AN EXTENDED TRANSACTION MODEL APPROACH FOR MULTILEVEL SECURE TRANSACTION PROCESSING

Vijayalakshmi Atluri

MSIS Department and CIMIC, Rutgers University, Newark, NJ 07102

atluri@andromeda.rutgers.edu

Ravi Mukkamala

Department of Computer Science Old Dominion University, Norfolk, VA 23529

mukka@cs.odu.edu

Abstract Multilevel secure transaction processing has been well explored in the past decade. Despite this research, the proposed secure concurrency control protocols are not completely satisfactory because of the stringent constraints imposed by multilevel security. In this paper, we argue that modeling a transaction as an extended transaction model could significantly reduce the performance penalty. We accomplish this by minimizing (1) the probability of restarting a high security level transaction; and (2) the portion of the transaction to be reexecuted, when a restart becomes inevitable. In particular, we exploit the non-flat nature of transactions by identifying dependencies among various components of a transaction and portraying a transaction as an advanced transaction model. We demonstrate, via formal proofs, that our approach preserves the semantics of the transaction, and our concurrency control algorithm guarantees serializability.

Keywords: Multilevel Security, Transaction Processing, Advanced Transaction Model

1. INTRODUCTION

Access control requirements on sensitive data can be broadly classified into two categories: (1) *discretionary access control* (DAC), in which users, at their discretion, are allowed to grant permissions on the data they own, and (2) *mandatory access control* (MAC), in which all data are labeled based on their level of sensitivity and all users are given clearances, where users are allowed to access data with a given label only if their level of clearance permits them. Although many commercial applications employ DAC, it is not adequate where more stringent security requirements are needed as they are vulnerable to sophisticated attacks, such as Trojan Horse Attacks. Such application domains call for multilevel secure (MLS) systems that enforce MAC.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35587-0_24](https://doi.org/10.1007/978-0-387-35587-0_24)

M. S. Olivier et al. (eds.), *Database and Application Security XV*

© IFIP International Federation for Information Processing 2002

The area of transaction processing in multilevel secure (MLS) database management systems (DBMSs) has substantially progressed in the past few years. The major emphasis of this progress can be seen in secure concurrency control, one of the essential components of secure transaction processing. Concurrency control aims at synchronizing the operations of concurrent transactions to ensure correct execution.

Despite the substantial research activity, the proposed secure concurrency control protocols are not completely satisfactory because of the stringent constraints imposed by multilevel security. Researchers have proposed transaction processing protocols, in particular, concurrency control protocols using locking, timestamping, hybrid (combination of locking and timestamping) techniques. Other solutions try to tackle this difficult problem by maintaining multiple versions of data. Unfortunately, these solutions do not meet all the requirements of a secure concurrency control protocol; they trade either performance or recency for achieving security.

More specifically, the problem with the secure transaction processing is that if a data conflict between a low security level transaction and a high security level transaction should occur, for security reasons, the conflict must *always* be resolved in favor of the low security level transaction. In other words, the low level transaction has the right of way, whereas the high level transaction is hindered or restarted.

In the cases where a transaction has to be restarted, some solutions require that all of the transaction's effects must be undone and then the transaction is reexecuted from the beginning [1]. Other solutions reexecute starting from the first low read operation [7, 4]. These approaches however, cause resource wastage and in some cases results in starvation of high security level transactions. Much of the prior research in this area has been limited to the traditional transaction concepts.

Motivated by the need for releasing the rigidity, traditional transaction models have been extended in various directions. The objective of this paper is to investigate how the properties of the extended transaction models can be utilized to yield better secure concurrency control solutions for MLS DBMSs.

We have recognized that modeling a transaction as an extended transaction model, in particular as a workflow model, could reap significant gains in performance. In our work, we explore mechanisms that are specifically tailored for reducing the performance penalty to be paid for restarts by designing protocols such that (1) the probability of restarting a high security level transaction is minimum; and (2) even when a restart becomes inevitable, the portion of the transaction to be reexecuted is minimum.

This paper is organized as follows. In section 2, we introduce the preliminaries of our extended transaction model and the security model. In section 3, we review the related work in this area. In section 4, we present an overview of

our extended transaction model approach. In section 5, we present our secure concurrency control algorithm using the extended transaction model. Proofs of correctness that demonstrate that our redesign algorithm does indeed generate extended transactions that are equivalent to the original transactions, and the concurrency control algorithm guarantees (one-copy) serializability, have been omitted in this paper due to space limitations. Finally, in section 6, we summarize the results and discuss future work.

2. THE MODEL

In this section, we present our extended transaction model, and review the multilevel security model. Our approach assumes multiple versions of data and relies on the multiversion serializability theory reviewed in [7].

2.1. Extended Transaction Model

Driven by the needs of advanced application domains such as design, engineering, manufacturing and commerce, several extended transaction models have been proposed [6]. Many of these models recognize the fact that the transactions are not always flat. Some such extended models include nested transactions and workflow transactions. There has been an established model of nested transaction processing [3], which describes the way to ensure the correctness under the scenario where a transaction is split-up into various subtransactions. Splitting of a transaction into subtransactions based on semantic information is described in [2, 12]. Thus, in an extended transaction model, a transaction can be chopped into meaningful subtransactions, where there exist dependencies among them. In this paper, we recognize two types of dependencies among subtransactions within a transaction: data dependency and value dependency. In addition, we recognize another type of dependency between different transactions.

In the following, we develop the necessary formalism of the extended transaction model employed in this paper. Let T_i be a transaction. We use T_{ij} to represent the j th subtransaction of T_i .

Definition 1 We say that a data dependency $T_{ij} \rightarrow T_{ik}$ exists, if the results of the write operations in T_{ik} are dependent on the read operations of T_{ij} .

Definition 2 We say that a value dependency $T_{ij} \xrightarrow{exp} T_{ik}$ exists if T_{ik} has to be executed only if exp is true. Here, exp is an expression over the data items (accessed or modified) in T_{ij} .

The third type of dependencies that we present below are specified between either subtransactions of two *different* transactions, or between two transactions. We assume that every (sub)transaction is associated with a set of transaction management primitives that move the (sub)transaction from one state to the

other. We assume for each transaction and subtransaction, the set of primitives and states as follows:

	primitives	states
transaction	{begin, precommit, commit, abort}	{initial, done, commit, abort}
subtransaction	{begin}	{initial, done}

Definition 3 A control-flow dependency $T_{ij} \xrightarrow{s_{ij}, p_{kl}} T_{kl}$ states that primitive p_{kl} of T_{kl} is invoked only if T_{ij} enters state s_{ij} .

Although several types of control-flow dependencies may be defined based on the above states and primitives [5], in this paper, we use the following two.

1. $T_{ij} \xrightarrow{bd} T_{kl}$: This states that T_{kl} begins only if T_{ij} enters a “done” state.
2. $T_i \xrightarrow{t} T_j$: This states that T_j terminates (either commit or abort) only if T_i terminates.

2.2. Multilevel Security Model

We assume the security structure to be a partially ordered set $\mathcal{S}$ of security levels with ordering relation $\leq$. A class $s_i \in \mathcal{S}$ is said to be *dominated* by another class $s_j \in \mathcal{S}$ if $s_i \leq s_j$.

Let D be the set of all data objects. Each data object $d \in D$ is associated with a security level. Every transaction T_i is associated with a security level. We assume that there is a function L that maps all data objects and tasks to security levels. That is, for every $d \in D$, $L(d) \in \mathcal{S}$, and for every transaction $T_i \in W$, $L(T_i) \in \mathcal{S}$. We require every transaction to obey the following two security properties — the simple security and the restricted $\star$ -property.

1. A transaction T_i is allowed to read a data object d only if $L(d) \leq L(T_i)$
2. A transaction T_i is allowed to write to a data object d only if $L(d) = L(T_i)$.

In addition to these two restrictions, a *secure* system must prevent illegal information flows via covert channels.

3. RELATED WORK

Secure concurrency control with traditional transaction models has been well understood [8, 1, 7, 9, 10]. In the following, we only review the solutions that are closely relevant to our work. The protocol proposed by Keefe and Tsai [8] uses a priority queue where transactions are placed in the queue based on the security level of the transaction. A transaction is assigned a timestamp that is smaller than all active transactions executing at lower security levels.

As a result, a *high* transaction's read operation does not interfere with a *low* transaction's write operation since the *high* transaction is considered as an older transaction though it arrives later than the *low* transaction. Although this requires a trusted code for implementation, it ensures one-copy serializability (the usual notion of correctness when multiple versions are maintained in the database) without compromising security. Moreover, transactions are never subjected to starvation. However, high transactions are sometimes given stale versions of the low data.

The protocol proposed by Jajodia and Atluri [7] assigns timestamps to transactions as in a conventional timestamp ordering protocol. According to this protocol, transactions are sometimes made to wait for their commit. Whenever a *high* transaction reads *low* data, it is not allowed to commit until all *low* transactions with smaller timestamps than that of itself commit. This is because a *low* level transaction with a smaller timestamp can always invalidate the read operation of the *high* transaction by issuing a write. In such an event, the *high* transaction is reexecuted. This protocol guarantees one-copy serializability, is secure, and free of starvation. In addition, it can be implemented with completely untrusted code. Restarting a transaction may have significant impact on performance, especially if they are not short lived.

Researchers have proposed other solutions that compromise correctness for ensuring security. Jajodia and Atluri [7] have proposed three increasingly stricter notions of correctness for multiversion multilevel databases — *level-wise serializability*, *one-item read serializability* and *pairwise serializability* — that are weaker than one-copy serializability.

Bertino et al. [4] proposed a lock-based scheme for implementing transactions in a multilevel secure file storage system. Assuming the availability of a *trusted* lock manager and a *trusted* file manager, they implement a covert-channel-free MLS system. The main contribution of their work is the introduction of a "signal lock" along with the usual read lock and write lock in a 2PL scheme. They do assume a single version for each data item and use strict 2PL for concurrency control for transactions at the same security level. A high-level transaction intending to read a low-level data item obtains a signal-lock on the item. If the item is already write-locked by a low-level transaction, then the high-transaction has to wait until the write-lock is released. A low writer however does not need to wait when a data item is signal-locked by a high-level reader. In addition, they provide language primitives by which a high-level transaction can roll-back to a prespecified step when it receives a signal from the low-level indicating that the data item that it read has now been written. A transaction can then rollback to the state prior to the read statement. Similar strategy is also explained by Ray et al. in [11]. Both the approaches in [4, 11] incur the same penalty in performance as that in [7].

Our work deviates from prior work significantly. The novel part of our work is that we exploit the non-flat nature of transactions by identifying the dependencies among various components of a transaction. In other words, we redesign a transaction and model it as an advanced transaction model. As a result of this redesign, we can minimize the number of restarts or even unnecessary rollbacks to a prior save-state.

4. OVERVIEW OF THE EXTENDED TRANSACTION MODEL APPROACH

Our approach to achieve efficient execution of multilevel transactions relies on first recognizing the data dependencies among the various operations (sets of operations), and the non-flat nature of the transaction.

As mentioned in section 1, the goal of our approach is to reduce the amount of reexecution of the transaction when restart becomes inevitable. In the following, we describe three cases where such optimizations are possible. In the first case, a restart is necessary, however, the transaction need not have to be restarted in full. In the second case, a particular data modification by a low security level transaction does not necessitate any form of corrective action by a high security level transaction. In the third case, a data modification may require some corrective action, but the compensatory action is pre-specified and can be efficiently accomplished than by restarting the transaction. All these three techniques can also be used in conventional transaction processing systems. However, we expect that using them in the secure transaction processing scenario will significantly enhance the performance.

4.1. Exploiting non-flat nature of a transaction

Let us consider a transaction T_1 with subtransactions and dependencies as shown in Figure 1. As can be seen, the subtransaction T_{17} comprises of $read[x]$ where $L(x) < L(T_1)$. Suppose T_1 reads x . But, T_1 may have to be reexecuted due to a conflicting operation by a lower level transaction to ensure serializable execution. In such a scenario, not all parts of the transaction need to be reexecuted.

Figure 1. An example of partial reexecution

Our approach exploits the dependencies among subtransactions so that only those subtransactions which are effected by the modified low data item are reexecuted. For example, in Figure 1, subtransactions T_{11} , T_{12} , T_{16} , T_{18} and T_{19} are not required to be reexecuted for ensuring correctness since the change in the value of x does not affect those subtransactions.

4.2. Exploiting the value dependencies among subtransactions

Consider the scenario featured in Figure 2. Depending on the value of x , the transaction could follow either the execution sequence A (T_{18} , T_{19}) or the sequence B (T_{13} , T_{14} , T_{15}). Now assume that the value of x is 5 when the high level transaction begins its execution. Since the condition $x < 20$ is satisfied, the transaction follows the sequence A. Now if the value of x was changed to 6 at a later point of time due to a lower security level transaction, the transaction need not roll back since the condition $x < 20$ still holds. Therefore, by modeling this as a value dependency, one may evaluate the condition and thereby avoid reexecution entirely, and reexecute it only when it is necessary.

Figure 2. An example of value dependency approach

4.3. Employing compensating subtransactions

Now consider the case in Figure 3. In this case, the value of x changes from 6 to 10, while the transaction is executing. In a conventional scenario, this requires a transaction restart. However, the restart could be prevented here by executing a compensating module. We realize that devising a compensating action is much more difficult, and moreover, some subtransactions may not be compensatable. In this paper, we do not address this issue.

4.4. Example

In the following, we present an example where the dependencies introduced in section 2.1 exist among a real-world database scenario. Consider a corporate database that contains several types of information about its employees: social security number, name, address, rank, years of service, weekly pay (p), number of hours worked per week (h), number of overtime hours worked per week (o), hourly rate (r), additional compensation (c), yearly bonus, etc. The information

Figure 3. An example of compensation

such as weekly pay, hourly rate and additional compensation are considered sensitive, and are labeled *high*, while the rest of the information is classified *low*. Thus, $L(h) = L(o) = \text{low}$, and $L(p) = L(c) = L(r) = \text{high}$.

At the end of each week, the number of hours worked by an employee as well as the weekly pay of each employee is computed. The weekly pay of each employee is computed as follows:

Pay = ((number of regular hours + number of overtime hours)*hourly rate) + c, if overtime hours ≥ 20
 = ((number of regular hours + number of overtime hours)*hourly rate), otherwise.

That is there will not be any additional compensation if an employee works less than 20 overtime hours per week. Since h and o are *low* data items, a *low* transaction (T_1) is initiated to enter h and o . A *high* transaction (T_2) is initiated to compute the weekly pay. They are as follows:

$T_1 = \text{write}[h] \text{write}[o]$
 $T_2 = \text{read}[h] \text{read}[o] \text{read}[r]$
 $\text{read}[c] [p = ((h + o) * r) + c] \text{write}[p], \text{ if } o \geq 20$
 $[p = (h + o) * r] \text{write}[p], \text{ otherwise}$

Based on the semantics of the transaction, the dependencies as shown in Figure 4 can be seen among the operations within a transaction. Note that there exist both data dependencies as well as value dependencies.

5. PROPOSED CONCURRENCY CONTROL ALGORITHM

Algorithm 1 [The Scheduler]

- 1 There is a separate scheduler for each security level s .
- 2 Whenever a new transaction T_i arrives, the scheduler performs the following steps:
  - (a) Assign a unique timestamp $ts(T_i)$. We assume that there is a shared hardware clock at system low which is used by each scheduler at

Figure 4. An extended transaction model for T_1 and T_2

level s to assign a unique timestamp $ts(T_i)$ to every transaction T_i at its level.

(b) Each transaction T_i declares its write- and read-sets. The scheduler at level s maintains $W(T_i)$ and $R(T_i)$.

(c) If there exists a $read_i[x]$ such that $L(x) = s' < L(T_i)$,

- i find every T_j such that $L(T_j) = s'$, $ts(T_j) < ts(T_i)$ and $W(T_j) \cap R(T_i)$. Add dependencies $write_j[x] \xrightarrow{bd} read_i[x]$ and $c_j \xrightarrow{t} c_i$.

ii Restructure T_i according to the restructuring algorithm.

3 Each version x_j of a data item x 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 with $rts(x_0) = wts(x_0) = ts(T_0)$.

4 When a transaction T_i wants to read a data item x and if $L(T_i) = L(x)$, the scheduler selects a version x_k with the largest $wts(x_k)$ such that $wts(x_k) < ts(T_i)$, processes $read_i[x_k]$, and modifies $rts(x_k)$ as $rts(x_k) = \text{maximum}\{ts(T_i), rts(x_k)\}$.

5 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 $write_i[x_i]$ and modifies the timestamps of the new version x_i as $rts(x_i) = wts(x_i) = ts(T_i)$.

6 Suppose a transaction T_i wants to read a data item x . If $L(T_i) > L(x)$, the scheduler at level $L(T_i)$ selects a version x_k with the largest $wts(x_k)$ such that $wts(x_k) < ts(T_i)$. Although the operation $read_i[x_k]$ is processed, the read timestamp $rts(x_k)$ is not modified for security reasons.

Algorithm 2 [Restructuring]

- 1 Add data dependencies between the operations of each transaction based on their semantics.
- 2 If there exist a value dependency $d_k = o_i[x] \xrightarrow{exp} o_i[y]$, replace d_k with the following: $o_i[x] \xrightarrow{exp} inc[counter_k] \xrightarrow{counter_k < 2} o_i[y]$.

Definition 4 We say that a transaction T is *semantically equivalent* to another transaction T' , (i) if the set of data operations of both T and T' are equal, and (ii) if the values read by each read operation in both T and T' are the same, then the values written by each write operation in T and T' are the same.

Theorem 1 [Proof of Equivalence] Let T be a transaction. Let T' be the restructured transaction using algorithm 2. Then T' is semantically equivalent to T .

Theorem 2 [Proof of Correctness] Let H be a multiversion history produced by algorithm 1. Then H is one-copy serializable. In fact, H is equivalent to a one-serial history in which transactions are placed in a timestamp order.

5.1. Discussion

According to Step 2(c) of algorithm 1, the restructuring of a transaction is done only if a transaction reads from low. Otherwise, it is executed as a flat transaction. If it reads from low, the first step of algorithm 2 is to exploit the partial order of the operations and the data dependencies among them. In the second step of algorithm 2, we add counters to make sure that the operations are redone only if the condition has changed since the prior read. Let us consider various cases:

1. Suppose the schedule in the above example is

T2	read2[h]	read2[o]
T1	write1[h]	write1[o]

Then the read2[o] operation is redone after write1[o] because of the “bd” dependency, as shown in Figure 5. Therefore this will result in a serializable schedule.

2. Suppose the schedule is

T2	read2[h]	read2[o]
T1	write1[h]	write1[o]

The above read2[o] is in fact the one triggered by write1[o]. This is also correct. So by adding these two dependencies, it makes sure that the write of

low precedes the read of low if it hasn't already occurred in that order. That is, only the out-of-order read operation is redone but not all. This way, we are saving on the number of redoes. The dotted arrows indicate the added dependencies between transactions according to step 2(c)(i) of algorithm 1. Moreover, according to step 1 of algorithm 2, only parts of the transaction are to be redone, but not all.

Figure 5. Dependencies added between T_1 and T_2 as in step 1 of algorithm 1

Now the idea of the second step of the restructure algorithm is to provide further optimization. The objective is to exploit the nature of the value dependency to avoid the redoing certain parts of the transaction by examining the value. For example, during the high transaction execution, the low value of “o” has changed. This does not have to necessarily trigger a redo. Consider the value read by `read[o]` is 10. Now it has changed to 15. This should not have any effect on the redoing part, and therefore, the `write[p]` operation (in Figure 6) does not have to be redone. This is accomplished by incorporating counters in that path and incremented whenever transaction executes that part of the transaction.

Figure 6. Exploiting the value dependencies in T_2

5.2. Implementation

The proposed system can be implemented in several ways. Here, we describe one such implementation. We assume that our system is implemented using the kernelized architecture. Thus, both the transaction manager (TM) and the Scheduler are untrusted. (We assume the functionalities of algorithms 1 and 2 are implemented in the TM and the scheduler. When a new transac-

tion is received by the appropriate level TM, it assigns the transaction a unique timestamp by reading from a clock at system low. We assume that the granularity of the clock is high enough so that the timestamps are unique among the transactions at all the security levels.

A TM, upon receiving a transaction, invokes its redesign module. This module first identifies the low-level reads, and their dependencies with write operations at its own level. It builds both the value and data dependencies among the operations, as per algorithm 2. We refer to this as the dependency graph for a transaction. At this stage, however, there is no knowledge of the actual low-level transactions that the high-level transaction under consideration may be in conflict with. The TM, now, sends low-level read requests to the scheduler at its level. Since we assume a timestamp-based concurrency control, each read request is associated with a timestamp of the transaction that originated it. Accordingly, the scheduler selects the value of the latest version whose write timestamp is smaller than that of the transaction issuing the read request. The read requests are then sent to the trusted OS to retrieve the data from the appropriate DB (as in a kernelized architecture). In addition to the data, the request also reads the active transaction log maintained by the scheduler, and determines the active transactions and their timestamps that conflict with its own read. (An active transaction is one that has started but not yet terminated.) Thus, the low-level read request returns the data value, its timestamp, and a set (possibly empty) of active low-level transactions that conflict with the high-level read.

The high-level TM, after receiving the transaction sets from the low-level read operations, saves them for use at the time of transaction commit. Actually, it is only concerned with low-level transactions with timestamps smaller than the high-level transaction and with read-write conflicts. Let us call this set the active-transaction set (ATS). At the time of transaction commit, the TM needs to determine the state of the transactions in the ATS. Accordingly, it sends "read-log" request to the low-level DBMS through the TCB. In return, it is presented with the current list of low-level active transactions. If none of the transactions in its ATS are still active, then it proceeds with the next step of examining the dependency graph. Otherwise, it waits and repeats the above two steps periodically. When all transactions in the active-set have terminated, the high-level TM resends all low-level read operations to the low-level scheduler. When the new values are returned, it checks with the dependency graph to determine the operations that need to be reexecuted. The reread values are used for both data and value dependencies.

In summary, our implementation uses untrusted TMs and schedulers at each security level, logs maintained by the schedulers indicating the current active transactions and their read- and write-sets, and multiple versions of data items.

6. SUMMARY AND FUTURE WORK

In this paper, we described an extended model for multilevel secure transaction processing. In a traditional MLS model, a high-level transaction is aborted and restarted whenever its low-level reads conflict with low-level transactions' writes. This is often a concern for high-level users in real systems. Here, we suggest a way to extend the rigid transaction model by using some of the workflow transaction redesign techniques. The proposed scheme uses timestamping as a means of concurrency control. In addition, it employs multiversion data. It employs value and data dependencies to identify the relationship between low-level read data and high-level writes within a transaction. This information is then used at the time of transaction commit. Depending on the conflicts during execution between a high-level transaction and low-level transactions, only some parts of a high-level transaction may need to be reexecuted. But what parts, if any, are to be reexecuted is determined by the data and value dependencies identified earlier and the actual conflicts and data values. We show the correctness of the proposed scheme by proving the serializability of the resulting transaction histories.

In the future, we plan to evaluate the performance of the proposed method in terms of the throughput offered at higher security levels as well as the response time. We also plan to measure the additional protocol overhead imposed by the proposed system, especially the cost of maintaining additional versions of data, the list of active transactions, and the cost of active transaction enquiry and reread of low-level data. The costs and benefits will be compared with those of a traditional scheme where a high-level transaction is simply aborted when its low-read operation conflicts with a low-level write operation. In addition, we plan to compare the performance of our scheme with other schemes [11, 4] that have similar objectives.

Acknowledgments

The work of V. Atluri was supported in part by the National Science Foundation under grant IRI-9624222. We would like to acknowledge Binto George for taking part in the initial discussions on this work, while he was at Rutgers.

References

- [1] Paul Ammann and Sushil Jajodia. A timestamp ordering algorithm for secure, single-version, multi-level databases. In Carl Landwehr and Sushil Jajodia, editors, *Database Security, II: Status and Prospects*, pages 23–25. North Holland, 1992.
- [2] Paul Ammann, Sushil Jajodia, and Indrakshi Ray. A Semantic-Based Transaction Processing Model for Multilevel Transactions. *Journal of Computer Security*, 6(3), 1998.
- [3] C. Beeri, P. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of ACM*, 36(2), 1989.

- [4] Elisa Bertino, Sushil Jajodia, Luigi Mancini, and Indrajit Ray. Advanced transaction processing in multilevel secure file stores. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):120–135, 1998.
- [5] Panos K. Chrysanthis and Krithi Ramamritham. ACTA, A framework for specifying and reasoning about transaction structure and behavior. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 194–203, June 1990.
- [6] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, California, 1992.
- [7] 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 216–224, Oakland, California, May 1992.
- [8] T. F. Keefe and W. T. Tsai. Multiversion Concurrency Control for Multilevel Secure Database Systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, California, May 1990.
- [9] 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 137–147, Tucson, Arizona, December 1990.
- [10] John McDermott and Sushil Jajodia. Orange Locking: Channel-free Database Concurrency Control via Locking. In *Proc. of the 6th IFIP WG 11.3 Workshop on Database Security*, Vancouver, BC, August 1992.
- [11] Indrajit Ray, Luigi V. Mancini, Sushil Jajodia, and Elisa Bertino. Asep: A secure and flexible commit protocol for mls distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):880–899, 2000.
- [12] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple Relational Guidance for Chopping Up Transactions. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 298–307, San Diego, CA, June 1992.