Transaction Processing Environment Kernelized Architecture in Multilevel Secure Application Policies
Jaychand Vishwakarma
Information Technology, VBS Purvanchal University, Jaunpur,
Uttar Pradesh, India
Email: jaychandvbs04@gmail.com

Abstract—Multilevel security poses many challenging problems for transaction processing. The challenges are due to the conflicting requirements imposed by confidentiality, integrity, and availability—the three components of security. We identify these requirements on transaction processing in Multilevel Secure (MLS) database management systems (DBMSs) and survey the efforts of a number of researchers to meet these requirements. While our emphasis on centralized system based on kernelized Architecture, we briefly overview the research in the distributed MLS DBMSs as well.

Keywords—Database systems, transaction processing, security, concurrency control, mandatory access control, covert channel, kernelized architecture, replicated architecture.

I. INTRODUCTION
MULTILEVEL SECURE (MLS) databases are characterized by systems that contain sensitive data classified at different levels of sensitivity. An MLS database management system (DBMS) enforces controlled access to data by assigning a security clearance to every user. Unlike conventional databases, users of MLS databases cannot, at their discretion, give away the access privileges on their data to other users, even though they are the owners of the data. An MLS DBMS decides the access privileges of the users based on the classification label associated with the data and the clearance level of the user. These are the so-called mandatory access controls [12]. Classifications and clearances are collectively known as security classes or levels. A security class is made up of two components: a hierarchical component and a (possibly empty) non-hierarchical component, called a category. As an example, the different security levels distinguished in the structure of U.S. military security are unclassified, confidential, secret, and top secret, ordered in the increasing order of their importance to the national security. Categories are independent of each other, and are not ordered. NATO, Nuclear, and Crypto are examples of some of the categories used by U.S. military. They form the basis for enforcing need-to-know.

The general structure of security classes’ forms a lattice using the following dominance relation: A security class is said to dominate another security class if the hierarchical component of the first is greater than or equal to that of the second and the category set of the first contains all the categories of the second. (See [1],[12],[28] for additional details on multilevel security.) To enforce mandatory access control, all MLS systems use a trusted computing base (TCB), a small part of the operating system, which is responsible for all security relevant actions of the system. The TCB always has to be invoked, cannot be bypassed, and must be shown to perform only its intended functions (the code that implements the TCB must be verified for the presence of any malicious code). Concurrency control is crucial to transaction processing since databases are generally meant to cater to multiple users. Therefore, every DBMS must be equipped with a proper concurrency control mechanism to ensure correct concurrent execution of transactions. Concurrency control in multilevel secure databases, in addition to ensuring correct concurrent execution of transactions, must also preserve security. The security community recognizes secrecy (also known as confidentiality), integrity and availability as inherent components of security (see [1] for details). However, designing a concurrency control protocol that preserves all of the above three components of security is not trivial. For example, the conventional concurrency control protocols such as two-phase locking and timestamp ordering [8] do not meet the multilevel confidentiality requirements. They are prone to signaling channels, a type of illegal information flow channels that can be used by malicious transactions to leak sensitive information. (A detailed discussion of signaling channels can be found in Sections 2.1 and 4.)

This article is organized as follows. Section 2 identifies the requirements to be met by every secure concurrency control protocol. Section 3 presents the scope of this survey. In Section 4, we present how conventional protocols do not meet the security requirements with the help of examples. In Section 5, we present the efforts of a number of researchers in designing concurrency control protocols that meet all the multilevel security requirements. Since research
in multilevel secure databases is relatively new, most of the research efforts in the area of secure transaction processing are focused on centralized databases. In addition to providing secure concurrency control, distributed transaction processing involves preserving atomicity without introducing signaling channels. Section 6 presents the research efforts in the area of secure distributed transaction processing. Finally, Section 7 presents conclusions.

II. REQUIREMENTS ON SECURE CONCURRENCY CONTROL PROTOCOLS

In this section, we discuss how the three components of security, namely, secrecy, integrity, and availability impose restrictions on database concurrency control.

2.1 Secrecy Requirement

A concurrency control protocol to be used in an MLS DBMS must meet all the multilevel security requirements described below. Mandatory access controls are usually based on the Bell-LaPadula model [7]. This model is defined in terms of subjects and objects. Subjects are the active entities in the system such as, users, processes initiated by the users etc., and objects are the passive entities that contain information such as, files, memory, I/O devices etc. Mandatory access control consists of two properties that restrict the read and write accesses by subjects (transactions) on objects (data items). These properties result in the following mandatory security requirements on transactions: • A transaction is not allowed to read a data item unless the security level of the transaction dominates the security level of the data item. • A transaction is not allowed to write a data item unless the security level of the data item dominates the security level of the transaction. These two requirements are intended to prevent information flow from higher security levels to their lower level counterparts. While the intention of the first condition is obvious, the latter restriction is to guard against Trojan horses. A Trojan horse is a malicious piece of code that when initiated by unsuspecting higher level users may insidiously copy higher level data into lower level data domains. Therefore in a secure system, although users are trusted not to deliberately divulge information, the processes initiated by the users need not be trusted since they may be programmed (without the knowledge of the user) to cause leakage of information. When a higher level user wants to update lower level data, he or she must do so by first logging in as a lower level user and then initiating a lower level transaction [12]. Although the second requirement of the Bell-LaPadula (known as the ⋄-property) allows transactions to write data items at higher levels, most MLS systems disallow such write-ups for integrity reasons [19]. Unfortunately, although the above two restrictions prevent direct leakage of sensitive information from higher to lower security levels, there are, however, other means of leakage via signaling channels. A signaling channel [19] is an indirect way of leaking higher level information to lower security levels. Such channels can arise due to data contentions among transactions executing at different security levels. For example, a higher level transaction by reading a lower level data item may cause a lower level transaction attempting to write this data item to be delayed or aborted. Thus, a higher level transaction by choosing either to read or not to read a lower level data item can signal higher level information to lower level subjects. (A more concrete example of a signaling channel can be found in Section 4.) Effectively, a higher level transaction can establish a communication channel by simply accessing selected lower level data items according to some agreed upon code. Therefore, to prevent such indirect leakage of sensitive information, a concurrency control protocol must meet the following requirement:

REQUIREMENT 1. A secure concurrency control protocol must be free of signaling channels.

2.2 Integrity Requirement

The aim of a concurrency control protocol is to ensure correct execution when a number of transactions are executing concurrently in the database system. Correct execution of a set of transactions simultaneously accessing the database is characterized by the effect of these transactions on the database. The traditional notion of correctness is serializability [8].Serializability means that the effect of concurrent execution of transactions is equivalent to that of the transactions executed serially. Thus, to ensure database integrity, the following requirement has to be met:

REQUIREMENT 2. A secure concurrency control protocol must guarantee serializability.

2.3 Availability Requirement

Often, confidentiality requirements conflict with database integrity requirements [15],[30],[34]; requirements 1 and 2 are no exception. In order to meet these two requirements, a concurrency control protocol may either subject transactions to indefinite delays or repeatedly abort them, a problem known as starvation. Although a number of transactions are competing to concurrently access the database, the database system must be made available to all the transactions. This requirement can be stated as follows:

REQUIREMENT 3. A secure concurrency control protocol should not cause starvation.
2.4 Single-Level Implementation Requirement
In addition to the above three requirements, it is always desirable that the secure concurrency control protocol meet the following:

REQUIREMENT 4. A secure concurrency control protocol must be implementable with untrusted code.

There are two reasons for this requirement. First, in high assurance systems it is crucial that the size of the trusted code be kept as small as possible since all the code used for implementation has to be verified for its trustworthiness [14]. A rigorous verification of the code is considered to be a very expensive task. Therefore, it is desirable that the scheduler providing concurrency control be implementable with code that is not required to be trusted. Second, the trusted scheduler needs additional security verification and must be layered on top of the already evaluated TCB. There is no guarantee that the composition of these two secure components will be secure [32]. Notice that for implementing the scheduler with completely untrusted code, the scheduler must be single-level, i.e., it must be separate for each security level. A concurrency control protocol that satisfies this requirement, automatically meets requirement 1 since there are no trusted components in the protocol. The challenge, therefore, is to build a protocol that meets all the four requirements stated in this section.

III. SCOPE OF THE SURVEY
Most of the recent research and development in secure transaction processing can be categorized broadly into two different areas: one based on kernelized architecture (see Fig. 1) and the other based on replicated architecture (see Fig. 2). These two are among the number of architectures proposed by the Woods Hole study group [10] as a solution to build multilevel secure DBMSs by reusing the existing DBMS technology instead of building a trusted DBMS from scratch. These two architectures differ in the way in which they physically store data. Both these architectures use the notion of trusted front end (TFE), which cannot be bypassed. In the kernelized architecture, a multilevel database is partitioned into single-level databases, in which data at each security level is stored separately. This architecture uses a separate DBMS for each security level to manage data at or below that level. The trusted front end ensures that the users’ queries are submitted to the DBMS with the same security level as that of the user, while the trusted back end makes sure that a DBMS at a specific security level accesses data without violating the mandatory security policy (in other words, without circumventing the BellLaPadula restrictions).

Processing of a user’s query accessing data from multiple security levels involves expensive joins that may degrade the performance since the different levels of data are stored separately. On the other hand, since this architecture has separate DBMSs for each security level, the scheduler that is responsible for concurrency control can also be separated for each level. Therefore, the scheduler can be implemented with untrusted code and need not be part of the TCB. In replicated architecture, like kernelized, data at each security level is stored in separate containers; however, in addition to the data at that level, copies of data at all lower levels are also maintained at each level. This architecture also uses a separate DBMS to manage data at each security level. Thus, if a high transaction wishes to read data from a low 1 level, it will be given the replica of the low level data maintained in the high container.

As a result, this architecture is impractical for large number of security levels. Though query processing is not expensive as in the kernelized architecture, the critical issue of transaction processing in this architecture is the propagation of updates of the lower level data to higher level DBMSs in a secure and correct manner. Jajodia and Kogan [19] first proposed a secure protocol to propagate the updates which uses a little amount of trusted code. This and algorithms which followed [11],[24] shared a flaw that they did not generate a consistent ordering either for general partial orders or even for lattices [3]. It was shown that consistent orderings are generated if the security structures are restricted to planar lattices [3] or to multilevel-acyclic partial order [23]. Finally, a standard characterization of partial orders known as the crown-free property was found to be necessary and sufficient for generating consistent orderings [4]. This article does not discuss the research on transaction processing in systems based on replicated architecture any further; henceforth, it concentrates only on those based on kernelized architecture.

IV. IMPACT OF SECURITY REQUIREMENTS ON TRADITIONAL CONCURRENCY CONTROL PROTOCOLS
Because of its ease of implementation, two-phase locking protocol has been extensively used for concurrency control in almost all commercially available database management systems. According to this protocol, a transaction should acquire a read (write) lock before it reads (writes) a data item, and every transaction must go through two-phases:

- a growing phase, and
- a shrinking phase.

A transaction should acquire all the required locks during the growing phase and should relinquish these locks during the shrinking phase. However, once a transaction releases a lock, it cannot acquire any more locks. Unfortunately,
as stated earlier, this protocol is not suitable for multilevel secure environments since it is prone to signaling channels. In the following, we show with the help of an example how a signaling channel can be established using the two-phase locking protocol. Consider a database that stores information of two types: low and high. Any low information is made accessible to all users of the database by the DBMS; on the other hand, high information is available only to a select group of users with special privileges. In accordance with the secrecy requirements, a transaction executing on behalf of a user with no special privileges would only be able to access (read and write) low data elements, while a high transaction (initiated by a high user) would be given full access to the high data elements and read-only access to the low elements. A transaction should acquire all the required locks during the growing phase and should relinquish these locks during the shrinking phase. However, once a transaction releases a lock, it cannot acquire any more locks. Unfortunately, as stated earlier, this protocol is not suitable for multilevel secure environments since it is prone to signaling channels. In the following, we show with the help of an example how a signaling channel can be established using the two-phase locking protocol. Consider a database that stores information of two types: low and high. Any low information is made accessible to all users of the database by the DBMS; on the other hand, high information is available only to a select group of users with special privileges. In accordance with the secrecy requirements, a transaction executing on behalf of a user with no special privileges would only be able to access (read and write) low data elements, while a high transaction (initiated by a high user) would be given full access to the high data elements and read-only access to the low elements.

Imagine a “conspiracy” of two transactions: $T_1$ and $T_2$. $T_1$ is a transaction confined to the low domain that wishes to write two low data elements $x$ and $y$; $T_2$ is a transaction initiated by a high user and, therefore, able to read all data elements, wishes to read the low data elements $x$ and $y$ and update a high data element $z$. Assume only these two transactions are currently active. If $T_2$ first requests to read $x$, a lock will be placed on $x$ for that purpose. Suppose that next $T_1$ wants to write $x$. Since $x$ has been locked by another transaction, $T_1$ will be forced by the scheduler to wait. $T_1$ can measure such delays, for example, by going into a busyloop with a counter. Thus, by selectively issuing requests to read low data elements, transaction $T_2$ could modulate delays experienced by transaction $T_1$, effectively sending signals to $T_1$. Since $T_2$ has full access to high data, by transmitting such signals, it could pass on to $T_1$ the information that the latter is not authorized to see. The information channel thus created is a signaling channel, which is shown

- **high**: $T_2$, $z$; read-lock2[$x$], $r2[x]$; read-lock2[$y$], $r2[y]$; write-lock2[$z$], $w2[z]$
- **low**: $T_1$, $x$, $y$; $w1[x]$ will be delayed

![Fig.1 The kernelized architecture](image-url)
in Fig. 3. (In this figure and in all the subsequent figures in the paper, the horizontal axis represents time and the vertical axis the security level.) The signaling channel, however, can simply be eliminated if a *high* transaction does not acquire a lock on *low* data elements, or relinquish the lock when a *low* transaction requests it. But, this may lead to the history shown in Fig. 4. This history is not *Serializable* because the corresponding serialization graph (shown in Fig. 5) contains a cycle (the Edge from T1 to T2 is due to the conflicting operations \(r_2[x]\) and \(w_1[y]\) where \(r_2[x]\) precedes \(w_1[y]\), and the edge from T2 to T1 is due to the conflicting operations \(w_1[y]\) and \(r_2[y]\) where \(w_1[y]\) precedes \(r_2[y]\)).

![Serialization Graph](image)

**Fig. 5.** The serialization graph corresponding the history in Fig. 4.

A signaling channel can be established in a similar manner even if a timestamp-ordering protocol is used instead. A timestamp-ordering protocol assigns a unique timestamp to every transaction and maintains two values for each data item: a read timestamp and a write timestamp. When a transaction issues a read (write) operation on a data item, the scheduler allows this operation only if the write (read) timestamp of the data item is not larger than the timestamp of the transaction; otherwise the scheduler rejects the operation. When a transaction reads (writes) a data item, the scheduler modifies the read (write) timestamp of the data item with the timestamp of the transaction. In our example, assume T1 and T2 are assigned uniques timestamps, \(ts(T1)\) and \(ts(T2)\), respectively. Let \(ts(T1) < ts(T2)\). In this case, T2 modifies the write timestamp of x to \(ts(T2)\). Since it is greater than \(ts(T1)\), T1’s write is rejected, thus establishing a signaling channel. In the rest of this article, we first discuss transaction processing in centralized multilevel secure databases. Then we examine how these solutions can be adapted for distributed MLS databases.

**V. TOWARDS A SOLUTION**

The problem of secure concurrency control has attracted the attention of many researchers since seventies. Lamport [26], Reed and Kanodia [38], and Schaefer [39] have provided solutions to secure readers/writers problem that employ
optimistic concurrency control techniques. Unfortunately, these concurrency control algorithms, although meet the secrecy requirements, do not satisfy correctness requirements when applied to transactions (since transactions contain read and write sets that are often related) [2]. Moreover, they suffer from starvation.

5.1 Secure Lock-Based Protocols
The signaling channels described in Section 4 can be eliminated if the high transactions are allowed to read low data items without acquiring a read lock. Although this approach meets the secrecy requirement, it does not meet the integrity requirement since all transactions are not two-phase. For example, in the history shown in Fig. 3, T1 will not be delayed since T2 does not acquire a read-lock on x. However, as can obviously be seen from Fig. 4, it does not ensure serializability. One way to circumvent both the problems mentioned above is as follows. When a high transaction wishes to read a low item, it first acquires a read lock on that low data item as in the traditional two-phase locking protocol, but should release this lock if a low transaction requests a write lock on the same data item. When a read lock by a high transaction is broken, the high transaction is to be aborted. Thus, this approach eliminates signaling channels since high transactions cannot in any way effect the execution of the low transaction, and at the same time guarantees serializability since it aborts all transactions violating two-phase property. If we revisit our example in Fig. 3, under this approach, T2 releases its read-lock on x since the lower level transaction T1 has requested for a conflicting lock on the same item x. Since it has prematurely released its lock, T2 aborts. Although this approach meets both secrecy and integrity requirements, under this approach, a malicious low transaction may cause a high transaction to be aborted repeatedly, resulting in starvation. In [33], McDermott and Jajodia provide a way to reduce the amount of starvation. According to their approach, whenever a high transaction prematurely releases its read lock on a low data item due to security reasons, it does not abort and rollback entirely, but holds its write locks on high data items, marks the low data item in its private workspace as unread and retries reading this data item by entering into a queue. This queue maintains the list of all high transactions waiting for retrial to read that particular data item and enables the first transaction in the queue to be serviced first. The modified approach, however, does not always produce serializable schedules [20]. Recently, Jajodia et al. [20] proposed two secure locking protocols that attempt to detect all cycles in the serialization graph by “painting” certain transactions and the data items accessed by the high transactions whose low locks are broken and by detecting a cycle at the moment it is imminent in the serialization graph. The first protocol produces pair wise Serializable histories (see Section 5.3.3), while the second protocol produces serializable histories if the security levels form a total order. It should be noted that the aforementioned lock-based protocols use a common lock table for all security classes; thus, it is not possible to implement them with completely untrusted code.

5.2 Secure Timestamp-Based Protocols
Since locking has been found to be unsuitable for concurrency control in multilevel secure databases, researchers have experimented with other techniques, such as time stamping. However, the problem with the secure timestamp ordering protocol is that a high transaction cannot modify the read timestamp after reading a low data item (which is necessary to guarantee serializability) since doing so amounts to writing down. Therefore, the challenge is to devise a protocol that does not require such write-downs yet ensures correct concurrent executions of transactions. Ammann and Jajodia [2] have proposed two secure timestamp-based algorithms that meet all the secrecy and integrity requirements; however, they are prone to starvation. One of the solutions to eliminate starvation is to maintain, instead of single version, multiple versions of data. The contention to the same data items by high and low transactions can be resolved by maintaining multiple versions of data where high transactions are given older versions of data. The usual notion of correctness for concurrent execution of transactions when multiple versions are maintained in the database is said to be correct when its effect is equivalent to that of a serial execution of the same transactions on a one-copy database, which is called one copy serializability [8]. The significant efforts based on multiversion timestamp-ordering technique are due to Main one and Greenberg [31], Keefe and Tsai [25], and Jajodia and Atluri [18]. Although these three protocols meet secrecy and availability requirements, Main one and Greenberg’s protocol does not meet the integrity requirement (i.e., it does not generate one-copy serializable histories). We defer the discussion of the first protocol [31] until Section 5.3 and present the latter two [18], [25] in this section.

5.2.1 Keefe-Tsai’s Scheduler
This scheduler uses a timestamp-based protocol, which is similar to the conventional multiversion timestamp ordering protocol, but differs in the way it assigns the timestamps to every transaction. It uses a priority queue, and transactions are placed in this queue based on the security level of the transaction: A transaction is assigned a timestamp that is smaller than all other 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. For example, consider the history in Fig. 3 once again. Keefe-
Tsai’s scheduler assigns a lower timestamp to T2 even though it arrives later than T1. Therefore, T2’s r2[x] will not invalidate T1’s w2[x]. This scheduler meets the first three requirements presented in Section 2: It is secure, it guarantees one-copy serializability, and it eliminates starvation; however, it uses a multilevel scheduler which, therefore, has to be trusted.

5.2.2 Jajodia-Atluri’s Scheduler
This scheduler, like Keefe-Tsai’s, is based on the multiversion time stamping protocol, but differs from it in many respects. It assigns timestamps to transactions in the order of their arrival and thus transactions are not given unusually old versions of data. However, according to this protocol, transactions are sometimes made to wait for the commit: Whenever a transaction reads from a lower level, it is not allowed to commit until all the transactions from that lower level with smaller timestamps commit. This is because a lower level transaction with a smaller timestamp can potentially invalidate the read operation of the higher level transaction by issuing a write. This protocol uses single-level schedulers and therefore is secure and is also implementable with untrusted code. It guarantees one-copy serializability. Though transactions are made to wait for commit, this protocol is free of starvation since this wait is finite. Thus, it meets all the four MLS requirements presented in Section 2.

5.3 An Alternative Approach: Relaxing the Integrity Requirement
A solution to the problem of secure concurrency control can be obtained by viewing it from a different perspective. One can argue that the traditional notion of correctness is too restrictive for multilevel secure databases [18], [31]. This is supported by the fact that the integrity constraints in multilevel databases are different from those in conventional databases. Enforcing integrity constraints in MLS databases is difficult or even impossible, especially those constraints that are defined over data at different security levels [34]. Since one cannot enforce the integrity constraints, there is no reason to insist on preserving serializability for these systems. Thus, serializability requirements can be relaxed for MLS systems. In [18], Jajodia and Atluri have proposed three alternative notions of correctness, namely, level wise serializability; one-item read serializability, and pair wise serializability, for multilevel multiversion databases. These weaker notions of correctness can be used as alternatives for one-copy serializability. They exploit the nature of integrity constraints in MLS databases to improve the amount of concurrency. Additionally, pairwise serializability is useful in cases where there exist only two security levels. In such cases, the effect of ensuring pair wise serializability is equivalent to that of ensuring serializability. This distinction is important because the algorithms to provide pairwise serializability (presented in Section 5.3.3) are less expensive compared to those providing the usual serializability and, therefore, for an MLS system that has just two levels, there is no reason to use an expensive algorithm that provides serializability.

5.3.1 Levelwise Serializability
This is the weakest form of correctness among all the three. This criterion may be used in situations where each integrity constraint is defined over data belonging to only one security level. Informally, levelwise serializability preserves the following two properties:

1) It ensures correct execution of transactions if transactions executing at a single security level alone are considered, and
2) It gives consistent data whenever a transaction reads from a lower security level.

What we mean by consistent data is explained with the help of an example. Consider a corporate database that contains several types of information about its employees: social security number, name, address, rank, years of service, regular hours worked, overtime hours worked, hourly rate, weekly pay, and yearly bonus. The information about hourly rate, weekly pay, and yearly bonus is considered sensitive, and is classified high, while the rest of the information is classified low. At the end of each week, weekly payroll is processed: First, a low transaction is processed that updates the hours accumulated by each employee, followed by a high transaction that computes each employee’s weekly pay. Obviously, when this high transaction is executed it is important that it is given a consistent copy of the low data. It would be undesirable to give the high transaction the regular hours worked by an employee during one week, but overtime hours for some other week. The second requirement has been added to prevent this type of anomalies. It is easy to build schedulers that satisfy levelwise serializability requirements. In fact, the concurrency control protocol that has been implemented in Trusted Oracle [31] generates levelwise serializable histories. This protocol:

- uses a combination of locking and timestamping techniques,
- employs strict two-phase locking for scheduling the operations of transactions accessing data items at their own level, and
- maintains multiple versions of data for those transactions that read lower level data items.
Whenever a transaction updates a data item, a new version is created. Every version of the data item is associated with a timestamp, which equals the time at which the transaction that has written that version commits. Every transaction is assigned a timestamp to select an appropriate version of the lower level data item; the version selected is the version with the largest timestamp, yet smaller than the timestamp of the transaction. In [6], Atluri, Jajodia, and Bertino propose a different protocol that exploits the presence of multiple versions of the data. This algorithm, unlike that of Trusted Oracle:

- Uses multiversion timestamping ordering protocol for scheduling the operations of transactions accessing data items at their own level, and therefore, offer better concurrency, and
- Selects an appropriate committed version of the lower data items by computing a read consistency point for each transaction, which is the minimum of timestamps of all active transactions at that level.

The version given to a transaction reading lower level items is the latest version whose timestamp is smaller than its read consistency point. Since levelwise serializability provides a weak form of correctness, it suffers from inconsistent retrieval problems, especially when a transaction attempts to read data from more than one security level. Consider once again the corporate database of the above example. Suppose some high transactions are executed from time to time that require some low information about an employee in addition to some high information. Under levelwise serializability, there is no guarantee that these high transactions will see a consistent view of the database.

As an example, consider the following high transaction T1: T1 reads the number of hours worked and the total pay for an employee. It is reasonable to expect that whenever T1 is executed, the values returned for an employee will satisfy the following constraint:

\[(\text{Number of hours worked}) \times (\text{Hourly rate}) = \text{Total pay}\]

Since this integrity constraint involves data at two levels, there is no guarantee with just levelwise serializability that T1 will be provided with a consistent view. It is possible that T1 will be given for an employee the hours worked for one week, but the total pay for a different week. We elaborate on this next. For an employee, let n denote the number of hours worked during a week, p the total pay. Also assume that the database maintains for each employee the raise in benefits, denoted by b. Thus, u, p, and b are high items, and n is a low item. Assume that a low transaction T2 updates the hours worked by an employee, and another high transaction T3 reads the hours worked and hourly rate and writes the weekly pay, and as mentioned above, T1 reads the hours worked and weekly pay for an employee and computes the raise in benefits. Assume that the database is initialized with versions n0, u0, p0, and b0. Consider the following scenario: Suppose the low transaction T2 arrives first, followed by the high transaction T1. T1 issues the read request for the low item n and is given the version n0. Before it can issue the next instruction, T1 is suspended. T2 completes the execution, at which point the high transaction T3 arrives and starts execution. Once T3 completes, T1 has a chance to complete as well, yielding the multiversion history in which the high transaction T1 reads n0 which is the hours worked by an employee during the previous week, while it reads p1 which is the pay for the current week. Although this history is levelwise serializable, it is clearly not desirable. We need a stricter correctness criterion to prevent such anomalies.

5.3.2 One-Item Read Serializability

In addition to levelwise serializability, one-item read serializability additionally guarantees the following property:

Whenever two transactions of the same security level read a data item from a lower level, each of them is given an appropriate version of that data, i.e., the version given to the later transaction (later in the serialization order) never precedes the version given to the earlier transaction.

To see how one-item read serializability avoids the anomaly described in the history presented in the previous section, notice that T3 precedes T1 in the history and version n0 precedes n2. Therefore, it will not be the case that T1 is given an earlier version n0 while T3 is given a latter version n2 of the low item n. Returning to the corporate database example, suppose at the end of each year, each employee’s rank is evaluated, and each employee is given an yearly bonus based on the performance during that year. Also, at the end of each year, every employee’s hourly rate is updated based on the year of service and the yearly bonus, for the future year. For an employee, let x and z denote the years of service and rank, respectively, and let y denote the yearly bonus. Thus, x and z are classified low, while y is classified high. Suppose that the database has been initialized with versions x0, y0, and z0, respectively. Assume T2 is a low transaction that updates the years of service and the rank for an employee, T3 is a high transaction that updates the yearly bonus based on the years of service and the bonus for the previous year, and T1 is a high transaction that updates the hourly rate of an employee based on the years of service and the yearly bonus. Suppose that the high transaction T1 arrives first, issues the read request for the low item x and is given the version x0. Before it can issue the next instruction, T1 is suspended. Later the low transaction T2 arrives which completes the execution by updating the low items x and z,
thus creating versions \( \mathcal{x}_2 \) and \( \mathcal{z}_2 \). At this point, the high transaction \( T_3 \) arrives and creates a new version \( \mathcal{y}_3 \) of \( \mathcal{y} \) by reading \( \mathcal{z}_2 \) and \( \mathcal{y}_0 \). After the execution of \( T_3 \), \( T_1 \) has a chance to complete as well. Therefore, \( T_1 \) issues its read operation to read \( \mathcal{y} \). Although the resulting multiversion history is one-item read serializable, it suffers from a yet another inconsistent retrieval problem: The high transaction \( T_1 \) reads \( \mathcal{x}_0 \), the years of service for the previous year, and \( \mathcal{y}_3 \), the bonus for the current year. This is surely not desirable in this case. The next correctness criterion is designed to correct this anomaly.

5.3.3 Pairwise Serializability

This is a stricter form of correctness than those provided by both levelwise serializability and one-item read serializability. It ensures correct execution of transactions at any two levels. In other words, it ensures the following property:

The combined history of any two security levels is one-copy Serializable

Both the protocols described in Section 5.3.1 that guarantee levelwise serializability can be modified to generate pairwise serializable histories. Jajodia and Atluri [18] proposed a modified version of the Trusted Oracle protocol. The only modification is that this protocol:

- Uses both timestamp ordering and two-phase locking to schedule the operations of transactions accessing data items at their own level.

The levelwise serializability algorithm can also be modified to produce pairwise serializable histories. It differs in the way in which it computes the read consistency point of a transaction:

- The read consistency point of a transaction \( T_i \) is determined in such a way that it is never larger than the read consistency point of any transaction \( T_j \) at the level of \( T_i \) whose timestamp is larger than that of \( T_i \).

Pairwise serializability does not guarantee one-copy serializability if transactions read from more than two security levels. The following example demonstrates the inconsistencies encountered with pairwise serializability.

Returning once again to the corporate database, suppose that there is yet another security level, called very high. At this level, data about future corporate plans are kept. At this level typically read data at two lower levels and make some projections based on the data that is read. Pairwise serializability cannot guarantee that transactions at very high level will see a consistent database state if they read data from both the lower two security levels.

Consider the following three transactions: Suppose a high transaction \( T_1 \) wishes to update the high data item \( x \) by first reading low data item \( y \) and then reading \( x \), a low transaction \( T_2 \) simply updates \( y \), and a very high transaction \( T_3 \) that reads both \( y \) and \( x \). Consider the following concurrent execution of these three transactions. Suppose first \( T_1 \) arrives and issues read operations on \( y \) and \( x \). Before it issues the write operation on \( x \), it gets suspended. Meanwhile, \( T_2 \) arrives and finishes its execution and thus creates a new version \( \mathcal{y}_2 \) of \( y \). As soon as \( T_2 \) terminates, \( T_3 \) arrives and is fortunate enough to successfully read both \( y \) and \( x \). At this point of time, \( T_1 \) wakes up and writes \( x \) thereby creating \( \mathcal{x}_1 \). Notice that, by the time \( T_3 \) issues both its read operations, only \( \mathcal{y}_2 \) and \( \mathcal{x}_0 \) are available. Thus, \( T_3 \) observes inconsistent data since it is given a new version \( \mathcal{y}_2 \) but an old version \( \mathcal{x}_0 \). One-copy serializability eliminates such anomalies.

VI. SECURE DISTRIBUTED TRANSACTION PROCESSING

Until now, we have confined the discussion of secure transaction processing to centralized multilevel secure databases. In this section, we turn our attention to transaction processing in distributed MLS databases. A distributed database (DDB) consists of several logical objects that are physically located at different sites (or nodes). Each site consists of an independent processor connected via communication links to other sites. Transactions executing in these systems may require to access (either update or retrieve) data objects from more than one site. The site at which a transaction originates is usually referred to as the coordinator and other sites participating in the execution are called subordinate sites.

To guarantee correct executions of local and distributed transactions, each site in the DDB is equipped with a concurrency control protocol and an atomic commit protocol. An atomic commit protocol is required to ensure that either all components of a distributed transaction (often referred to as sub transactions or subordinates) commit or they all abort. A concurrency control protocol is necessary at each site to ensure that concurrent execution of all transactions (distributed as well as local) at that site is serializable. Therefore, in a DDB, the integration of the concurrency control and the atomic commit protocols must satisfy the four requirements stated in Section 2.

Two-phase commit protocol (2PC), one of the most popular atomic commit protocols, consists of the following two phases: a prepare phase during which the coordinator of a distributed transaction (the site at which the distributed transaction originates) collects the votes of all the participants (sites participating in the execution of the distributed
transaction) that are willing to commit the transaction, and a decision phase during which the coordinator reviews the votes and decides whether to commit or abort the transaction. To commit a transaction, 2PC requires a total of \( 6n \) messages where \( n \) is the number of participants of the distributed transaction: \( n \) messages for the distribution of subtransactions, \( n \) done responses from each participant, \( n \) prepare messages to the participants, \( n \) yes/no votes from the participants, \( n \) commit/abort messages to the participants, and \( n \) acknowledgments from the participants.

Because of its high communication cost, often several variations of 2PC including presumed abort (PA), presumed commit (PC), and early prepare (EP) are used instead of 2PC itself [36], [40]. While PA and PC avoid the last round of messages by eliminating the acknowledgment and therefore require approximately \( 5n \) messages, EP combines the distribution of the subtransactions with the prepare phase and thus requires only \( 4n \) messages. Most commercial systems (e.g., IBM’s LU6.2 and Digital’s DECdtm), in fact, use EP as the atomic commit protocol.

6.1 Lock-Based Protocols

Secure two-phase locking (S2PL) and commit protocols that guarantee serializability have been studied in [21], [22]. Jajodia and McCollum [21] have modified 2PL to give a secure 2PL protocol (S2PL) that yields serializable schedules. Every transaction in S2PL must satisfy the following four properties:

1) **Well-formed**: A transaction must obtain a shared lock (S-lock) before reading a data item and an exclusive lock (X-lock) before writing a data item.

2) **Two-phase**: A transaction cannot request additional locks once it has issued an unlock action.

3) **Strict**: A local transaction holds on to all its locks until it completes. A subtransaction, on the other hand, may release any S-locks once it enters a prepared state. The subtransaction keeps all X-locks until it receives a commit or abort decision from its coordinator.

4) **Signaling channel free**: A high transaction must release its S-lock on a low data item when a low transaction requests an X-lock on the same data item. In such an event, the high transaction can either abort or start over.

In [22], Jajodia, McCollum, and Blause Stein describe how 2PC, EP, and PA can be modified to be secure. They show that the integration of S2PL with either PA or the conventional 2PC satisfies all the MLS requirements, but integration of S2PL with EP does not satisfy requirement 2. This is because integration of EP and S2PL cannot guarantee that the distributed transaction is always two-phase.

Later, Atluri et al. [5] showed how a secure concurrency control protocol can be integrated with EP so that the integration meets all the requirements by proposing a secure locking protocol SLP (which is similar to S2PL in [22]) and a secure analog of EP (called SEP). SLP provides different degrees—0, 1, 2, and 3—of isolation. While SLP satisfies all the requirements for degrees 0, 1, and 2 isolation, it is susceptible to starvation for degree 3 isolation.

The steps of SEP are as follows: Assume \( Ti \) is decomposed into \( n \) subtransactions, and \( Ti,j \), the subtransaction at the originating node \( Sj \), is one among them. In the prepare phase, the coordinator generates subtransactions \( Ti,1, Ti,2, ..., Ti,n \) and sends them to the participating sites \( S1, S2, ..., Sn \), respectively. The coordinator also sends the security level \( s \) with each subtransaction. At each participant \( Sk, Ti,k \) must first acquire an S-Lock (X-lock) on an item before it is read (written). S-locks as well as X-locks are long locks. If \( Ti,k \) completes successfully, the participant augments its yes vote to the coordinator with a read-low indicator. A one-bit read-low indicator is added whenever \( Ti,k \) has read an item from a lower level. Otherwise, it sends a no vote. During the decision phase, if the coordinator receives yes votes from all the participants and if no subtransaction has read data from lower levels, it commits \( Ti \) and then sends commit messages to all its participants. An extra round of messages is required between the coordinator and all those participants \( Sj \) that sent the readlow indicator with their yes vote as follows: The coordinator first sends to each \( Sj \) a confirm message to confirm the commit. If \( Sj \) has not released its S-lock on any lower level data item during the time it has been in the prepared state, it responds with a confirmed message; otherwise, it sends a not-confirmed message. If the coordinator receives a confirmed message from all \( Sj \)’s to which the coordinator has sent the additional round of messages, then it sends commit messages to all its participants; otherwise it sends abort message. On the other hand, if the coordinator receives at least one no vote or if it times out waiting for a vote, it aborts the transaction, and sends abort messages to all participants.

SEP has two drawbacks. First, if we compare the number of messages required by SEP to that required by EP, EP always requires about \( 4n \) messages (where \( n \) is the number of participants), while SEP sometimes requires more than \( 4n \) messages. An extra round of messages is required between the coordinator and those nodes where the subtransactions have read data from the lower levels. Second, SEP is overly pessimistic. Any high subtransaction

3. Most commercial systems that use lock based mechanisms for concurrency control do not always use 2PL but use less restrictive protocols that hold locks for shorter duration to increase the amount of concurrency. While degree 3
isolation is same as serializability, degrees 0, 1, and 2 isolation offer lower levels of correctness than serializability. The details on different degrees of isolation can be found in [16], [17].

That reads low data is aborted if any of its S-locks on the low data are broken while it waits for the confirm message from the coordinator. Thus, SEP aborts a transaction if there is a possibility of a violation of the two-phase requirement. It is entirely possible that the transaction is two-phase, even though some of the S-locks are broken. Atluri, Bertino, and Jajodia [5] have proposed an optimization to SEP, called O3SEP. O3SEP assumes that the clocks in the distributed system are synchronized. During the prepare phase, the coordinator first computes the latest time at which locks by all subtransactions have been acquired, and it sends an additional round of messages to only those participants who has voted before this latest lock acquisition time. During the decision phase, the coordinator computes the earliest time at which any lock has been released by any subtransaction, and it aborts the transaction only if this earliest lock release time is less than the latest lock acquisition time. Thus, O3SEP reduces both the number of messages and the number of transactions being aborted.

Recently, Ray et al. [37] proposed an advanced secure commit protocol (ASEP), a modification of SEP. Like SEP, ASEP sends an additional round of messages to all subtransactions which have read low data items. However, in ASEP, subtransactions can roll back to a prespecified save point and reexecute the subtransaction (as in [9]) if they release their read locks on low data items. If a subtransaction successfully completes, then it sends a yes message. ASEP checks this by sending messages to these participants repeatedly until it receives all yes responses. However, if any of the participants sends a no message, the coordinator aborts the transaction. ASEP additionally offers a user to choose various levels of consistency for achieving atomicity by allowing them to specify several language primitives. When a high subtransaction is forced to relinquish its locks on a low data item, the user may choose:

1) to abort the high subtransaction (which may result in the abort of the entire transaction),
2) to perform a partial roll-back of the high subtransaction so that the entire transaction need not abort, or
3) to conduct forward recovery which may trade-off consistency.

6.2 Timestamp-Based Protocols

No prior work has been reported that extends the secure timestamp-based protocols to a DDB environment. In this section, we compare the applicability aspects of the two secure schedulers Keefe-Tsai’s and Jajodia-Atluri’s—in an MLS DDB.

The correctness of a timestamp-based protocol in a distributed environment relies on synchronized clocks. For the distributed timestamp-ordering protocol to work correctly, every transaction, either distributed or local, must be assigned an unique timestamp. Unlike distributed two-phase locking schedulers that require coordination among sites to handle distributed deadlocks, the decision to schedule or reject an operation depends solely on the information maintained by local schedulers at each site.

Clocks can be synchronized using either the technique described by Lampport [27] or by using time services such as network time protocol [35] or Digital time service [13]. Liskov [29] suggests that it is possible to use these time services to have distributed clocks that are synchronized within a millisecond, “even after extended periods when synchronization to primary reference sources has been lost” [35]. To guarantee globally unique timestamps, a two component timestamp may be used: the first component is the timestamp assigned by reading from a globally synchronized clock and the second component is the site identifier that is unique and orders all the sites in the DDB.

Suppose every site in the MLS DDB uses Keefe-Tsai’s scheduler for concurrency control. According to this protocol, every transaction, distributed or local, must be assigned a timestamp such that it is smaller than the timestamps of all active transactions executing at all lower levels. Suppose that there are four sites S1, S2, S3, and S4 in a DDB. Let Ti be a distributed transaction that originates at site S1 such that Ti requires subtransactions to be executed at sites S2 and S3. As per Keefe-Tsai’s protocol, Ti should be assigned a timestamp that is smaller than all active transactions at all sites, S1, S2, S3, and S4. Otherwise, it is possible that a lower level transaction with a smaller timestamp at some other site may issue write operations that can invalidate the read operations of Ti. Therefore, although determining the timestamp of a transaction may not cause any additional problems for a local transaction, it may require additional communications between the coordinator and the participants.

Note that even though Ti has subtransactions at S2 and S3 only, determining Ti’s timestamp requires exchange of messages between S1 and all other sites in the DDB since it must be smaller than minimum of all active transactions at all four sites.

On the other hand, suppose each site in the MLS DDB employs Jajodia-Atluri’s scheduler for concurrency control. Since, according to this protocol, timestamps to transactions are assigned in a conventional timestamp-ordering protocol (that is a transaction that arrives earlier is given a smaller timestamp), it does not require any additional message exchanges among the sites participating in the execution of the distributed transaction. If we consider the earlier example
once again, the coordinator $S_i$ can simply assign a timestamp to $Ti$ (note that we assume synchronized clocks) and send the timestamp of $Ti$ along with the subtransactions of $Ti$ to $S_2$, $S_3$, and $S_4$. However, Jajodia-Atluri’s scheduler requires $Ti$ to wait for its commit until all transactions at all sites with smaller timestamps than that of $Ti$ commit. This is because, the purpose of this waiting is to make sure that no transaction that has arrived earlier than $Ti.j$ writes a data item that has been previously read by $Ti.j$. Since extending both Keefe-Tsai’s and Jajodia-Atluri’s schedulers to distributed environment results in unsatisfactory solutions, more investigation is required to address this issue.

VII. CONCLUSIONS

This article has elaborated on the research efforts of several researchers to develop a secure concurrency protocol for multilevel secure database management systems based on the kernelized architecture. This problem is of considerable interest since the traditional concurrency control protocols such as two-phase locking and timestamp ordering cannot satisfy the MLS requirements. Although researchers made modifications to the two-phase locking protocol to make it secure, as seen in this article, this protocol suffers from starvation. Even the timestamp-ordering protocols suffer from the same problem when modified to meet the secrecy requirements. Therefore, maintaining multiple versions of data seems to be unavoidable for MLS databases. The other solution discussed in this article is to relax the traditional correctness requirements. It is reasonable to require weaker notions of correctness for these systems since it is not possible to enforce all the MLS integrity constraints.

ACKNOWLEDGMENTS

The work of Vijayalakshmi Atluri was partially supported by the National Science Foundation under Grant No. IRI-9624222 and by the National Security Agency under Grant No. MDA904-96-1-0127. The work of Sushil Jajodia was partially supported by the National Science Foundation under Grant No. INT-9412507, and by the National Security Agency under Grant No. MDA904-96-1-0103 and Grant No. MDA904-96-1-0104.

REFERENCES

Vishwakarma, International Journal of Advanced Research in Computer Science and Software Engineering 8(2)
ISSN(E): 2277-128X, ISSN(P): 2277-6451, pp. 87-100


