A Formal Protection Model of Security in Centralized, Parallel, and Distributed Systems

GLENN S. BENSON, IAN F. AKYILDIZ AND WILLIAM F. APPELBE
Georgia Institute of Technology

One way to show that a system is not secure is to demonstrate that a malicious or mistake-prone user or program can break security by causing the system to reach a nonsecure state. A fundamental aspect of a security model is a proof that validates that every state reachable from a secure initial state is secure. A sequential security model assumes that every command that acts as a state transition executes sequentially, while a concurrent security model assumes that multiple commands execute concurrently. This paper presents a security model called the Centralized-Parallel-Distributed model (CPD model) that defines security for logically, or physically centralized, parallel, and distributed systems. The purpose of the CPD model is to define concurrency conditions that guarantee that a concurrent system cannot reach a state in which privileges are configured in a nonsecure manner. As an example, the conditions are used to construct a representation of a distributed system.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, scheduling, synchronization*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants, specification techniques*

General Terms: Design, Security

Additional Key Words and Phrases: Access control, concurrency control, distributed system security, operating system security, protection model

1. INTRODUCTION

A security model has two components: a security predicate and a model of computation. A security predicate is a Boolean function that is satisfied by a state of privileges only if the state is secure. An example security predicate is the conjunction of the Bell–La Padula *ss-property*, **-property*, and *ds-property* [7],

The work of I. F. Akyildiz was supported in part by the National Computer Security Center under grant MDA 904-90-C-7030.
Authors' current addresses: G. S. Benson, Trusted Information Systems, 3060 Washington Road, (Rt. 97), Glenwood, MD 21738; I. F. Akyildiz and W. F. Appelbe, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

which describes privileges for accessing information. A model of computation is a state machine that moves between states by executing *commands* from a *command set*, where each command is a sequence of atomic *operations*.

This paper presents the Centralized-Parallel-Distributed model (CPD model), a *privilege-based* security model whose state is defined in terms of privileges, where a privilege is either a permission to access data (e.g., an access right, or a privilege to continue execution, for instance, a synchronization condition). Most privilege-based models [7, 12, 26, 27], represent a centralized, sequential system; yet, the CPD model provides concurrency and ensures security for every state reachable from a secure initial state. If concurrency were not controlled in the CPD model, then concurrently executing commands could potentially interleave their operations and reach a nonsecure state. For example, multiple invocations of a command that trades high-sensitivity-level privileges for low-sensitivity-level privileges, if not correctly executed, could potentially erroneously yield an intermediate state that grants both high and low sensitivity-level privileges simultaneously.

The purpose of the CPD model is to formally guarantee that the problem of proving security for a concurrent model is *reducible* to the problem of proving security for a sequential model, where solutions to the sequential problem are well known (e.g., [7, 10, 12, 26, 27, 40]). A polynomial time test is presented that is satisfied only if a given concurrent model can be *reduced* to a sequential model. A command is called *security-preserving* (S_PRES) if the command yields a secure state when given a secure state as input; a command is called *sequential-security-preserving* (SS_PRES) if the command yields no nonsecure intermediate or final state when given a secure initial state as input. Consider, for example, a command that first yields a nonsecure state and then yields a secure state. In this case the command is S_PRES, but not SS_PRES. The CPD model reduces a distributed model to a sequential (SS_PRES) model by proving that if every command in a given command set satisfies the CPD model *reduction conditions* (i.e., *nested critical section condition* and *least privilege condition*), then concurrent execution is *reducible* to sequential execution.

The nested critical section condition stipulates that every command must nest its critical sections. A *critical section* is a sequence of operations in a command that executes sequentially with respect to other interfering commands [8]. In the CPD model, a critical section's entry and exit is implemented by a *lock*, where the purpose of the lock is to guarantee mutual exclusion. The purpose of the nested critical section condition is to provide *serializability*, where the property of nested critical sections that ensures serializability is two-phase locking.[1]

Although serializability is a desired attribute in a model, serializability does not necessarily imply that concurrent model is *reducible* to a sequential model. Consider, for example, a simple Generic security model, G model, whose commands are serializable and SS_PRES. The syntax of the model of computation of the G model provides two operations, *a* and *r*, whose semantics are that a privilege is (*a*)*cquired* and (*r*)*eleased*, respectively. An example command set of

[1] Two-phase locking exists if locks are taken and released in two phases. "In the first phase, locks are acquired but not released. In the second phase, locks are released but not acquired" [41].

the G model is as follows:

$$\{c_1(w, x, y, z) = a_1(w)r_1(x)a_1(y)a_1(z),$$
$$c_2(w, x, y, z) = r_2(w)a_2(x)r_2(y)a_2(w)a_2(z)\}.$$

Command c_1 accepts four formal parameters, w, x, y, and z, and executes operations that acquire access to w, release access to x, acquire access to y, and acquires access to z. Command c_2 accepts four parameters and executes five operations. Consider a state with four resources: (h)ost, (s)ecure printer, (t)ape drive, and (d)isk drive, and the command invocations $c_1(t, s, h, d)$ and $c_2(h, s, t, d)$ whose operations are instantiated with the actual parameters, respectively, as follows:[2]

(i) $a_1(t)r_1(s)a_1(h)a_1(d)$
(ii) $r_2(h)a_2(s)r_2(t)a_2(h)r_2(d)$.

Further consider an example security predicate that prohibits access to all four resources simultaneously. Given this security predicate, both of the commands are SS_PRES if the initial state has no privileges. However, if the commands are executed concurrently, the following sequence could be executed:

$$\underbrace{r_2(h)a_1(t)r_1(s)a_1(h)a_1(d)}_{\text{not secure}}\underbrace{a_2(s)r_2(t)a_2(h)r_2(d)}_{\text{secure and serializable}}$$

In the sequence,[3] after the sixth state transition, privileges for all four resources have been acquired, which is defined by the security predicate as a security violation. However, after the sequence completes, the final state is the same as the one produced by command (i) followed by command (ii)—privileges for the printer and host are acquired, but privileges for the tape and disk are not.

The concurrent history is not secure because there exists a state reachable from a secure initial state that is not secure. The problem with the command set is that it does not satisfy the *least privilege condition*. "Subjects should be given no more privilege than is necessary to enable them to do their jobs. In that way the damage caused by erroneous or malicious software is limited" [21]. Any command that acquires new privileges and releases old privileges, in effect, trades old privileges for new ones. The least privilege condition in the CPD model stipulates that every transition must release all of its old privileges, before acquiring any of its new ones. Neither transition in the example above adheres to this condition.

The scope of this paper concerns demonstrating security for reachable states. However, there exist aspects of security in privilege-based systems beyond the scope of this paper. For example, we do not address liveness concerns, which may arise in a definition of secure auditing, for instance, [48]. Also, we do not describe transition constraints [21, 30, 37], that is, "constraints that hold for the

[2] These sequences are called *command histories*, and are defined in Section 2.
[3] This sequence is called a *concurrent history*, and is formally defined in Section 2.

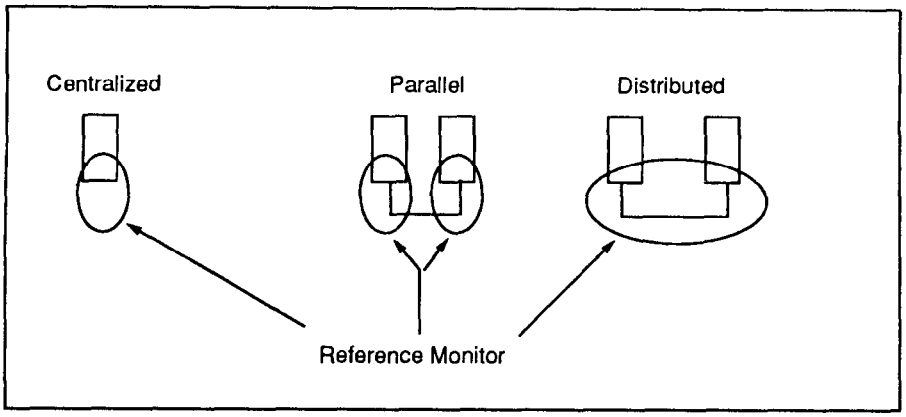

Fig. 1. Secure architectures.

relation between secure states, and hence, can be checked only by comparing two or more states" [30]. An example transition constraint from the Military Message System model is that "no classification ranking can be downgraded except with the role of downgrader who has invoked a downgrade operation" [30].

The purpose for providing concurrency is to formally represent a centralized, a parallel, or a distributed Trusted Computing Base (TCB)—the portion of the system that maintains the state of privileges. As shown in Figure 1, a centralized TCB resides on a single processor and may either be sequential or multiprogrammed. A parallel TCB resides on multiple processors that access a common clock and storage and communicate via shared memory. A distributed TCB resides on multiple processors that do not access a common clock or storage and exchange information via communication lines [42]. The TCB state (i.e., the collection of privileges) is centralized in the case of a centralized or parallel TCB, and is distributed in the case of a distributed TCB. Security-relevant facilities in the TCB execute by concurrently updating the TCB state.

The remainder of this paper is organized as follows. Section 2 defines the CPD model and proves that the reduction conditions guarantee that a concurrent command set is *reducible* to a sequential command set. In other words, Section 2 proves that if every command in a command set both satisfies the reduction conditions and is SS_PRES, then every state reachable from a secure initial state is secure. Section 3 evaluates the CPD model with respect to other related security models. Privilege-based models are compared with others to show that the CPD model is useful for defining both *nondisclosure* and *integrity* for centralized, parallel, and distributed TCBs. Section 4 gives an example distributed model that satisfies the reduction conditions. Finally, Section 5 concludes the paper.

2. CPD MODEL

The CPD model is a general-purpose security model that represents centralized, parallel, and distributed TCBs, independently of a particular security policy or environment. This section formally defines the model of computation and a security predicate.

Table I. Notation

	Individuals	Sequences
Formal	operation (p_a) operation set (P_a)	command (c_k) command set (C_j)
Actual	instruction (i_a) instruction set (I_a)	history (h_k) history set (H_j)

Table II. Example Notation

	Individuals	Sequences
Formal	$a_1(w)$ $\{a_1(w), r_1(x)\}$	$a_1(w)r_1(x)a_1(y)a_1(z)$ $\{c_1, c_2\}$
Actual	$a_1(t)$ $\{a_1(t), r_1(s)\}$	$a_1(t)r_1(s)a_1(h)a_1(d)$ $\{h_1, h_2\}$

The CPD model is expressed in terms of operations, operation sets, instructions, instruction sets, commands, command sets, histories, and history sets, as shown in Table I. The first two rows of Table I list notation for items expressed in terms of formal parameters, and the latter two rows notation for items expressed in terms of actual parameters. The first column defines individuals and the second defines sequences. An operation p_a is a parameterized state transition; a command c_k is a sequence of operations; an operation set P_a is a set of operations;[4] and a command set C_j is a set of commands. An instruction i_a is an operation that has been instantiated with an actual parameter. A history h_k is a sequence of instructions. An instruction set I_a and history set H_j are sets of instructions and histories, respectively.

For example, consider the G model of Section 1. The first row of the first column of Table II depicts an *operation*, with formal parameter w. An instantiation of this operation with actual parameter t is shown in the *instruction* in the third row of the first column. An instruction, i_a, is normally denoted with the same subscript as its corresponding operation. The second and fourth rows of the first column illustrate an operation set and instruction set, respectively. The first row of the second column denotes a *command*. When clear from the context, as an abbreviated notation, a command may be written as c_k. The second row of the second column is an abbreviated notation that represents the following command set:

$$\underbrace{\{a_1(w)r_1(x)a_1(y)a_1(z),}_{c_1} \quad \underbrace{r_2(w)a_2(x)r_2(y)a_2(w)a_2(z)\}}_{c_2}$$

The third row of the second column denotes a history. Although any sequence of instructions is a history, for clarity, different kinds of histories are denoted with

[4] Operation sets and instruction sets are not used in this paper, and are only defined here for completeness of Table I.

different notations. A *command history* is the history that corresponds to a particular command where each instruction is an instantiation of its corresponding operation. A command history is denoted by h_k where k is the subscript of the corresponding command. For example, the third row of the second column denotes a command history h_1. A *concurrent history* is an interleaving of instructions from multiple command histories, for example,

$$r_2(h)a_1(t)r_1(s)a_1(h)a_1(d)a_2(s)r_2(t)a_2(h)r_2(d).$$

Concurrent histories and arbitrary histories are denoted by h, h', h'', and so on. The fourth row of the second column denotes a history set (denoted by H_j). A history set, H_j, that contains only command histories is called a *command history set*. The notation $|c_k|$, $|C_j|$, $|h_k|$, and $|H_k|$ denote the number of operations in c_k; the number of non-null commands in C_j; the number of instructions in h_k; and the number of non-null command histories in H_k, respectively.

The set of all states is denoted by $\mathscr{M}$, and the undefined state is denoted by $\perp$. The symbol $\underline{\mathscr{M}}$ denotes the union of $\mathscr{M}$ and $\perp$.

$$\underline{\mathscr{M}} \stackrel{\text{def}}{=} \mathscr{M} \cup \{\perp\}.$$

The set of all atomic operations and instructions are denoted by $\mathscr{P}$ and Ψ, respectively, for example, $P_a \subseteq \mathscr{P}$ for any P_a, and $I_a \subseteq \Psi$ for any I_a. The deterministic transition function, τ, defines a transition, as follows:

$$\tau \mid \Psi \times \underline{\mathscr{M}} \rightarrow \underline{\mathscr{M}}.$$

A behavior is a sequence:

$$M_0 \stackrel{i_1}{\mapsto} \tau(i_1, M_0) \stackrel{i_2}{\mapsto} \tau(i_2, \tau(i_1, M_0)) \stackrel{i_3}{\mapsto} \tau(i_3, \tau(i_2, \tau(i_1, M_0))) \ldots$$

where i_a is the ath instruction in the behavior, $\tau(i_a, \tau(i_{a-1}, \ldots, \tau(i_1, M_0)))$ is the ath state reached by the behavior, and M_0 is an initial state. Lamport conjectures that "the behavior of every discrete system, be it hardware or software, can be formally represented as such a sequence [behavior]" [29]. Since the transition function is deterministic, a behavior is uniquely determined by a history and an initial state. For example, the behavior given above is uniquely determined by the history, $h = i_1 i_2 i_3$, and the initial state M_0.

The model of computation consists of a nondeterministic *front end* and a deterministic *state machine*, as shown in Figure 2.

Input into the front end is a command history set, and output from the front end is a single concurrent history. The front end executes by computing concurrent (interleaved) histories (formally defined in Definition 1 of Sect. 2.3) from the input command history set and nondeterministically choosing for output one of the computed concurrent histories. Subsequently, the concurrent history is input into the state machine which sequentially executes the transition function τ. The purpose of the front end is to formally define every concurrent history that may potentially be executed by the state machine, and the purpose of the state machine is to represent machine execution.

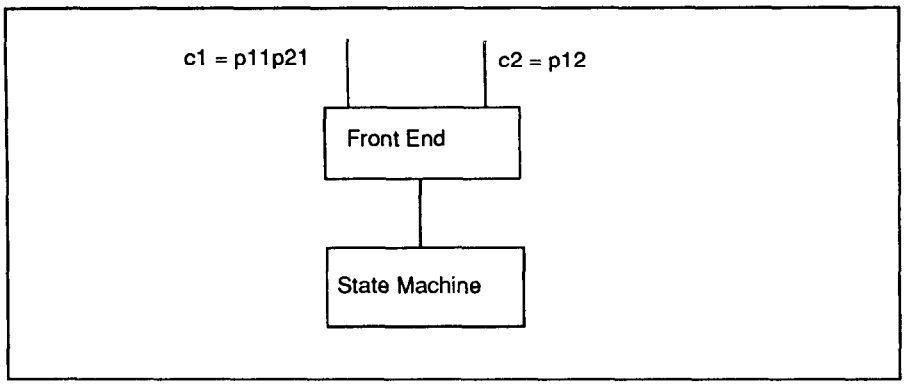

Fig. 2. Example two-command model of computation.

For example, the model of computation shown in Figure 2 corresponds to the following concurrent program.

cobegin
  i_{1_1}
  i_{2_1}
□
  i_{1_2}
coend

The command history set H_1 in the example is $\{i_{1_1}, i_{2_1}, i_{1_2}\}$. The set of potential concurrent histories output by the front end for this command history set is as follows:

(i) $i_{1_1} i_{2_1} i_{1_2}$
(ii) $i_{1_1} i_{1_2} i_{2_1}$
(iii) $i_{1_2} i_{1_1} i_{2_1}$

The front end nondeterministically chooses one of the three concurrent histories. If, for example, the front end chooses concurrent history (i) and the initial state is M_0, then the state machine reaches $\tau(i_{1_1}, M_0)$, $\tau(i_{2_1}, \tau(i_{1_1}, M_0))$, and $\tau(i_{1_2}, \tau(i_{2_1}, \tau(i_{1_1}, M_0)))$, respectively.

The purpose of a command set is to represent a set of TCB utilities, and the purpose of a command history is to represent a particular thread of execution through the TCB. For example, Figure 2 represents two concurrent threads of execution, where one thread executes an invocation of the TCB utility represented by the command history h_1, and the other thread executes an invocation of the TCB utility represented by the command history h_2.

A primary difference between the CPD model and sequential models is that sequential models do not provide synchronization. A CPD model instruction i_a is *enabled* in state M_x if and only if $\tau(i_a, M_x) \neq \perp$. An *enabling condition* is a Boolean function that is satisfied by instruction i_a and state M_x if and only if i_a is *enabled* in M_x. An enabling condition for each CPD model operation is presented in

Section 2.2. The CPD model front end nondeterministically chooses a concurrent history only if the state machine executes enabled instructions.

For example, suppose, in the command history set described above, i_{2_1} is not enabled in $\tau(i_{1_1}, M_0)$, that is, $\tau(i_{2_1}, \tau(i_{1_1}, M_0)) = \perp$. In this case, concurrent history (i) cannot be chosen by the front end. The front end does not represent any implementable system because it predicts the execution of the state machine a priori. However, by definition, the front end describes all potential execution histories of an implemented system, provided correct implementation of synchronization, that is, the implemented system does not execute nonenabled instructions.

2.1 Syntactic Definition

The CPD model consists of:

(i) *A set of tokens*: Every token has a *type*. Every type has a *class* where there are a bounded number of classes. However, there may be an unbounded number of tokens of a given type. The type of token *tok* is denoted by type (tok); and the class of token *tok* is denoted by class(tok). The predefined class *lock* is used for synchronization. A token whose type is of class lock is denoted by l_j. Another predefined class is *index*, which is used for indexing into the state. A token whose type is of class index may be denoted by *row*, *col*, or x_a. Classes that are not predefined have semantics specific to the represented system. For example, an instantiation-specific class that is ignored by the security predicate is described in [10].

(ii) *A finite set of commands*: A command is of the form:

$$\begin{aligned}&\text{command } c_k(\text{tok}_1{:}\text{type}_{\text{tok}_1}, \ldots, \text{tok}_n{:}\text{type}_{\text{tok}_n}) =\\ &\quad p1\\ &\quad p2\\ &\quad \ldots\\ &\quad p_{|c_k|}\end{aligned}$$

Here, c_k is a name and n is a constant. Each formal parameter tok_j is a token of type $\text{type}_{\text{tok}_j}$. Each p_a for $a = 1, \ldots, |c_k|$ is one of the following operations, where the indices of p_a are sequentially ordered natural numbers beginning with 1.

```
enter(tok,row,col)
delete(tok,row,col)
present(tok,row,col)
absent(tok,row,col)
```

A command is a sequence of operations written as $c_k = p_1, \ldots, p_{|c_k|}$ which means command c_k is the operation sequence $p_1, \ldots, p_{|c_k|}$. A state M_x is either the undefined state $\perp$ or a two-dimensional matrix indexed by tokens of class index. The operations *enter* and *delete* put and remove a token into and from the M_x[row,col] coordinate of the state matrix, respectively. The operations *present* and *absent* determine the existence and nonexistence of tok in the M_x[row,col] coordinate of the state matrix, respectively. The *coordinate* (a row and column of the state) referenced in operation p_a or instruction i_a is denoted by $coord(p_a)$

Table III. Operation and Instruction Components

p_a = enter(tok, row, col)	i_a = enter(r_1, x_2, x_3)
coord(p_a) = [row, col]	coord(i_a) = [x_1, x_2]
op(p_a) = *enter*	op(i_a) = *enter*
token(p_a) = tok	token(i_a) = r_1

or *coord*(i_a), respectively. The *kind* of operation or instruction (*enter*, *delete*, *present*, or *absent*), of p_a or i_a is denoted by op(p_a) and op(i_a), respectively. The token of an operation or instruction is denoted by *token*. Example applications of the component definitions are presented in Table III. Consider, for example a system with two hosts, one shared disk and two access rights (read and write). Only one host may access the disk at a time. A possible model of the system contains five types: host, disk, read_t, write_t, and mutex_t of classes, *index*, *index*, *right*, *right*, and *lock*, respectively (the class *right* is specific to this representation). Assume that two host tokens and one token from each of the other types are defined. The command set may contain a command c_k that represents a TCB utility that ensures that only one host has disk access at a time. The command c_k provides mutual exclusion by executing the *enter* and *delete* operations on the lock l_1 of type mutex_t.

```
command c_k(row:host, col:disk, r_1:read_t, r_2:write_t) =
  enter(l_1, row,col)
    absent(r_1, row,col)
    enter(r_1, row,col)
    enter(r_2, row,col)
    delete(r_1, row,col)
    delete(r_2, row,col)
  delete(l_1, row,col)
```

The first and last operations in c_k reference a lock used to provide mutual exclusion for the remainder of the command's operation sequence (the formal semantics of locks is presented in Section 2.2, below). The second operation checks that r_1 is not in the [*row,col*] coordinate of the state matrix, and the remaining operations first *enter* and then *delete* r_1 and r_2 in the state matrix.

2.2 Semantic Definition

For a given instruction i_a, the form of τ is given below:

$$\tau(i_a, M_x) = \begin{cases} \perp & \text{if } (M_x = \perp) \\ t(i_a, M) & \text{else if } (\text{enabled}(i_a, M)) \\ \perp & \text{otherwise} \end{cases}$$

where *enabled* is a Boolean function (enabled and t are to be defined subsequently). In other words, the semantics of τ is that if an operation is enabled in the current state, then τ returns the result of a state transition t; otherwise, τ returns the error symbol $\perp$. Note that the semantics of the instructions ensures that the only state reachable from $\perp$ is itself.

The CPD model operations are as follows:

(i) **enter**(tok,row,col)

τ(enter(tok,row,col), M_x) =

$$\begin{cases} \perp & \text{if } M_x = \perp \\ (t(\text{enter(tok,row,col)}, M_x) & \\ \quad \text{else if } (\text{class(tok)} \neq \text{lock} \vee \text{tok} \notin M_x[\text{row,col}]) & \\ \perp & \text{otherwise} \end{cases}$$

where

$\forall$ row′,col′ t(enter(tok,row,col), M_x)[row′,col′] =

$$\begin{cases} M_x[\text{row}',\text{col}'] & \text{if } ((\text{row}' \neq \text{row}) \vee (\text{col}' \neq \text{col})) \\ M_x[\text{row}',\text{col}'] \cup \{\text{tok}\} & \text{otherwise} \end{cases}$$

This operation puts tok in M_x[row,col]. If tok is a lock, then the operation blocks until tok is not in M_x[row,col]. This implies t(enter(tok,row,col), M_x) $\neq M_x$ if $M_x \neq \perp$ and class(tok) = lock.

(ii) **delete**(tok,row,col)

τ(delete(tok,row,col), M_x) =

$$\begin{cases} \perp & \text{if } M_x = \perp \\ t(\text{delete(tok,row,col)}, M_x) & \\ \quad \text{else if } (\text{class(tok)} \neq \text{lock} \vee \text{tok} \in M_x[\text{row,col}]) & \\ \perp & \text{otherwise} \end{cases}$$

where

$\forall$ row′,col′ t(delete(tok,row,col), M_x)[row′,col′] =

$$\begin{cases} M_x[\text{row}',\text{col}'] & \text{if } ((\text{row}' \neq \text{row}) \vee (\text{col}' \neq \text{col})) \\ M_x[\text{row}',\text{col}'] - \{\text{tok}\} & \text{otherwise} \end{cases}$$

This operation removes tok from M_x[row,col]. If tok is a lock, then the operation blocks until tok is in M_x[row,col]. This implies t(delete(tok, row,col), M_x) $\neq M_x$ if $M_x \neq \perp$ and class(tok) = lock.

(iii) **present**(tok,row,col)
This operation blocks if and only if tok $\notin M_x$[row,col].

τ(present(tok,row,col), M_x) =

$$\begin{cases} \perp & \text{if } M_x = \perp \\ M_x & \text{else if } \text{tok} \in M_x[\text{row,col}] \\ \perp & \text{otherwise} \end{cases}$$

(iv) **absent**(tok,row,col)
This operation blocks if and only if tok $\in M_x$[row,col].

τ(absent(tok,row,col), M_x) =

$$\begin{cases} \perp & \text{if } M_x = \perp \\ M_x & \text{else if tok} \notin M_x[\text{row,col}] \\ \perp & \text{otherwise} \end{cases}$$

The semantics of an example command is given below.

command c_k(x_1:host, l_1:disk_lock, l_2:request_lock, l_3:buff_lock) =
delete(l_1, x_1, 4)
enter(x_1, 6, 7)
enter(l_2, 6, x_1)
absent(r_1, 3, 4)
delete(l_3, 3, 4)

Command c_k accepts four formal parameters (x_1, l_1, l_2, l_3), of types host, disk_lock, request_lock, and buff_lock, respectively. The class of the host type is *index*, and the classes of disk_lock, request_lock, and buff_lock are *lock*. The token r_1 is a constant whose class is *right*. The command waits until l_1 may be deleted from $M_x[x_1, 4]$ (M_x is the current state and M_{x+a} is the resultant state after a instructions have been executed). Next, index x_1 is entered into $M_{x+1}[6, 7]$. The command then waits until l_2 may be entered into $M_{x+2}[6, x_1]$. Next, the command waits until r_1 is not an element of $M_{x+3}[3, 4]$. The command then waits until l_3 can be deleted from $M_{x+4}[3, 4]$. Assuming that the command runs to completion, during which no concurrent commands are executing, the resultant matrix has three, four, or five changes with respect to the original matrix, depending on whether or not x_1 and r_1 are in the original matrix.

As the example illustrates, the classes of tokens, *index* and *lock* correspond to indices into the state matrix and synchronization locks, respectively. Other representation-specific classes, such as *right*, could potentially represent privileges, degrees of trust, inheritance, or other TCB-specific semantics. The next section shows how distinct command histories interleave their instruction to provide concurrency and how lock tokens are used for synchronization.

2.3 Scheduler

Concurrency is defined as an interleaving of the instructions in distinct commands histories. In a sequential environment, each command history is a state transition. In a parallel or distributed system, however, command histories are not atomic, so each interleaving of command instructions is a sequence of state transitions.

Execution of history h is given by the scheduler $\mathcal{T}$, which sequentially applies the instructions in the history to the state.

$$\mathcal{T}(h, M_x) = \begin{cases} M_x & \text{if } |h| = 0 \\ \mathcal{T}(\text{rest}(h), \tau(\text{first}(h), M_x)) & \text{otherwise} \end{cases}$$

where if $|h| > 0$, the first instruction in h is denoted by first(h), and the remainder of h is denoted by rest(h), and if $|h| = 0$, then first(h) is undefined and rest(h) = h. For example, for $h_j = i_1 i_2 i_3$, first(h_j) = i_1 and rest(h_j) = $i_2 i_3$.

Definition 1. Multiple command histories can be executed concurrently by interleaving the instructions in the command histories. The set of all possible concurrent histories generated from a command history set H_j is an interleaved set (iset).

$$\text{iset}(H_j) = \{h \mid \text{is\_iset}(H_j, h)\}$$

where

$$\text{is\_iset}(H_j, h) = \begin{cases} \text{true} & \text{if } |h| = 0 \wedge |H_j| = 0 \\ \text{true} & \text{if } \exists h_k \in H_j \text{ first}(h_k) = \text{first}(h) \wedge \\ & \quad \text{is\_iset}(((H_j - \{h_k\}) \cup \{\text{rest}(h_k)\}), \text{rest}(h)) \\ \text{false} & \text{otherwise} \end{cases}$$

In other words, the *iset* of a command set is a *concurrent history set.* Iset is defined recursively, where, for each h in iset, first(h) is equal to the first instruction in some element h_k of H_j. An interleaving contains all the instructions in the histories and preserves the relative ordering of instructions. For example, let $H_1 = \{i_{1_1} i_{2_1}, i_{1_2} i_{2_2}\}$, then

$$\text{iset}(H_1) = \{i_{1_1} i_{2_1} i_{1_2} i_{2_2}, i_{1_1} i_{1_2} i_{2_1} i_{2_2}, i_{1_1} i_{1_2} i_{2_2} i_{2_1}, i_{1_2} i_{1_1} i_{2_1} i_{2_2}, i_{1_2} i_{1_1} i_{2_2} i_{2_1}, i_{1_2} i_{2_2} i_{1_1} i_{2_1}\}.$$

Definition 2. The *schedule set* (scheduleset) of a command set and an initial state is the largest subset of the iset in which $\perp$ is not reached.

$$\text{scheduleset}(H_j, M_x) = \{h \in \text{iset}(H_j) \mid \mathscr{T}(h, M_x) \neq \perp\}.$$

For example, if i_{1_2} is not enabled in $\tau(i_{1_1}, M_x)$, then

$$\text{scheduleset}(H_1, M_x) = \{i_{1_1} i_{2_1} i_{1_2} i_{2_2}, i_{1_2} i_{1_1} i_{2_1} i_{2_2}, i_{1_2} i_{1_1} i_{2_2} i_{2_1}, i_{1_2} i_{2_2} i_{1_1} i_{2_1}\}.$$

While the *iset* defines all possible concurrent histories, the *scheduleset* defines only those concurrent histories that may be chosen by the front end. In other words, the set difference between the iset and the scheduleset is the set of histories that execute blocked instructions.

Serializable histories are defined next. The serializability definition is in terms of the permutation (*perm*) of a command history set.

Definition 3. The set of *permutations* of a command history set H_j is called a perm.

$$\text{perm}(H_j) = \{h \mid \text{is\_perm}(h, H_j)\}$$

where

$$\text{is\_perm}(h, H_j) = \begin{cases} \text{true} & \text{if } |h| = 0 \wedge |H_j| = 0 \\ \text{true} & \text{else if } \exists h_k, h' \; h = h_k h' \wedge h_k \in H_j \wedge \text{is\_perm}(h', H_j - \{h_k\}) \\ \text{false} & \text{otherwise} \end{cases}$$

where $h_k h_l$ denotes the history h_l appended onto the end of history h_k. For example, let $H_j = \{h_1, h_2, h_3\}$, then

$$\text{perm}(H_j) = \{h_1h_2h_3,\ h_1h_3h_2,\ h_2h_1h_3,\ h_2h_3h_1,\ h_3h_1h_2,\ h_3h_2h_1\}.$$

Definition 4. A command history set H_j is serializable if and only if every scheduleset history h that does not reach $\perp$ yields the same final state as if the commands were executed in some serial order.

$$\begin{aligned}&\text{serializable}(H_j) \text{ iff}\\ &\quad \forall M_x \in \mathscr{M}\ \forall h \in \text{scheduleset}(H_j, M_x)\ \exists h' \in \text{perm}(H_j)\\ &\quad \mathscr{T}(h, M_x) = \mathscr{T}(h', M_x)\end{aligned}$$

For each M_x in which $\text{scheduleset}(H_j, M_x) = \varnothing$, $\text{serializable}(H_j)$ is vacuously satisfied. Otherwise, for each element of $\text{scheduleset}(H_j, M_x)$, there must exist some sequential schedule of commands that returns the same final state.

The serializability conditions are satisfied whenever all critical sections are nested and all instructions in distinct command histories that reference common coordinates are in *shared critical sections*. A critical section is a mutually exclusive sequence of instructions. Distinct instructions that reference common coordinates are called *interfering* instructions, and distinct histories that contain interfering instructions are called *interfering* histories.

Definitions 5–7 reference histories and instructions, but may also be applied to commands and operations. As ancillary functions *last* and *start* define the last instruction, and all but the last instruction in a history, respectively. Both *last* and *start* are undefined for the null history and are defined in terms of *reverse*, the instructions of a history in reverse order.

$$\begin{aligned}\text{last}(h) &= \text{first}(\text{reverse}(h))\\ \text{start}(h) &= \text{reverse}(\text{rest}(\text{reverse}(h)))\end{aligned}$$

where

$$\text{reverse}(h) = \begin{cases} h & \text{if } |h| = 0\\ \text{reverse}(\text{rest}(h))\text{first}(h) & \text{otherwise}\end{cases}$$

The ancillary function get_lock extracts all the instructions from a history that reference locks.

$$\text{get\_lock}(h) = \begin{cases} h & \text{if } |h| = 0\\ \text{first}(h)\text{get\_lock}(\text{rest}(h)) & \text{else if } \text{class}(\text{first}(h)) = \text{lock}\\ \text{get\_lock}(\text{rest}(h)) & \text{otherwise}\end{cases}$$

Definition 5. A history h is nested if and only if all critical sections are nested.

$$\text{nest}(h) = \begin{cases}\text{true} & \text{if } \text{is\_nest}(\text{get\_lock})(h))\\ \text{false} & \text{otherwise}\end{cases}$$

where

$$\text{is\_nest}(h) = \begin{cases}\text{true} & \text{if } (\text{coord}(\text{first}(h)) = \text{coord}(\text{last}(h))) \wedge\\ & \quad (\text{op}(\text{first}(h)) = \text{enter} \wedge \text{op}(\text{last}(h)) = \text{delete}) \wedge\\ & \quad \text{is\_nest}(\text{rest}(\text{start}(h)))\\ \text{false} & \text{otherwise}\end{cases}$$

Definition 6. The *critical sections* (crit) of an instruction in a history is a set. Each element of crit is a locked coordinate.

$$\text{crit}(h, i_a) = \text{is\_crit}(h, i_a, \varnothing)$$

$$\text{is\_crit}(h, i_a, S) =$$

$$\begin{cases} \varnothing & \text{if} \quad |h| = 0 \\ S & \text{else if} \quad \text{first}(h) = i_a \wedge \\ & \qquad \text{class}(\text{first}(h)) \neq \text{lock} \\ S \cup \{\text{coord}(\text{first}(h))\} & \text{else if} \quad \text{first}(h) = i_a \wedge \\ & \qquad \text{class}(\text{first}(h)) = \text{lock} \wedge \\ & \qquad \text{op}(\text{first}(h)) = \text{enter} \\ \text{is\_crit}(\text{rest}(h), i_a, S) & \text{else if} \quad \text{class}(\text{first}(h)) \neq \text{lock} \\ \text{is\_crit}(\text{rest}(h), i_a, S \cup \{\text{coord}(\text{first}(h))\}) & \text{else if} \quad \text{op}(\text{first}(h)) = \text{enter} \\ \text{is\_crit}(\text{rest}(h), i_a, S - \{\text{coord}(\text{first}(h))\}) & \text{else if} \quad \text{op}(\text{first}(h)) = \text{delete} \end{cases}$$

For example, consider the history h, given in the following, with critical sections marked by the braces.

$$h = \underbrace{i_1 \underbrace{i_2 i_3 i_4} i_5}$$

Here,

$$\text{crit}(h, i_1) = \text{crit}(h, i_2) = \text{crit}(h, i_5) = \{\text{coord}(i_1)\}$$
$$\text{crit}(h, i_3) = \text{crit}(h, i_4) = \{\text{coord}(i_1), \text{coord}(i_3)\}$$

Definition 7. A command history set has *proper critical sections* (pcs) if every pair of interfering instructions from distinct command histories have a common critical section.

$$\text{pcs}(H_j) = \begin{cases} \text{true} & \text{if} \quad \text{nest}(H_j) \wedge \forall h_k, h_l \in H_j \quad h_k \neq h_l \Rightarrow \\ & \qquad \forall (i_a, i_b) \in \text{interfere}(h_k, h_l) \\ & \qquad\qquad ((\text{crit}(h_k, i_a) \cap \text{crit}(h_l, i_b)) \neq \varnothing) \\ \text{false} & \text{otherwise} \end{cases}$$

where

$$\text{interfere}(h, h') = \{(i_a, i_b) \mid i_a \in \text{in}(h) \wedge i_b \in \text{in}(h') \wedge \text{coord}(i_a) = \text{coord}(i_b)\}$$

and

$$\text{in}(h) = \begin{cases} \varnothing & \text{if} \quad |h| = 0 \\ \text{in}(\text{rest}(h)) \cup \{\text{first}(h)\} & \text{otherwise} \end{cases}$$

Nested critical sections provide *dynamic two-phase locking*: "Lock each entity accessed by the transaction immediately before the corresponding action; release all locks immediately following the last step of the transaction" [41]. The theorem that dynamic two-phase locking ensures serializability is given by Papadimitriou in [41], and is utilized in our model.

THEOREM 1. *If a command history set H_j is pcs, then the command history set is serializable.*

$$pcs(H_j) \Rightarrow serializable(H_j).$$

A polynomial time validation algorithm that tests if the conditions of Theorem 1 are satisfied is straightforward and, for brevity, its details are omitted here.

In general, the number of possible states that result from interleaved executions is factorial in the number of operations in a given set of commands. The $|\,\text{iset}(H_j)\,|$ is the upper bound of the number of different schedules because $\text{iset}(H_j)$ is all possible interleavings of the commands. Any algorithm that verifies security by checking every schedule must, as an upper bound, check $|\,\text{iset}(H_j)\,|$ different schedules.

Definition 8. The $\text{size}(H_j)$ is the total number of operations in the commands in H_j.

$$\text{size}(H_j) \stackrel{\text{def}}{=} \sum_{h_k \in H_j} |\,h_k\,|$$

The $|\,\text{iset}(H_j)\,|$ is given by the following formula:[5]

$$|\,\text{iset}(H_j)\,| = \frac{(\text{size}(H_j))!}{\prod_{h_k \in H_j}(|\,h_k\,|!)}.$$

Since the formula is exponential in $\text{size}(C_j)$, in many cases it may not be practical to validate security by enumerating every element of *iset*. The formula for $\text{size}(H_j)$ is analogous.

Since a polynomial time algorithm exists that ensures serializability, the security of results of concurrent execution for two-phase loacking can be verified in polynomial time. If we assume security for sequential execution is ensured, security for the results of concurrent execution is ensured. Section 2.4 presents a polynomial time algorithm for demonstrating security for intermediate states of concurrent execution.

2.4 Principle of Least Privileges

The definition of serializability (Definition 4) does not distinguish between S_PRES commands and SS_PRES commands because serializability considers only the final state.

The security predicate sp is a function that maps each state into a Boolean value:

$$\text{sp}\,|\,\mathcal{M} \rightarrow \text{Boolean}.$$

The purpose of a security predicate is to formalize a security policy. The intuition is $\text{sp}(M_x)$ is satisfied if and only if M_x is "secure" according to some given security policy.

Not every policy defined in terms of states is a security policy. For example, every reasonable security policy describes a state with no privileges as secure. Below, three sp-assumptions are defined that restrict arbitrary security policies by restricting the definition of allowable *sp* functions. The sp-assumptions assume

[5] Consider $\text{size}(H_j)$ balls numbered between 1 and $\text{size}(H_j)$ inclusive, and $|\,H_j\,|$ boxes, where box_k holds h_k balls. The formula (multinomial [15]) is the number of ways to put the balls in the boxes. Each configuration of balls in boxes corresponds to a member of iset. For example, suppose ball 1 and ball 3 are in box 5, and ball 2 is in box 6. This configuration corresponds to an iset where the first three instructions are $p1_5$, $p1_6$, $p2_5$ such that p_{a_k} is the ath instruction in c_k.

that the initial state, M_0, is empty, that is M[row,col] = $\varnothing$ for each row,col, and sp(M_0). *Sp-assumption* 1 is defined in terms of Definitions 9, 10, 11, and 12, given below.

Definition 9. The *initial sequence* (initseq) of a history h is the history set that consists of all subsequences of instructions beginning with the first instruction in the history.

$$\text{initseq}(h) = \{h' \mid \text{is\_initseq}(h', h)\}$$

where

$$\text{is\_initseq}(h', h) = \begin{cases} \text{true} & \text{if } \mid h' \mid = 0 \\ \text{true} & \text{else if } ((\text{first}(h') = \text{first}(h)) \wedge \\ & \quad \text{is\_initseq}(\text{rest}(h'), \text{rest}(h))) \\ \text{false} & \text{otherwise} \end{cases}$$

For example,

$$h = i_1 i_2 i_3$$

initseq(h) is the set:[6]

$$\{\text{null}, i_1, i_1 i_2, i_1 i_2 i_3\}.$$

As abbreviated notations, the *initiset* and *initperm* combine iset and perm with initseq, respectively.

Definition 10. The *initial list* (initiset) and *initial perm* (initperm) of a command history set are the respective sets of every initial sequence of an iset and perm, respectively.

$$\text{initiset}(H_j) = \{h' \mid \exists h \in \text{iset}(H_j) \quad h' \in \text{initseq}(h)\}$$
$$\text{initperm}(H_j) = \{h' \mid \exists h \in \text{perm}(H_j) h' \in \text{initseq}(h)\}$$

The set of reachable states is the set of states that can be reached through some interleaved execution.

Definition. 11. The set of reachable states from M_x is every state reachable by an initiset.

$$\text{reachable}(H_j, M_x) = \{M_y \mid \exists h \in \text{initiset}(H_j) \mathscr{T}(h, M_x) = M_y\}.$$

A token is called a *privilege* if the token is not a lock, for example, *tok* is a privilege if class(tok) $\neq$ lock. A state M_y is a *privileged subset* of a state M_x, denoted by $M_y \sqsubseteq M_x$, if, the set of privileges in each coordinate of M_y is a subset of its corresponding coordinate in M_x. $\perp$ is in the privileged subset of every state.

$M_y \sqsubseteq M_x$ iff
$M_y = \perp \vee$
$\forall$row,col,tok class(tok) $\neq$ lock $\wedge$ tok $\in M_y$[row,col] $\Rightarrow$ tok $\in M_x$[row,col].

For example, consider states M_x and M_y presented in Table IV. From the definition of privileged subset, $M_y \sqsubseteq M_x$, but $M_x \not\sqsubseteq M_y$.

[6] $\mid h \mid = 0 \Rightarrow h$ = null.

Table IV. Notation

	M_x			M_y	
	col_1	col_2		col_1	col_2
row_1	$\{l_1, r_1, r_2\}$	$\varnothing$	row_1	$\{r_1\}$	$\{l_2\}$
row_2	$\{r_3\}$	$\{r_4\}$	row_2	$\{l_1\}$	$\{r_4\}$

Definition 12. The privsub of M_x is the set of all privileged subsets of M_x.

$$\text{privsub}(M_x) = \{M_y \mid M_y \sqsubseteq M_x\}.$$

In the following, three sp-assumptions are presented which restrict the class of possible functions that map $\mathcal{M}$ to {true, false}.

sp-assumption 1. Every reachable subset of a reachable state is secure.

$$\forall M_x, M_y \in \text{reachable}(H_j, M_0)\ \text{sp}(M_x) \wedge M_y \sqsubseteq M_x \Rightarrow \text{sp}(M_y).$$

sp-assumption 2. The error state is secure.

$$\text{sp}(\perp) = \text{true}.$$

sp-assumption 3. The security predicate ignores locks.

$$\forall M_x, \text{tok,row,col}\ \text{sp}(M_x) \wedge \text{class(tok)} = \text{lock} \Rightarrow \text{sp(enter(tok,row,col))}.$$

Sp-assumption 1 formalizes the statement that loss of privileges should not imply less security. For example, if a user looses access to a file, then the result state should not be less secure than the initial state.

Sp-assumption 2 defines the error state as secure. Since the error state cannot be reached without violating the instruction definitions (the error state can only be reached by executing a blocked instruction), sp-assumption 2 does not constrain any reasonable security policy.

Sp-assumption 3 ensures that locks and access privileges cannot be confused by the security predicate.

Definition 13 represents a secure command history set.

Definition 13. A command history set H_j is secure (security) if every state reachable by the command history set from a secure initial state M_x is secure.

$$\text{security}(H_j) \text{ iff } \forall M_x \in \text{reachable}(H_j, M_0)\ \text{sp}(M_x).$$

Definitions 14–16 formally define the principle of least privileges.

Definition 14. The begin and end of a history h are the instruction sequences up to the first enter instruction that does not reference a lock and the remaining instructions, respectively.

$$\forall h\ \exists h', h''\ h = h'h'' \wedge \text{is\_begin}(h') \wedge \text{is\_end}(h'')$$

such that

$$h' = \text{begin}(h) \text{ and } h'' = \text{end}(h)$$

where

$$\text{is\_begin}(h') = \begin{cases} \text{true} & \text{if } |h'| = 0 \\ \text{true} & \text{else if } (\text{op}(\text{first}(h')) \neq \text{enter} \vee \text{class}(\text{first}(h')) = \text{lock}) \wedge \\ & \quad \text{is\_begin}(\text{rest}(h')) \\ \text{false} & \text{otherwise} \end{cases}$$

and

$$\text{is\_end}(h'') = \begin{cases} \text{true} & \text{if } |h''| = 0 \\ \text{true} & \text{else if } \text{op}(\text{first}(h'')) = \text{enter} \wedge \text{class}(\text{first}(h'')) \neq \text{lock} \\ \text{false} & \text{otherwise} \end{cases}$$

For example, let $h = \text{enter}(l_1, 1, 2)\text{delete}(r_1, 1, 2)\text{enter}(r_2, 1, 2)$. In this example,

$$\text{begin}(h) = \text{enter}(l_1, 1, 2)\text{delete}(r_1, 1, 2)$$
$$\text{end}(h) = \text{enter}(r_2, 1, 2)$$

Definition 15. A command history h_k is *least privileged* (leapriv) if all delete instructions that do not reference locks precede all enter instructions that do not reference locks.

$$\text{leapriv}(h_k) = \text{is\_leapriv}(\text{end}(h_k))$$

$$\text{is\_leapriv}(h') = \begin{cases} \text{true} & \text{if } |h'| = 0 \\ \text{true} & \text{else if } \neg(\text{op}(\text{first}(h')) = \text{delete} \wedge \text{class}(\text{first}(h')) \neq \text{lock}) \wedge \\ & \quad \text{is\_leapriv}(\text{rest}(h')) \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 16. A command history set is a *least privileged set* (leaprivset) if every command history is least privileged.

$$\text{leaprivset}(H_j) \text{ iff } \forall h_k \in H_j \ \text{leapriv}(h_k).$$

Definition 16 is the formal definition of a least privileged command set. The conditions of Section 2.3 (Definition 7) when combined with Definition 16 provide security for all *reachable* states, provided all commands are *ss_pres*, defined as follows.

Definition 17. A history set is *ss_pres* if every state reachable from the secure initial state via sequential execution is secure.

$$\text{ss\_press iff } \forall h \in \text{initperm}(H_j) \ \text{sp}(h, M_0).$$

2.5 Theorem for Security in Centralized, Parallel, and Distributed Systems

This section presents and proves Theorem 2, a general-purpose security theorem for centralized, parallel, and distributed systems.

The proof strategy is to show that each reachable state is either $\perp$ or a subset of some state reachable through sequential execution. Then, security is established by applying the sp-assumptions.

Definition 18. The *reachable sequential set* (reach_seq) is the set of final states reachable through sequential execution.

$$\text{reach\_seq}(H_j, M_x) = \{M_y \mid \exists H_k \subseteq H_j \; \exists h \in \text{perm}(H_k) M_y = \mathcal{T}(h, M_x)\}.$$

For example,

$$\text{reach\_seq}(\{h_1, h_2\}, M_x) = \{M_x, \mathcal{T}(h_1, M_x), \mathcal{T}(h_2, M_x), \mathcal{T}(h_1h_2, M_x), \mathcal{T}(h_2h_1, M_x)\}.$$

Definition 19. The *sub_reach_seq* is the set of states that are reachable subsets of reachable sequential states.

$$\text{sub\_reach\_seq}(H_j, M_x) = \{\perp\} \cup \text{is\_sub\_reach\_seq}(H_j, M_x)$$

where

$$\text{is\_sub\_reach\_seq}(H_j, M_x) = \{M_y \in \text{reachable}(H_j, M_x) \mid \exists M_z \in \text{reach\_seq}(H_j, M_x) M_y \in \text{priv\_sub}(M_z)\}.$$

For example, $\text{sub\_reach\_seq}(\{h_1, h_2\}, M_x)$ is

$$\left\{ M_y \in \text{reachable}(H_j, M_x) \mid M_y = \perp \vee M_y \in \bigcup_{M_z \in \text{reach\_seq}(\{h_1, h_2\}, M_x)} \text{priv\_sub}(M_z) \right\}.$$

Definition 20. A command history set is ss_sub if every intermediate state reached through sequential execution is a subset of some final state reached through sequential execution.

$$\text{ss\_sub}(H_j, M_x) = \begin{cases} \text{true} & \text{if } \forall h \in \text{init\_perm}(H_j) \mathcal{T}(h, M_x) \in \text{sub\_reach\_seq}(H_j, M_x) \\ \text{false} & \text{otherwise} \end{cases}$$

LEMMA 1. *If $M_x \sqsubseteq M_y$, then for any history h, $\mathcal{T}(h, M_x) \sqsubseteq \mathcal{T}(h, M_y)$.*

PROOF. If $|h| = 0$, then Lemma 1 is obvious. Otherwise, assume, by induction, Lemma 1 for all but the last instruction, $i_{|h|}$ of h. By considering each possible kind of operation, that is, enter, delete, present, and absent, it is easy to see that Lemma 1 is satisfied after i_h executes. □

Definition 21. The *not subset* (not_sub) of a command history set is the set of interleaved histories that reach a state that is not a subset of a state reachable through sequential execution.

$$\text{not\_sub}(H_j, M_x) = \{h \in \text{iset}(H_j, M_x) \mid \exists h' \in \text{initseq}(h) \mathcal{T}(h', M_x) \notin \text{sub\_reach\_seq}(H_j, M_x)\}.$$

As an ancillary function, Lemma 2 references *incomplete*, the number of command histories that have not completed execution.

Definition 22. *Incomplete* denotes the number of command histories that have not yet completed execution.

$$\text{incomplete}(H_j, h) = |H_j| - \text{comp}(H_j, h)$$

$$\text{comp}(H_j, h) = \begin{cases} 0 & \text{if } |h| = 0 \\ 1 + \text{comp}(H_j - h_k, \text{rest}(h)) & \text{else if } \exists h_k \in H_j \ \text{first}(h_k) = \text{first}(h) \wedge |h_k| = 1 \\ \text{comp}((H_j - h_k) \cup \{\text{rest}(h_k)\}), \text{rest}(h)) & \text{else if } \exists h_k \in H_j \ \text{first}(h_k) = \text{first}(h) \wedge |h_k| > 1 \\ 0 & \text{otherwise} \end{cases}$$

For example, let $H_j = \{h_1, h_2, h_3\}$, where $h_1 = i_{1_1}$, $h_2 = i_{1_2} i_{2_2}$, and $h_3 = i_{1_3} i_{2_3} i_{3_3}$. Let h be defined as follows:

$$h = i_{1_1} i_{1_2} i_{1_3} i_{2_3}.$$

Here, $\text{incomplete}(H_j, h) = 2$ because h_1 has completed execution, but h_2 has not yet executed i_{2_2} and h_3 has not yet executed i_{2_3} and i_{3_3}.

LEMMA 2. *The nested critical section condition and the least privilege condition ensure that the set of reachable states that are not subsets of states reachable through sequential executions is the empty set.*

$$\forall M_x \in \mathcal{M}\ pcs(H_j) \wedge leapriv(H_j) \Rightarrow |\ not\_sub(H_j, M_x)\ | = 0.$$

PROOF. Suppose not. Let $h \in \text{not\_sub}(H_j, M_x)$. Let h' be the shortest initial sequence of h that yields a state that is not a privileged subset of a state reached through sequential execution. In other words:

(i) $h' \in \text{initseq}(h)$

(ii) $\mathcal{T}(h', M_x) \notin \text{sub\_reach\_seq}(H_j, M_x)$

(iii) $\forall h'' \in \text{initseq}(h')\mathcal{T}(h'', M_x) \notin \text{sub\_reach\_seq}(H_j, M_x) \Rightarrow h'' = h'$

Let $i_b = \text{last}(h')$, and $h_k \in H_j$ be the history that contains i_b. Let $i_a = \text{first}(h_k)$ and $i_c = \text{last}(h_k)$. Let h''' be the sequence of instructions in h between i_a and i_b inclusive, and h^{IV} be the sequence in h after i_b and up to and including i_c, as shown in the following.

$$h = \overbrace{i_1 \cdots \underbrace{i_a \cdots i_b}_{h'''} \cdots \underbrace{i_c}_{h^{IV}} \cdots i_{|h|}}^{h'}$$

Since i_b is at the end of the shortest initial sequence that yields a nonprivileged subset state, $\text{op}(i_b) = \text{enter}$ and $\text{class}(i_b) \neq \text{lock}$. The remainder of the proof has two cases, depending on the form of h_k.

Case I. In h_k, at least one instruction enters a lock after i_b.

In other words, in h^{VI}, shown below,

$$h_k = \underbrace{\cdots}_{h^V} \underbrace{i_b \cdots}_{h^{VI}}$$

there exists at least one enter lock instruction. Thus, from the definition of *nest* (Definition 5), no delete lock instruction precedes i_b, that is, in h^V. Thus, from the definition of nest, there exists histories, $h^{VII} \in \text{iset}(H_j)$, and $h^{VIII} \in \text{initseq}(h^{VII})$ such that instructions of h_k are not interleaved before i_b in h^{VII}, and h^{VIII} yields the same final state as h'. Formally,

(i) $h^{VII} \in \text{iset}(H_j)$
(ii) $h^{VIII} \in \text{initseq}(h^{VII})$
(iii) $\mathcal{T}(h^{VIII}, M_x) = \mathcal{T}(h', M_x)$
(iv) $\exists h^{IX} h^{VIII} = h^{IX} h^V i_b$

From Lemma 1, if $\mathcal{T}(h^{IX}, M_x) \in \text{sub\_reach\_seq}(H_j, M_x)$, then $\mathcal{T}(h^{IX} h_k, M_x) \in \text{sub\_reach\_seq}(H_j, M_x)$. However, this is a contradiction because $\mathcal{T}(h^{IX} h_k, M_x) = \mathcal{T}(h', M_x)$. As a result, assume $\mathcal{T}(h^{IX}, M_x) \notin \text{sub\_reach\_seq}(H_j, M_x)$. Note that h^{IX} has no instructions from h_k. Thus, $|\text{not\_sub}(H_j - \{h_k\}, M_x)| \neq \varnothing$. So, by induction on $|H_j|$, it can be shown that Case I is proved.

Case II. In h_k, no instruction enters a lock after i_b.

From the definition of nest, there exists a history h^X such that

(i) $h^X \in \text{iset}(H_j)$
(ii) $\mathcal{T}(h^X, M_x) = \mathcal{T}(h, M_x)$
(iii) $\exists h^{XI} h^X = h' h^{VI} h^{XI}$

If $\mathcal{T}(h' h^{VI}, M_x) \neq \perp$, then from the definition of leapriv, (h_k has no instruction after i_b that deletes a token that is not a lock),

$$\mathcal{T}(h' h^{VI}, M_x) \notin \text{sub\_reach\_seq}(H_j, M_x).$$

In other words, after h_k completes, the resultant state is not a subset of a state reached through sequential execution. The remainder of the proof is by induction on incomplete(H_j, h).

If $\mathcal{T}(h' h^{VI}, M_x) = \perp$, then some *present* or *absent* instruction executes after i_b, when the present or absent instruction is not enabled. However, it can be shown that such an instruction is unnecessary. □

THEOREM 2. *If every history in a command history set satisfies the proper critical section, the least privilege, and the ss_pres condition, then every state reachable from the secure initial state is secure.*

$$pcs(H_j) \wedge leapriv(H_j) \wedge ss\_pres(H_j) \Rightarrow security(H_j).$$

PROOF. Suppose not for $h \in \text{initiset}(H_j)$. If $\mathcal{T}(h, M_0) = \perp$, then from sp-assumption 2, there is a contradiction and, as a result, Theorem 2 is proved. Otherwise, assume $\mathcal{T}(h, M_0) \neq \perp$. From Lemma 2, there exists an $h' \in \text{initperm}(H_j)$ such that $\mathcal{T}(h, M_0) \sqsubseteq \mathcal{T}(h', M_0)$. From the definition of ss_pres, sp($\mathcal{T}(h', M_0)$). Thus, from sp-assumption 1, sp-assumption 3, and ss_pres, sp($\mathcal{T}(h, M_0)$). □

3. EVALUATION

The purpose of a security model is to bridge the semantic gap between a security policy and a specification, as shown in Figure 3 (this section considers only the formal development path of system definition [21]). A policy is informal, while a specification is formal. A policy reflects administrative decisions, while a specification reflects the behavior of an implementation. Furthermore, a policy does not define a system state. However, a specification defines execution on an abstract model of computation and is expressed in terms of states and state transitions. Finally, a policy is architecture-independent, while a specification defines a particular implementation. For example, the military security policy [2] does not define the number of nodes in a network, while a specification of a particular system that enforces the military security policy may define this characteristic.

A security model has characteristics of both policies and specifications. First, a model is formal. It would otherwise be difficult, if not impossible [39], to verify a specification with respect to a model. Second, a model defines states. Otherwise, a model would not reflect the discrete nature of computing resources. Third, a model is architecture-independent. Otherwise, a model would be overly specific.

This section evaluates the CPD model by presenting a taxonomy of security models (Sect. 3.1) and a comparison of the CPD model with other related security models (Sect. 3.2).

3.1 Security Model Taxonomy

The taxonomy describes different types of security problems and their corresponding models. The purpose of many different kinds of system models, such as security models, deadlock models, and fault-tolerance models, is to analyze and prove predicates. A state machine model's predicate divides states into "good" states (e.g., secure states, deadlock-free states, k-resources-available states, and "bad" states, e.g., nonsecure states, deadlocked states, not-k-resources-available states). The purpose of a state machine model is either to prove a *safety* property, that is, that "something (presumably bad) will not happen" [29], a *liveness* property, that "something (presumably good) will eventually happen" [29], or some combination of safety and liveness properties.

The first division of the security model taxonomy divides security models according to safety and liveness.[7] The safety category contains all security models that define *safety properties* but not *liveness properties*, and the liveness category contains security models that define a combination of safety and liveness properties.

The security policies that can be expressed by safety models are *nondisclosure* and *integrity* policies:

—*Nondisclosure*. "The assets of a computing system are accessible only by authorized parties" [43].

[7] Security model taxonomies are controversial. For example, at the highest layer, other taxonomies divide security models into the classes: *nondisclosure*, *integrity*, and *denial of service*.

	Security policy	Security model	Specification
Formal	No	Yes	Yes
State definition	No	Yes	Yes
Architecture definition	No	No	Yes

Fig. 3. Policy-specification semantic gap.

—*Integrity.* "Assets can be modified only by authorized parties" [43]. Assets may only be modified in an authorized manner.

These are safety policies because they both define security for systems that prohibit a user or program from acquiring unauthorized access to an item. The security policies that can be expressed by liveness models are *nondenial of service* policies.

—*Nondenial of Service.* "Assets are available to authorized parties. An authorized party should not be prevented from accessing those objects to which he or she or it has legitimate access" [43].

Nondenial of service is a liveness policy because it defines security for systems that assure that an authorized user or program will eventually obtain access to a desired item. The CPD model is a safety model and, as a result, the taxonomy of liveness models is outside the scope of this paper.

The safety category is divided into privilege-based models and information-flow models. The distinction between the two classes is that the state in a privilege-based model is defined solely in terms of privileges, while the state in an information-based model includes the values of information storage units.

Privilege-based models include access control models (e.g., [7, 26]) and other models that define both access control and synchronization (e.g., the CPD model and [6]). In a privilege-based model, access rights, such as read and write, are considered privileges to access information; and synchronization primitives, such as locks, are considered "privileges" to continue execution. Privilege-based models have been used to express nondisclosure [3, 4, 7, 9, 12, 40] and integrity [6, 11, 31, 33].

In an information flow model, "information is transmitted along an object when variety in the events engaged by a source user can be conveyed to a destination user as a result of their interaction with the object" [19]. Here, "interaction with an object" changes the object's value (e.g., the value of a collections of bits, a file, or an encrypted message). The taxonomy divides information flow models into two categories: noninterference and deducibility. A noninterference model (e.g., [23, 25, 35, 36, 44, 45]) provides information flow restrictions that may potentially prohibit one user from knowing that another user is on the system. A deducibility model (e.g., [17, 50]) may potentially prohibit one user from deducing "anything about the sequence of inputs made by a second user" [36]. Determining the difference between these categories is an open research problem. Currently, the difference depends upon the precise definition of *interference*, and the presence of determinism in the model of computation.

3.2 Security Model Comparison

This section evaluates the CPD model in terms of the three evaluation metrics. The first metric, distribution (Sect. 3.2.1) describes a model's ability to represent security in a distributed system. The second metric (Sect. 3.2.2) describes a model's ability to accurately represent a design specification. The third metric, policy (Sect. 3.2.3) describes a model's flexibility to represent different types of security policies.

3.2.1 *Distribution.* Historically, most secure systems (e.g., [1, 16, 20, 46, 47]) implement a centralized or possibly a parallel TCB, and security models for these systems (e.g., [7, 11]) typically prohibit concurrency. Some new systems (e.g., [13, 18]) however, are currently being designed with distributed TCBs. Currently, there exist many noninterference models (e.g., [22, 36, 44, 45]), but only a few privilege-based models (e.g., CPD model and [10, 40]) that define security for distributed systems. No privilege-based model, however, other than the CPD model, provides concurrency yet guarantees that every state reachable from a secure initial state is secure.

Since the CPD model provides concurrency and references a global state, it is relatively easy to see that it can represent a centralized TCB. The CPD model may also represent a parallel or distributed TCB because the CPD model accounts for concurrency (e.g., Sect. 4 presents a representation of a distributed system with a shared printer resource). As a result, the CPD model can be used to represent all three types of architectures.

Five aspects of the CPD model that are used to define distribution may require further motivation.

(i) The CPD model prohibits feedback from affecting the input command set.
(ii) The CPD model defines sequential instruction execution, but not sequential history execution.
(iii) Some behaviors defined secure in the CPD model could potentially deadlock.
(iv) The CPD model does not contain the operations create or destroy.
(v) For some TCBs, sp-assumption 1 appears too restrictive.

In some models [36, 44] a user can query the state and use the result of the query to define the next input. The CPD model prohibits this type of feedback because the entire command set is input into it at initialization time. The input command set is defined in the CPD model as a free variable, which implies that any input that could potentially arise through feedback is a possible interpretation of the free variable.

Another aspect is the apparent lack of sequencing between commands. In some situations, a user may wish to designate that command history, h_k, execute before a second designated command history, h_l. In this case, explicit synchronization is required in the definitions of the respective commands. For example, the respective commands may be defined such that the ninth operation in h_l blocks until the third operation in h_k executes. "The result of this policy [for command ordering] is that the desired policy is hidden within the program [commands], rather than being stated as an explicit rule that the system can then enforce" [14].

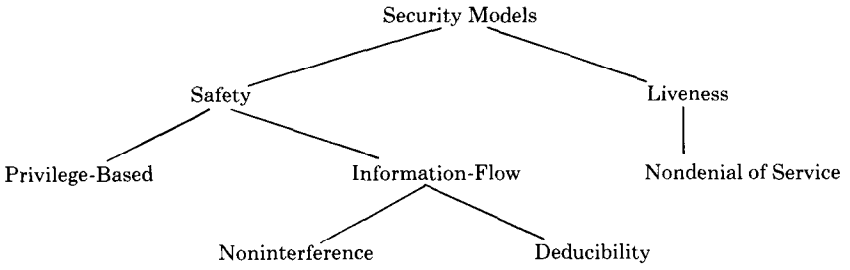

Fig. 4. Security model taxonomy.

The third aspect is deadlocks. Security and deadlock avoidance are two different safety properties, where the set of secure deadlock-free behaviors is the set of all behaviors that satisfy both a CPD model security predicate and a deadlock-avoidance predicate. As a result, security and deadlock avoidance can be treated separately, where the CPD model neither helps nor hinders identifying deadlocks and a given deadlock-avoidance model neither helps nor hinders identifying nonsecure states.

The fourth aspect is the apparent lack of create or destroy subject and object operations. In other security models (e.g., [7, 12, 26, 27]), explicit create and destroy operations increment or decrement the list of subjects and objects. These operations can be represented in the CPD model as moving a subject or object off a free list onto an active list, and off an active list onto a destroyed list, respectively. Since the CPD model uses an unbounded size matrix as its state, the alternative representation can be explicitly coded in the state by index tokens that represent the free list, the active list, and the destroyed list, respectively [9].

The fifth aspect is the apparent over-restrictiveness of sp-assumption 1. Upon close examination, sp-assumption 1 is defined only over reachable states. As a result, every security model that reaches only secure states satisfies sp-assumption 1. In other words,

$$\text{security}(H_j) \Rightarrow \text{sp-assumption 1}.$$

3.2.2 *Design.* This section argues that a privilege-based model, such as the CPD model, can be used to provide a good specification correctness criterion for a system that enforces a safety security policy. A security model is associated with two mappings, as shown in Figure 5, which indicates that the security model *enforces* the security policy, and the system specification *enforces* the security model. Since a security policy is informal, the mapping from a security policy to a security model is consequently informal [39]. However, the mapping from a specification to a security model may be formal [24, 28, 32, 38].

As shown in the taxonomy of Figure 4, there are two categories of safety models: privilege-based and information flow. The primary advantage of a privilege-based model is that it may be easier to justify the mapping between the specification and the model, while the primary advantage of an information-flow model is that it may be easier to justify the mapping from the security model to the policy. The former potential advantage exists because a privilege-based model,

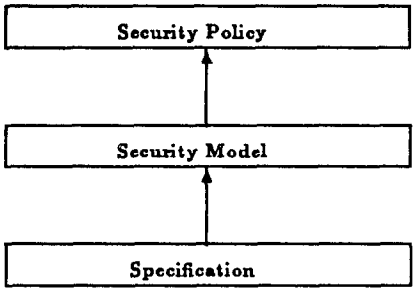

Fig. 5. Security model mappings.

but not an information-flow model, can ignore internal TCB state variables without affecting the model's formalism. As a result, privilege-based models have advantages in systems that permit some "legal covert channels." Consider, for example, a system that contains a large number of sensitivity levels (e.g., 2^{64}) in which it is not practical to allow the scheduler to allocate a fixed time slice to each sensitivity level. The system contains a covert channel because low-sensitivity-level users may obtain information about high-sensitivity-level users by monitoring the system load average. The difference between the two categories of models, with respect to this example, is that security for internal TCB variables is not affected by changes in access permissions, but internal TCB variables are conduits for transmitting information.

The primary advantage of an information-flow model compared with a privilege-based one is that an information-based model is a superior representation of a security policy. In particular, in a privilege-based model "it is not clear what possibilities for security violations through covert channels still exist in the actual system" [49], while in an information-flow model, covert channels may be prohibited. Consider for example, a policy that is defined in terms of information values, for example, the Clark–Wilson model [14]. In this case the policy has an application-independent portion that can be represented by a privilege-based model and an application-dependent portion that cannot be represented by a privilege-based model. Here, the application-dependent portion requires that *transformation procedures* be *certified* to cause transitions between valid object states. Since a privilege-based model cannot define the information values, no privilege-based model security predicate can distinguish a valid from an invalid information value.

3.2.3 *Policy.* Since the CPD model is a privilege-based safety model, it cannot be used to define a nondenial of service policy. However, it can implement a distributed version of many privilege-based security models (e.g., [7, 11, 26, 27, 40]). Since the CPD model does not define a specific instance of a security predicate, as in the case of [7, 11, 30, 34], it can be used to formalize a variety of different security policies. For example, the CPD model may represent a distributed version of the Bell–La Padula model [10] and a multilevel secure file system [5].

4. A DISTRIBUTED SYSTEM EXAMPLE

This section presents an example application of the CPD model—a representation of a distributed system. The example depicts a system with three nodes, two

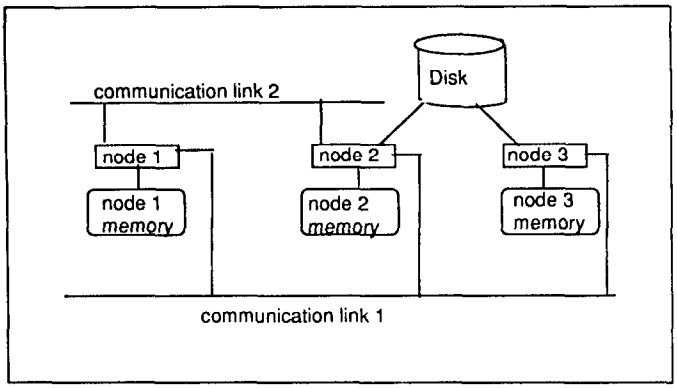

Fig. 6. Example distributed system.

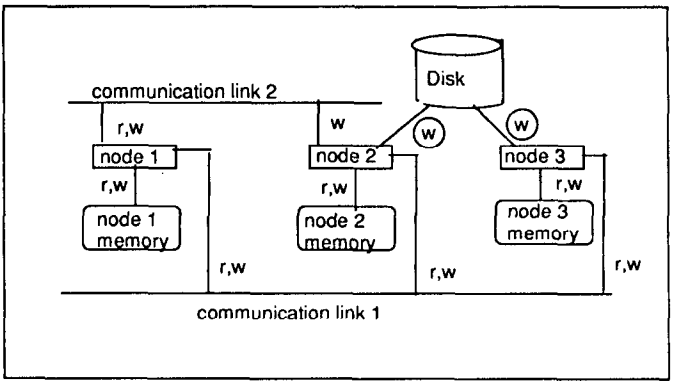

Fig. 7. Nonsecure distributed system.

communication links, and one shared disk, as shown in Figure 6. Communication link 1 ($comm_1$) connects all three nodes, communication link 2 ($comm_2$) connects $node_1$ and $node_2$, and the disk connects $node_2$ and $node_3$. Each node is connected to its own local memory.

A security policy for the distributed system defines privileges for communication and memory access. For example, consider a security policy that restricts access to the shared disk.

> The distributed system is secure unless $node_2$ and $node_3$ have simultaneous write access to the shared disk.

An example nonsecure configuration of privileges is shown in Figure 7 (nonsecure privileges are circled). The figure shows that all three nodes have (*r*)ead and (*w*)rite access to $comm_1$. Also, $node_1$ has (*r*)ead and (*w*)rite access to $comm_2$ and $node_2$ has (*w*)rite access to $comm_2$. However, the distributed system is not secure because both $node_2$ and $node_3$ have (*w*)rite access to the shared disk.

	$comm_1$	$comm_2$	disk	$node_1$ mem	$node_2$ mem	$node_3$ mem
$node_1$	α	α		α		
$node_2$	α	α	α		α	
$node_3$	α		α			α

Fig. 8. Initial state.

The CPD model can represent this distributed system by defining states where the rows represent nodes and the columns represent memory, communication links, and the disk. The initial state, M_0[8] (shown in Figure 8) defines the system architecture, where the token α denotes a hardware link. For example, since there exists a link between $node_2$ and $comm_2$, $\alpha \in M_0[node_2, comm_2]$. However, since there does not exist a link between $node_3$ and $comm_2$, $\alpha \notin M_0[node_3, comm_2]$.

The formalism for the security predicate (Sect. 2.4) is defined in the following.

$$\begin{aligned} &\text{sp}(M_x) \text{ iff} \\ &\quad M_x = \perp \vee \\ &\quad \neg\exists \text{row}_1, \text{row}_2 \\ &\quad \alpha, w \in M_x[\text{row}_1, \text{disk}] \\ &\quad \alpha, w \in M_x[\text{row}_2, \text{disk}] \end{aligned}$$

The initial state M_0 (Sect. 2.4) is secure because w is not in any coordinate. Also, the security predicate satisfies all three sp-assumptions:

—*sp-assumption* 1. The distributed system can transition to a nonsecure state by gaining (as opposed to losing) a hardware connection or write privilege.

—*sp-assumption* 2. The disjunction in sp ensures that the error state is secure.

—*sp-assumption* 3. The security predicate does not reference a token of class *lock*.

Depending on the system being modeled, there exist many potential command sets. So this section adds a new command, c_k, to a previously existing command set, C_j, forming $C_j \cup \{c_k\}$, where it is assumed that C_j satisfies the conditions of Theorem 2, that is, every instantiation H_j of C_j satisfies *pcs* (Definition 7), *leaprivset* (Definition 16), and *ss_pres* (Definition 17). An example new command, c_k, is defined in the following:

```
command c_k(n:node, c:comm, d:disk) =
  enter(l_1, n, d)
    present(α, n, d)
      enter(l_1, n, c)
        present(α, n, c)
        delete(w, n, c)
      delete(l_1, n, c)
    enter(r, n, d)
  delete(l_1, n, d)
```

[8] For brevity, unused rows and columns are not shown in Figure 8.

Command c_k *deletes* w from the $[n, c]$ coordinate and *enters* r to the $[n, d]$ coordinate. Provided every command in C_j follows the convention that a coordinate is locked before it is referenced, $\text{pcs}(H_j \cup \{h_k\})$, for each instantiation. Since c_j executes its delete privilege operation, delete(w, n, c), before its enter privilege operation, enter(r, n, d), every instantiation of c_j satisfies leapriv. Finally, since c_j enters neither α nor w, c_j satisfies ss_pres. As a result, since (i) the security predicate satisfies all three sp-assumptions, (ii) $\text{sp}(M_0)$, and (iii) every instantiation of $(C_j \cup \{c_k\})$ satisfies all three conditions of Theorem 2, for each instantiation,

$$\text{security}(H_j \cup \{h_k\}).$$

The example shown in this section is relatively simple because it contains only a few simple nodes and communication devices. Each command represents a single thread of execution and may only modify the state in a single row or column. In general, there may be multiple active entities (subjects) on each node, multiple slots on the communication media, and multiple partitions on the disk. A CPD model representation of the more complex distributed system contains a row corresponding to each active entity and a column corresponding to each passive entity. A security predicate could potentially distinguish between entities on each node. For example, a security predicate may permit a *trusted* subject residing on node_2 access to the disk, but prohibit an *untrusted* subject on the same node from accessing the same disk. A detailed example of a CPD model representation of a nontrivial distributed file system is described in [9].

5. CONCLUSION

The contribution of this paper is a formal protection model for centralized, parallel, and distributed systems. In general, parallel and distributed systems are difficult to model because of the complicated interactions of concurrent executions. This problem is solved by proving Theorem 2, which demonstrates conditions for ensuring security for parallel and distributed systems. The conditions are relatively easy to validate.

In future research, the CPD model will be used as a fundamental building block of a composability model. Composability will show that if a countable number of command history sets satisfy their respective local security predicates, then the composition command history set satisfies the composition security predicate. The approach is to define a set of *composition operators* that guarantee composability. One composition operator will use the CPD model results to prove that the conditions of Theorem 2 and the sp-assumptions guarantee composability. Further research will define classes of composable safety properties that are applicable to nonserializable systems.

ACKNOWLEDGMENTS

The referees made valuable comments on earlier drafts of this paper. Tim Redmond and Charles Pfleeger also provided many useful insights.

REFERENCES

1. *System Overview Gemini Trusted Multiple Microcomputer Base (version 0).* Carmel, Calif., 0 ed., May 1985.
2. *Trusted Computer Systems Evaluation Criteria.* Tech. Rep. DoD 5200.28-STD, National Computer Security Center, Fort Meade, Md., Dec. 1985.
3. AKYILDIZ, I., AND BENSON, G. Security models of distributed systems. In *Proceedings of the 4th International Symposium on Computer and Information Sciences*, A. Dogac and E. Gelenbe, Eds. (Cesme, Turkey, Oct. 1989). Vol. 2, 1225–1235.
4. AKYILDIZ, I., AND BENSON, G. A security level reclassifier for a local area network. In *Proceedings of the European Symposium on Research in Computer Security* (Toulouse, Oct. 1990). AFCET.
5. AKYILDIZ, I., BENSON, G., AND APPELBE, W. A multilevel secure file server for a local area network. Tech. Rep. GIT-ICS-89/27, Georgia Institute of Technology, Atlanta, Aug. 1989.
6. BADGER, L. A model for specifying multi-granularity integrity policies. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1989). 269–277.
7. BELL, D., AND LAPADULA, L. Secure computer system unified exposition and multics interpretation. Tech. Rep. MTR-2997, MITRE Corp., Bedford, Mass., July 1975.
8. BEN-ARI, M. *Principles of Concurrent Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
9. BENSON, G. A formal protection model of security in distributed systems. Ph.D. dissertation, Georgia Institute of Technology, Atlanta, Aug. 1989.
10. BENSON, G., APPELBE, B., AND AKYILDIZ, I. The hierarchical model of distributed system security. In *1989 IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1989). 194–203.
11. BIBA, K. Integrity considerations for secure computer systems. Tech. Rep. TR-3153, MITRE Corp., Bedford, Mass., April 1977.
12. BISHOP, M. Practical take-grant systems: Do they exist? Ph.D. dissertation, Purdue Univ., West Lafayette, In., May 1984.
13. BRANSTAD, M., TAJALLI, H., MAYER, F., AND DALVA, D. Access mediation in a message passing kernel. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1989). 66–72.
14. CLARK, D., AND WILSON, D. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (Oakland, Calif., April 1987). 184–194.
15. COHEN, D. *Basic Techniques of Combinatorial Theory.* John Wiley, New York, 1978.
16. COHEN, E., AND JEFFERSON, D. Protection in the Hydra operating system. In *Proceedings of the 5th SOSP* (Nov. 1975). 141–160.
17. DENNING, D. A lattice model of secure information flow. In *Commun. ACM 17*, 5 (May 1976), 236–243.
18. WONG, R., ET AL. The SDOS system: A secure distributed operating system prototype. In *12th National Computer Security Conference* (Baltimore, Md., Oct. 1989). National Institute of Standards and Technology, 1989, 172–183.
19. FOLEY, S. A universal theory of information flow. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (Oakland, Calif., May 1987). 116–121.
20. FRAIM, L. SCOMP: A solution to the multilevel security problem. *IEEE Comput. 16*, 7 (July 1983), 26–34.
21. GASSER, M. *Building a Secure Computer System.* Van Nostrand Reinhold, New York, 1988.
22. GLASGOW, J., AND MACEWEN, G. Reasoning about knowledge in multilevel secure distributed systems. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy* (Oakland, Calif., 1988), 122–128.
23. GOGUEN, J., AND MESEGUER, J. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy* (Oakland, Calif., 1982). 11–20.
24. GOOD, D., DIVITO, B., AND SMITH, M. Using the Gypsy methodology. Tech. Rep., Computational Logic Inc., Austin, Tex., 1988.
25. HAIGH, J., AND YOUNG, W. Extending the noninterference version of MLS for SAT. In *IEEE Trans. Softw. Eng.* (Feb. 1987), 141–150.
26. HARRISON, M., RUZZO, W., AND ULLMAN, J. Protection in operating systems. In *Commun. ACM 17*, 8 (Aug. 1976), 461–471.

27. JONES, A., LIPTON, R., AND SNYDER, L. A linear time algorithm for deciding security. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 1976.
28. KEMMERER, R. FDM—a specification and verification methodology. In *Proceedings of the 3rd Seminar on the DoD Computer Security Initiative Program* (Gaithersburg, Md., Nov. 1980). NBS.
29. LAMPORT, L. A formal basis for the specification of concurrent systems. In *Distributed Operating Systems: Theory and Practice*, Vol. F28, Y. Paker et al., Eds., NATO Advanced Study Institute, Springer-Verlag, Berlin, 1987, 4–46.
30. LANDWEHR, C., HEITMEYER, C., AND MCLEAN, J. A security model for military message systems. *ACM Trans. Comput. Syst.*, ACM (Aug. 1984), 198–222.
31. LEE, T. Using mandatory integrity to enforce "commercial" security. In *1988 IEEE Symposium on Security and Privacy* (Oakland, Calif., April 1988). 140–146.
32. LEVITT, K., ROBINSON, L., AND SILVERBERG, B. The HDM handbook. Tech. Rep., Computer Science Lab., SRI International, Menlo Park, Calif., June 1979. Vols. 1–3.
33. LIPNER, S. Non-discretionary controls for commercial applications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy* (Oakland, Calif., April 1982). 2–10.
34. LU, W., AND SUNDARESHAN, M. A model for multilevel security in computer networks. In *Infocom* (New Orleans, La., April 1988). 1095–1104.
35. MACEWEN, G., POON, V., AND GLASGOW, J. A model for multilevel security based on operator nets. In *1987 IEEE Symposium on Security and Privacy* (Oakland, Calif., April 1987). 150–160.
36. MCCULLOUGH, D. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy* (Oakland, Calif., 1988), 177–186.
37. MCLEAN, J. Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (Oakland, Calif., 1987), 123–131.
38. MILLEN, J. Operating system security verification. Tech. Rep. M79-223, MITRE Corp., Bedford, Mass., Sept. 1979.
39. DE MILLO, R., LIPTON, R., AND PERLIS, A. Social processes and proofs of theorems and programs. *Commun. ACM 20*, 5 (May 1979), 271–280.
40. MINSKY, N. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.* (Oct. 1984), 573–602.
41. PAPADIMITRIOU, C. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Md., 1986.
42. PETERSON, J., AND SILBERSCHATZ, A. *Operating System Concepts*, 2nd ed. Addison-Wesley, Reading, Mass., 1985.
43. PFLEEGER, C. *Security in Computing.* Prentice-Hall, Englewood Cliffs, N.J., 1989.
44. RUSHBY, J. Proof of separability: A verification technique for a class of security kernels. In *Proceedings of the 5th International Symposium on Programming.* Springer-Verlag, Berlin, 1982, 352–362.
45. RUSHBY, J. Security policies for distributed systems. Unpublished draft, SRI International, Sept. 1988.
46. SAYDJARI, O. S., BECKMAN, J., AND LEAMAN, J. Locking computers securely. In *Proceedings of the 10th DoD/NBS Computer Security Conference* (Gaithersburg, Md., Sept. 1987). 129–140.
47. SCHROEDER, M., AND SALTZER, J. The MULTICS kernel design project. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles* (Nov. 1977). 57–65.
48. SEIDEN, K., AND MELANSON, J. The auditing facility for a vmm security kernel. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy* (Oakland, Calif., May 1990). 262–277.
49. SUTHERLAND, I. Relating Bell–LaPadula-style security models to information models. In *Proceedings of the Computer Security Foundations Workshop* (Franconia, N.H., June 1988), 112–126.
50. SUTHERLAND, I., PERLO, S., AND VARADARAJAN, R. Deducibility security with dynamic level. In *Proceedings of the Computer Security Foundations Workshop II* (Franconia, N.H., June 1989). IEEE, New York, 1989, 3–8.

Received March 1988; revised January 1990; accepted July 1990