The Hierarchical Model of Distributed System Security

G. Benson, W. Appelbe and I. Akyildiz

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

The Hierarchical Model (H_Model) is an access matrix-based model used to define non-disclosure in distributed multilevel secure applications such as secure file systems, secure switches, and secure upgrade/downgrade facilities. The H_Model explicitly encodes access rights, synchronization primitives, and indirection in its state matrix. Serializability of concurrent commands is formally defined in terms of the H_Model syntactic model of computation. H_Model serializability conditions are independent of the semantic security predicate.

1 Introduction

There are two reasons for building secure distributed systems: (1) to link secure machines in a secure manner, and (2) to satisfy security requirements for distributed applications. A security model for the first problem, e.g., [8,15,16,19], calculates an aggregate security model from a group of security models. A security model for the second problem provides a global view that satisfies the requirements for a distributed application. An example of a distributed application is a distributed Multilevel Secure (MLS) file system where the file system\(^3\) consists of multiple secure file servers and untrusted hosts that communicate over a local area network. Untrusted hosts access the communications media via trusted interface units. This paper presents the Hierarchical Model (H_Model) which models non-disclosure in the trusted subjects implemented in the secure file servers. The trusted subjects are modeled as concurrently executing components that enforce a global file system security policy.

A security model consists of semantic and syntactic components. The semantic component provides a security predicate that defines security. Example security predicates are the s-property and the e-property [2] for sequential systems, and the

\(^3\)The file system is being implemented by Martin Marietta Corporation, and is being modeled at the Georgia Institute of Technology.

restrictiveness property [18] for distributed systems. The syntactic component defines a model of computation. Example models of computation are state machines with sequential schedulers for sequential systems [2], and operator nets for distributed systems [8]. The semantic component of the H_Model is identical to a security predicate that describes security in a sequential environment; however, the syntactic component of the H_Model additionally provides concurrent processing. An important aspect of the H_Model is that the syntactic component (model of computation) is hidden from the security predicate. This is done by assuring that state transitions appear as if they are executed sequentially, even though they can be executed concurrently.

The H_Model is a distributed analog of the Harrison, Ruzzo, Ullman model (HRU) [12]; both models define a syntactic model of computation, independent of a semantic security predicate. The H_Model provides concurrent processing which may be used to model either distributed systems or centralized systems with concurrent commands. The syntactic model used in the H_Model is a deterministic finite state machine whose input is a set of concurrent commands, where each command is a sequence of atomic operations that act as transitions. The H_Model uses blocking operations to control concurrency. Concurrent commands in the H_Model are analogous to concurrent database transactions [5]. In both cases, a global state is concurrently accessed by multiple users. In a database, a given set of transactions are considered "correct" if [5]:

i) The transactions correctly implement the database transaction policy when executed sequentially.

ii) The transactions appear to the user to execute sequentially even when executed concurrently.

The method for achieving objective (i) is specific to the database transaction policy, and objective (ii) is obtained through serializability. The database analogy to the H_Model is that instead of transactions the H_Model uses commands, and instead of a database the H_Model uses an access matrix called a state matrix.
Objective (ii) in a database hides concurrent execution from the database transaction policy because transitions appear to execute sequentially. The same concept applies to the H.Model because a semantic security predicate defined for sequential commands is applicable even if the commands merely appear to execute sequentially, as opposed to actually executing sequentially.

The motivation for providing concurrency is that sequential models do not reflect all security-relevant events. For example, consider a policy that allows users to access directories of files. Suppose the policy were designed such that a distinct access right is added or removed between a user and every file in a directory whenever access to the directory is granted or removed, respectively. A concurrent model could allow two users to concurrently alter their access rights to the same directory by interleaving their respective sequence of access right modifications to distinct files. We cannot merely assume that a sequential model would be correct if it were executed concurrently. For example, does a sequential model allow a user to fork two processes that add and remove access to the same directory simultaneously? If so, are we assured that there is not some subtle bug in the sequential model that could potentially result in some undefined state? We have found that synchronization problems often result in security problems [3] and as a result, they should be modeled.

Security proofs are inductive over the length of all legal schedules of concurrent commands. Constraints on legal schedules should not violate the constraints imposed by the architecture. For example, commands that are executed on different machines should be allowed to execute concurrently, unless a distributed synchronization constraint (such as secure remote procedure call, or secure distributed semaphore) is assumed or explicitly modeled.

The remainder of this paper is organized as follows. Section 2 presents the H.Model. In section 3 we evaluate our model and compare it with other security models. Section 4 is an example, and section 5 summarizes the paper.

2 The H.Model

The H.Model consists of:

i) A Set of Tokens: Every token has a unique type. Every type has a unique class. There are three classes: \textit{index}, \textit{lock}, or \textit{right}. A token in the class \textit{index} is denoted by \textit{x}, a token in the class \textit{lock} is denoted by \textit{l}, and a token in the class \textit{right} is denoted by \textit{r}. There is an unbounded number of types of class \textit{index}, and a bounded number of types of classes \textit{lock} and \textit{right}.

A type of class index has potentially an unbounded number of tokens, and a type of class lock or right has a bounded number of tokens. A definition that admits parameters from more than one class is denoted by a concatenated name. For example, a parameter that may be in any of the three classes is denoted by \( (xlr)_m \). We also denote index parameters, \( a \) or \( o \) when we want to denote a subject or an object as intended in the HRU model [12].

ii) A Finite Set of Commands: A command is of the form:

\[
\text{command } c_j( x_1 : \text{type}_{s_1}, \ldots , x_u : \text{type}_{s_u}, \\
\quad l_1 : \text{type}_{b_1}, \ldots , l_v : \text{type}_{b_v}, \\
\quad r_1 : \text{type}_{c_1}, \ldots , r_w : \text{type}_{c_w} ) = \\
\quad P_{t_1} \\
\quad \ldots \\
\quad P_{t_n}
\]

Here, \( c_j \) is a name, and \( u, v, \) and \( w \) are constants. Each formal parameter is a token. The formal parameters \( x_1, \ldots , x_u \) are of types \( \text{type}_{s_1}, \) for \( k = 1 \ldots u \). The formal parameters \( l_1, \ldots , l_v \) are of types \( \text{type}_{b_1}, \) for \( k = 1 \ldots v \). The formal parameters \( r_1, \ldots , r_w \) are of types \( \text{type}_{c_1}, \) for \( k = 1 \ldots w \). Each \( P_{t_a} \) for \( a = 1 \ldots n \) is one of the following operations:

i) \text{enter}(zlr), s, o

ii) \text{delete}(zlr), s, o

where \( s \) and \( o \) are formal parameters whose type is of class index.

2.1 Components

The operations in a command are transitions on the global state. The global state is an unbounded state matrix \( M \), with a row and column for every token whose type is of class index. A coordinate of \( M \) is denoted \( [s, o] \). The value of \( M[s, o] \) is a subset of the tokens. A scheduler accepts a set of commands as input and issues the operations in a given command in the order in which they are written. A command in a set that contains a formal parameter of the wrong type is ignored. The scheduler may interleave operations in distinct commands.

For example, consider 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 model of the system may contain five types: host, disk, read, write, and mutex, of classes, index, index, right, right, and lock, respectively. Assume two host
tokens, and one token from each of the other types is defined. A command assures that only one host has disk access by entering and deleting the lock, l1. An example command for this model has the following form:

command c1( x1 : host, x2 : disk, r1 : read, l1 : mutex, l2 : write) =
  enter(l1, x1, x2)
  enter(r1, x1, x2)
  enter(l2, x1, x2)
  delete(l1, x1, x2)
  delete(l2, x1, x2)
  delete(l1, x1, x2)

The body of a command is a sequence of operations. The semantics of an operation is defined in terms of the transition function, $t$:

$$set \ of \ operations \times \ set \ of \ states \rightarrow set \ of \ states$$

The H_Model has two operations: enter and delete which insert, and remove a token to or from the state matrix, respectively. A token's type distinguishes tokens with different semantics, e.g., a host, a disk, an access right. A type's class distinguishes types of different purposes. The index, lock, and right classes represent indirection, synchronization constraints, and access privileges, respectively. An enter and delete operation may modify the value of at most a single coordinate.

The H_Model operations are analogous to the HRU [12] model enter and delete operations which insert a token and delete a token in the state matrix, respectively. A difference between the H_Model and the HRU model concerns lock variables. A lock may not be entered in the state matrix where the lock already exists; and lock may not be deleted where the lock does not exist. If an operation cannot execute, the operation blocks. The H_Model does not contain create or destroy subject operations as in the case of the HRU model, because subjects and objects and their respective status (created, not created, and destroyed) can be constructed using the H_Model primitives [4].

The formal description of the H_Model operations is given below:

i) $t(enter((zlr)_a, s, o))$

$$\forall s', o'. \ t(enter((zlr)_a, s, o), M)[s', o'] =$$

$$\begin{cases} 
M[s', o'] & \text{if } s' \neq s \text{ or } o' \neq o \\
M[s', o'] \cup \{(zlr)_a\} & \text{otherwise}
\end{cases}$$

This operation puts $(zlr)_a$ in $M[s, o]$. If $(zlr)_a$ is a lock, then the operation blocks until $t(enter((zlr)_a, s, o), M)[s', o'] \neq M$; otherwise, the operation is not blocked. Informally, if $(zlr)_a$ is a lock, the operation blocks until $(zlr)_a$ is not in $M[s, o]$. If $(zlr)_a$ is not a lock, then the operation is executed regardless of the value of $M[s, o]$.

ii) $t(delete((zlr)_a, s, o))$

$$\forall s', o'. \ t(delete((zlr)_a, s, o), M)[s', o'] =$$

$$\begin{cases} 
M[s', o'] & \text{if } s' \neq s \text{ or } o' \neq o \\
M[s', o'] \setminus \{(zlr)_a\} & \text{otherwise}
\end{cases}$$

This operation removes $(zlr)_a$ from $M[s, o]$. If $(zlr)_a$ is a lock, then the operation blocks until $t(delete((zlr)_a, s, o), M)[s', o'] \neq M$; otherwise, the operation is not blocked. Informally, if $(zlr)_a$ is a lock, the operation blocks until $(zlr)_a$ is in $M[s, o]$. If $(zlr)_a$ is not a lock, then the operation is executed regardless of the value of $M[s, o]$.

The semantics of an example command is given below.

command c2(m1 : host, l1 : dsk1, l2 : req, l3 : bbuf) =
  delete(l1, m1, 4)
  enter(m1, 6, 7)
  enter(l3, 6, m1)
  enter(q, 3, 4)
  delete(l3, 3, 4)

Command, $c_2$, accepts four formal parameters $m_1, l_1, l_2, l_3$, of types host, dsk1, req, and bbuf, respectively. The class of the host type is index, and the classes of dsk1, req, and bbuf are lock. The token q is a constant whose class is right. The command waits until $l_1$ may be deleted from $M[m_1, 4]$. Next, index $m_1$ is entered into $M[6, 7]$. The command then waits until $l_2$ may be entered into $M[6, m_1]$. Next, right $q$ is entered into $M[3, 4]$. The command then waits until $l_3$ can be deleted from $M[3, 4]$. Assuming that the command runs to completion when 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 $m_1$ and $q$ are in the original matrix.

A command defines the correctness criteria of the security enforcing mechanisms. Since distinct security enforcing mechanisms execute concurrently, an execution history is represented as a sequence of atomic operations which may be interleaved with operations from different commands. Whenever commands must be synchronized in order to enforce the security predicate, explicit synchronization constraints must be modeled. The synchronization constraints are presented in the form of locks. An important
synchronization constraint is a critical section. A critical section is a sequence of operations that execute atomically with respect to operations from different commands. In section 2.2 we show how to build critical sections using locks, and demonstrate that correctly formed critical sections imply serializability.

2.2 Scheduler

This section presents recursive equations that define the H.Model. The equations represent our recursive specification of the H.Model being implemented in the Gypsy verification environment.[10] The equations could have been specified iteratively, but are specified recursively in order to simplify the proofs.

When a command is issued, the command runs to completion by executing its operations in order. We write $c_j = p_1, \ldots, p_n$ to denote that command $c_j$ is the operation sequence $p_1, \ldots, p_n$, where $|c_j| = n$. We denote the first operation in a command $c_j$ by $\text{first.c}(c_j)$, and the remainder of the operation sequence $\text{rest.c}(c_j)$, where $|c_j| = 0$ implies $\text{first.c}(c_j)$ is undefined and $\text{rest.c}(c_j) = c_j$. The coordinate referenced in operation $p_n$ is denoted by $\text{coord}(p_n)$; the kind of operation (enter or delete) referenced in operation $p_n$ is denoted by $\text{op}(p_n)$; the token referenced in operation $p_n$ is denoted $\text{tok}(p_n)$; and the class of the token referenced in operation $p_n$ is denoted $\text{class}(p_n)$. Execution of command $c_j$ is given by the function $T$ which sequentially applies the operations in $c_j$ to the state matrix:

$$T(c_j, M) = \begin{cases} M & \text{if } |c_j| = 0 \\ \text{t(first.c(c_j), M)} & \text{if } |c_j| = 1 \\ T(\text{rest.c}(c_j), T(\text{first.c}(c_j), M)) & \text{otherwise} \end{cases}$$

Definition 1. Multiple commands can be executed concurrently by interleaving operations in the commands. The set of all possible interleaving histories, $h$, of commands of a set of commands, $C_i$, is an interleaved set (iset).

$$\text{iset}(C_i) = \{ h | \text{iset}(C_i, h) \}$$

where

$$\text{iset}(C_i, h) = \begin{cases} \text{true} & \text{if } |h| = 0 \land \forall c_k \in C_i : |c_k| = 0 \\ \text{true} & \text{if } \exists c_k \in C_i : \text{first.c}(c_k) = \text{first.c}(h) \land \text{iset}(C_i - \{c_k\}) \land (\text{rest.c}(c_k), \text{rest.c}(h)) \\ \text{false} & \text{otherwise} \end{cases}$$

In other words, the iset of a set of commands is a set of sequences of operations. Iset is defined recursively, where for each $h$ in iset, first.c(h) is equal to the first operation in some element $c_k$ in $C_i$. An interleaving contains all the operations in the commands, and preserves the relative ordering of operations. For example,

$$\text{iset}(\{ p_1, p_2, p_3, p_4, p_5 \} ) =$$

$$\{ p_1, p_2, p_3, p_4, p_5 \} \cup \{ p_3, p_4, p_5, p_4, p_5 \}$$

Not every interleaving of a set of commands may be scheduled because of the semantics of the blocking operations given in section 2.1. For this reason, we define a scheduleset.

Definition 2. A scheduleset of a set of commands and an initial matrix is the set of all legal schedules of the respective commands according to the semantics presented in section 2.1.

$$\text{scheduleset}(C_i, M) = \{ h | h \in \text{iset}(C_i) \text{and legal}(h, M) \}$$

where

$$\text{true} \text{ if } |h| = 0$$

$$\text{true} \text{ if } \text{first}(h) = \text{enter}(s, a, o) \land \text{legal}(\text{rest.c}(h), \text{t(first}(h), M))$$

$$\text{true} \text{ if } \text{first}(h) = \text{delete}(s, a, o) \land \text{legal}(\text{rest.c}(h), \text{t(first}(h), M))$$

$$\text{false} \text{ otherwise}$$

In other words, a scheduleset of a set of commands and an initial matrix is a subset of the iset of the commands. Operation sequences in an iset in which locks are entered where they already exist in the state matrix, or deleted where they do not exist in the state matrix, do not appear in a scheduleset.

The next two definitions provide serializability. A command sequence $c_1, \ldots, c_n$, is denoted $\hat{c}$. The first command in $\hat{c}$ is denoted first.p(\hat{c}), and the remainder of the command sequence is denoted rest.p(\hat{c}), where $|\hat{c}| = 0$ implies first.p(\hat{c}) is undefined, and rest.p(\hat{c}) = $\hat{c}$. The definitions of first.p and rest.p are compared with first.c and rest.c in the example below.

$$\hat{c} = p_1 p_2 p_3 p_4 p_5$$

- first.p($\hat{c}$) = $p_1 p_2 p_3$
- rest.p($\hat{c}$) = $p_4 p_5$
- first.c($\hat{c}$) = $p_1$
- rest.c($\hat{c}$) = $p_2 p_3 p_4 p_5$
Definition 3. The set of permutations of commands in a command set is a perm.

\[ \text{perm}(C_i) = \{ [\epsilon, \text{perm}(C_i, \epsilon)] \} \]

\[ \text{perm}(C_i, \epsilon) = \begin{cases} 
\text{true} & \text{if } |C_i| = 0 \land \epsilon = 0 \\
\text{true} & \text{if } \text{first.p}(\epsilon) \in C_i \land \\
\text{perm}(C_i - \{ \text{first.p}(\epsilon) \}, \text{rest.p}(\epsilon)) & \text{otherwise}
\end{cases} \]

For example,

\[ \text{perm}(\{c_1, c_2, c_3\}) = \{c_1c_2c_3, c_1c_3c_2, c_2c_1c_3, c_2c_3c_1, c_3c_1c_2, c_3c_2c_1\} \]

Definition 4. A command set is serializable if and only if every schedule is executable, and every schedule yields the same final state as if the commands were executed in some serial order.

\[ \text{serializable}(C_i) \iff \\
\forall M \forall h \in \text{scheduleset}(C_i, M) \\
\exists \epsilon \in \text{perm}(C_i) \quad T(h, M) = T(\epsilon, M) \]

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

The number of serializable, concurrent schedules is not always polynomial in the size of the commands in a given command set [4]. Therefore, an algorithm that determines if a given set of commands is serializable is not necessarily efficient, if the algorithm executes by enumerating every possible concurrent schedule. We avoid enumeration by providing a polynomial time algorithm that computes two serializability conditions. The correctness criterion of the serializability conditions is if the serializability conditions are satisfied by a given command set, then the command set is serializable. The serializability conditions are satisfied whenever all critical sections are nested, and all operations in all distinct commands that reference common coordinates are in shared critical sections. The algorithm that computes the serializability conditions is a straightforward application of the respective definitions of the serializability conditions given below.

Definition 5. The first operation in a critical section (crit) enters a lock in a coordinate \((s, o)\), and the last operation in the critical section deletes the lock from the same coordinate.

\[ \text{crit}(p_m, p_n, c_j) \iff \\
p_m, p_n \in c_j \land \\
op(p_m) = \text{enter} \land op(p_n) = \text{delete} \land \text{tok}(p_m) = \text{tok}(p_n) \land \\
\text{coord}(p_m) = \text{coord}(p_n) \land a < b \land \text{class}(p_m) = \text{lock} \]

Definition 6. \( p_{m_j} \) and \( p_{m_k} \) in \( c_k \) share a critical region, \( \text{scr}(c_j(p_{m_j}), c_k(p_{m_k})) \), if and only if \( p_{m_j} \) and \( p_{m_k} \) are in critical sections that share the same lock and coordinate.

\[ \text{scr}(c_j(p_{m_j}), c_k(p_{m_k})) \iff \\
\exists p_{m_j}, p_{m_k}, p_{m_k} \quad \text{crit}(p_{m_j}, p_{m_k}, c_j) \land \\
\text{crit}(p_{m_k}, p_{m_k}, c_k) \land \\
\text{coord}(p_{m_j}) = \text{coord}(p_{m_k}) \land \\
\text{tok}(p_{m_j}) = \text{tok}(p_{m_k}) \land \\
a_j \leq m_j \leq b_j \land a_k \leq m_k \leq b_k \]

Definition 7. A command set, \( C_i \), has proper critical regions (\( \text{pcr}(C_i) \)) if every pair of operations in distinct commands that reference a common coordinate are in shared critical regions.

\[ \text{pcr}(C_i) \iff \\
\forall c_j, c_k \in C_i \forall p_{m_j} \in c_j \forall p_{m_k} \in c_k \\
\text{coord}(p_{m_j}) = \text{coord}(p_{m_k}) \Rightarrow \text{scr}(c_j(p_{m_j}), c_k(p_{m_k})) \]

Definition 8. A command, \( c_j \), is nested, \( \text{nest}(c_j) \) if and only if all critical sections in the command are nested.

\[ \text{nest}(c_j) \iff \\
\forall p_{m_j}, p_{m_k}, p_{m_k} \quad \text{crit}(p_{m_j}, p_{m_k}, c_j) \land \\
\text{crit}(p_{m_k}, p_{m_k}, c_j) \Rightarrow \\
a < d < c < b \lor d < a < b < c \]

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" [17]. The theorem that dynamic two-phase locking assures serializability is proved by Papadimitriou in [17].

Theorem 1. If a set of commands has proper critical regions and is nested, then the set of commands is serializable.

\[ \text{pcr}(C_i) \land \text{nest}(C_i) \Rightarrow \text{serializable}(C_i) \]

Theorem 1 is a corollary of Papadimitriou's theorem.

3 Evaluation

Sequential security models, e.g., [2,7,9,11,12,13], are usually defined in terms of finite state machines. Each model defines a secure initial state, and proves by induction that every state reachable from a secure initial state is itself secure. Most models are in one of two general categories: access matrix, e.g., [12], or information flow, e.g., [9]. An access matrix model defines access privileges possessed by different entities in a system. An information flow model defines properties of input and output. An
advantage of an access matrix-based model is it defines the implementation of operating system security-enforcing mechanisms.\(^2\)

A disadvantage of an access matrix-based model is that it does not define security for all types of information flow. For a stronger definition of security, an information flow model is required. An advantage of an information flow model is that it defines security for some legitimate or storage channels that are considered covert with respect to an access control model. A disadvantage of an information flow model is that it is difficult to implement a system that actually enforces the model. The H.Model is an access matrix-based model, which coupled with a covert channel analysis such as the one described in [8], provides enough security assurances for many applications.

This section compares the H.Model with some related models: the Goguen-Meseguer model [9], the Odyssey Restrictiveness model [15], and the Lucid knowledge model [8]. All of the related models discussed in this section express abstract properties of security predicates. The Restrictiveness model and the Lucid knowledge model use their abstract properties to compose secure components into an aggregate system. A specific security predicate (such as the is-property and the s-property described in the Bell and La Padula model [2]) can be designed and verified by showing that the predicate is a special case of some abstract semantic property. The H.Model differs from the example models because the H.Model provides a syntactic security property. The H.Model syntactic serializability property can be used in conjunction with a semantic security property provided by another model.

The Goguen-Meseguer model [9] is historically an important security model in the context of secure networks because it introduces the MLS non-interference property. The MLS non-interference property states that one process should be prohibited from detecting any operation executed by another process unless allowed by the information flow rules stated in the security policy. The MLS non-interference property has the appeal of being close to the intuitive notion of security. The primary problems with the MLS non-interference property are: it is too restrictive, and it does not provide concurrency. The other related models presented in this section [8,15] extend the MLS non-interference property.

The Odyssey Restrictiveness model [15] provides composability, non-determinism, and interrupts. A basic difference between the Restrictiveness model and the H.Model is their respective definitions of secure buffers: use of buffers. The Restrictiveness model does not use a blocking bounded buffer because of a potential covert storage channel. The H.Model uses a blocking bounded buffer, but hides the buffer so that it may not be directly perceived by an untrusted subject. The hidden buffer is indirectly accessed via a trusted subject that maintains its own buffer queue. Eventually, every item in the trusted subject queue will be placed in the buffer, but flow control is restricted by explicit locks defined in the H.Model. By hiding bounded buffers the H.Model is not able to eliminate the covert channel, but is able to assure that the covert channel is presented in the form of a timing channel as opposed to a storage channel. Timing channels generally have lower bandwidth than storage channels, and as a result have lower risk.

The Lucid knowledge model [8] provides composability and temporal logic. The Lucid knowledge model is an operator net model which is syntactically similar to a data flow model. Semantically, it provides reasoning processes that are able to deduce knowledge concerning other processes. By using knowledge and reasoning processes, one may potentially be able to define a wider class of semantic information than is available using standard modeling techniques. A problem with the Lucid knowledge model is it uses unconventional syntax that is incompatible with existing verification environments (e.g. Gypsy [10]). Unlike the other related models discussed in this section (9,15 and the H.Model) which are defined using first order logic, the Lucid knowledge model uses modal operators which may express temporal ordering. The H.Model may express some aspects of temporal ordering by saving information tokens that represent properties of old states in its state matrix. Such representations are clumsy when compared with Lucid.

A potential drawback of the H.Model with respect to the sample models is that the H.Model has an implicit semantic notion of security which is somewhat weaker than the notions expressed in the sample models. All of the sample models express properties that are historically derivative from Goguen-Meseguer’s MLS non-interference property. Non-interference allows for a stronger notion of security than can be expressed in access matrix models because some covert communication paths could potentially be established without defining a specific access path in an access matrix.

We believe the disadvantage of the higher risk approach of the H.Model is offset by its advantages:

- The H.Model can easily define many important currently-existing models. The HRU [12], and the take-grant [13] models are two example models that can be expressed in terms of the H.Model [4]. Other models such as the Bell and La Padula model [2] which can be syntactically defined in terms of HRU [18] can also be syntactically defined in terms of the H.Model. The security predicate of such models is
4.1 Overview

The Bell-La Padula model defines a system as a set of appearances. An appearance is a sequence of the form:

\[ M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots \]

where each \( M_i \) is a state, and each \( e_j \) is a state transition (command). In an unrelated paper [14], Lamport suggests that the behavior of any discrete system can be described as such a set of appearances. The Multics Interpretation of the Bell-La Padula model [2] provides eleven "rules" that act as state transitions. The set of rules, combined with an initial state, \( M_0 \), provide a definition of a system, i.e., a set of appearances. The purpose of the Bell-La Padula model is to define a secure system. A given implementation is proven secure if its behavior (set of appearances) is a subset of a Bell-La Padula secure system. A system-security predicate is function computed over a system that is satisfied only if the system is secure. The Bell-La Padula system-security predicate is satisfied if and only if every state in every appearance satisfies a state-security predicate. The Bell-La Padula state-security predicate is the conjunction of the ss-property, the +-property, and the da-property. Since the definition of system is discrete, the states in each appearance can be enumerated with the non-negative integers, and security can be proved by induction. The base case of the inductive proof demonstrates that the initial state satisfies the state-security predicate, and the inductive case demonstrates that no rule may move from a secure state to a non-secure state.

DBL provides an alternative description of a secure system. The DBL system is a super set of the Bell-La Padula system because concurrent transitions are allowed. Therefore, any system proved secure according to the Bell-La Padula model is also secure according to the DBL model, and in addition, some distributed systems are also secure.

4.2 Security Predicate

As described above, the Bell-La Padula security predicate is the conjunction of the ss-property, the +-property, and the da-property. For simplicity, we only describe the +-property in DBL, but an extension that includes the other two properties is straightforward. Also, the Bell-La Padula model distinguishes between the current security level and the security level of a subject or object, but for simplicity, DBL omits this distinction. The +-property is a function defined in terms of subjects, objects, and a partially ordered set of security levels. The partial ordering relation is called dominates. The +-property states:
i) If a subject has append access to an object, then the current security level of the object dominates the current security level of the subject.

ii) If a subject has write access to an object, then the current security level of the object dominates the current security level of the subject.

iii) If a subject has read access to an object, then the current security level of the subject dominates the current security level of the object.

The DBL represents the *-property by data structures that define privileges, security levels, and security level functions. A privilege is an access permission (right). Example privileges are append (a), write (w), and read (r). Security levels are partially ordered levels of trust. The partial order relation (dom) is explicitly represented in the state. The security level functions are the mappings from subjects and objects to security levels, and are also explicitly encoded in the state. The form of the DBL *-property is given below:

\[ a \in M[s, o] \Rightarrow \text{security level of } o \text{ dominates security level of } s \]

\[ w \in M[s, o] \Rightarrow \text{security level of } o \text{ dominates security level of } s \]

\[ r \in M[s, o] \Rightarrow \text{security level of } s \text{ dominates security level of } o \]

DBL represents security levels by encoding the dom right between the indices that represent security levels according to the partial order. Consider a policy that defines three security levels, \( lev_1, lev_2, \) and \( lev_3 \), such that \( lev_1 \) dominates \( lev_2 \), \( lev_1 \) dominates \( lev_3 \), and \( lev_2 \) and \( lev_3 \) are not related. This configuration is represented in the state below.

\[
\begin{array}{c|c|c|c}
lev_1 & lev_2 & lev_3 \\
\hline
dom & dom & dom \\
\hline
lev_2 & dom \\
\hline
lev_3 & dom \\
\end{array}
\]

The security level functions are defined using the right, \( f \). For example, if \( f \in M[lev_1, o] \), then the security level of \( o \) is \( lev_1 \). An example configuration is given in the state below.

- The security level of \( s_1 \) is \( lev_2 \)
- The security level of \( s_2 \) is \( lev_1 \)
- The security level of \( s_3 \) is \( lev_3 \)
- The security level of \( o_1 \) is \( lev_2 \)
- The security level of \( o_2 \) is \( lev_1 \)
- The security level of \( o_3 \) is \( lev_3 \)

Access privileges are indicated by right tokens. A state satisfies the *-property if and only if the star predicate given below is satisfied.

\[ \star(M) \text{ iff } \forall s, o, lev_1, lev_3
\]

\[
(a \in M[s, o] \land f \in M[lev_3, s] \land f \in M[lev_1, o] \Rightarrow dom \in M[lev_1, lev_3]) \\
(w \in M[s, o] \land f \in M[lev_3, s] \land f \in M[lev_3, o] \Rightarrow dom \in M[lev_3, lev_1]) \\
(r \in M[s, o] \land f \in M[lev_3, s] \land f \in M[lev_1, o] \Rightarrow dom \in M[lev_1, lev_3])
\]

An example state matrix that satisfies the star predicate is given below. The example state matrix has three privileges:

i) \( a \in M[s_1, o_1] \)

ii) \( r \in M[s_2, s_1] \)

iii) \( w \in M[s_3, o_3] \)

Privilege (i), for example, indicates that \( s_1 \) has append access to \( o_1 \). Since \( f \in M[lev_3, s_1] \), and \( f \in M[lev_1, o_1] \), the security levels \( s_1 \) and \( o_1 \) are \( lev_2 \) and \( lev_1 \), respectively. Since, dom \( \in M[lev_1, lev_3] \), the security level of \( o_1 \) dominates the security level of \( s_1 \). Therefore, privilege (i) satisfies the *-property as given in the star predicate. The other two privileges can be shown to satisfy the star predicate through similar reasoning.
4.3 Model of Computation

The DBL model of computation is a list of commands. Each DBL command is analogous to a Bell-La Padula rule. The commands change the state by modifying privileges or security level functions. The purpose of the commands is to define a secure system, e.g. set of appearances. As described in section 4.1, a specification is proved secure according to DBL by demonstrating that the specified system is a subset of a DBL system. Some appearances in a DBL system need not be included in any specification, because they contain deadlocks. Although deadlocks should be avoided in any practical implementation, deadlocks are not non-secure unless the deadlock occurs in a non-secure state. An example command called get-read which enters the read (r) privilege is given below. The get-read command proceeds to completion if its result satisfies the stor predicate, and blocks without changing the state otherwise.

\[
\text{get-read}(s, o, lev_s, lev_o) = \\
\text{enter}(1, lev_s, lev_o) \\
\text{delete(lock: dom, lev_s, lev_o)} \\
\text{enter(dom, lev_s, lev_o)} \\
\text{enter}(1, lev_s, o) \\
\text{delete(lock: f, lev_s, s)} \\
\text{enter}(f, lev_s, s) \\
\text{enter}(1, lev_s, o) \\
\text{delete(lock: f, lev_s, o)} \\
\text{enter}(f, lev_s, o) \\
\text{enter}(1, s, o) \\
\text{enter}(r, s, o) \\
\text{delete}(1, s, o) \\
\text{enter}(1, lev_s, o) \\
\text{enter}(1, lev_s, s) \\
\text{enter}(1, lev_s, lev_o)
\]

The get-read command executes by computing three tests. If any of the tests are not satisfied, then get-read deadlocks. The first test validates that the security level of the subject dominates the security level of the object. The next two tests validate that the security level function parameters are associated with the respective subject and object parameters.

The get-read command is secure, because the r right is not entered into the state unless the security level of the subject dominates the security level of the object. We claim but do not formally prove here, that if the state is secure according to the stor predicate before get-read executes, then the resultant state after get-read executes is also secure. Furthermore, get-read satisfies the conditions of theorem 1 of section 2.2.

The get-read command, as presented here, is intended to illustrate the H-Model, and is not intended as an alternate formalism of the Bell-La Padula model. Some aspects are the Bell-La Padula model are omitted for simplicity, and other aspects of the Bell-La Padula model should be enhanced in an analogous distributed model. For example, the DBL as presented here, does not distinguish between active subjects and passive objects, but DBL can be extended to include these features [4]. Also, DBL can be augmented with a more complex data structure for locks, so that more concurrency can be modeled.

5 Conclusion

The H-Model defines security for distributed applications and concurrent applications implemented on centralized systems. The H-Model differs from other security models for distributed environments because the H-Model hides the distributed model of computation from the security predicate. In section 2.2 we prove this result. The H-Model is currently being used to define security for a distributed Multilevel Secure file system.

Our definitions in section 2.2 are being implemented in the Gypsy [10] formal programming environment. We structure our Gypsy proof on an abstract machine. The abstract machine is the unbounded size state matrix and a transition function. The transition function accepts a sequence of operations and produces a final state. We use Gypsy lemmas to prove the serializability theorem. All of our functions in the Gypsy implementation are functional and zero assignment - all code is implemented in the form of functions, and local variables are not used. Each lemma is defined in terms of an arbitrary vector and functions that operate on elements of the vector. As a result, universal quantification is implicit without the need for explicit quantifiers.

In a related paper, [4], we extend our results to include a wider class of secure systems. We show that the critical section conditions of theorem 1 combined with a least privilege condition,
assures security for intermediate states as well as final states. The purpose of the extended result is to prove that if the critical section conditions, and the least privilege conditions are satisfied by a given set of commands, and the set of commands satisfies the security predicate when executed sequentially, then every state reachable from a secure initial state through concurrent execution is secure.

In future work we will formalize a mapping between a specification language described in [14] and the H-Model. The mapping provides a formal framework for modeling and specifying secure distributed systems. In addition, the specification language augments the model by defining liveness predicates.

References


