Database Concurrency Control in Multilevel Secure Database Management Systems

Thomas F. Keefe, Member, IEEE, W. T. Tsai, Member, IEEE, and Jaideep Srivastava

Abstract—Transactions are vital for database management systems (DBMSs) because they provide transparency to concurrency and failure. Concurrent execution of transactions may lead to contention for access to data, which in a multilevel secure DBMS (MLS/DBMS) may lead to insecurity. In this paper we examine security issues involved in database concurrency control for MLS/DBMSs and show how a scheduler can affect security. We introduce Data Conflict Security, (DC-Security) a property that implies a system is free of covert channels due to contention for access to data. We present a definition of DC-Security based on noninterference. Two properties that constitute a necessary condition for DC-Security are introduced along with two other simpler necessary conditions. We have identified a class of schedulers we call Output-State-Equivalent for which another criterion implies DC-Security. The criterion considers separately the behavior of the scheduler in response to those inputs that cause rollback and those that do not. We characterize the security properties of several existing scheduling protocols and find them to be insecure.

Index Terms—Covert channel analysis, database concurrency control, multilevel secure database management systems, multilevel security, multiversion concurrency control, noninterference.

I. INTRODUCTION

COMPUTER security addresses the need for controlled access to the information in a computer system. Multilevel secure (MLS) computer systems provide strong mechanisms for controlling the disclosure of sensitive information [8]. Database management systems (DBMS's) [29] support efficient access to large amounts of shared data. They are important because they allow for centralized control of an enterprise's operational data [5]. MLS database management systems (MLS/DBMSs) apply the access controls of multilevel secure computers to database management systems [13], [14], [16], [22].

Transactions are commonly used to support concurrent access to shared data as they simplify database interaction by providing transparency to concurrency and to failure. Concurrent execution of transactions may lead to contention among subjects for access to data. In MLS/DBMSs this contention may lead to security problems.

In this paper we attempt to characterize the relationship between security and scheduling requirements. We apply noninterference [9] to an abstract model of a transaction scheduler. The resulting security requirement we call DC-Security. We study the implications of DC-Security by developing several necessary and sufficient conditions.

Following this, we study the relationship between security and scheduling requirements. We characterize several existing scheduling protocols. Every protocol considered produces serializable schedules and thus meets our scheduling requirements. We use the necessary and sufficient conditions to determine the DC-Security of the protocols.

The distribution of a database over a network of computers complicates many issues in the design of a DBMS. The effect of distribution on the security of a system is unclear [24], [31]. We consider here only centralized DBMS's.

A. Multilevel Security

Here we consider systems that enforce a military security policy [6]. In this policy, elements of data are assigned a sensitivity level. This level reflects the value of the information and the care that must be taken to avoid its disclosure. Users are assigned clearance levels that reflect their perceived level of trustworthiness. A multilevel secure system is one that can be shared by users from more than one clearance level and contains data of more than one sensitivity level.

There are two parts to the military security policy, mandatory and discretionary security. Mandatory security controls the flow of information based on the trustworthiness of the individual, while discretionary security controls the flow of information based on the individual's need-to-know.

1) Security Policy: Mandatory security is based on security classifications. These classification levels are used to arbitrate access to data elements by users as we describe below. Discretionary security is based on the identity of the user. Associated with each element of data is a list of one or more user names and the allowed modes of access for each one. Access to the data element is arbitrated based on this information. In this work we consider only mandatory security.

The set of security classifications is partially ordered by the $\geq$ relation. When two security classifications $l_1$
and \( l_1 \) satisfy \( l_1 \succeq l_2 \) we say \( l_1 \) dominates \( l_2 \). If \( l_1 \succeq l_2 \) and \( l_1 \neq l_2 \) we say that \( l_1 \) strictly dominates \( l_2 \), or \( l_1 > l_2 \).

Military security classifications consist of two parts: a hierarchical level and a set of nonhierarchical categories. The set of hierarchical levels is totally ordered by the \( \succeq_h \) relation. In examples, we will use the following hierarchical levels ordered as follows:

\[
\text{Top Secret} \succeq_h \text{Secret} \succeq_h \text{Confidential} \succeq_h \text{Unclassified}.
\]

The second part of a military security classification is the category set. Each category set is a subset of the set of nonhierarchical categories. For the set of categories \{Crypto, Comsec\} we have the following possible category sets: \{\}, \{Crypto\}, \{Comsec\} and \{Crypto, Comsec\}. Category sets are partially ordered by set inclusion, i.e., \( \supseteq \).

Let \( H(l) \) denote the hierarchical classification and \( CS(l) \) denote the category set of a security classification \( l \). The partial order \( \succeq \) of the elements in \( SC \) is defined as follows. For \( l_1 \) and \( l_2 \) that are members of \( SC \), \( l_1 \succeq l_2 \) iff:

1) \( H(l_1) \succeq_h H(l_2) \);

and

2) \( CS(l_1) \supseteq CS(l_2) \).

Consider the following examples. The classification Secret \{Crypto\} strictly dominates the classification Confidential \{Crypto\} because the hierarchical level Secret is greater than Confidential and they share the same category set. The classification Unclassified \{Crypto, Comsec\} dominates Unclassified \{Crypto\} because the hierarchical level of the first is at least as great as the second and the first category set contains all the elements in the second category set. Consider the two classifications Confidential \{Crypto\} and Unclassified \{Comsec\}. Even though Confidential is greater than Unclassified, neither category set includes the other and therefore the classifications are incomparable.

Elements of information in the system are assigned security classifications called sensitivity levels that represent their sensitivity. Users are assigned security classifications called clearance levels that represent the levels to which they are trusted. The military security policy requires that a user’s clearance level dominates the sensitivity levels of information the user is allowed to view.

The policy does not deal directly with the modification or creation of information. There is an implicit assumption that the modification or creation of an element of information with sensitivity level \( l \) only incorporates information with a sensitivity level dominated by \( l \).

A user is trusted to assign the sensitivity level of the data appropriately. A program acting on a user’s behalf is called a subject and is not trusted. It is assigned a security classification called a classification level. The classification level dominates the sensitivity levels of all information that the subject is allowed to view. The classification level is dominated by the sensitivity levels of all information that the subject creates or modifies.

2) Security Models: A security model is an abstract model of how a secure system enforces the security policy. One popular model was developed by Bell and LaPadula [1]. The model defines two entities, subjects and objects. Subjects represent active elements in the system (e.g., processes, users). A subject is assigned a security classification level, which we refer to as its classification level. Objects represent passive data (e.g., relations, tuples). An object is assigned a security classification level, which is called its sensitivity level. Information is allowed to flow from an object (subject) with security classification level \( l_1 \) to a subject (object) with classification level \( l_2 \) only if \( l_2 \succeq l_1 \).

The “Orange book” [6] defines several evaluation classes for secure systems representing varying levels of assurance. The evaluation class B1 requires the enforcement of a mandatory access control (MAC) policy. This requires that the system satisfy the following properties.

Simple Security Condition

A subject may have read access to an object only if the subject’s classification level dominates the object’s sensitivity level.

*-Property (Star Property)

A subject may have write access to an object only if the object’s sensitivity level dominates the subject’s classification level.

The simple security condition and *-property are described in [1].

We must also consider information flow through covert channels. A covert channel allows information to be transferred in violation of the security policy (i.e., either from a high-level subject to a low-level subject or between two subjects with incomparable security levels). Covert channels are associated with a shared resource and can be categorized as either storage or timing channels. Timing channels require that a subject or user be able to measure time, while storage channels do not.

We will now give an example of a storage channel. Consider a secure operating system that insists that a set of file names be unique. A file named “SDI” exists and is classified Secret. The mandatory access control does not allow an Unclassified subject to see “SDI” in the directory catalog. If the subject attempts to create a new file named “SDI” the request is denied. Through this denial, the subject learns of the existence of some file named “SDI” at an invisible level. A cooperating Secret subject can remove and create “SDI” to signal information to the Unclassified subject. This storage channel is a manifestation of the multiparty update conflict [17].

In contrast to storage channels, a timing channel signals information by modulating an observable delay. A timing channel depends on a resource shared between subjects with different security classification levels, such as a processor or a disk drive and requires the ability to measure time. Consider Unclassified and Secret subjects sharing
a single disk drive. By modulating the rate of disk accesses, the Secret subject can delay the Unclassified subject’s computation, and transfer information to the Unclassified subject. The Secret user may perform many disk access to transmit a “one” and no disk access to transmit a “zero.” In this way it can communicate a string of binary digits. The receiving process measures the delay experienced by its disk requests. If the receiver cannot measure the delay, the channel disappears. A discussion of these issues can be found in [8]. The “Orange book” [6] requires the designers of systems evaluated at class B3 or above to conduct a thorough search for covert channels and make a determination (either by actual measurement or by engineering estimation) of the maximum bandwidth of each identified channel.

At higher levels of assurance, a limit is placed on the maximum bandwidth a covert channel may have.

The Trusted Computing Base (TCB) includes all the hardware, firmware, and software responsible for enforcing the security policy. Every trusted program is included in the TCB. It is important that the size of the TCB be as small as possible to allow an effective evaluation of its security.

B. Database Concurrency Control

This section considers the problem of scheduling several concurrent transactions. We adopt a scheduling model similar to that of Papadimitriou [26]. The purpose of a scheduler is to order the actions of several transactions in a way that maintains correctness and allows concurrency. We say a scheduler maintains correctness if every schedule it outputs is view serializable [26]. Two schedules s and s’ are view equivalent if:

1) they include the same transactions; and
2) if transaction T₁ reads the value of an element x written by Tᵢ in s, it does so in s’ as well; and
3) if transaction Tᵢ writes the final value of an element x in s, it does so in s’ as well.

We say a schedule is view serializable if it is view equivalent to some serial schedule. A serial schedule executes a set of transactions with no interleaving (i.e., each transaction runs to completion before the next one begins). A serial schedule limits the concurrency among transactions as each transaction must wait until all previous transactions have completed.

Other important properties of schedules include reliability and concurrency. A schedule is reliable if the failure of a transaction is transparent to other transactions in the schedule. A schedule is concurrent if it is an interleaved sequence of actions from two or more transactions. A serial scheduler (i.e., one that only outputs serial schedules) is correct by definition, but is inefficient.

C. The Threat to Security

One common transaction scheduling protocol is two-phase locking [2]. The idea is as follows. There is a lock corresponding to each element in the database. When a transaction T attempts to access an element x, the scheduler attempts to acquire the lock for x on behalf of T. If the lock is already held by another transaction, T is required to wait until the transaction holding the lock releases it. Otherwise, the lock is acquired for T and the access to x is carried out. When transaction T commits, the locks acquired for T are released.

This scheduling protocol exhibits a covert timing channel which we now briefly explain. Consider two subjects S₁ and S₂ with subject classification levels l₁ and l₂, respectively, such that l₁ > l₂. If S₁ acquires a lock to read element x, then S₂ will be required to wait until S₁ commits its transaction before it will be allowed to write element x. The subject S₁ can covertly signal S₂ by modulating the time it holds the lock on element x. The subject S₂ receives the message by measuring the delay it experiences when it attempts to access element x.

We can use this channel to send a sequence of m bits from S₁ to S₂ in the following way. The subject S₁ submits transaction T₁₀ and a selection of the transactions in T₁₁, · · · , T₁m. Specifically, S₁ submits a transaction T₁ₜ only if the ith bit of the message is “1.” The transaction T₁ₜ reads element xᵢ and commits. The subject S₂ submits one transaction T₁ that attempts to write elements x₀ through xₘ in sequence.

Synchronization is achieved as follows: the subject S₁ submits transaction T₁₀, which reads an element x₀, waits a suitably long time, and commits. The subject S₂ submits the first action of transaction T₂, a write of element x₀, sometime between the read and commit of transaction T₁₀. The write of x₀ by T₂ is delayed until after the commit step of T₁₀ is executed.

Each of the m bits in the message is sent as follows: to send a “1,” S₁ submits the transaction T₁ₜ that reads xₜ, waits for time tₜ, and commits. The time tₜ is chosen to allow S₂ to detect the delay experienced by its write of xₜ. To send a “0,” S₁ does nothing but wait for time tₜ. The time tₜ is chosen to approximate the duration of one of the reading transactions (T₁₅, · · · , T₁ₙ). In this case, the write of xₙ by T₂ will be executed without delay. Following the execution of its write of element xₙ, subject S₂ waits for time tₙ and issues a write of element xₙ₊₁. The time tₙ is chosen to maintain synchronization. Appropriate choices for tₙ, tₜ, and tₖ depend on several factors, including the precision with which real-time can be measured, storage access times, and recovery protocols.

We cannot predict the bandwidth of this channel accurately as we lack detailed information. However, we can give an approximate analysis. Let t_response be the time it takes to execute a transaction that writes one element and commits (i.e., from the time the transaction is submitted until it completes). Then, the time required to send a bit is roughly t_response (the time to send a “0” may be shorter),
and the bandwidth is approximately $1/t_{\text{response}}$. The bandwidth of this channel could be quite high in the case that no logging is performed for read-only transactions and when all of the elements being read and written are contained in buffers.

We can increase the effective bandwidth of the channel by operating several instances of the channel concurrently. Each pair of subjects can communicate through a disjoint set of elements. However, the effective bandwidth will be limited above by the throughput of the system (i.e., the total number of transactions executed per unit time). The bandwidth of the channel is related to a common measure of the performance of the system. Thus, the threat will become increasingly important as the performance of transaction processing systems increase.

The discussion above gives an example of the security threat that we address in this work. The rest of the paper is organized as follows: Section II describes our research methodology. Section III introduces data conflict security (DC-Security), which implies a system is free of covert channels due to contention for access to data. We present a definition of DC-Security based on noninterference and present several necessary and sufficient conditions. Section IV uses these necessary and sufficient conditions to analyze the security properties of several existing scheduling protocols. Section V discusses related work. Finally, Section VI concludes this paper.

II. RESEARCH METHODOLOGY

The goal of our research is to characterize the relationship between security and scheduling requirements. In analyzing the security properties of a transaction scheduler, we distinguish between:

1) the security properties of the transaction scheduling protocol; and
2) the security properties of a specific implementation of the protocol.

This paper considers only the first case. By focusing on the protocol, we can evaluate a scheduler's inherent security properties. This allows us to recognize insecure scheduling protocols early, before we devote effort to design. Once we determine that a scheduling protocol is secure, we can analyze the security of the implementation.

A scheduler for an MLS/DBMS must produce correct schedules and do so in a secure manner. We assume that subjects at multiple classification levels share a single scheduler. Consequently, we must verify that the scheduler does not violate security. Our assumption allows us to adopt well-known scheduling protocols for use in MLS/DBMSs and allows a simpler analysis of their correctness. One goal in the design of any secure system is to minimize the amount of trusted code and thereby the amount of verification required. Once we demonstrate the correctness and security of a particular scheduling protocol, we can attempt to limit the amount of the scheduler we need to trust. This may required further arguments that the changes maintain the correctness and security of the scheduler. Alternatively, if we show that the protocol is not secure or not correct, there is no reason to consider it further. This approach allows a simple and uniform model of a scheduler for a large class of scheduling protocols. We can also imagine scheduling being carried out by several untrusted schedulers, each one scheduling the actions of subjects at a single classification level. This is the approach taken in [4], [7], [15], [23], and [28]. Adopting this approach can simplify arguments about the security of scheduling. However, it may complicate descriptions of the protocols and reasoning about their correctness.

A central issue in multilevel secure concurrency control is covert channel analysis. This is a difficult problem for secure systems in general [3, 12]. The notion of noninterference [9] and the related MLS property [10], [11] provide a simple and intuitively satisfying definition of security that considers both access control and covert channels.

Noninterference treats a system as a black box. The input and output behavior determine the security of the system. Consider a sequence $p$ of actions from several subjects. For every classification level $l$ we construct a new sequence $p_l$ by removing all the actions in $p$ with subject classification levels not dominated by $l$. The MLS property requires that a subject with classification level $l$ not be able to distinguish the behavior of the system in response to $p$ from the behavior of the system in response to $p_l$.

It is not possible to discover all covert channels in a real system's scheduler through an analysis of the protocol as some appear only in the implementation. We limit our attention to the covert channels that exist when we remove the contention for the scheduler itself. We remove this contention from the views of the executing subjects by assuming the scheduler can respond to inputs instantaneously. We call this an instantaneous scheduler and say a scheduling protocol is DC-secure if it satisfies noninterference when executed by an instantaneous scheduler.

A. Multilevel Transactions

Subjects issue transactions to the transaction scheduler one action at a time. Each action reads or writes an object. The military security policy places some restriction on the type of transactions that a subject can issue. Individual security models may add additional requirements. One requirement that is often imposed allows a subject to write an object only if the sensitivity level of the object equals the subject's classification level. We call this class of security models Class 1. Models that in addition allow a write when the sensitivity level of the object strictly dominates the subject classification level of the writer we call Class 2.

A characteristic of a transaction is the number of subjects involved in its execution. If only one subject is involved we call it Single-Subject (SS), if more than one is involved, we call it Multiple-Subject (MS). [19] con-
tains a discussion of some of the security issues involved with MS transactions. We will focus here on the SS transaction model, i.e., Class 1-SS and Class 2-SS.

Below is an example of a multilevel transaction $T_1$:

$$ T_1: R(x, U, S) \ W(y, y, S) \ W(T_2, S, S). $$

Our notation is based on that of [26]. Here, $R(x, U, S)$ denotes a Read accessing an element $x$ with classification level $U$ (Unclassified) by a subject with a classification level $S$ (Secret). Similarly, $W(y, y, S)$ is a Write action. The action $T_1: W(T_2, S, S)$ is a special type of action that identifies the end of the transaction. Rather than complicate the model with another type of action, we represent this step with a write of some element that is only accessible to the committing transaction. We call this the commit step as it indicates the transaction’s readiness to commit. When the context is clear we will represent the commit step of transaction $T_1$ as $T_1: C$.

Transactions in Classes 1-SS and 2-SS contain actions from only one subject. For these kinds of transactions we can simplify the notation by associating the subject classification level with the transaction’s name as shown below:

$$ T_1(S): R(x, U) \ W(y, S) \ C. $$

The set of transactions generated by security models in Class 2-SS properly contains the set of Class 1-SS transactions. This is easy to see because the Class 2-SS models loosen the restriction on writing up but are otherwise just like Class 1-SS models.

B. Schedules

In scheduling transaction in a MLS/DBMS we must consider possible interference among subjects. The concurrency control mechanism provides a limited interaction among subjects of different classification levels. We are concerned with this kind of interference and its effect on the security of the system. For example, consider the access requests of two transactions as they arrive at the scheduler:

$$ p =\begin{align*}
T_1(S): & R(x, U) & R(x, U) & C \\
T_2(U): & W(x, U) & C.
\end{align*}$$

The transaction $T_1$ is unusual in that it reads the entity $x$ twice. No transaction needs to do this. However, it provides a simple example of an interesting scheduling problem we now explain. Transaction $T_1$ reads the entity $x$, transaction $T_2$ writes $x$, and finally $T_1$ reads $x$ again. As is, this sequence of actions suffers from non-repeatable reads, i.e., the value read by the two $R(x, U)$ actions in the same transaction can be different. Several correct schedules can be produced in response to this input. One possible schedule would delay the write of $x$ by transaction $T_2$ until $T_1$ has completed as shown below:

$$ s_1 = T_2(U): W(x, U) \ C.$$

The actions of the Secret transaction $T_1$ delays the write by the Unclassified transaction $T_2$. This constitutes a timing channel. The Secret subject may signal information to the Unclassified subject by modulating the duration of its transaction. Two other possible schedules $s_2$ and $s_3$ do not delay the low-level transaction. Now, we consider their correctness. The schedule $s_2$ shown below is correct.

$$ s_2 = T_2(U): W(x, U) \ C.$$

It represents the serial schedule $T_2T_1$. In this schedule, the action $T_1(S): R(x, U)$ arrives before any others but must follow $T_2(U): W(x, U)$ in the output. This is a reasonable action only if the scheduler makes decisions based on some portion of the input schedule that has not yet arrived. In most cases the complete schedule is unknown when scheduling begins, as an application program determines the actions it will submit dynamically. Thus, $s_2$ is not, in general, a realistic solution.

The next schedule we consider is equivalent to the scheduler input:

$$ s_3 = T_2(U): W(x, U) \ C.$$

This schedule is not serializable. 1 Transaction $T_1$ will read two different values of $x$, which would not be the case in any serial schedule.

C. Schedulers

We classify schedulers by the type of information they have about the input schedule. Papadimitriou [26] defines three modes of information acquisition: static, declaration, and dynamic. A static mode scheduler receives information about all transactions in the schedule, before scheduling begins. In the declaration mode, the first step of a transaction carries information describing the rest of the actions. A dynamic mode scheduler initially has no information about the schedule. The scheduler knows nothing about an action until it arrives for scheduling. We consider here only dynamic mode schedulers.

Multiversion schedulers allow more than one version of an element in the database. This reduces the contention for an element. A schedule produced by a multiversion scheduler is a full schedule $s, V$. The $s$ portion of the schedule represents the output actions. The $V$ portion is the version function. Let $s$ be the schedule $s$ with the transaction $A_0$ added at the beginning and a transaction $A_\infty$ at the end. Transaction $A_0$ consists of a write action $W(x, l_0, L)$ for each element $x$ read or written in $s$ and transaction $A_\infty$ consists of a read action $R(x, l_0, T)$ for each $x$ read or written in $s$. A version function maps each

1The schedule is multiversion view serializable [26]. To execute the view serializable schedule the system must maintain more than one version of $x$. The first read step of $T_1$ reads the original version of $x$. The step $T_2(U): W(x, U)$ creates and writes a new version of $x$. The second read step of $T_2$ then ignores the version written by $T_2$ and reads the original version. This schedule has the effect of the serial schedule $T_1T_2$.
read step in $s$ to a previous write step of $s$ and maps each write step of $s$ to a previous write step or one of the symbols $r$ and $l$, where $r$ means write a new version and $l$ means the write step is ignored. The schedule output by a single version scheduler is also a full schedule ($s$, $V$), where the actions in the schedule are represented by $r$ and $V$ is the standard version function. The standard version function refers each read and write of an element to the most recently written version. Thus, each read returns the last value written and each write overwrites the preceding one.

The notion of view equivalence can be extended for multiversion schedulers [26]. We say two full schedules are view equivalent if they have the same transactions and the two version functions agree on all read steps. A full schedule is view serializable if it is view equivalent to some full schedule ($s$, $V$) where $s$ is a serial schedule and $V$ is the standard version function.

Actions arrive for scheduling on the input ($I$). In response to each action input, the scheduler outputs a possibly empty sequence of triples on the output. The output triples are of the form ($A$, $V$, $R$), where $A$ is an action, $V$ is its associated version, and $R$ is a sequence of names of rolled back transactions. The domains of $A$, $V$, and $R$ are described below.

$$A: \text{ACTION} \cup \{\text{nil}\}$$

$$V: \text{ACTION} \cup \{r\} \cup \{l\} \cup \{\text{nil}\}$$

$$R: \text{SEQ(TRANSACTION)}.$$  

The domain ACTION includes actions of the form $T_i; A(x, \ell, l)$ where $A$ is either a read or write action. The domain TRANSACTION includes task names of the form $T_i$. Elements of the domain SEQ(TRANSACTION) are sequences of the elements of TRANSACTION.

We assume for simplicity that the scheduler does not both output an action and roll back a sequence of actions in the same output triple. Thus, for any output triple $t$, either $A(t) \neq \text{nil}$ or $R(t) \neq \langle \rangle$, but not both.

Each triple output by the scheduler at time $t$ represents the output of one action or the rollback of a sequence of transactions. If an action is output, it is either the action that arrived at time $t$ or some action that arrived earlier and was delayed. If a sequence of transactions is rolled back, each transaction in the sequence started before $t$.

We model a scheduler as a deterministic state machine. The scheduler begins prior to time $t_0$ in some initial state $s_{i0}$. At each time $t_i$ beginning with $t_0$ an action $T_j; A(x, \ell, l)$ arrives for scheduling. In response to the input, the scheduler outputs a sequence of zero or more output triples. The scheduler then enters a new state $s_{i+1} = \delta_i[st_i, T_j; A(x, \ell, l)].$

We refer to the sequence of output triples generated at time $t_i$ in response to the input schedule $p$ as $O_i(p)[t]$. We refer to the cumulative output between time $t_i$ and $t_j$ inclusively as $O_i(p)[t_i, t_j]$. The sequence of actions output from $t_i$ up to but not including $t_j$ is $O_i(p)[t_i, t_j]$. We denote the projections of $O_i(p)[t_i]$ as $A(O_i(p)[t_i]), V(O_i(p)[t_i])$ and $R(O_i(p)[t_i]).$

We will represent the sequence of output triples at time $t_i$ with all empty triples removed (i.e., a triple is empty if $A(t) = \text{nil}$, $V(t) = \text{nil}$, and $R(t) = \langle \rangle$) as $O_i(p)[t_i]$. Similarly, $A(O_i(p)[t_i]), V(O_i(p)[t_i])$ and $R(O_i(p)[t_i])$ are the projections of $O_i(p)[t_i]$ with nil actions, nil versions, and $\langle \rangle$ rollback sets, respectively, removed.

To deal with the possibility of rollback, we require that if the scheduler rolls back transaction $T_i$ at time $t_j$, no action from transaction $T_i$ will appear in $I$, following $t_j$. We also define $A_{\text{effective}}(O_i(p)[t_i])$ and $V_{\text{effective}}(O_i(p)[t_i])$ as $A(O_i(p)[t_i])$ and $V(O_i(p)[t_i])$, respectively, with the actions and corresponding versions of all transactions in $R(O_i(p)[t_i])$ removed. We define the effective outputs in this way because we assume that no reasonable scheduler would output an action from a transaction that is previously rolled back. We can also define $A_{\text{effective}}(O_i(p)[t_i], t_j)$ and $V_{\text{effective}}(O_i(p)[t_i], t_j)$ as $A(O_i(p)[t_i], t_j)$ and $V(O_i(p)[t_i], t_j)$, respectively, with the actions and corresponding versions of all transactions in $R(O_i(p)[t_i], t_j)$ removed.

We have examined the security issues involved in database concurrency control for MLS/DBMSs and shown how a scheduler can affect the security of the system in this framework. In the next section we apply noninterference to the scheduler model introduced above.

III. DC-SECURITY

We base our analysis of schedulers on the noninterference approach [9]-[11]. Consider a system that acts on a stream of input actions $i$ from one or more subjects. Let the view of the system by a subject be the complete picture the subject has of the system.

Definition: A subject $S_i$ is noninterfering with a subject $S_j$ in a system iff for every stream of commands $i$, executing $i$ results in a view of the system by $S_j$ that is identical to that which results from executing the stream $i$ with all commands issued by $S_j$ removed.

In other words, if $S_i$ is noninterfering with $S_j$, the actions of $S_i$ have no effect on $S_j$'s view of the system. We concern ourselves here with multilevel security and so we require that $S_i$ can interfere with $S_j$ only if the subject classification level of $S_j$ dominates that of $S_i$. A system that satisfies this requirement is secure with respect to the MLS property [10].

A system that satisfies the MLS property is free of covert channels between levels [12]. Thus the MLS property is a sufficient condition for a system to be free of covert channels.

It is important to remember we are analyzing the model

\footnote{For an output triple $t$, the projections $A(t), V(t)$, and $R(t)$ refer to the first, second, and third component of the triple, respectively. A projection of a sequence is obtained by applying the projection to each element in the sequence.}
of a system. Thus, the system may have covert channels not present in the model. Our model includes the approximation that once an action arrives for scheduling, the scheduler will generate a response instantaneously. This is never true, but it allows us to mask out certain covert channels that may arise within the scheduler. In this way we can separate the covert channel analysis of the scheduling protocol from the analysis of the scheduler implementation.

We introduce the *purge* function [11]. The function takes as arguments a subject classification level $l$ and a schedule $p$. The result of *purge* $(l, p)$ is $p$ except that every action $A(x, l_i, t_i)$ such that $l_i \neq l$ is removed. We can also apply the purge function to a sequence of elements $R$ composed of transaction names. The result is $R$ except that every transaction $T_i$ with subject classification level $l_i$ such that $l_i \neq l$ is removed.

We define the view of a subject with classification level $l$ of a sequence of output triples $O$, denoted by view $(l, O)$, to be the sequence of output triples obtained by applying *view triple* $(l, i)$ to each triple $i$ in $O$ and removing from the result all triples which evaluate to $(\text{nil}, \text{nil}, \langle \rangle)$. We define the *view triple* function below, where class $(A)$ represents the subject classification level of the action $A$:

$$
\text{view triple}(l, (A, V, R)) =
\begin{cases}
(A, V, \text{purge} (l, R)) & \text{if } l \geq \text{class}(A) \\
(\text{nil}, \text{nil}, \text{purge} (l, R)) & \text{if } l \neq \text{class}(A).
\end{cases}
$$

We also assume that whenever $\text{view} (l, O, (p)[t])$ is nonempty, a subject with classification level $l$ knows the time $t$. Therefore, we insist not only that $\text{view} (l, O, (p)) = \text{view} (l, O, (\text{purge} (l, p))[t])$ but that $\text{view} (l, O, (p)[t]) = \text{view} (l, O, (\text{purge} (l, p))[t])$ for all $t$, such that $t_0 \leq t \leq t_n$, where $t_n$ is the time at which the last action of $p$ arrives for scheduling. We need only consider the behavior of the scheduler until $t_n$ as no output occurs after this. We base our definition of DC-security on noninterference.

**Definition:** A scheduler is *DC-Secure* with respect to a class of input schedules $S$ if for every schedule $p$ in $S$, and every subject classification level $l$ in $p$, view $(l, O, (p)[t])$ is for every prefix $p'$ of every schedule $p$, as it is defined in [11]. The equivalent follows from the fact that $\text{view} (l, O, (p)[t]) = \text{view} (l, O, (\text{purge} (l, p'))[t])$, where $A_i$ is the action in $p$ that arrives at time $t_i$. $p'$ is the prefix of $p$ that precedes $A_i$ and $p'[A_i]$ is the prefix constructed by appending $A_i$ to the end of $p'$.

To prove a scheduler satisfies DC-Security we can apply this definition directly as is done in [19]. In the rest of this section we simplify this proof by dividing the class of schedules into two groups, those that are schedulable (i.e., without rollback) and those that are not. If the scheduling of a transaction causes one or more transactions to be rolled back we say it is unschedulable. We will introduce a necessary condition for DC-Security with separate conditions regarding schedulable and unschedulable schedules. Following this we introduce two necessary conditions that are simpler to apply as they consider the scheduling of a single schedule $p$ rather than the scheduling of both $p$ and *purge* $(l, p)$. Finally, we describe a class of schedulers called output-state-equivalent for which another criterion implies DC-Security. The criterion has two parts, one for schedulable inputs and one for unschedulable inputs.

### A. Value Security

In this section we introduce a property of schedulers called **Strong-Value-Security**. The definition applies the *purge* function to a full schedule $(s_p, V_p)$. Let $(s_{\text{purge}}, V_{\text{purge}})$ be the result of *purge* applied to $(s_p, V_p)$. Then $s_{\text{purge}} = \text{purge} (s_p, s_p)$ and $V_{\text{purge}}$ agrees with $V_p$ for all actions in $s_{\text{purge}}$ and is undefined for every other action in $s_p$.

**Definition:** A scheduler is **Strong-Value-Secure** with respect to a class of input schedules $S$ if for all schedules $p$ in $S$, and every subject classification level $l$ in $p$, the full schedule output by the scheduler in response to $p$ call it $(s_p, V_p)$, is such that *purge* $(l, s_p)$ is equal to the full schedule output by the scheduler in response to *purge* $(l, p)$.

This definition ensures DC-Security for schedulable inputs. It does not consider actions that are output and later rolled back. For example, consider the following input schedule prefix $p$:

$$
C
$$

$$
R(y, U) 
$$

$$
W(x, U)
$$

The scheduler employs strict two-phase locking [2]. The deadlock graph is constructed each time an action is delayed. If delaying an action adds a cycle to the deadlock graph, the corresponding transaction is aborted. The schedule output by the scheduler in response to $p$ is the schedule $s_1$ shown below:

$$
T_1(S): R(z, U)
$$

$$
C
$$

$$
T_1(S): R(z, U) 
$$

$$
W(y, U)
$$

$$
T_3(U): R(y, U) 
$$

$$
W(z, U) 
$$

$$
R(x, U)
$$

and every subject classification level $l$ in $p$, view $(l, O, (p)[t]) = \text{view} (l, O, (\text{purge} (l, p))[t])$ for all $t_i$ such that $t_0 \leq t_i \leq t_n$, where $t_n$ is the time at which the last action of $p$ arrives for scheduling.

We associate the time $t_i$ with the arrival of some action $T_i: A(x, l_i, t_i)$ in a schedule $p$. If $l \neq l_i$, $T_i: A(x, l_i, t_i)$ is not included in *purge* $(l, p)$. In this case we define $O, (\text{purge} (l, p))[t_i]$ to be empty.

Equivalently, we can require that $\text{view} (l, O, (p')) = \text{view} (l, O, (\text{purge} (l, p'))[t])$ for every prefix $p'$ of every schedule $p$, as it is defined in [11].
Transaction \( T_3 \) does not appear in the schedule because it is aborted when \( T_3(U) : W(y, U) \) arrives for scheduling. The transaction is aborted because delaying \( T_3(U) : W(y, U) \) would cause deadlock. The schedule\( \text{purge}(\text{Unclassified}, s_i) \) consists of transaction \( T_2 \) only. Now, consider the input \( p_{\text{purge}} = \text{purge}(\text{Unclassified}, p) \) shown below:

\[
p_{\text{purge}} = T_3(U) ; T_3(U) : W(z, U) R(x, U)
\]

Again, transaction \( T_1 \) will be aborted when the action \( T_3(U) : W(y, U) \) arrives for scheduling. Thus the output of the scheduler in response to \( p_{\text{purge}} \) is simply transaction \( T_2 \). Thus, we have \( \text{purge}(\text{Unclassified}, s_i) \) is equal to the schedule output in response to \( p_{\text{purge}} \). Even though \( T_1 \) clearly interferes with transaction \( T_2 \), the input \( p \) satisfies Strong-Value-Security. This is because Strong-Value-Security does not consider transactions that are rolled back. We deal with this possibility in the next section.

**B. Recovery Security**

Rollback allows transactions at two different security levels to interfere with each other. The following property deals with this type of interference.

**Definition:** A scheduler is **Recovery-Secure** with respect to a class of inputs \( S \) iff for every input \( p \) in \( S \), on the arrival of an action \( A_t \) at time \( t \) for scheduling:

1. If the first nonempty output triple at time \( t \) rolls back a sequence of transactions \( T \), then for every subject classification level \( l \) in \( p \) for which \( \text{purge}(l, T) \) is nonempty, the first nonempty output triple at time \( t \) in scheduling \( \text{purge}(l, p) \) must roll back the sequence of transactions \( \text{purge}(l, T) \).
2. If no rollback occurs at time \( t \), then for every subject classification level \( l \) in \( p \), no rollback occurs at time \( t \) in the schedule \( \text{purge}(l, p) \).

**Recovery Security** ensures that a subject’s view of the first rollback at time \( t \) does not depend on the actions of subjects having a higher or incomparable subject classification levels. The first part ensures that any subject that is aware of the rollback in \( p \) (i.e., those that dominate a transaction in \( T \)) experiences the rollback at the same point in \( \text{purge}(l, p) \). Second, it ensures that the scheduler rolls back an equivalent set of actions in both schedules. Finally, the condition ensures that if rollback does not occur in \( p \) at \( t \), it does not occur in \( \text{purge}(l, p) \) at \( t \).

**C. Necessary Conditions for DC-Security**

We now present the following necessary condition for DC-Security based on Strong-Value-Security and Recovery-Security.

**Theorem 1:** (Necessary condition for DC-Security)—a scheduler that is DC-Secure with respect to the class of input schedules \( S \) is

1. Strong-Value-Secure with respect to all prefixes of inputs in \( S \); and
2. Recovery-Secure with respect to all prefixes of inputs in \( S \).

The proof of this theorem, along with those of all other theorems and lemmas presented, are available from the authors. Briefly, if a scheduler is not Strong-Value-Secure, there exists some input prefix \( p \) and some subject

\[
R(y, U) \ W(x, U) \ W(y, U).
\]

with classification level \( l \) such that the subject’s view of the output schedule in response to \( p \) is different than in response to \( \text{purge}(l, p) \). If a scheduler is not Recovery-Secure, there exists some input prefix \( p \) and some transaction \( T \), with classification level dominated by \( l \) such that \( T \) is aborted (not aborted) in \( p \) at time \( t \), but not aborted (is aborted) in \( \text{purge}(l, p) \) at time \( t \). In both cases, there is a schedule \( p \) such that \( \text{view}(l, O_t(p)) \neq \text{view}(l, O_t(\text{purge}(l, p))) \). Thus, the scheduler is not DC-secure.

We now consider two simpler properties of schedulers that can be used to identify schedulers that are not DC-Secure. The criteria are easier to apply because they consider a single schedule \( p \) rather than a pair of schedules, \( p \) and \( \text{purge}(l, p) \). The criteria apply to schedulers that do not delay or roll back any action in a schedule containing only one transaction. This is a reasonable requirement because in a schedule containing only one transaction there are no conflicts between transactions to cause delays or rollbacks. The first property we consider deals with interference through the delay of an action. The second deals with interference through rollback.

The task of a scheduler is to reorder its inputs to create a correct output schedule. If an action appears later in the output schedule than in the input schedule, it has experienced a delay. The following definition characterizes this delay.

**Definition:** An action \( A_t \) is delayed with respect to an action \( A_{t'} \) iff:

1. the action \( A_t \) appears before \( A_{t'} \) in the input schedule; and
2. the action \( A_t \) follows \( A_{t'} \) in the output schedule; and
3. the action \( A_t \) is the last action in the input schedule that satisfies 1) and 2).

The conditions 1) and 2) detect when the action \( A_t \) is delayed. Condition 3) limits the definition to cover only the last action in the input schedule that is moved ahead of \( A_t \). This definition ensures that each action is delayed with respect to at most one action. In addition, it distinguishes between necessary waiting and incidental waiting. Consider the following input schedule:

\[
T_3(S) : a_3
T_2(U) : a_2
T_3(U) : a_1.
\]

The output schedule produced in response is shown be-
Thus, to show that a scheduler is not DC-Secure we can show that it is not Schedule-Secure or that it is not Co-Schedule-Secure. These properties apply to one schedule rather than the relationship between the scheduling of two different schedules. Thus, they are somewhat easier to apply than Theorem 1.

D. A Sufficient Condition for DC-Security

In defining a sufficient condition we need to make stronger assumptions about the scheduler. We assume the model shown in Fig. 1.

The scheduler consists of an input selector, an ordered queue, a queue controller, and a protocol controller. Delayed actions are placed in the queue. The queue controller determines whether the next action will come from the input or the queue. The protocol controller given an action as input generates the outputs of the scheduler.

The actions in the queue are ordered by the time at which they arrived for scheduling. Following an input on "Reset," the next element removed from the queue will be the first element in the queue. The "Remove" input, given a sequence of transaction names as input, removes from the queue all actions from each of the transactions. The "Last" output signals when the last element of the queue has been retrieved.

The queue controller determines the input to the protocol controller. The finite state machine shown in Fig. 2 represents our assumptions about the queue controller. We label each state transition with a condition under which it can be traversed (above) and a pair of output values (below). The first output value selects the source of the next action, $I_i$ or Queue. The second controls the "Reset" input of the queue. The queue controller starts in state $s_0$. If no action is output or rolled back (i.e., $A = \text{nil} \land R = \langle \rangle$), the machine stays in this state and continues scheduling actions from the scheduler input $I_i$. When an action is output or rolled back (i.e., $A \neq \text{nil} \lor R \neq \langle \rangle$), the machine enters state $s_1$, when it resets and begins scheduling actions from the queue. Testing continues as long as there are actions in the queue and no action is output or rolled back (i.e., $A = \text{nil} \land R = \langle \rangle \land \neg \text{Last}$). During testing, if an action is output or rolled back (i.e., $A \neq \text{nil} \land R \neq \langle \rangle$), the queue is reset and the testing starts over. Thus, we are giving more priority to the scheduling of actions from the queue than to the actions from the input. When the last element in the queue has been tested (i.e., $A = \text{nil} \land R = \langle \rangle \land \text{Last}$), the controller begins scheduling actions from $I_i$ once again.

A protocol controller is a deterministic state machine. The protocol controller begins prior to time $t_0$ in some initial state $s_{00}$. At each time $t_i \geq t_0$ an action $T : A(x, I_i, I_i)$ arrives on $I_i$ for scheduling. This causes a series of events to occur which we refer to as $e_{\tau}$ through $e_{\tau}$. We use $(t_i, e_{\tau})$ to denote the $k$th event at time $t_i$. At each event $(t_i, e_{\tau})$ an action $T : A(x, I_i, I_i)$ appears on $I_i$. The protocol controller transfers the action either to $A$, along with a corresponding version on $V$, or to $f_{\tau}$. During each event,
a possibly empty sequence of transaction names is output on \( R \), which represent transactions to be rolled back. To be consistent with our previous assumption about output triples, we assume that \( R \neq \langle \rangle \) only when an action is transferred to \( Q_1 \). The output triple that is output on \( O_1 \) is composed of the outputs on \( A \), \( V \), and \( R \). Following the output on \( O_1 \), the protocol controller enters a new state \( s_{t_1, i+1} = \delta_p(s_{t_1, i}, T; A(x, l_1, l_1)) \).

We say that two states \( s_{t_1, i} \) and \( s_{t_1, i} \) are equivalent if the following conditions are satisfied:

1) The output of the protocol controller (including \( A \), \( V \), and \( Q_1 \)) in state \( s_{t_1, i} \) in response to the action \( T; A(x, l_1, l_1) \) is equal to its output in state \( s_{t_1, i} \) in response to the same action \( T; A(x, l_1, l_1) \).

2) The state \( \delta_p(s_{t_1, i}, T; A(x, l_1, l_1)) \) is equivalent to \( \delta_p(s_{t_1, i}, T; A(x, l_1, l_1)) \).

Now, we define a class of schedulers for which the cumulative effective output determines the state of the protocol controller. Here we refer only to the actions output but not yet rolled back by the scheduler. We do not consider the versions assigned to actions.

**Definition:** A scheduler is Output-State-Equivalent if for any two states of the protocol controller \( s_{t_1, i} \) and \( s_{t_1, i} \) with effective cumulative output \( A_{\text{effective}}(O_1(p)) \), \( (t_0, e_1), (t_1, e_1) \)) and \( A_{\text{effective}}(O_1(p_2)) \), \( (t_0, e_1), (t_1, e_1) \)), respectively, if \( A_{\text{effective}}(O_1(p_1)) \), \( (t_0, e_1), (t_1, e_1) \)) is equal to \( A_{\text{effective}}(O_1(p_2)) \), \( (t_0, e_1), (t_1, e_1) \)), then \( s_{t_1, i} \) is equivalent to \( s_{t_1, i} \).

In other words, for an Output-State-Equivalent scheduler, if the actions output but not yet rolled back by the scheduler in two different states are identical, the states of the protocol controller are equivalent. We now define a new property of schedulers called Rollback-Input Driven.

**Definition:** A scheduler is Rollback-Input Driven if rollback occurs at event \( (t_i, e_i) \) only if it occurred at \( (t_i, e_i) \), where \( k > 1 \).

In other words, if an action arriving at time \( t_i \) does not cause rollback, a scheduler that is Rollback-Input Driven will not roll back on any of the following events at time \( t_i \).

To prove that a scheduler is DC-Secure we need to show that for every subject classification \( l \) in \( p \), view \( (l, O_1(p)[t_i]) = \text{view}(l, O_1(\text{purge}(l, p))[t_i]) \) for all time \( t_i \) during the scheduling of \( p \). The proof follows from Strong-Value-Security if we ignore rollback. When rollback occurs we must show three things. First, before the rollback, the view in \( p \) is the same as the view in \( \text{purge}(l, p) \). Second, rollback provides the same view in \( p \) as in \( \text{purge}(l, p) \). Finally, we must show that following the first rollback the view of a subject with classification level \( l \) is equivalent in \( p \) and \( \text{purge}(l, p) \). The first two parts of the proof follow easily from Strong-Value and Recovery-Security. The last part of the proof relies on our ability to create a new schedule that has the same behavior as the original following recovery, but rolls back one fewer times. Thus, we can decompose the problem and induct over the number of times rollback occurs in the schedule. Lemma 1 gives us a method for constructing these equivalent schedules. The following lemma assumes a scheduler that does not delay actions unnecessarily, which we now define.

**Definition:** A scheduler does not delay actions unnecessarily if any output schedule prefix the scheduler produces can be used as inputs to the scheduler with the result that all actions are output without delay and without being rolled back.

**Lemma 1:** For any scheduler that satisfies the queuing model described in Section III-D and

1) does not delay actions unnecessarily; and
2) is Output-State-Equivalent; and
3) has completed the event \( (t_i, e_i) \) in scheduling the prefix \( p^* \) of a schedule \( p \),

there exists a sequence \( p_{eq} = p^* \delta_{eq}[p^* \delta_{eq}] \) (i.e., the actions in \( p_{eq} \) followed by those in \( p_{eq} \) followed by those in \( p_{eq} \)) with the first action of \( p_{eq} \) \( p_{eq} \) arriving at time \( t_i \) and the first action of \( p_{eq} \) \( p_{eq} \) arriving at \( t_m \) such that

1) The output (excluding any empty outputs) of the scheduler with input \( p_{eq} \) during \( [t_i, t_m] \) is equal to the output of the scheduler with input \( p \) during \( [t_i, t_m] \), i.e., \( \delta_{eq}(p_{eq})[t_i, t_m] = \delta_{eq}(p)[t_i, t_m] \),

2) The state of the scheduler at time \( t_m \) in scheduling \( p_{eq} \) is equivalent to the state of the scheduler at time \( t_{i+1} \) in scheduling \( p \).

3) The remaining input sequence at time \( t_m \) in scheduling \( p_{eq} \) is equal to the remaining input sequence at time \( t_{i+1} \) in scheduling \( p \).
Below, we present a sufficient condition for DC-Security for Output-State-Equivalent schedulers. The condition requires that all schedulable prefixes of inputs in $S$ be Strong-Value-Secure. The condition also requires the scheduler to be Recovery-Secure for all prefixes that roll back, if at all, only on the arrival of the last action.

**Theorem 4**: (Sufficient condition for DC-Security) — a scheduler is DC-Secure with respect to all inputs in $S$ if it is Output-State-Equivalent, satisfies the queuing model described in Section III-D and

1) does not delay actions unnecessarily; and
2) is Rollback-Input Driven; and
3) is Strong-Value-Secure with respect to all schedulable prefixes of inputs in $S$; and
4) is Recovery-Secure with respect to all prefixes of inputs in $S$, which roll back, if at all, only during the last time period.

We have presented a model of a scheduler and defined DC-Security in terms of the model. The definition of DC-Security states the requirements of interference applied to our scheduler model. To simplify the analysis of scheduling protocols, we assume that the scheduler exhibits no delays. In this way, we factor in the channels due to contention for access to the scheduler. We have introduced two properties of schedulers that constitute a necessary condition for DC-Security. We have also introduced two simpler necessary conditions. We have identified a class of schedulers we call Output-State-Equivalent for which another criterion implies DC-Security. The criterion considers separately the behavior of the scheduler in response to schedulable and unschedulable inputs.

In the next section, we will use the necessary and sufficient conditions we have developed in this section to analyze the security of various schedulers.

**IV. An Analysis of Several Existing Schedulers**

In this section, we examine the security properties of several existing schedulers using the framework developed in the previous section. We also introduce multiversion schedulers and discuss the relationship between various classes of secure schedulers (e.g., Class 1-SS and Class 2-SS) and those that are schedulable by multiversion schedulers. The section is divided into two parts. In the first, we consider single version schedulers. In the second, we examine multiversion schedulers.

**A. Single Version Schedulers**

In this section, we examine dynamic two-phase locking and conflict graph schedulers.

1) **Dynamic Two-Phase Locking (2PL) Schedulers:** The dynamic 2PL schedulers are characterized by the following test taken from [26]:

A step can always proceed, unless it conflicts with a previous step of an active transaction (other than its own).

These schedulers are dynamic because they make decisions based solely on the actions that have already arrived. These schedulers are not DC-Secure.

**Theorem 5**: Dynamic two-phase locking schedulers are not DC-Secure with respect to Class 1-SS schedules.

**Proof**: The dynamic 2PL scheduler does not delay actions in a schedule containing only one transaction. If there is only one transaction in the schedule, it cannot conflict with the step of other transactions. Thus, we can demonstrate that the scheduler is not DC-Secure by showing that is not Schedule-Secure for some Class 1-SS input with two transactions. Consider the input shown below:

$$
\begin{align*}
T_1(S): & \quad R(y, U) \quad R(z, U) \quad C \\
T_2(U): & \quad W(y, u) \quad W(z, U) \quad C.
\end{align*}
$$

The schedule output by the 2PL scheduler in response to $p$ is shown below:

$$
\begin{align*}
T_1(S): & \quad R(y, U) \quad R(z, U) \quad C \\
T_2(U): & \quad W(y, U) \quad C.
\end{align*}
$$

The action $T_2(U): W(y, U)$ is delayed with respect to $T_1(S): C$. Because Unclassified does not dominate Secret, the scheduler is not Schedule-Secure with respect to $p$. By Theorem 2, the scheduler is not DC-Secure with respect to Class 1-SS schedules.

From the fact that Class 1-SS is a subset of Class 2-SS, the dynamic two-phase scheduler is not DC-Secure with respect to Class 2-SS schedules either.

2) **Conflict Graph Schedulers**: In this section, we consider conflict graph schedulers [26]. A conflict graph is a directed graph that consists of a set of nodes representing the transactions in the schedule and a set of directed edges representing conflicts. There is an edge from transaction $T_1$ to $T_2$ if a step of $T_1$ conflicts with a previously scheduled step of $T_2$. Two steps conflict with each other if they access the same entity and one or more of the accesses are writes. Papadimitriou describes a subset of view serializable schedules called conflict serializable schedules and shows that a schedule is conflict serializable if its corresponding conflict graph is acyclic [26].

A conflict graph scheduler augments the conflict graph every time it outputs an action. A step proceeds if the edges added to the conflict graph by scheduling the step do not create a cycle [26]. If the step does not proceed, it never will. Thus, if the action cannot proceed, the transaction is rolled back. To recover properly, we must also roll back every transaction that read a value written by the transaction that is rolled back. To ensure that this is always possible, we must delay the commit of a transaction until all transactions it read from have committed.

**Theorem 6**: Conflict graph schedulers are not DC-Secure with respect to Class 1-SS schedules.
Proof: The conflict graph scheduler allows each action in a schedule containing only one transaction to proceed immediately. This is because a transaction cannot conflict with itself, which means the conflict graph will never have a cycle. Because every action is allowed to proceed, no transaction is rolled back. Thus, we can demonstrate that the scheduler is not Co-Schedule-Secure by showing that it is not Co-Schedule-Secure for some Class 1-SS input with two transactions. Consider the Class 1-SS schedule shown below:

\[ p = T_1(S): R(y, U) \quad R(z, U) \quad C \]
\[ T_2(U): W(y, U) \quad W(z, U) \quad C. \]

When transaction \( T_1 \) is ready to execute its commit step, the conflict graph has one edge from the node labeled \( T_1 \) to the node labeled \( T_2 \). The commit of \( T_1 \) is required to follow that of \( T_2 \) because it read the value of \( y \) written by \( T_2 \). The scheduler cannot output the write of \( z \) for \( T_2 \) because this would add an edge from \( T_2 \) to \( T_1 \) and create a cycle. Thus, the scheduler aborts \( T_2 \) and also \( T_1 \) as it read from \( T_2 \). The subject classification level of \( T_2 \) does not dominate that of \( T_1 \), and thus, the scheduler is not Co-Schedule-Secure with respect to \( p \). By Theorem 3, the scheduler is not DC-Secure with respect to Class 1-SS schedules.

B. Multiversion Schedulers

A multiversion scheduler allows multiple versions of the same entity. This allows enhanced concurrency by reducing the contention for access to an entity. We are interested in these schedulers because this contention is the cause of Data Conflict covert channels. In this section, we examine three multiversion scheduling protocols, multiversion locking, multiversion timestamp [26], and optimistic [20].

1) Multiversion Locking: Multiversion locking is described in [26]. The scheduler maintains two versions of an entity, the current version and an uncommitted one. There is always at most one uncommitted version. The protocol is defined by tests that determine when a read or write step can proceed. Separate tests apply to actions that are the last operation in a transaction. We quote the test descriptions given in [26]. The test for operations that are not last is as follows.

Any read step which is not the last step of its transaction can proceed; the version read is either the current one for the entity read, or an uncommitted one. Any write step which is not the last step of its transaction can proceed only if the last transaction that wrote this entity has committed. If it proceeds, the version written is kept.

The test for actions which are the last step in a transaction is as follows.

Any step which is the last step of its transaction cannot proceed until the following two categories of transactions have committed: a) transactions which have read the current version of an entity this transaction writes, and b) transactions from which the present transaction has read a version.

Theorem 7: The multiversion locking scheduler is not DC-Secure with respect to Class 1-SS transactions.

Proof: The multiversion locking scheduler does not delay actions in a schedule containing only one transaction. The reasons for delaying an action all involve other transactions. If there is only one transaction in the schedule, none of its actions will be delayed. Thus, we can demonstrate that the scheduler is not DC-Secure by showing that it is not Schedule-Secure for some Class 1-SS input with two transactions. Consider the Class 1-SS input schedule:

\[ T_1(S): R(x, U) \quad W(y, S) \quad C \]
\[ T_2(U): W(x, U) \quad C. \]

The step \( T_1(S): R(x, U) \) arrives and is executed. It reads the current version as this is the only version available. Next, the \( T_2(U): W(x, U) \) step arrives. It proceeds immediately as no other transaction has written \( x \). The action \( T_2(U): C \) is the last action of \( T_2 \). It is required to follow \( T_1(S): C \) as \( T_1 \) read the current version of an element that \( T_1 \) writes (i.e., \( x \)). The action \( T_1(U): C \) is delayed with respect to the commit step of \( T_1 \). Because Unclassified \( \neq \) Secret, the scheduler is not Schedule-Secure with respect to Class 1-SS schedules. By Theorem 2, the scheduler is not DC-Secure with respect to Class 1-SS schedules.

In the next section, we consider another multiversion scheduler.

2) Multiversion Timestamp: The multiversion timestamp scheduler uses the order of transaction starting times as its serialization order. Any action that would violate this order is not output and its transaction is rolled back.

In this scheduler, a timestamp is associated with each transaction. Each version of an entity has a read timestamp and a write timestamp. The read timestamp of a version is equal to the timestamp of the youngest transaction that read it (zero if it has not been read). The write timestamp is equal to the timestamp of the transaction that wrote the version. The timestamp of the transaction represents the time at which the transaction began execution. Initial versions of entities have read and write timestamps of zero. The appropriate version of an entity for a transaction \( T \) is the version whose write timestamp is a) smaller than the timestamp of \( T \), and b) as large as possible subject to a). The following test taken from [26] characterizes the scheduler:

An action \( T: R(x) \) always proceeds; it is presented with the version of \( x \) appropriate for \( T \). An action \( T: W(x) \) proceeds if the timestamp of \( T \) is larger than the real timestamp of the version of \( x \) appropriate for \( T \).

If a step is not allowed to proceed, the transaction is rolled
back. To ensure serializability we must roll back every transaction that read a value written by the transaction that is rolled back. To ensure that this is always possible we must delay the commit of a transaction until all transactions it reads from have committed.

Theorem 8: The multiversion timestamp scheduler is not DC-Secure with respect to Class 1-SS schedules.

Proof: The multiversion timestamp scheduler does not roll back the only transaction in a schedule. Because there is only one transaction in the schedule, no younger transaction can read a version of an element out of timestamp order and thereby cause a rollback to occur. Thus, we can demonstrate that the scheduler is not DC-Secure by showing that it is not Co-Schedule-Secure for some Class 1-SS input with two transactions. Consider the input shown below:

\[ T_1(S): R(x, U) \quad C \]
\[ T_2(U): R(x, U) \quad W(x, U) \quad C. \]

We start with a version of x whose write and read timestamps are both zero. The step \( T_1(U): R(x, U) \) arrives first and the scheduler assigns \( T_1 \) a timestamp of one. The read is executed and the read timestamp of x (the original version) is updated to one. The step \( T_1(S): R(x, U) \) arrives next and the scheduler assigns \( T_1 \) a timestamp of two. The read is executed on the original version of x and its read timestamp is changed from one to two. Transaction \( T_1 \) then executes \( T_1(S): C \) and commits. The next action to arrive is \( T_2(U): W(x, U) \). The version of x appropriate for \( T_2 \) is the only version available, the original. The read timestamp of this is two. Because this is not less than the timestamp of \( T_2 \) (i.e., zero), the step cannot proceed and \( T_2 \) is rolled back. Because Unclassified \( \not\in \) Secret, the multiversion timestamp scheduler is not Co-Schedule-Secure with respect to Class 1-SS schedules. By Theorem 3, the scheduler is not DC-Secure with respect to Class 1-SS schedules.

Neither the multiversion nor the multiversion timestamp scheduler is DC-Secure. In the following section we consider optimistic schedulers, which are DC-Secure.

3) Optimistic Schedulers: An optimistic scheduler does not reorder its input, but merely verifies that the input is a correct schedule. The actions of a transaction proceed immediately each write modifying a local copy of the element. When the transaction attempts to commit, the scheduler applies a test to determine if the transaction is an allowable continuation of the previously committed portion of the schedule. If so, all writes performed by the transaction are applied to the database atomically. If not, the writes are discarded. These local writes require the optimistic scheduler to maintain multiple versions of some objects.

The particular scheduler we consider in what follows is due to [20]. The criterion the scheduler uses in deciding if a transaction can commit is the following:

A transaction attempting to commit is aborted if its read set conflicts with the write set of a transaction that committed after it started.

This test is sufficient if the test is executed and the writes are preformed atomically. This is referred to as serial validation. We will discuss another possibility called parallel validation later in this section.

We will now show that the optimistic scheduler is DC-Secure with respect to Class 2-SS transactions. We will do this using the sufficient condition for DC-Security presented in Theorem 4. First we introduce the following six lemmas. The first lemma shows that if a prefix \( p \) does not cause rollback then neither does \( \text{purge}(l, p) \). We will use it to simply the proof of many of the other properties.

Lemma 2: If no rollback occurs when a schedule prefix \( p \) is used as input to the optimistic scheduler, no rollback occurs during the scheduling of the prefix \( \text{purge}(l, p) \) either.

Lemma 3: The optimistic scheduler is Strong-Value-Secure with respect to all schedulable prefixes of Class 2-SS schedules.

Lemma 4: The optimistic scheduler is Rollback-Input Driven.

Lemma 5: The optimistic scheduler is Recovery-Secure with respect to all prefixes of Class 2-SS inputs that roll back, if at all, only during the last time period.

Lemma 6: The optimistic scheduler is Output-State-Equivalent.

Lemma 7: The optimistic scheduler does not delay actions unnecessarily.

Theorem 9: The optimistic scheduler is DC-Secure with respect to Class 2-SS schedules.

Proof: The optimistic scheduler satisfies the queuing model of Section III-D as it does not delay actions. This together with Lemmas 3–7 and Theorem 4 imply the scheduler is DC-Secure.

There is a drawback to this scheduler. In case of conflicts, a transaction must roll back. This is inefficient as it requires the same processing to be repeated possibly many times before the transaction can commit. The policy is biased against high-level transactions because they are required to roll back in case of a conflict. This may make it difficult to commit high-level transactions and may cause starvation. An intruder can thus block the normal operations of high-level transactions by executing numerous low-level transactions. In this way, security conflicts with availability.

The problem of starvation is addressed in [20]. A transaction after having been aborted some number of times is allowed to execute to completion in a critical section. This ensures that the transaction will commit. The solution implies a loss of concurrency as no other transaction will be allowed to perform its validation while the starved transaction is executing. In addition, this critical section introduces a covert channel.

A transaction being processed by an optimistic scheduler goes through two phases. The first is called its action
phase. During this time it performs reads and writes. The
writes to the database are tentative and do not appear to
other transactions until the transaction commits. When the
transaction is ready to commit, it enters the validation phase. In this phase the transaction checks for conflicts
with other transactions. At the end of the validation phase,
the transaction is either rolled back or commits. If it com-
mits, its updates are made permanent.

The validation phase is usually considered a critical
section, i.e., when a transaction commits, its validation
is carried out completely before another validation may
begin. This is called serial validation and is the type we
consider above. The critical section necessary for serial
validation represents a covert timing channel. However,
it is not a data conflict covert channel, but a channel due
to contention for the scheduler. It also restricts perform-
ance. To counter this problem we can use a variation of
the optimistic scheduler employing parallel validation.
We now consider the security properties of this scheduler.

Parallel validation allows several transactions to check
for conflicts simultaneously. There are two critical sec-
tions each of which is shorter than the one required for
serial validation. Thus, it is likely that a transaction will
spend less time waiting to enter the critical section. When

$$p = \frac{R(x, U)}{T_1(U)} + \frac{W(y, U)}{T_2(U)}$$

a transaction is ready to commit, it enters the first critical
section. While in the critical section, it reads the sequence
number of the last transaction to commit, creates a snapshot
of the set of transactions that are in the validation
phase, called the active set, and adds itself to the set.

After leaving the first critical section, the transaction
performs several conflict tests [20]. It cannot commit if:

1) the transaction's read set conflicts with the write set
of a transaction that committed between the time this
one started and the time it entered the first critical
section; or

2) the transaction's read set or write set conflicts with
the write set of a transaction in the active set.

If the transaction discovers no conflicts, it makes its
writes permanent. Following this the transaction enters
the final critical section. While there, it removes itself
from the active set and either commits (if it found no con-
flicts) or is rolled back.

To represent the concurrency in the validation phase
using our scheduler model we specifically represent each
of the events associated with entering a critical section in
the input schedule. We can represent these events as writes
of special elements associated with the transaction as we
have done to represent the commit step. We will represent
the event in which transaction $T_i$ enters the first critical
section with the action $T_i; W(T_{i1}, I, I)$ and we represent
$T_i$ entering the second critical section with the action $T_i;
W(T_{i2}, I, I)$. When the context is clear we will represent
these actions as $T_i; C_1$ and $T_i; C_2$, respectively.

The second test in the parallel validation protocol poses
a difficulty for Class 2-SS schedules. It requires a trans-
action to abort when an element it has written is also writ-
then by another transaction. In Class 2-SS schedules, a low-
level transaction can write the same data element written
by a high-level subject requiring a low-level transaction
to abort. Because of this the scheduler is not DC-Secure
with respect to Class 2-SS schedules.

**Theorem 10:** The optimistic scheduler with parallel
validation is not DC-Secure with respect to Class 2-SS
transactions.

**Proof:** The optimistic scheduler with parallel vali-
dation never aborts the only transaction in a schedule be-
cause there are no other transactions with which it could
conflict. Thus, we can demonstrate that the scheduler is
not DC-Secure by showing that it is not Co-Schedule-Sec-
ure for some Class 2-SS input with two transactions.
Consider the following input schedule to an optimistic
scheduler:

$$p = \frac{R(z, U)}{T_1(U)} + \frac{W(x, U)}{T_2(U)} + \frac{W(z, U)}{T_3(U)}$$

The schedule is Class 2-SS because of the write of the
Secret element $z$ by $T_2$. When $T_1$ enters the first critical
section, it adds itself to the active set. When $T_2$ enters
the first critical section, it adds itself to the active set. When
$T_3$ enters the first critical section, it adds itself to the active set. Transaction
$T_1$ is forced to abort as $x$ is in both $T_1$'s and $T_2$'s write set.

Because Unclassified $\neq$ Secret, the scheduler is not Co-
Schedule-Secure with respect to Class 2-SS schedules. By
Theorem 3, the scheduler is not DC-Secure with respect to
Class 2-SS schedules.

The optimistic scheduler with parallel validation is not
Output-State-Equivalent as we will now demonstrate.
Consider the input schedule shown below:

$$p = \frac{R(z, U)}{T_1(U)} + \frac{W(x, U)}{T_2(U)} + \frac{W(z, U)}{T_3(U)}$$

When the action $T_1(U); C_i$ arrives for scheduling, the
active set is empty and no transactions have committed be-
tween the time $T_1$ started and now. Thus, when $T_3(U); C_i$
arrives, $T_1$ will commit. When $T_2(U); C_i$ is executed, $T_2$
is already in the active set. Because $T_1$ writes $z$ and $T_2$
reads it, when $T_2(U); C_i$ is executed, $T_2$ is aborted. When
$T_1(U); C_i$ is executed, the active set contains $T_2$. Because
$T_2$ writes $x$ and $T_1$ reads it, when $T_1(U); C_i$ arrives for
scheduling, $T_1$ will be aborted. When $T_1(U); C_i$ arrives
the effective cumulative output is the schedule prefix $s$
shown below:

\[ T_1(U): R(x, U) \quad C_1 \]
\[ s = T_2(U): W(z, U) \quad C_1, C_2. \]

Now consider the input schedule \( p_2 \) shown below:

\[ T_1(U): R(x, U) \quad C_1, C_2 \]
\[ p_2 = T_2(U): W(z, U) \quad C_1, C_2. \]

Once again, when \( T_2(U) : C_2 \) arrives, \( T_1 \) will commit as no transactions are in the active set and none have committed between the time \( T_1 \) started and now. When \( T_1(U) : C_1 \) executes the active set is empty. The only transaction that committed between the time \( T_1 \) started and now is \( T_3 \). Because \( T_1 \) and \( T_2 \) do not access any common elements, when \( T_1(U) : C_2 \) arrives for scheduling, transaction \( T_2 \) will commit. The effective cumulative output when \( T_1(U) : C_2 \) arrives for scheduling in both \( p_1 \) and \( p_2 \), and it is committed in one case and rolled back in the other, the scheduler is not Output-State-Equivalent. Because the scheduler is not Output-State-Equivalent we cannot apply Theorem 4 and must resort to proving the scheduler DC-Secure using the definition given in the beginning of Section III. Rather than carry through with this we leave open the question of whether the optimistic scheduler with parallel validation is DC-Secure with respect to Class 1-SS. The results of our characterization of the schedulers discussed above are shown in Table I.

### Table I

<table>
<thead>
<tr>
<th>Type of Scheduler</th>
<th>Schedule-Secure</th>
<th>Co-Schedule-Secure</th>
<th>DC-Secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic two-phase-locking</td>
<td>No</td>
<td>Unknown</td>
<td>No</td>
</tr>
<tr>
<td>Conflict graph</td>
<td>Unknown</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multiversion locking</td>
<td>No</td>
<td>Unknown</td>
<td>No</td>
</tr>
<tr>
<td>Multiversion timestamp</td>
<td>Unknown</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Optimistic with serial validation</td>
<td>Yes</td>
<td>Yes for Class 2-SS schedules</td>
<td>No</td>
</tr>
<tr>
<td>Optimistic with parallel validation</td>
<td>Yes</td>
<td>Unknown for Class 1-SS schedules</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No for Class 2-SS schedules</td>
<td></td>
</tr>
</tbody>
</table>

V. RELATED WORK

Hinke and Schaefer did some of the earliest work in this area [14]. The authors consider the problem of atomic reads and writes of a single data element and point out the problems associated with locking [14]. They provide a method for arbitrating access to a single data element but do not consider the problem in the context of transactions. Transactions imply an ability to read and write consistent versions of more than one data element. The approach can be generalized for use with transactions as described in [27]. The scheme is similar to the optimistic approach. Transactions are serialized at each level, i.e., two transactions at the same level cannot overlap. A high-level transaction attempting to commit must abort if any transaction at a lower-level committed while it was active, even if there is no actual conflict.

Several untrusted schedulers have been proposed. In [7], the authors suggest modifications to the optimistic scheduling protocol [20] for use in MLS/DBMS's. In [20], two variations on the optimistic approach are described; they are serial validation and parallel validation. Both approaches make use of at least one critical section. This presents a problem for an untrusted scheduler as a TCB cannot make this type of functionality available without introducing a covert channel. Dowling et al. describe a modification to the protocol that does not include critical sections but only briefly argue the correctness of the modified protocol.

Maimone and Greenberg have proposed two untrusted schedulers [23]. The first was implemented as part of Trusted Oracle. (An earlier report on some of this work can be found in [30].) The authors state that the default behavior of Oracle is to provide read-consistency [25], which is a weaker condition than serializability. The second scheduler proposed is based on the scheduling protocol proposed in [18].

In [15] the authors propose a scheduling protocol for Class 1-SS schedules (i.e., those that contain transactions able to write at a single level only) for use in replicated architectures. Replicated architectures dedicate a separate DBMS for each security level of data stored in the database. Each DBMS maintains the data classified at its own level along with a copy of the data stored at each level it dominates. An untrusted scheduler executes in each DBMS and schedules the transactions at that classification level. When a transaction commits, the TCB propagates the updates the transaction performed to other schedulers operating at higher classification levels. Each scheduler ensures the serializability of the schedule that it produces. To ensure consistency among the individual schedulers, the protocol requires that if transaction \( T_1 \) precedes \( T_2 \) in the equivalent serial ordering on a DBMS at some classification level \( l_4 \), then the update projections of \( T_1 \) must precede those of \( T_2 \) in the equivalent serial ordering on every DBMS's with classification levels dominating \( l_4 \). This scheduling protocol depends on some trusted code. Because of this the authors argue the scheduling protocol's security. The authors also sketch the proof of the scheduling protocol's correctness.

The protocol in [15] requires that update projections be routed along a topological ordering, which is consistent
with the partial ordering of security classification levels assigned to the DBMSs (the topological ordering extends the partial ordering of the security classification levels to a total order). Routing along the topological ordering is necessary for correctness. However, routing updates along the topological ordering can violate security if there is a pair of DBMSs that have incomparable classification levels.

Recently, Costich [4] proposed a scheduling protocol for replicated architectures that addresses this problem. The scheduler is untrusted except for a small "garbage collection" algorithm.

In [19] the Class 2-SS Multiversion Timestamp protocol is proposed. It is based on the multiversion timestamp protocol of [26]. The scheduler is shown to be correct and DC-Secure with respect to Class 2-SS schedules.

VI. CONCLUSION

We examined the security issues involved in database concurrency control for MLS/DBMSs. We have shown how a scheduler can affect security and introduced the concept of DC-Security, which implies a system is free of covert channels due to contention for access to data.

We presented a model of a scheduler and defined DC-Security in terms of the model. The definition of DC-Security is based on an application of noninterference to our scheduler model.

We introduced two properties of schedulers which constitute a necessary condition for DC-Security. These are Strong-Value-Security and Recovery-Security. We have introduced two other necessary conditions, Schedule-Security and Co-Schedule-Security. Each is easier to apply as it considers the behavior of the scheduler in response to one input only.

We have identified a class of schedulers we call Output-State-Equivalent. In these schedulers the behavior of the protocol controller is determined by the actions that have been output but not yet rolled back. For this class of scheduler, another criterion implies DC-Security. The criterion considers separately the behavior of the scheduler in response to schedulable and unschedulable inputs.

We also characterized the security of the Dynamic Two-Phase Locking, Conflict Graph, Multiversion Locking, Multiversion Timestamp and Optimistic scheduling protocols. Only the optimistic scheduler was found to be DC-Secure. All of the other protocols failed to be either Schedule-Secure or Co-Schedule-Secure. The results are shown in Table I.

We have shown that optimistic scheduling with serial validation is DC-Secure even through it will introduce a covert storage channel due to contention for the scheduler. We have not completely answered the question for optimistic scheduling with parallel validation. We hope to investigate this in the future.

A scheduler that is DC-Secure is free of covert channels due to contention for access to data. The issue of covert channels due to contention for the scheduler needs to be addressed. In analyzing the scheduler we assume it can output multiple actions without delay. This assumption masks out channels arising within the scheduler. This may be a reasonable approximation in the case of scheduling actions. When we consider the execution of these actions as well, this approximation is probably not adequate. We need to consider covert channels that can result due to these delays.

ACKNOWLEDGMENT

The authors wish to thank the anonymous referees, whose constructive comments greatly improved this paper.

REFERENCES


W. T. Tsai (S'81-M'85) received the M.S. degree in 1982 and the Ph.D. degree in 1985, both in computer science from University of California at Berkeley, and the S.B. degree in 1979 in computer science and engineering from MIT.
He is currently an associate professor with the Department of Computer Science, University of Minnesota, Minneapolis. His research interests are software engineering, computer systems, and computer security.
Dr. Tsai serves as an Associate Editor for the Journal of Software Maintenance, the International Journal on Software Engineering and Knowledge Engineering, and the International Journal on AI Tools.

Jaideep Srivastava received the B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur, India, in 1983, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1985 and 1988, respectively.
Since 1988 he has been an Assistant Professor with the Computer Science Department, University of Minnesota, Minneapolis. In 1983 he was a research engineer with Upton Digital Systems, Lucknow, India. He has published over 35 papers in journals and refereed conferences in the areas of databases, networks, and artificial intelligence. His research interests lie in the areas of the active and real-time databases, parallel databases, federated databases, parallel production systems, and database support for molecular biology and manufacturing applications.
Dr. Srivastava is a member of the IEEE Computer Society and the Association for Computing Machinery. His professional activities have included being on various program committees, and refereeing for journals, conferences, and the National Science Foundation.