15

Providing different degrees of recency options to transactions in multilevel secure databases

V. Atluri[a], *E. Bertino*[b] *and S. Jajodia*[c]
[a]*MS/CIS Department, Rutgers University, Newark, NJ 07102, U.S.A.*

[b]*Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy*

[c]*Center for Secure Information Systems and Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030-4444, U.S.A.*

Abstract

Although several secure multiversion concurrency protocols have been proposed by researchers, only two protocols produce histories that are one-copy serializable — one proposed by Keefe and Tsai and the other by Jajodia and Atluri. However, neither of these are completely satisfactory. Keefe and Tsai protocol sacrifices recency for correctness by providing a high transaction with very old versions of low data. Jajodia and Atluri protocol, on the other hand, sacrifices performance for correctness by making high transactions that read low data wait before they can commit. The first contribution of this paper is to provide different recency options to transactions, where each transaction can receive data with a desired degree of recency. These options are provided without sacrificing one-copy serializability. In fact, we propose four types of degrees of recency and present protocols for each type. The traditional timestamp-based protocols require that transactions be assigned unique timestamps, which is necessary to ensure the correctness of the protocols. The second contribution of this paper is to recognize that this requirement need not be met in a multilevel secure environment to guarantee correctness. The protocols to determine the timestamps for providing different types of recency options are based on this property.

Keywords

Security and Protection; Information Systems, General; Systems; Multilevel Security; Recency, Concurrency Control; Serializability, One-copy serializability

1. INTRODUCTION

Although locking is one of the most popular techniques for providing concurrency control in conventional (untrusted) database management systems (DBMSs), it has been found to be unsuitable in multilevel secure (MLS) DBMSs. To overcome its shortcomings, researchers have proposed several secure protocols (for example, [MG90, KT90, JA92]) that are timestamp-based and use multiple versions. In these protocols, timestamps are used to give high transactions older versions of low data, thus eliminating both the signaling channel and starvation problems.

Among all such protocols, two protocols stand out since only they produce histories that are one-copy serializable. The first protocol has been proposed by Keefe and Tsai [KT90] and the other by Jajodia and Atluri [JA92]. The difference between the two protocols is the way the scheduler assigns timestamps to the arriving transactions. In the rest of this paper, the schedulers proposed by Keefe and Tsai and Jajodia and Atluri are referred as scheduler K and scheduler J, respectively.

Scheduler K places a high transaction behind *all* active transactions executing at lower levels by assigning the high transaction a timestamp that is smaller than those given to all active transactions at lower levels. As a result, high transactions receive much older versions of low data, older than those given by the traditional multiversion timestamp ordering protocol (MVTO) [BHG87]). (Example 2 in section 4 provides further details on how schedulers K and J assign timestamps to transactions.) By contrast, scheduler J assigns transactions timestamps in the same order as they arrive; as a result high transactions are given the same version of the low data as MVTO. However, if a high transaction reads from a lower level, Scheduler J makes high transaction wait to commit until all transactions that are at that lower level and have arrived earlier than this transactions finish their execution.

Thus, these two solutions can be viewed at being at the two extremes; one gives the transactions very old versions of data but does not require them to wait, and the other gives the transactions most recent versions of data but makes them wait. In other words, one solution sacrifices recency for correctness and the other performance for correctness.

The first contribution of this paper is to propose a solution to this dilemma by providing different options to transactions with respect to the recency of data. According to our approach, each transaction can choose a desired degree of recency for the low data, thereby reducing the time of wait. Thus, a transaction does not have to wait for the completion of *all* earlier active transactions, nor does it have to settle for very old versions of the data. Thus, our approach provides more flexibility to transactions since they have the liberty to choose an appropriate version without opting for one of the two extremes.

Our notion of providing varying degree of recency has some similarities to epsilon serializability proposed by Pu et al. [$P^{+}93$]. However, their approach does not deal with multilevel databases and, moreover, it sacrifices consistency for recency and allows only read-only transactions to enjoy this flexibility. In addition to dealing with multilevel security, the distinguishing feature of our approach is that it provides different degree of recency options to transactions *without compromising one-copy serializability.* Additionally, our approach extends this option for both read-only as well as update transactions.

The idea behind our approach is to manipulate the timestamps that are assigned to transactions as follows. We first model all active transactions as points on a recency spectrum by assigning a recency value to each one of them. The timestamp of a new transaction is selected in such way that this new transaction is placed at an appropriate position on this recency spectrum based on the specified degree of recency. We propose four different types of degrees of recency and present protocols that provide these types.

To ensure correctness of the protocols, the traditional timestamp-based protocols (for example, MVTO [BHG87]) require that each transaction be assigned a unique timestamp. Another important contribution of this paper is to identify a property (see Section 3) which states that the preceding requirement need not be met in a multilevel secure environment to guarantee correctness. The protocols to determine the timestamps for providing different types of recency options are based on this property.

The remainder of this paper is organized as follows. In section 2, we present our security model. Section 3 first makes two important observations related to timestamp based protocols in multilevel secure systems and formalizes them as a property. Using this property, we give an implementation for Scheduler K that uses single-level schedulers. In section 4, we first present the notion of degree of recency and then propose four different types of recency in which a transaction can specify its desired degree of recency. In section 5, we first propose protocols to determine the timestamp of a transaction based on the desired type and degree of recency. Then we give a multiversion timestamp based protocol. All the protocols proposed in this paper are single-level and therefore are secure. Finally, section 6 presents the conclusions.

2. THE SECURITY MODEL

In this section, we give a brief description of our security model. We refer the reader to [Den82] for additional details related to multilevel security and to [BHG87] for details relevant to multiversion serializability.

The secure system S consists of a set D of *data items* (objects), a set T of *transactions* (subjects) and a partially ordered set S of access classes (or security levels) with ordering relation $\leq$. "A class s_i is said to be *dominated* by another class s_j if $s_i \leq s_j$. A class s_i is said to be *strictly dominated* by another class s_j (denoted as $s_i < s_j$) if $s_i \leq s_j$ and $i \neq j$". 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 $T_j \in T$, $L(T_j) \in S$. In other words, every data item as well as every transaction has a security class associated with it.

The following two conditions are *necessary* for a system to be secure:

1. A transaction T_j is allowed to read a data element x only if $L(x) \leq L(T_j)$
2. A transaction T_j is allowed to write a data element x only if $L(x) = L(T_j)$.

In addition to these two restrictions, a *secure* system must guard against illegal information flows through signaling and covert channels. Notice that unlike [KT90], we restrict transactions to write only at their levels. We believe that it is prudent to disallow transactions that write to higher levels for integrity and security reasons [JK90].

Definition 1 Given two security classes s_i and s_j such that $s_j < s_i$, s_j is said to be the *child* of s_i iff there exists no s_k such that $s_j < s_k < s_i$.

3. AN IMPLEMENTATION FOR SCHEDULER K

A good concurrency control protocol has the following four properties: (1) it is one-copy serializable (serializable if single version is kept), (2) it is free of covert channels, (3) it is starvation-free, and (4) it can be implemented with single level schedulers (i.e., they do not require any trusted code.)

A difficulty with Scheduler K as described in [KT90] is that it does not specify how timestamps are to be assigned to the high transactions that read low data, which is crucial for the implementation of the protocol. To correct this, Maimone and Greenberg [MG90] have proposed an implementation for scheduler K using single level schedulers. Their implementation uses a three component timestamp $(t.l.s)$, the first component (t) represents the values of the system clock, the second component (l) the security level of the transaction, and the third component (s) the sequence number. Every transaction is assigned a timestamp as follows: If there are no active transactions at lower levels, then the value of t is assigned by reading from the clock at system low. Otherwise, the value of t is computed by scanning all the lower security levels. Two transactions may have the same value of t. $t.l_1.s$ is considered smaller than $t.l_2.r$ if $l_2 < l_1$, irrespective of the third component. Similarly, $t.l.s$ is smaller than $t.l.r$ if $s < r$.

A limitation of the Maimone and Greenberg's implementation is that it requires the security levels to be hierarchically ordered. In addition, it requires a critical section, meaning that certain operations involved in determining the timestamp must be executed as an atomic unit. This creates the potential for a covert channel. We suggest that this protocol is complicated because besides placing transactions in a specific order in accordance with their security level, it requires that every transaction be assigned a unique timestamp.

In this section, we propose another implementation for scheduler K that requires single level schedulers as in [MG90]; however, we use single component timestamps, and our implementation does not require a critical section. Since our transaction model does not allow write-ups, to distinguish it from [KT90] and [MG90], we call our implementation secure MVTO (SMVTO). Our solution is based on the following two important observations. Suppose the multiversion scheduler wishes to employ a timestamp based protocol in a secure system S.

Observation 1 Transactions at different security levels need not be assigned unique timestamps.

Observation 2 Transactions at the same security level must be assigned unique timestamps.

The first observation says that, two transactions may have the same timestamp if they belong to two different security levels. The following two reasons support our observation. First, suppose there exist two transactions T_i and T_j such that $L(T_i) \neq L(T_j)$. These two transactions do not conflict if they both read data from another lower level since two read operations never conflict. Second, T_i and T_j may conflict with each other if they contain conflicting operations, in which case the conflicting operations need to be ordered. A conflict may occur as follows: Suppose $L(T_j) < L(T_i)$. In this case, T_i and T_j conflict if T_i reads a data item x and T_j updates the same data item. The usual timestamp based protocols serialize these two transactions in the order of their timestamps since they assign unique timestamps to transactions. We will show that one can still order these two transactions even though their timestamps are equal. However, we can no longer use the conventional MVTO, but need to modify it. The modified MVTO (which we denote by SMVTO) is presented in algorithm 3.2 later in this section.* A similar argument can be made in the case where $L(T_i) < L(T_j)$. It is easy to preserve the property in the second observation. We can always assign progressively increasing timestamps by serializing arriving transactions.

The above two observations lead to the following property, which is *necessary* for every timestamp based secure concurrency control protocol to guarantee correctness.

Property 1 In a system S, for every pair of transactions T_i and T_j such that $L(T_i) = L(T_j)$, $ts(T_i) \neq ts(T_j)$.[†]

In the following, we present a protocol for generating timestamps. However, unlike conventional timestamp based protocols, we do not determine the timestamp by reading from a clock, but we compute it by scanning the timestamps of transactions at lower levels. Another distinguishing feature of this protocol is that it may assign the same timestamp to two different transactions, provided they belong to different security levels.

Algorithm 3.1 [An Algorithm to Determine the Timestamps]

1. Each transaction T_i is assigned a time, time(T_i) as soon as it arrives by reading from a global clock at system low.
2. A timestamp $ts(T_i)$ is computed for every transaction T_i as soon as it arrives. (The exact details as how it is computed is explained in the following steps.) Only after assigning a timestamp, it is allowed to execute any operations.
3. Each scheduler at level s maintains two values: min-ts$_s$ and max-ts$_s$, where min-ts$_s$ = min $\{ts(T_i) : L(T_i) = s\}$ and max-ts$_s$ = max $\{ts(T_i) : L(T_i) = s\}$
4. We assume there is an initial transaction T_0, which initializes min-ts$_s$ and max-ts$_s$ as follows: min-ts$_s$ = max-ts$_s$ = $ts(T_0)$.
5. The scheduler updates min-ts$_s$ whenever a transaction completes its execution at level s, and modifies max-ts$_s$ whenever a new timestamp at level s is computed.
6. The timestamp, $ts(T_i)$ of each transaction T_i at level s is determined as follows: $ts(T_i) <$ min{min-ts of all the children of level s, time(T_i)} and $>$ max-ts$_s$. If s has no children, i.e., s itself is the lowest security level, then $ts(T_i)$ = time(T_i).

Since this algorithm determines the timestamp of a transaction only by examining the children of its security level, it may sometimes give excessively old versions of data to a transaction if all the children of the transaction's level have no active transactions. We suggest two solutions to alleviate this problem. First, a *dummy* transaction can be sent to each scheduler at a certain interval. The exact frequency of these dummy transactions can be determined based on the specific application. Second, all security levels can be examined while determining the timestamp, and if there are no active transactions at all the lower security levels, timestamp can be assigned by reading from the clock at system low.

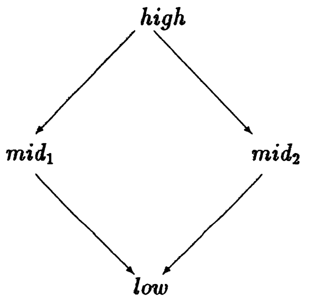
*Note that we cannot allow two transactions to have the same timestamp if we allow write-ups since two such transactions at different levels may create two new versions with the same version number when both of them update a data item.

[†]It also means for every pair of transactions T_i and T_j such that $L(T_i) \neq L(T_j)$, $ts(T_i)$ may equal $ts(T_j)$.

Example 1 We explain with an example how the above protocol assigns timestamps to transactions. Consider the security lattice as shown in figure 1 with four elements. Suppose the first transaction to arrive is T_1 at level *low*. Since this level does not have any children, timestamp of T_1 is determined by reading from a real-time clock at system low. In other words, $ts(T_1) = time(T_1)$. Suppose $time(T_1) = 8$. Since T_1 is the only transaction at *low*, min-ts$_{low} = 8$. While T_1 is still active, assume two transactions T_2 and T_3 arrive at levels mid_1 and mid_2, respectively. They receive timestamps that are less than 8. Let $ts(T_2) = ts(T_3) = 7$. All later transactions at these two levels may receive timestamps between 7 and 8 (excluding these two values), for example, 7.1, 7.2, and so on. It is important to note that the dot "." in the timestamp is the decimal point but does not represent the separator used to separate the two components of timestamp as in [AJ92] or [MG90]. Accordingly, min-ts$_{mid_1}$ = min-ts$_{mid_2} = 7$.

Suppose, at this point of time, another new transaction T_4 arrives at *high*. It receives a timestamp which is less than both min-ts$_{mid_1}$ and min-ts$_{mid_2}$. Let $ts(T_4) = 6$.

A new transaction T_5 at level *low* arriving after the completion of T_1 receives its timestamp from the clock. Suppose $ts(T_5) = 9$. At this point, another new transaction, T_6 arriving at either mid_1 or mid_2 may be assigned 8, which is same as the timestamp of the *low* transaction T_1. According to property 1, even though T_1 and T_6 are assigned the same timestamp, it does not cause any harm to the correctness. □



$$\begin{tikzcd} & high \arrow[dl] \arrow[dr] & \\ mid_1 \arrow[dr] & & mid_2 \arrow[dl] \\ & low & \end{tikzcd}$$

Figure 1. A Security Lattice

The following theorem shows the necessity of choosing > in step 6 of algorithm 3.1.

Theorem 1 Algorithm 3.1 satisfies property 1.

Proof: Consider two transactions T_i and T_j such that $L(T_i) = L(T_j) = s$. Without loss of generality, assume T_i has arrived earlier than T_j. From step 3 of algorithm 3.1, min-ts$_s \leq ts(T_i)$ and max-ts$_s \geq ts(T_i)$. From step 6, $ts(T_j) >$ max-ts$_s \geq ts(T_i)$. Thus, $ts(T_j) > ts(T_i)$. In case where T_j arrives earlier than T_i, $ts(T_i) > ts(T_j)$. But $ts(T_j)$ is never equal to $ts(T_i)$. □

In step 6 of algorithm 3.1, < has been chosen instead of ≤ because, by using ≤, it may not be possible to assign distinct timestamps to two transactions at the same security level.

Algorithm 3.2 [Secure MVTO (SMVTO)]

1. There is a separate scheduler at each security level.
2. Every transaction T_i is assigned a timestamp $ts(T_i)$ using algorithm 3.1.

3. The database maintains multiple versions of data. Each version x_i of a data item x has a read timestamp $rts(x_i)$ and a write timestamp $wts(x_i)$ associated with it. We assume there is an initial transaction T_0, whose timestamp $ts(T_0) = 0$, that writes into the database, and for each version x_0 of an item x, $rts(x_0) = wts(x_0) = ts(T_0)$.
4. If a transaction T_i wishes to read a data item x such that $L(x) < L(T_i)$, then it selects a latest version x_k such that $wts(x_k) < ts(T_i)$.
5. If a transaction T_i wants to read a data item x such that $L(T_i) = L(x)$, then the scheduler selects a version x_k with the largest $wts(x_k)$ such that $wts(x_k) \leq ts(T_i)$, processes $r_i[x_k]$, and modifies $rts(x_k)$ as $rts(x_k) = \max\{ts(T_i), rts(x_k)\}$.
6. When a transaction T_i wants to write a data item x, scheduler selects a version x_k with the largest $wts(x_k)$ such that $wts(x_k) < ts(T_i)$. It rejects $w_i[x_i]$ if $rts(x_k) > ts(T_i)$; otherwise, it processes $w_i[x_i]$ and modifies the timestamps of the new version x_i as $rts(x_i) = wts(x_i) = ts(T_i)$.

The only difference between MVTO and SMVTO lies in step 4. Notice that according to SMVTO, a transaction is allowed to read a version x_j of x only if $wts(x_j) < ts(T_i)$. (In the original MVTO, this condition is $wts(x_j) \leq ts(T_i)$.) This slight modification allows us to place T_i before T_j in the serialization order even though their timestamps happen to be equal.

Theorem 2 Suppose the scheduler in a system S uses SMVTO and the timestamps assigned to all transactions in a history H satisfy property 1. Then H is one-copy serializable. □

We prove this theorem using the following three lemmas. We assume the version order as follows: $x_i \ll x_j$ iff $wts(x_i) < wts(x_j)$.

Lemma 1 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) < L(T_j)$, then $ts(T_i) < ts(T_j)$.

Proof: If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) < L(T_j)$, there must exist a data item x such that $L(x) = L(T_j)$ and $r_j[x_i] \in H$. From step 4 of the above algorithm, it follows that $wts(x_i) < ts(T_j)$. From step 6 of this algorithm, $wts(x_i) = ts(T_i)$. These two expressions can be combined and thus, $ts(T_i) < ts(T_j)$. □

Lemma 2 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) = L(T_j)$, then $ts(T_i) < ts(T_j)$.

Proof: Let $L(T_i) = s$. Since T_i and T_j are transactions of the same level, there must exist a data item x such that at least one of the following three cases is true.

Case 1. $r_j[x_i] \in H$

Case 2. $r_i[x_n], w_j[x_j] \in H$ with $x_n \ll_s x_j$

Case 3. $r_k[x_j], w_i[x_i] \in H$ with $x_i \ll_s x_j$

Consider case 1. Since x_i is the version created by T_i, according to step 5 of algorithm 3.2, $wts(x_i) \leq ts(T_j)$. From step 6, $wts(x_i) = ts(T_i)$. From property 1, $ts(T_i) \neq ts(T_j)$. Combining these three expressions, we get $ts(T_i) < ts(T_j)$.

Consider case 2. The read operation $r_i[x_n]$ and the write operation $w_j[x_j]$ can occur in any order, i.e. either the read operation precedes the write operation, or vice versa.

1. Consider the case where $r_i[x_n]$ precedes $w_j[x_j]$.

   According to step 5 of algorithm 3.2, $rts(x_n) = ts(T_i)$. From step 6, $wts(x_j) = ts(T_j)$. Again from step 6, $rts(x_n) \leq ts(T_j)$. From property 1, $ts(T_i) \neq ts(T_j)$. Combining the above, we get $ts(T_i) < ts(T_j)$.

2. Consider the other case where $r_i[x_n]$ succeeds $w_j[x_j]$. Then, $wts(x_n) < wts(x_j)$, $wts(x_n) \leq ts(T_i)$. Thus, $wts(x_j) \not\leq ts(T_i)$. Otherwise, T_i would have selected the version x_j of x instead of x_n. In other words, $wts(x_j) > ts(T_i)$. Therefore, $ts(T_i) < ts(T_j)$.

Now, consider case 3. Since $x_i \ll x_j$, $wts(x_i) < wts(x_j)$. From step 6, $wts(x_i) = ts(T_i)$, and $wts(x_j) = ts(T_j)$. Since $L(T_i) = L(T_j)$, from step 2, $ts(T_i) \neq ts(T_j)$. Therefore, $ts(T_i) < ts(T_j)$. □

Lemma 3 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_j) < L(T_i)$, then $ts(T_i) \leq ts(T_j)$.

Proof: Let $L(T_j) = s$. If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_j) < L(T_i)$, this edge implies that there must exist a data item x such that $L(x) = s$ and $r_i[x_n], w_j[x_j] \in H$ with $x_n \ll_s x_j$ since $x_n \ll_s x_j$, $wts(x_n) < wts(x_j)$. From step 4, $wts(x_n) < ts(T_i)$.

Now we must consider the following two cases:

1. $r_i[x_n]$ precedes $w_j[x_j]$. In this case, since T_j is still an active transaction, from algorithm 3.1, $ts(T_i) \leq$ min-ts$_s$ and min-ts$_s \leq ts(T_j)$. Therefore, $ts(T_i) \leq ts(T_j)$.
2. $w_j[x_j]$ precedes $r_i[x_n]$. Now we need to consider the following three cases.
   (a) $ts(T_i) < ts(T_j)$. In this case, the proof trivially follows.
   (b) $ts(T_i) = ts(T_j)$ In this case, the proof trivially follows.
   (c) $ts(T_i) > ts(T_j)$. Since $wts(x_j) = ts(T_j) < ts(T_i)$, according to step 4 of the algorithm, T_i selects the version x_j of x instead of x_n. Therefore, the edge $T_i \rightarrow T_j$ no longer exists.

□

Proof of Theorem 2: Suppose there exists a cycle in $MVSG(H)$. We must consider the following two cases of cycles in $MVSG(H)$.

Suppose all the transactions in the cycle belong to the same security level. We denote this cycle by a single level cycle. From lemma 2, for every edge $T_i \rightarrow T_j$ such that $L(T_i) = L(T_j)$, $ts(T_i) < ts(T_j)$. In other words, all edges in this cycle follow the timestamp order, and therefore, this cycle cannot exist.

Suppose that in the cycle at least one pair of transactions belong to different security levels. We denote such a cycle by a multiple level cycle. Let this cycle be $T_1 \rightarrow T_2 \rightarrow T_3 \ldots T_n \rightarrow T_1$. For every edge $T_l \rightarrow T_{l+1}$, from lemmas 1, 2 and 3, we get $ts(T_l) \leq ts(T_{l+1})$. However, every multiple level cycle must contain an edge $T_k \rightarrow T_{k+1}$ such that $L(T_k) < L(T_{k+1})$. For such an edge $ts(T_k) < ts(T_{k+1})$. Therefore, if we combine all the results, we arrive at a relation $ts(T_1) \leq ts(T_2) \ldots ts(T_k) < ts(T_{k+1}) \ldots ts(T_n) \leq ts(T_1)$, which is impossible. Thus there cannot be a multiple level cycle in $MVSG(H)$. □

4. DEGREE OF RECENCY

In this section, we begin with an overview of our approach with the help of examples and then formally present our notion of degree of recency. As described in the introduction, neither scheduler K nor scheduler J provide a completely satisfactory solution to the problem of secure concurrency control. We explain these two extreme solutions with the following example.

Example 2 Suppose a transaction T_i wishes to read a data item x from a lower level s. Assume that there are 100 transactions currently executing at level s. These transactions arranged in the increasing order of their timestamps are shown in figure 2, denoted by $T_1, T_2, \ldots, T_{100}$. Scheduler K places T_i ahead of all these 100 transactions by assigning a timestamp to T_i that is smaller than the timestamps of all the 100 transactions. That is, T_i is placed at position "a" in figure 2. If there are any active transactions at levels between $L(T_i)$ and s whose timestamps are smaller than T_1, scheduler K may place T_i much ahead than shown here even though T_i does not read data from those levels.

On the other hand, scheduler J serializes T_i after all the 100 transactions by assigning a timestamp to T_i that is larger than the timestamps of all the 100 transactions. That is, T_i is positioned at "c" in figure 2. In this case, T_i has to wait for its commit for the completion of all 100 transactions.

Notice that the version of data given to a transaction T_i by scheduler K is affected by transactions at all lower levels although T_i does not read from all lower levels. On the other hand, the version given by scheduler J depends only on the transactions at the level from which T_i reads data. In case of scheduler J, it is possible that these lower level transactions may themselves been waiting for the completion of transactions from yet another lower level. □

In summary, a transaction need not wait if it is willing to accept a very old version of the data. If it wishes to obtain the most recent version of the data, then it has to wait; the duration of the wait is dependent on the duration of active transactions at s. Neither of these extreme solutions are desirable even though they both preserve one-copy serializability.

In this paper, we propose a solution where a transaction can choose an appropriate version of a data item from among a number of versions. If it does not choose the most recent version of a data item, it need not wait for the completion of all the active transactions at lower levels for its commit. According to our approach, a transaction chooses its desired version of data item by specifying it in terms of *degree of recency*. Our notion of degree of recency is explained below.

Example 3 Consider once again example 2. Suppose T_i wishes to read 60% recent data of x. Then T_i has to specify its degree of recency as 0.6. We serialize T_i just after the first 60 active transactions but before the 61st transaction. In other words, although T_i arrives after T_{100}, it is executed as if it has arrived after T_{60}. Our approach is simple and is based on manipulating the timestamps so as to meet the specified degree of recency needs. We assign a timestamp to T_i whose value is larger than the timestamp of the first 60 transactions but still smaller than the 61st transaction, i.e., at position "b" in figure 2. As a result, T_i need to wait for the completion of only 60 transactions instead of 100. The distinguishing feature of our approach is that we provide these different degrees of recency options to transactions *without compromising one-copy serializability.* It is important to note that T_i is guaranteed to be given at least 60% recent data but will never be given less recent data. However, T_i may be given more than 60% recency since all the 100 transactions may not modify the same data item that T_i has read. □

Strictly speaking, in the above example, while computing the timestamp of a transaction T_i according to the specified degree of recency, it is more appropriate to consider only those active transactions at s that update x. However, such a specification requires transactions to predeclare their write-sets since at the time of computation of the timestamp of T_i all

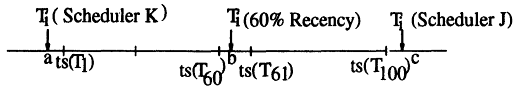


Figure 2. The Recency Sequence

active transactions at s can potentially update x. To avoid this requirement, we assume that all active transactions at s will update x.

In the remainder of this section, we formalize the notion of degree of recency and propose four different types of degree of recency. Below, we use $\mathcal{T}$ denote the set of active transactions.

Definition 2 Given a set of transactions $\mathcal{T}$, a *recency sequence* $P(\mathcal{T})$ is a linearly ordered sequence of transactions in $\mathcal{T}$ such that for every T_i preceding T_j in the sequence, $ts(T_i) \leq ts(T_j)$.

Definition 3 Given a recency sequence $P(\mathcal{T})$, the *sequence number* of a transaction T_i in $\mathcal{T}$ ($sn(T_i)$) is defined as $n_i + 1$ where n_i is the number of transactions preceding T_i in $P(\mathcal{T})$.

Definition 4 The *recency value* of a transaction T_i in $P(\mathcal{T})$ ($rv(T_i)$) is defined as follows: $rv(T_i) = sn(T_i)/N$, where N is the total number of transactions in $P(\mathcal{T})$.

Definition 5 Given a set of transactions $\mathcal{T}$, a *recency spectrum* $Q(\mathcal{T})$ is defined as follows: $Q(\mathcal{T}) = (rv_{min}(\mathcal{T}), rv_{max}(\mathcal{T}))$ where $rv_{min}(\mathcal{T}) = min\{rv(T_i) : T_i \in \mathcal{T}\}$ and $rv_{max}(\mathcal{T}) = max\{rv(T_i) : T_i \in \mathcal{T}\}$.

Definition 6 A transaction T_k is given at least *degree r recency* with respect to spectrum $Q(\mathcal{T})$, if $ts(T_k) > ts(T_i)$ where T_i is a transaction in $\mathcal{T}$ such that $rv(T_i) = r$.

Note that T_k in the above definition is not an element in $\mathcal{T}$. It is a new transaction whose timestamp is computed based on its desired degree of recency using the currently executing active transactions in $\mathcal{T}$. If T_k is the first transaction to arrive, then $\mathcal{T}$ would be an empty set.

Suppose a transaction T_j is running at degree 0 recency, then it does not mean that $rv(T_j) = 0$. In fact, $rv(T_j)$ is actually decided by its position in the recency sequence. Suppose another new transaction T_i wishes to receive degree 1 recency and T_j is the only transaction in $\mathcal{T}$. In such a case, $rv(T_j) = 1$ and T_i is assigned a timestamp that is larger than $ts(T_j)$.

The recency sequence of transactions in example 2 is shown in figure 2 as $T_1, T_2, \ldots T_{100}$, where $1, 2, \ldots 100$ denote the sequence numbers of the transactions. Figure 3 depicts the recency values of these transactions. A transaction may specify the degree of recency ranging from 0 to 1. This range, in fact, reflects the range of the timestamps of all the active transactions at levels lower than that of the transaction. If transaction T_i specifies its degree of recency as 1, then it is serialized according to scheduler J, and therefore is placed at the extreme right of the recency spectrum, i.e., at position "c" in figure 3. On the other hand, if it chooses its degree of recency as 0, it is placed according to scheduler K, and thus is placed at the extreme left end of the spectrum, i.e., at position "a" in

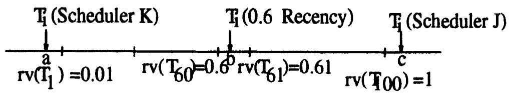


Figure 3. The Recency Spectrum

figure 3. If T_i specifies its degree of recency as 0.6, then it is placed at position b in figure 3. How would a transaction determine its desired degree of recency is beyond the scope of this paper.

We propose the following four different types of degree of recency. A transaction may specify its desired degree of recency according to one of these four types.

- Degree of recency by level (R_s)
- Degree of recency by data item (R_x)
- Degree of recency in general (R_g)
- Degree of recency by transaction (R_t)

In the first three types, the specified degree of recency should range from 0 to 1. However, in the fourth type it should be specified with respect to a particular transaction that is currently being executed, executed in the past, or going to be executed in near future.

4.1. Degree of recency by level (R_s):

With this type of recency, a transaction is allowed to specify its degree of recency with respect to a security level. By specifying a certain value of R_s, a transaction T_i is specifying that it wishes to wait for the completion of R_s fraction of all active transactions from the security level s. For example, a *very high*[‡] transaction T_i may specify its $R_{high} = 0.5$, meaning that it wishes to read only 50% recent values of the data from the security level *high*. According to our approach, T_i need not wait for the completion of all the earlier transactions at *high*, but needs to wait only for half as many. This type of recency can be defined as follows:

Definition 7 A transaction T_i is given *degree r recency by level* ($R_s(T_i) = r$) if it is given at least degree r recency with respect to spectrum $Q(\mathcal{T}_s)$ where $\mathcal{T}_s = \{T_j \in \mathcal{T} : L(T_j) = s\}$.

If a transaction specifies the degree of recency with respect to a level but also reads data from another lower level, then it may have to wait longer than required by the specified R_s. For example, consider once again the *very high* transaction T_i mentioned above. Suppose T_i, in addition to the data at *high* also has read data from *low*. Then it needs to wait for the completion of all active transactions at *low* whose timestamps are smaller than T_i.

4.2. Degree of recency by data item (R_x):

With this type, a transaction can choose to read a desired recent version on one or more data items. For example, if a transaction specifies its $R_x = 0.5$ means that it wishes to read at least 50% recent value of that specific data item x. This type of recency can be defined as follows:

[‡]We often use *very high, high* and *low* in our examples and discussions. They refer to three hierarchically ordered security levels in S such that *low* < *high* < *very high*.

Definition 8 A transaction T_i is given *degree r recency by data item* ($R_x(T_i) = r$) if it is given at least degree r recency with respect to spectrum $Q(\mathcal{T}_x)$ where $\mathcal{T}_x = \{T_j \in \mathcal{T} : L(T_j) = L(x)\}$.

A transaction may even specify different degrees of recency on more than one data item. For example, a transaction T_i may specify its $R_x(T_i) = 0.5$, $R_y(T_i) = 0.3$ and $R_z(T_i) = 0.2$. In such a case, if all the data items are of the same level, then T_i will be given 0.5 recency for all the three data items. This is because T_i needs to wait for a specified amount of time to read 50% recent value of x, and therefore, it can read a more recent versions of y and z as well.

On the other hand, if x, y and z belong to different security levels, then all these levels need to be considered to determine the timestamp of T_i. The computation of T_i will be carried out in two stages. In the first stage, a temporary timestamp is computed by examining each security level $L(x), L(y)$ and $L(z)$. In the second stage, the actual timestamp of T_i is computed, which is the largest of all the three temporary timestamps computed above. It is important to note that 0.5 recency at one level does not necessarily mean more recent than 0.3 at another level. We elaborate it with an example.

Example 4 Consider a system with three hierarchical security levels - *very high*, *high* and *low*. Suppose a transaction T_i at *very high* level wishes to read data items x and y where $L(x) = \textit{high}$ and $L(y) = \textit{low}$ with the degree of recency as follows: $R_x(T_i) = 0.5$ and $R_y(T_i) = 0.3$. Suppose there are 10 active transactions at *high* where $ts(T_5) = 58$ and $ts(T_6) = 62$. Similarly, assume that there are 100 active transactions at *low* where $ts(T_{30}) = 85$ and $ts(T_{31}) = 88$. $R_x(T_i)$ may result in a temporary timestamp of $T_i = 60$, while $R_y(T_i) = 0.3$ result in a temporary timestamp of $T_i = 87$. As can clearly be seen, $R_y(T_i) = 0.3$ selects more recent versions of data than $R_x(T_i) = 0.5$, although $R_x(T_i)$ is larger than $R_y(T_i)$. Therefore, in order to satisfy both the recency specifications, $ts(T_i)$ should lie between 85 and 88 rather than 58 and 62, as shown in figure 4. □

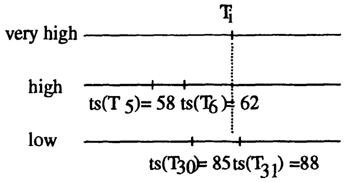


Figure 4. Computing $ts(T_i)$ in example 4

Notice that opting for R_x is same as opting for R_s if a transaction specifies a single item or number of items from the same level. But it differs from R_s if a transaction specifies R_x on number of items from different security levels.

4.3. Degree of recency in general (R_g):

In this case, a transaction may specify its degree of recency in general, that is without specifying either with respect to a level or with respect to a data item. It means that it does not wish to wait for the completion of all the active transactions. Degree of recency in general can be defined as follows:

Definition 9 A transaction T_i is given *degree r recency* in general ($R_g(T_i) = r$) if it is given at least degree r recency with respect to spectrum $Q(\mathcal{T}_g)$ where $\mathcal{T}_g = \{T_j \in \mathcal{T} : L(T_j) < L(T_i)\}$.

If a transaction specifies $R_g = 0$ and reads data from just one lower level, it may receive a version much older than the version it receives by specifying its degree of recency either as R_s or R_x. However, if a transaction specifies $R_g = 1$, then it needs to wait only for the completion of all active transactions at the level from which it has read, as opposed to waiting for all the earlier active transactions at all lower levels.

4.4. Degree of recency by transaction (R_t):

A transaction specifies its degree of recency such that it can choose to execute after a specified transaction. For example, suppose a transaction T_1 computes the salaries of every employee in an organization. This type of recency allows T_1 to choose to execute only after the completion of another transaction T_2 that updates the number of hours accumulated by each employee. This type of recency can be defined as follows.

Definition 10 A transaction T_i is given *degree of recency by transaction* with respect to a transaction T_j ($R_t(T_i) = T_j$) if $ts(T_i) > ts(T_j)$.

To specify such a degree of recency, a transaction is required to have some semantic knowledge of other transactions in the system. In other words, a transaction must have the knowledge of the purpose of the transactions that it is interested in. Notice that while the other three types of degrees of recency range between 0 and 1, there is no such range involved with this type of recency. However, by specifying $R_t(T_i) = T_j$, the degree of recency of data read by T_i depends on the degree of recency specified by T_j.

5. PROTOCOLS TO PROVIDE DIFFERENT DEGREES OF RECENCY

In this section, first we give protocols that determine the timestamp of a transaction based on the type and degree of recency specified by the transaction. Then we propose a single-level scheduler that provides concurrency control.

Algorithm 5.1 [Protocol to compute $ts(T_i)$, given $R_s(T_i)$]

1. There is a separate timestamp generator for each security level.
2. The timestamp generator at level $L(T_i)$ computes $ts(T_i)$ as follows:
   (a) Counts the total number of active transactions at level s. Let this value be N_s.
   (b) Forms a sequence P of these transactions by sorting them in the ascending order of their timestamps. Assigns a sequence number $sn(T_j)$ to each transaction T_j in the sequence as follows: $sn(T_j)$ = number of transactions preceding T_j in the sequence + 1.
   (c) Finds transactions T_k and T_l from P such that $sn(T_k) = \lceil R_s(T_i) * N_s \rceil$ and $sn(T_l) = sn(T_k) + 1$.[§]
   (d) Computes the timestamp of T_i as follows: $ts(T_i) > ts(T_k)$, $ts(T_i) \leq ts(T_l)$ and $ts(T_i) \neq ts(T_m)$ where T_m is any transaction at level $L(T_i)$.

[§]It may not always be possible to find both T_k and T_l when $R_s(T_i)$ is either 0 or 1. In such a case, some of the conditions in step d have to be ignored.

It is always possible to find a timestamp that satisfies all the conditions in step 2 (d) of the above algorithm since timestamps assigned to transactions that belong to the same level are unique. Therefore, it is always possible to find a timestamp that lies between two timestamps. The following example clearly explains the process of assigning timestamps.

Example 5 Suppose a transaction T_i specifies its $R_s(T_i) = 0.6$. The timestamp generator at level $L(T_i)$ first computes N_s, the total number of all active transactions at level s, and arranges them in the order of their timestamps. Let N_s be 101. Then it assigns a sequence number to each of these 101 transactions. Let this sequence of transactions be: $T_1, T_2 \ldots T_{101}$. Then the timestamp generator at $L(T_i)$ computes the timestamp of T_i as follows: First it computes $\lceil R_s(T_i) * N_s \rceil = \lceil 0.6 * 101 \rceil = 61$. Then $ts(T_i)$ is selected such that $ts(T_{61}) < ts(T_i) \leq ts(T_{62})$. Suppose $ts(T_{61}) = 893$ and $ts(T_{62}) = 897$. Then, $ts(T_i)$ may be assigned as 895. □

Algorithm 5.2 [Protocol to compute $ts(T_i)$, given $R_{x_1}(T_i), R_{x_2}(T_i) \ldots R_{x_n}(T_i)$]

```
1. There is a separate timestamp generator for each security level.
2. The timestamp generator at level $L(T_i)$ computes $ts(T_i)$ as follows:
   (a) Groups $x_1, x_2 \ldots x_n$ according to their security levels. Let these groups be
       $G_1, G_2 \ldots G_m$.¶
   (b) For each group $G_j$ the timestamp generator performs the following steps.
       i. Computes $R_{G_j}(T_i) = \max\{R_{x_k}(T_i) : x_k \in G_j\}$.
       ii. Assigns $s_j = L(x_k)$ such that $x_k \in G_j$.
       iii. Counts the total number of active transactions at level $s_j$. Let this value
            be $N_j$.
       iv. Forms a sequence $P_j$ by sorting all the active transactions at level $s_j$ in
           the ascending order of their timestamps.
       v. Assigns a sequence number $sn(T_m)$ to each transaction $T_m$ in $P_j$ as follows:
          $sn(T_m)$ = number of transactions preceding $T_m$ in the sequence + 1.
       vi. Finds transactions $T_{k_j}$ and $T_{l_j}$ from $P_j$ such that $sn(T_{k_j}) = \lceil R_{G_j}(T_i) * N_j \rceil$
           and $sn(T_{l_j}) = sn(T_{k_j}) + 1$.
       vii. Computes a temporary timestamp $ts_j(T_i)$ as follows: $ts_j(T_i) > ts(T_{k_j})$,
            $ts_j(T_i) \leq ts(T_{l_j})$.
   (c) The timestamp generator computes $ts(T_i) = \max\{ts_j(T_i)$, where $j = 1, \ldots m\}$
       and $ts(T_i) \neq ts(T_l)$ where $T_l$ is any transaction at level $L(T_i)$.
```

Algorithm 5.3 [Protocol to compute $ts(T_i)$, given $R_g(T_i)$]

```
1. There is a separate timestamp generator for each security level.
2. The timestamp generator at level $L(T_i)$ computes $ts(T_i)$ as follows:
   (a) Counts the total number of active transactions from all levels lower than $L(T_i)$.
       Let this value be $N$.
   (b) Forms a sequence $P$ of these transactions by sorting them in the ascending order
       of their timestamps. Assigns a sequence number $sn(T_j)$ to each transaction $T_j$
       in the sequence as follows: $sn(T_j)$ = number of transactions preceding $T_j$ in
       the sequence + 1.
```

¶Note that $m = n$ if $x_1, x_2 \ldots x_n$ belong to different security levels. Otherwise $m < n$.

(c) Finds transactions T_k and T_l from P such that $sn(T_k) = \lceil R_g(T_i) * N \rceil$ and $sn(T_l) = sn(T_k) + 1$.
(d) Computes the timestamp of T_i as follows: $ts(T_i) > ts(T_k)$, $ts(T_i) \leq ts(T_l)$ and $ts(T_i) \neq ts(T_m)$ where T_m is any transaction at level $L(T_i)$.

Algorithm 5.4 [Protocol to compute $ts(T_i)$, given $R_t(T_i) = T_k$]

1. There is a separate timestamp generator for each security level.
2. The timestamp generator at $L(T_i)$ computes $ts(T_i)$ as follows: $ts(T_i) > ts(T_k)$ and $ts(T_i) \neq ts(T_j)$ where T_j is any transaction at level $L(T_i)$.

Having assigned the timestamps to transactions by using one of the above four protocols based on the desired type and degree of recency, the following algorithm can be used for providing concurrency control. Notice that the protocols to compute timestamps are developed based on property 1 stated in section 3.

Algorithm 5.5 [The Scheduler]

1. Each transaction is given a timestamp as soon as it arrives using one of the four protocols depending on the type of recency specified by the transaction.
2. Each data item x of version j has a read timestamp $rts(x_j)$ and a write timestamp $wts(x_j)$ associated with it. We assume there is an initial transaction T_0 that writes into the database, such that $rts(x_0) = wts(x_0) = ts(T_0)$.
3. When a transaction T_i wants to read a data item x, and if $L(x) < L(T_i)$, then it selects a version x_k with the largest $wts(x_k)$ such that, $wts(x_k) < ts(T_i)$. However, T_i cannot commit until all active transactions whose timestamps are smaller than that of T_i finish their execution. If there is a $w_j[x_j]$ such that $ts(T_j) < ts(T_i)$, then T_i is reexecuted starting from $r_i[x_j]$.
4. When a transaction T_i wants to read a data item x such that $L(T_i) = L(x)$, then the scheduler selects a version x_k with the largest $wts(x_k)$ such that $wts(x_k) < ts(T_i)$, processes $r_i[x_k]$ and modifies $rts(x_k)$ as $rts(x_k) = \max\{ts(T_i), rts(x_k)\}$.
5. When T_i wants to write a data item x, the scheduler selects x_k as above. It rejects $w_i[x_i]$ if $rts(x_k) > ts(T_i)$; otherwise it processes $w_i[x_i]$ and modifies the timestamps of the new version x_i as $rts(x_i) = wts(x_i) = ts(T_i)$.

5.1. Proof of correctness

Theorem 3 Any schedule produced by the above algorithm is one-copy serializable. □

We prove this theorem using the following three lemmas. In the following proofs, we will use the following version order: $x_i \ll x_j$ iff $wts(x_i) < wts(x_j)$.

Lemma 4 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) < L(T_j)$, then $ts(T_i) < ts(T_j)$.

Proof: The proof is similar to the proof of lemma 1. □

Lemma 5 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) = L(T_j)$, then $ts(T_i) < ts(T_j)$.

Proof: The proof is similar to the proof of lemma 2. □

Lemma 6 If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_i) > L(T_j)$, then $ts(T_i) \leq ts(T_j)$.
Proof: Let $L(T_j) = s$. If there is an edge $T_i \rightarrow T_j$ in $MVSG(H)$ such that $L(T_j) > L(T_i)$, then this edge implies that there must exist a data item x such that $L(x) = s$ and $r_i[x_n], w_j[x_j] \in H$ with $x_n \ll_s x_j$. Now, the following two cases must be considered.
Case 1: Suppose $ts(T_i) \leq ts(T_j)$. This does not require any proof.
Case 2: Suppose $ts(T_i) > ts(T_j)$. In such a case, from step 2 of the above algorithm, T_i has to be reexecuted. Therefore, T_i reads a version x_n of x such that either $x_n = x_j$ or $x_n \gg_s x_j$. In either case, there cannot be an edge $T_i \rightarrow T_j$ (in the former case this edge would be a reads-from edge $T_j \rightarrow T_i$.). □
Proof of Theorem 3: The proof is similar to the proof of theorem 2. □

6. CONCLUSION

There are two multiversion concurrency control protocols that meet all the multilevel security requirements. The first solution, proposed by Keefe and Tsai [KT90] places transactions behind all active transactions at levels lower than its level and, as a result, transactions receive much older versions of data. The second solution, proposed by Jajodia and Atluri [JA92] gives high transactions the most recent version of data; however, it makes a high transaction that reads from a lower level wait for its commit until the completion of all earlier active transactions at that lower level. Since both solutions are not entirely satisfactory, we have proposed in this paper an approach in which each transaction can receive data having the degree of recency it desires. We have introduced four types of degrees of recency and have presented protocols for each type. As part of our future work, we plan to implement the proposed protocols and study the improvement in performance and recency of data with respect to schedulers J and K, respectively.

ACKNOWLEDGMENTS

The work of S. Jajodia was partially supported by National Science Foundation grants IRI-9303416 and INT–9412507 and by National Security Agency contract MDA904-94-C-6118.

REFERENCES

[AJ92] Paul Ammann and Sushil Jajodia. A timestamp ordering algorithm for secure, single-version, multi-level databases. In C. E. Landwehr and S.Jajodia, editors, *Database Security, II: Status and Prospects*, pages 23–5. North Holland, 1992.

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

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

[JA92] Sushil Jajodia and Vijayalakshmi Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure databases. In *Proc. IEEE Symposium on Security and Privacy*, pages 216–24, Oakland, California, May 1992.

[JK90] Sushil Jajodia and Boris Kogan. Transaction processing in multilevel-secure databases using replicated architecture. In *Proc. IEEE Symposium on Security and Privacy*, pages 360–8, Oakland, California, May 1990.

[KT90] T. F. Keefe and W. T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 369–83, Oakland, California, May 1990.

[MG90] William T. Maimone and Ira B. Greenberg. Single-level multiversion schedulers for multilevel secure database systems. In *Proc. 6th Annual Computer Security Applications Conf.*, pages 137–47, Tucson, Arizona, December 1990.

[P+93] Calton Pu et al. Distributed divergence control for epsilon serializability. In *Proc. IEEE International Conf. on Distributed Computing Systems*, pages 449–56, 1993.

Vijayalakshmi Atluri is an Assistant Professor of Computer Information Systems in the MS/CIS Department at Rutgers University. She received her B.Tech. in Electronics and Communications Engineering from Jawaharlal Nehru Technological University, Kakinada, India, in 1977, M.Tech. in Electronics and Communications Engineering from Indian Institute of Technology, Kharagpur, India, in 1979, and Ph.D. in Information Technology from George Mason University, USA, in 1994. Her research interests include Information Systems Security, Database Management Systems, and Distributed Systems.

Elisa Bertino is professor of computer science at the Department of Computer Science of the University of Milan where she heads the Database Systems Group. She has also been professor in the University of Genova, Italy. Until 1990, she was a researcher for the Italian National Research Council in Pisa, Italy. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation in Austin, Texas, at George Mason University. Her research interests include object-oriented databases, deductive databases, multimedia databases, database security. Prof. Bertino is a co-author of the book "Object-Oriented Database Systems - Concepts and Architectures" 1993 (Addison-Wesley). She is currently serving as program chair of the 1996 European Symposium on Research in Computer Security (ESORICS'96). She is on the editorial board of the IEEE Transactions on Knowledge and Data Engineering, and the International Journal of Theory and Practice of Object Systems.

Sushil Jajodia is Professor of Information and Software Systems Engineering and Director of Center for Secure Information Systems at the George Mason University. His research interests include information security, temporal databases, and replicated databases. He has published more than 125 technical papers in the refereed journals and conference proceedings and has edited or coedited nine books, including *Information Security: An Integrated Collection of Essays*, IEEE Computer Soc. Press (1995), *Multimedia Database Systems: Issues and Research Directions*, Springer-Verlag (1995), and *Temporal Databases: Theory, Design, and Implementation,* Benjamin/Cummings (1993). He is a coauthor of the forthcoming book *Principles of Database Security,* Benjamin/Cummings.