our case since we are explicitly restricted to creating at most one new version of a datum per version period. Similar examples can be developed for other concurrency control strategies, such as timestamp ordering.

The interference/starvation problem exhibited in the above example can be eliminated by having update transactions access stable versions, *i.e.* data from an an earlier version period, rather than current versions when reading data at a dominated class.

Figure 1 shows the general architecture by which read requests may be serviced with old versions. For the purposes of the paper, we assume that transaction access follows the *restricted *-property* rather than just the *-property. The restricted *-property says that if transaction T_a attempts to write datum x, then $L(T_a) = L(x)$. The restriction prevents transactions from writing up. Hence, it is only the servicing of transaction reads that need cross class boundaries. The general motivation for the restricted *-property over the *-property is integrity; rogue transactions from dominated classes cannot damage data at dominating classes. Use of the restricted *-property is a standard, although not universal, assumption in algorithms for secure multilevel databases.

Figure 2 gives the requirements – current or stable – for different types of transaction access. The first row gives the access requirements of update transactions. The second row gives the access requirements of long-read transactions.

Transaction Type \ Location of Data Item	Own Class	Dominated Class
update	Current	Stable
long-read	Stable	Stable

Figure 2: Data Requirements For Different Transaction Types

3 Serializability Constraints

Although avoidance of interference is most easily expressed in terms of stable vs. current versions of a datum, serializability arguments require more precise definitions of versions. Specifically we need two ways to relate transactions to versions of each datum. Consider a transaction T_a at some class S_i. Let the *version* function, denoted $v : \mathbf{T} \rightarrow N$, map T_a to version n if n is the first version in which T_a's effects can become visible. It is possible that T_a's updates do not appear in any version if some other transaction T_b of the same version as T_a subsequently overwrite the values T_a produces; hence the conditional in the definition. Also, we require that all updates of T_a first appear in the same

version; otherwise T_a is not atomic. Let the *assign* function, denoted $a : \mathbf{T} \rightarrow N$, map T_a to version n if T_a reads some datum x at a dominated class S_j, $S_i > S_j$, and the version of x read is n. Note that by insisting that $a(\cdot)$ be a function, we ensure that for all dominated data that T_a accesses, the version accessed be the same. It is straight forward to exhibit unserializable execution histories if T_a is allowed to access multiple versions of data at dominated classes [AJJ92, Jae92].

The definition of $v(\cdot)$ is essentially the same as the rule in [MPL92] for assigning update transactions to a version period. The definition of $v(\cdot)$ presented here is more general, in that it allows for concurrency control algorithms other than two-phase locking, such as timestamp ordering. The definition of $a(\cdot)$ is essentially the same as the rule in [MPL92] for assigning long-read transactions to a version period. In our case, it is helpful to have both definitions for each of the possible transaction types.

The definitions of $v(\cdot)$ is incomplete respect to read-only 'update' transactions and long-read transactions. If T_a is a read-only transaction that is executing as an update transaction, the value of $v(T_a)$ need only be consistent with neighboring, class S_i update transactions in some serialization order. Common scheduling algorithms can produce an explicit serialization order. For example, time of first lock release is suitable for determining $v(T_a)$ in a two-phase locking scheduler.

If T_a at class S_i is a long-read transaction (recall this means that T_a wishes a guarantee not to be blocked) then $v(T_a)$ is one plus the version of the data (if any) that T_a reads at class S_i. If T_a reads no data at class S_i, then $v(T_a)$ is one plus the version of the data (if any) that T_a reads at some dominated class. T_a must read the same version from all dominated classes, but T_a may possibly read a different version at its own class.

Note that $a(T_a)$ is not defined if the grade of S_i, $g(S_i)$ is zero. Fortunately, no definition is required in this case. Also, if $g(S_i)$ is not zero, and T_a does not read any data from a dominated class, $a(T_a)$ is again undefined. In this case a definition is necessary. As was done for $v(\cdot)$, let $a(T_a)$ be defined consistently with neighboring, class S_i transactions in some serialization order.

For any given transaction, the version function and the assignment function are related. Specifically, we restrict any versioning algorithm to satisfy $a(T_a) = v(T_a) - 1 \vee a(T_a) = v(T_a)$. In other words, a transaction always reads dominated data of its own version, or of the immediately preceding version. The reason for this restriction is as follows. First, it is clearly inconsistent to have the effects of transaction T_a appear in a version prior to some version from which T_a reads. Hence, we require $a(T_a) \leq v(T_a)$. Second, suppose that a transaction T_a were allowed to have $a(T_a) = n - 1$ and $v(T_a) = n + 1$ for some n. Consider the following scenario:

Let S_i, S_j and S_k be three classes such that $S_i > S_j > S_k$, and let, T_a, T_b, and T_c be three transactions at these classes, respectively. Suppose the three transactions obtain the following versions. The notation $o_a[x_n]$ indicates that operation o from transaction T_a accesses version n of datum x.

$$\begin{array}{ll} T_a: & r_a[x_n]\ r_a[y_n] \\ T_b: & r_b[x_{n-1}]\ w_b[y_{n+1}] \\ T_c: & w_c[x_n] \end{array}$$

Note that $a(T_b) = n - 1$ since $L(T_b) = S_j > L(x) = S_k$ and T_b reads the value of x from version $n - 1$. Also, $v(T_b) = n + 1$ since T_b writes the value of y in version $n + 1$. In terms of a serialization graph, $T_a \rightarrow T_b$ since T_a reads an earlier version of y than T_b writes, $T_b \rightarrow T_c$ since T_b reads an earlier version of x than T_c writes, and $T_c \rightarrow T_a$ since T_c writes the version of x that T_a reads. Hence, the result is the cycle $T_a \rightarrow T_b \rightarrow T_c \rightarrow T_a$.

To avoid such cycles in general, we prohibit transactions such as T_b above. by requiring, for all transactions T_a, $a(T_a) \geq v(T_a) - 1$. Together, the two constraints yield $a(T_a) = v(T_a) - 1 \ \vee \ a(T_a) = v(T_a)$.

3.1 A Serialization Sketch

The basis for the serialization argument is that, in any multiversion execution history H, any T_a where $v(T_a) = n$ may be serialized after any T_b where $v(T_b) = n-1$ and before before any T_c where $v(T_c) = n+1$. Figure 3 shows this basic serialization argument. Figure 3 essentially captures the correctness argument behind the scheme in [MPL92].

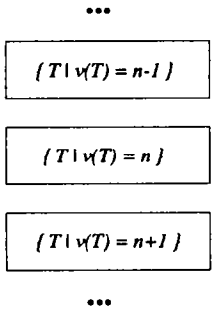

Figure 3: Serialization Of Transactions By Version Function

The scenario is more complicated in the case of hierarchically decomposed or multilevel secure databases. Here, a transaction T_a where $v(T_a) = n$ may read down to dominated classes. If version n is available for the

dominated classes, then $a(T_a)$ may be set to n. However, if version n is not available, it is efficient to assign $a(T_a)$ to $n-1$ instead. Thus the transactions T_a for which $v(T_a) = n$ are broken into two parts - those with $a(T_a) = n-1$ and those with $a(T_a) = n$. The concurrency control algorithm at each class S_i must ensure that local transactions (*i.e.* transactions T_a for which $L(T_a) = S_i$) for which $a(T_a) = n-1$ serialize before local transactions for which $a(T_a) = n$. The revised serialization argument is shown in figure 4.

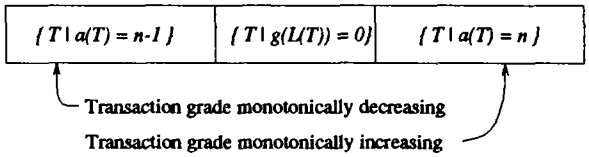

Figure 4: Serialization Of Version n Transactions

Figure 4 also shows the effect of the grade function, $g(\cdot)$, on the serialization of version n transactions. Specifically, version n transactions at classes of grade zero must serialize after version n transactions at dominating classes that read version $n-1$ data, *i.e.* those T_a for which $a(T_a) = n-1$. Extending this argument to other classes shows that all version n transactions must serialize after transactions at dominating levels that read version $n-1$. Ordering version n transactions for which $a(\cdot)$ has value $n-1$ monotonically *decreasing* by the grade function satisfies the objective. In addition, version n transactions that, in essence, read from version n transactions at dominated levels must serialize after such transactions. In this case ordering version n transactions for which $a(\cdot)$ has value n monotonically *increasing* by the grade function satisfies the objective. Remaining serialization decisions are up to the local scheduler at each class, and can be made in a conventional manner, *e.g.* via two-phase locking or timestamp ordering.

We omit a complete treatment of serializability, such as one be based on multiversion serializability [BHG87]. The interested reader is referred to the serializability argument for the similar case of the two-snapshot algorithm that appears in [Jae92].

4 A Secure Periodic Multiversion Algorithm

From the point of view of correctness, any versioning algorithm is satisfactory as long as it satisfies the constraints on the $a(\cdot)$ and $v(\cdot)$ functions. However, satisfying correctness constraints is not enough. For example, suppose one adopted the straightforward versioning algorithm of switching to new versions synchronously at all classes. This decision means that all classes always have the same version, say n, as the current version. The most recent stable version is $n-1$ at all classes. Thus for all T_a, $v(T_a) = n$ and $a(T_a) = n-1$. Suppose a transaction T_a reads a version of data item x, namely x_{n-1}, from some dominated class, and suppose the version period switches immediately after the read. The version switch ends the version period n, and T_a must now abort, since any values T_a may wish to write are too late to appear in version n.

Aborting T_a in the above example satisfies the correctness constraints, but it is hardly efficient. What is required is some time for a transaction that has read a value from a dominated level to complete its processing. A solution is to manage version switching in a progressive manner, beginning first at grade 0 classes, and proceeding up from grade i to grade $i+1$ classes. In between each switch, transactions accessing old versions are given time to complete. During this time, new transactions can access newer version of dominated data. The algorithm below is based on a similar approach is used in the two-snapshot algorithm [AJJ92, Jae92]:

The following pseudo-code uses a loop indexed by security grade. Let G denote the largest grade found in the multilevel database, and let n denote the current version period.

```
n := 1;
While Normal Operation
  For i in 0..(G − 1)
    Declare version n at each level S
      where G(S) = i
      --Subsequent transactions at class S
      --have v(·) ≥ n + 1
    Set a(·) = n for new grade i + 1 transactions
      --For old grade i + 1 transactions a(·)
      --remains n − 1
    Delay one period
      --To allow grade i + 1 transactions with
      --a(·) = n − 1 to finish
    Abort unfinished grade i + 1 transactions
      with a(·) = n − 1
  End-For
  Garbage collect data with version n − 1
  n := n+1
End-While
```

The steps within the For-loop are carried out in parallel at multiple classes. Thus all classes of a given grade declare new versions at the same time.

The algorithm given here does not forcibly abort transactions at grade zero, although in practice it may be desirable to do so. The algorithm does forcibly abort old transactions at higher grades, if they fail to finish within the time limit, where 'old' encompasses

any transaction that began execution prior to the availability of the new versions created in the first step of the For-loop, *i.e.* when $a(\cdot) = n - 1$. Transactions that begin after that point are considered new and are not forcibly terminated until the next pass through the While-loop. This means that every transaction has at least one period in which to execute, and some have considerably longer. A complete pass through the While-loop requires G periods, so the maximum time available to a transaction is $G + 1$ periods.

4.1 Satisfaction Of Desirable Properties

The above discussion has dealt primarily with correctness of the algorithm from the viewpoint of one-copy serializability. For completeness, we must also discuss the other properties that we have claimed of the algorithm.

First, we address correctness from the viewpoint of security. For direct access, we assume that the TCB implements the simple-security and restricted *-property on each access request. The real issue is covert channels - is it possible for a transaction at one class to influence any transaction at either a dominated or incomparable class. The only access across classes is a read by a transaction at a dominating level to a datum at a dominated level. Since no read-lock is necessary on the stable version that is used to satisfy the read request, there is no possibility for a transaction at the dominated level to infer that the read request is pending. Hence, there is no covert channel between transactions in the algorithm presented here.

Remaining claims are easily justified as follows. It is clear that only up to three versions of a datum are simultaneously maintained, namely the current version, and up to two stable prior versions. None of the concurrency control need be in the TCB, although the TCB is responsible for implementing the simple-security, and restricted *-properties. The correctness sketch in section 3 applies to general partial orders; *e.g.* a restriction to crown-free partial orders or some more other subset of partial orders is not needed. High transactions are not subject to starvation since at least one stable version is always available for read access at any given class. Further, a transaction at a dominating class always has at least one period to complete execution. Finally, long read-only transactions can execute at any level by accessing stable versions. As in [MPL92], long read-only transactions are subject to the deadline imposed by the versioning algorithm.

5 Conclusion

We have presented a periodic multiversion concurrency control algorithm suitable for secure multilevel databases. We argue that the algorithm yields a desirable tradeoff of properties. In return for setting a limit on transaction duration and having some transactions view a slightly dated version of the database, the algorithm efficiently uses a bounded amount of storage, keeps all concurrency control functions outside of the TCB, applies to arbitrary security structures, does not subject high transactions to starvation, and supports long read-only transactions with conflict-free access.

We developed a set of constraints that ensured one-copy serializability, and then presented an algorithm that satisfies these constraints. In addition, we show that the algorithm guarantees transactions a fixed time to complete, allows transactions to span version switching periods, and showed that security constraints are satisfied.

References

[AJ93] Paul Ammann and Sushil Jajodia. Distributed timestamp generation in planar lattice networks. *ACM Transactions on Computer Systems*, 11(3):205–225, August 1993.

[AJF93] Paul Ammann, Sushil Jajodia, and Phyllis Frankl. Globally consistent event ordering in one-directional distributed environments. Technical Report ISSE-TR-93-104, George Mason University, Fairfax, VA 22030, August 1993.

[AJJ92] Paul Ammann, Frank Jaeckle, and Sushil Jajodia. A two snapshot algorithm for concurrency control in secure multi-level databases. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 204–215, Oakland, CA, May 1992.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BL76] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, The Mitre Corporation, Bedford, MA, March 1976.

[CK93] Oliver Costich and M. Kang. Maintaining multilevel transaction atomicity in MLS database systems with replicated architecture. In *IFIP7: Proceedings of the IFIP WG 11.3 Seventh Annual Working Conference on Database Security*, pages 332–357, Huntsville, AL, September 1993.

[CM92] Oliver Costich and John P. McDermott. A multilevel transaction problem for multilevel

secure database systems and its solution for the replicated architecture. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 192–203, Oakland, CA, May 1992.

[Com83] Committee on Multilevel Data Management Security, Air Force Studies Board, National Research Council, Washington, DC. *Multilevel Data Management Security*, 1983.

[Cos92] Oliver Costich. Transaction processing using an untrusted scheduler in a multilevel database with replicated architecture. In Carl Landwehr and Sushil. Jajodia, editors, *Database Security V: Status and Prospects*, pages 173–190. North Holland, 1992.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982.

[DoD85] DoD Computer Security Center. *Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[FM89] Judy Froscher and Cathy Meadows. Achieving a trusted database management system using parallelism. In C. Landwehr and S. Jajodia, editors, *Database Security II: Status and Prospects*, pages 151–160. North Holland, 1989.

[HC86] Meichun Hsu and Arvola Chan. Partitioned two-phase locking. *ACM Transactions on Database Systems*, 11(4):431–446, December 1986.

[Jae92] Frank Jaeckle. A two snapshot algorithm for concurrency control in secure multi-level databases. Master's thesis, George Mason University, 1992.

[JK90] Sushil Jajodia and Boris Kogan. Transaction processing in multilevel-secure databases using replicated architecture. In *Proceedings of the Symposium on Research in Security and Privacy*, Oakland, CA, May 1990.

[KJ90] Boris Kogan and Sushil Jajodia. Concurrency control in multilevel-secure databases using replicated architecture. *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 153–162, May 1990.

[KK92] I.E. Kang and T.F. Keefe. On transaction processing for multilevel secure replicated databases. In *Proceedings European Symposium on Research in Computer Security*, pages 329–347, Toulouse, France, 1992. Springer-Verlag. Lecture Notes in Computer Science, Volume 648.

[KT90] T.F. Keefe and W.T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 369–383, Oakland, CA, May 1990.

[Mav93] Padmaja Mavuluri. On the fly reading of entire databases. Master's thesis, George Mason University, 1993.

[McD93] John P. McDermott. *Transaction Management in Replicated-Architecture Multilevel-Secure Database Systems.* PhD thesis, George Mason University, 1993.

[MJS91] John McDermott, Sushil Jajodia, and Ravi Sandhu. A single-level scheduler for the replicated architecture for multilevel-secure databases. In *Seventh Annual Computer Security Application Conference*, pages 2–11, San Antonio, TX, December 1991.

[MPL92] C. Mohan, Hamid Pirahesh, and Raymond Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 124–133, San Diego, CA, June 1992.

[PMC+92] Hamid Pirahesh, C. Mohan, Josephine Cheng, T. S. Liu, and Pat Sellinger. Parallelism in relational data base systems: Architectural issues and design approaches. In *Proceedings of 2nd International Conference on Databases in Parallel and Distributed Systems*, pages 4–29, Dublin, Ireland, July 1992.

[Pu86] Calton Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, October 1986.