A Rule-Set Approach to Formal Modeling of a Trusted Computer System

Leonard J. LaPadula The MITRE Corporation

ABSTRACT: This article describes a rule-set approach to formal modeling of a trusted computer system. A finite state machine models the access operations of the trusted system while a separate rule set expresses the system's trust policies. A powerful feature of this approach is its ability to fit several widely differing trust policies easily within the same model. The paper shows how this approach to modeling relates to general ideas of access control. Departing from the traditional abstractness of discussions of formal models, the paper also relates this approach to the implementation of real systems by connecting the rule set of the model to the system operations of a UNIX System V system. This gives high confidence that a real system could clearly derive from the elements of the formal model instead of additionally depending on numerous design and policy decisions not addressed in the model. Neither are the trust policies left largely to the imagination of the reader—the rule base has detailed specifications of the mandatory access control policy of UNIX System V/MLS, a version of the Clark-Wilson integrity policy, and two supporting policies that implement roles. A fundamental point established by the work reported in this article is that formal modeling can be moved considerably closer to implementation of real systems, a fact that has great beneficial impact on the possibility of building high assurance trusted systems.

1. Introduction

This paper describes a rule-set approach to formal modeling for a trusted computer system. The approach uses a State Machine Model and a Rule Set Model. This rule-based approach has its roots in several ideas fostered in the Generalized Framework for Access Control (GFAC) vision (Abrams, 1990). The modeling approach responds to the challenge of that vision, as expressed in these objectives:

- Make it easy to define and formalize access control policies besides traditional mandatory access control (MAC) and discretionary access control (DAC), to increase the availability of diverse security policies.
- Make it easy to add new security policies to a complete formal model without having to re-do the entire model.
- Make it feasible to configure a system with security policies chosen from a vendor-provided set of options with confidence that the resulting system security policy makes sense and will be properly enforced.
- Construct the model in such a way that it can be shown to satisfy an accepted definition of each security policy it represents.

Following an overview of the general framework, the article describes a rule-set approach to modeling that uses a State Machine Model and a Rule Set Model. To demonstrate the approach concretely, the article also gives examples of the model components, describing detailed rules of operation, several security policies, and formal expressions of the rules of operation and the policies. The remainder of this article has five sections.

- Formal Modeling Approach: This section presents general concepts of access control and shows how the rule-set approach relates to them.
- Interface Between the State Machine and the Rule Set: This section explains how the State Machine and the Rule Set relate to each other and gives a comprehensive list of specific interface messages.
- State Machine Model: This section describes a state machine model based

on the architecture of a UNIX®[1] system and gives several detailed examples of its rules of operation.

- Rule Set Model: This section specifies a rule set model having four policies:
  - Mandatory Access Control (MAC) Policy: The MAC modeled is the policy of UNIX System V/MLS.
  - Clark-Wilson Integrity (CWI) Policy: The CWI policy provides control over modification of information by regulating the transactions that users can apply to files of information. This policy employs roles and types and execute-control lists (the Clark-Wilson triples). Its inclusion shows how the commercial data processing requirements described by Clark and Wilson (Clark, 1987) can be modeled and integrated with the MAC policy for a UNIX system.
  - Functional Control (FC) Policy: This is a supporting policy based on roles and types.
  - Security Information Modification (SIM) Policy: This policy controls modification of security information through roles and types.
- Conclusions: Some comments about the rule-set approach are presented.

2. *Formal Modeling Approach*

2.1. *Background*

The Generalized Framework for Access Control (GFAC) (Fig. 1) thesis asserts that all access control is based on a small set of fundamental concepts (Abrams, 1990). Articulation of this view has been enhanced by the terminology and concepts in the ISO "Working Draft on Access Control Framework." (ISO, 1990).

All access control policies can be viewed as *rules* specified in terms of *attributes* by *authorities*. The three main elements of access control in a trusted computer system are:

Authority: An authorized agent must define security policy, identify relevant security information, and assign values to attributes.

Attributes: Attributes describe the characteristics of subjects and objects that will be used within the computer system for decision making about access control.

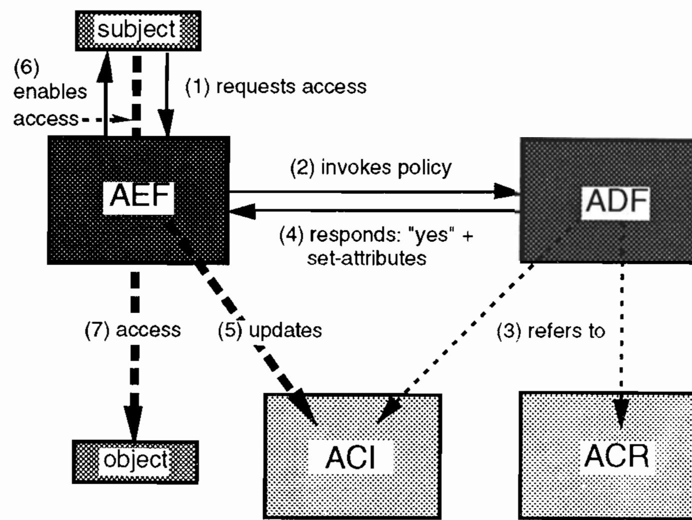

Figure 1. Overview of generalized framework for access control.

Rules: A set of formal expressions define the relationships among attributes and other security information for access control decision in the computer system, reflecting the security policies defined by authority.

The ISO working draft (ISO, 1990) collectively refers to attributes, Access Control Context (ACC) information, and other security-relevant information as Access Control Information (ACI), and it calls rules, appropriately enough, Access Control Rules (ACR). The generalized framework explicitly recognizes the two parts of access control—adjudication and enforcement. The ISO draft uses the terminology Access Control Decision Facility (ADF) to denote the agent that adjudicates access control requests and Access Control Enforcement Facility (AEF) for the agent that enforces the ADF's decisions. In a trusted computer system, the AEF corresponds to the system functions of the Trusted Computing Base (TCB) and the ADF corresponds to the access control rules that embody the system's security policy, also part of the TCB. Figure 1 depicts the generalized framework in the terms just described.

Rule-set modeling has a lot in common with traditional finite state machine modeling. It differs significantly, though, in the way it sets up access rules. Models like the Bell-LaPadula model (Bell, 1976) and the Compartmented Mode Workstation model (Millen, 1990) include access control constraints in their rules of operation. In these models, an Open File rule describes both access policy and system behavior. The Open File rule describes the behavior of the modeled system

as a state transition. It uses built-in criteria to decide if it should permit the Open File request. A typical non-disclosure criterion requires that the security level of a subject requesting the Open File dominate the security level of the object it wants to open. Information affected by the transition might include the set of objects currently held open by the subject that made the request.

The rule-set approach separates the decision criteria from the state transition descriptions. A rule set specifies the security policies of the modeled system while a finite state machine model describes the behavior of the system. In this way we partition the system function Open File into two operations:

- Decide if the request should be granted—is the process allowed to open the referenced file?
- Grant the request—open the file, or not—return an error indication.

Partitioning system functions this way creates a structure in which access policies can be changed without modifying the system operations. Figure 2 depicts this rule-set approach to modeling an Open File function.

The following steps occur when a process makes an Open request:

1. The Open function invokes the rule set, which consists of the access rules that define the trust policies for the system.
2. The rule set adjudicates the request and returns its decision to the Open function.
3. If the request is approved, the Open function performs the system operations necessary to enable the process to access the desired object and allows the process to continue.

2.2. *Structure of the Model*

A trusted computer system built according to this modeling plan would have two major parts inside its Trusted Computing Base (TCB):

- An Access Enforcement Facility (AEF): The AEF owns and operates the system functions available to computer programs.
- An Access Decision Facility (ADF): The ADF keeps the rule set that expresses the system's access policies.

When a program attempts a system function, the AEF appeals to the ADF for an access decision. The AEF will provide some set of arguments to identify the desired access. These arguments and additional access control context information (ACC) provide the information the ADF needs for decision making.

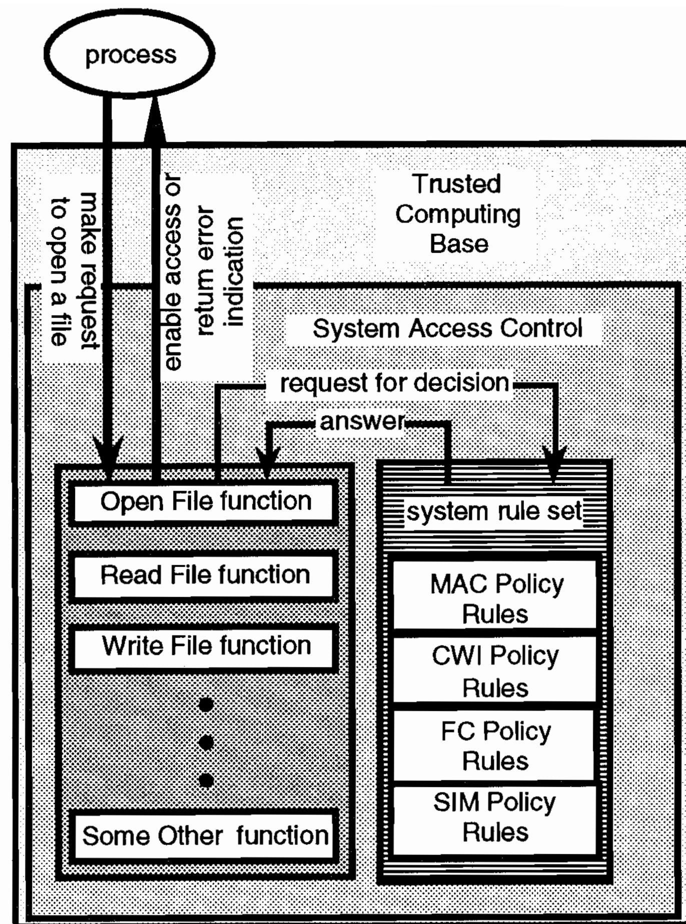

Figure 2. Rule-Set approach to modeling.

Using an AEF-ADF partitioning gives us the following modeling paradigm:

- A State Machine Model, corresponding to an AEF, describes the behavior of the system and the interface of computer processes to the system's TCB.
- A Rule Set Model, viewed as an ADF, defines the security policies of the system and interprets them for the operations of the State Machine Model.

An interface definition relates the State Machine and Rule Set Models to each other. The interface enables the state machine to invoke the rule set for adjudication of a process's request. The interface is defined in terms of requests and arguments used to convey the necessary access control information. The design of the interface depends on several critical factors:

- What system will the state machine model?
- How detailed is the state machine's representation of the system to be?
- Will the rule base deal with the same level of detail as the state machine, or will it deal with abstractions of the system's elements and behavior?

These questions have been settled as follows in this article:

- The state machine targets the class of UNIX System V systems.
- The state machine model includes an operation rule for each UNIX® System V system call that involves the access control policies of the system. Each operation rule is an abstraction of its corresponding system call, but the abstraction preserves the essential functionality of the system call. Having a rule of operation that clearly maps to a system call provides a bridge to the detailed design of a real system having clear correspondences to the elements of the formal model instead of additionally depending on numerous design and policy decisions not addressed in the model. Page, Heaney, Adkins, and Dolsen (Page, 1989) have also commented on the usefulness of emphasizing the operations that a model must support: "Another distinctive aspect of the SMDE is its emphasis on the set of operations which a model is to support. While many traditional model development methodologies postpone operational specifics until implementation, our experience has shown the operational considerations as indispensable to security model design."
- The rule set essentially addresses the same level of detail as the state machine. Still, it has enough generality that it could be useful with other state machines. The form of its generality will become apparent in the description of the interface between the state machine and the rule set in the next section.

3. *Interface Between the State Machine and the Rule Set*

Imagine that a UNIX System V kernel had two parts—an AEF part, which we call the AEF-kernel, and an ADF part, which we call the ADF-kernel. Imagine that a process invokes the Open system call to open a file for reading. The AEF-kernel sends a message to the ADF-kernel to find out if the process's request is valid. The message contains or references the access control information (ACI)

needed by the ADF-kernel to make its decision. The ACI could include many possible items of information. Some basic information items likely to be needed are identification of the requesting process, identification of the file to be opened, and attributes of the process and the file. The ADF-kernel may use other access control context (ACC) information, such as the time of day, to make its decision. The ADF-kernel returns the decision to the AEF-kernel. The AEF-kernel then completes the Open system call, enabling the requested access if the decision was favorable, returning an error message if not.

Each rule of operation in the State Machine Model "invokes" the Rule Set Model with a function called "Access-Rules." The arguments of Access- Rules correspond to the messages exchanged between the AEF-kernel and the ADF-kernel.

A rule of operation for abstractly describing the Open operation might be the following:

Open (file_name, mode):

CONDITION
Access-Rules (open, mode, current_process_aci, file_aci)

EFFECT
Open_Set (current_process) = Open_Set (current_process) UNION (file_name, mode)

The CONDITION means that if the function Access-Rules is true, then do the actions given in EFFECT. In this case, the EFFECT is to add the named file to the set of files accessible by the requesting process and to set its access mode.

We define the interface between the state machine and the rule set by specifying the valid arguments for Access-Rules. At each invocation of the Rule Set Model, the State Machine Model identifies the intended action of the process and a set of relevant attributes (ACI). So, we can define the needed interface by a set of requests with appropriate ACI. Several terms are needed, explained now so that the interface definition will be understandable to the reader.

file	a set of attributes associated with a UNIX file.
directory	a set of attributes associated with a UNIX directory.
ipc	a set of attributes associated with a UNIX storage object used for inter-process communication: these objects may be message queue or semaphores.

scd — a set of attributes associated with a UNIX system object that stores system control data (therefore the acronym "scd"); the UNIX inode is an example. When we use "scd" in the subsequent interface definition, we also show what the scd is referring to—either a file or a directory.

A final word is needed here before presenting the interface messages. The reader will see the requests CHANGE-ROLE and MODIFY-ATTRIBUTE in the interface. These requests have no counterpart in the set of UNIX System V system calls, but they are needed for the security policies we are interested in modeling. The meaning and use of these additional requests will become clear later.

Each element of the interface definition has the form "request (argument list)." The request usually identifies the action that a process wants to do, but it also may be used for simply sending information from the State Machine Model to the Rule Set Model. The argument list identifies a set of access control information (ACI). In the list that follows, terms like "process" or "object" are shorthand for "attributes associated with a process" or "attributes associated with an object." The requests described below seem appropriate for detailed modeling of systems like UNIX System V. Some notable features are as follows:

- For each system call of UNIX System V, as defined by Bach (Bach, 1986), there is at least one request that relates to its functionality. On the other hand, a single rule of operation that models a system call might use several requests. A Create File rule of operation, for example, might invoke Access-Rules twice. First it might need to know if the requesting process has permission to search the directory in which the file will be located. Then, if the search is valid, it again would appeal to the Access-Rules for a policy decision on creating the file.
- The set of requests we have defined here is not minimal. Several requests represent variations of writing to an object; for these separate requests we could have substituted a single request with arguments. I believe my choice enhances the intuitive understanding of the modeler and affords greater flexibility in modeling the class of systems we have targeted.

3.1. Interface Messages

ALIAS (process, file). The process is attempting to create an alternate name for the file. A state machine model of UNIX would use this request in its rule of operation for linking to a file.

ALTER (process, ipc). The process wishes to access the control information for an ipc-type object. This request relates to reading or modifying data about the ipc-type object. This is similar to the modify-permissions-data and get-permissions-data requests defined below for the access control information associated with files and directories. In a UNIX environment this request would be used by the system calls that control message queues, semaphores, and shared memory.

CHANGE-OWNER (process, scd(file/directory)). The process wants to change the owner of the indicated object. An scd object is one that contains system access control information. In UNIX this is the inode. The parenthetical remark "file/directory" means that the scd pertains to a file or directory. The attribute set passed to the Rule Set Model will consist of attributes of the file or directory and an attribute that identifies the object to be modified.

CHANGE-ROLE (process, role attribute, role value). The process wants to change the role of the owner of the process. The argument "role attribute" names the attribute to modify and the argument "role value" gives the desired role.

CLONE (process1, process2). Process1 wants to create a clone of itself, process2. In a UNIX environment this corresponds to a fork system call.

CREATE (process, file/directory/scd/ipc). The process wants to create a new file, directory, scd-type-object, or ipc-type-object.

DELETE (process, file/directory/ipc). The process wants to delete the indicated object.

DELETE-DATA (process, file). The process wants to truncate (remove all data from) the file.

EXECUTE (process, file). The process wants to execute the file. This request compares to the UNIX exec system call.

GET-PERMISSIONS-DATA (process, scd(file/directory)). The process wants to read discretionary access permissions for the indicated file or directory.

GET-STATUS-DATA (process, scd(file/directory)). The process wants to read status data about the file or directory. This corresponds to a UNIX stat system call, which can return information such as file type, file owner, access permissions, and file size.

MODIFY-ACCESS-DATA (process, scd(file/directory)). The process wants to modify access information about the object, such as the time of last modification. This compares to the UNIX utime system call.

MODIFY-ATTRIBUTE (process, user/process/object, attribute, value). The process wants to modify an attribute of the user, the process, or an object. The argument "attribute" names the attribute to change and the argument "value" gives the new value.

MODIFY-PERMISSIONS-DATA (process, scd(file/directory)). The process wants to modify discretionary access permissions of the object. This request parallels the UNIX chmod system call.

READ (process, directory). The process wants to read data from the indicated directory.

READ-ATTRIBUTE (process, user/process/object, attribute). The process wants to read an attribute of the user, the process, or an object. The argument "attribute" names the attribute to read.

READ&WRITE-OPEN (process, file/ipc). The process wants to open the object for reading and writing. In UNIX the object is either a file or a message queue.

READ-OPEN (process, file). The process wants to open the file for reading.

SEARCH (process, directory). The AEF-part of the TCB needs to read the directory as part of some other operation requested by the process. This corresponds to searching a directory in UNIX; so, all rules of operation that model system calls using the UNIX *namei* subroutine will invoke Access-Rules with this request.

SEND-SIGNAL (process1, process2). Process1 wants to send a signal to process2. This parallels the UNIX kill system call.

TERMINATE (process). The system has terminated the process. The State Machine Model gives this request to the policy model for information only. It empowers the Rule Set Model to update its information base, if necessary.

TRACE (process1, process2). Process1 wants to trace process2. The rule set will interpret this to mean "read/write the memory of process2." This equates to the UNIX ptrace system call.

WRITE (process, directory). The process wants to write data to the directory. The UNIX creat system call may have to search a directory before creating a file. Thus, in a UNIX system built according to our modeling paradigm, the creat system call would use this request to check the process's permission to search the directory involved.

WRITE-OPEN (process, file). The process wants to open the file for writing.

4. State Machine Model

The State Machine Model reflects the kernel architecture of the UNIX System V system as described by Bach (Bach, 1986). This article has several examples of rules of operation of the State Machine Model, not a complete set, since its purpose is to explain the modeling approach, not build a trusted system. Still, I have tried to achieve the breadth and depth of coverage needed to make obvious how a modeler may use this approach. This section defines rules of operation that abstractly describe many of the key system calls of UNIX System V. The selected rules are a subset of what is needed for a complete State Machine Model.

4.1. Introduction

This model uses the term "process" where earlier models used the familiar term "subject." "Process" is less general than "subject" but more suitable to what is being modeled here—the interface between a TCB and the processes it services.

The rules of operation are specified in a programming-like language that should be both readable and intelligible to a wide audience. This same language will be used again later to express the policy rules of the rule set. The reader with computer systems experience should have no trouble understanding the language that expresses the rules. The reader who may be unfamiliar with computer programming languages or who may want to verify the meaning of a language form should see the Appendix, which gives a description of the language as well as the modeling constructs employed.

The rules of operation of the State Machine Model define the valid transitions for the modeled system. Recall that the state machine has a transition rule for each UNIX System V system call. We validly could have chosen instead to make the model rules more primitive than system calls. This gives the benefit of simpler rules of operation but has the undesirable effect of moving the model another level away from the real system. The additional level means that system calls must in general be mapped to several rules of operation by the designer or evaluator of the system. Since this mapping must be made constructively—that is, it is not deducible from the specification of the model—modelers can do it far more easily at the time they create a State Machine Model. The result then is a one-to-one correspondence of system calls to rules of operation.

An example will show the difference between the two approaches. Taking the more abstract approach, we might define the Open rule of operation in the following form, as we saw earlier:

Open (file_name, mode):

CONDITION

Access-Rules (open, mode, current_process_aci, file_aci)

EFFECT

Open_Set (current_process) = Open_Set (current_process) UNION (file_name, mode)

The UNIX open system call is far more complicated than this. It has an option to create the named file under certain circumstances. It also provides an option for the process to cause the file to be truncated (have all its data erased) during opening. The abstract form of the open rule above ignores these options. To relate the UNIX open system call to such abstract rules of operation, then, requires an additional specification of the open system call showing the mapping of the call's functions to several abstract rules of operation. To represent the functions of the open system call, these rules of operation will specify the following transitions: searching a directory, truncating a file, opening a file, and creating a file. We take the other approach instead and show all the options of the open system call in a single corresponding rule of operation for opening a file. This single rule invokes the Rule Set Model as appropriate to determine the permissibility of searching a directory, truncating a file, opening a file, or creating a file.

4.2. Rules of Operation

This section gives the specification for the following rules of operation of the State Machine Model:

Open
Read
Fork
Kill
Unlink

4.2.1. Open

Open (file_name, mode, truncate_option, create_option[2]):

The open system call is the first operation a process performs to access data in a file. When successful, the call returns a file descriptor that will be used by other file operations, such as reading, writing, determining status, and closing the file. If the file does not exist and the create_option argument indicates that the process wishes to create the file in this case, then the call will create the file and open it

in the mode specified. The mode argument indicates the type of open, such as reading or writing, and the truncate_option shows whether the process wants all the current data in the file cleared.

```
IF
    Access-Rules(search, directory_name[current directory or directory from specified pathname
THEN
   SELECT CASE STATUS(file_name)
     CASE STATUS(file_name) == "active" (* the directory search was valid and the
          file exists *)
       SELECT CASE truncate_option
         CASE ON
           IF
             NOT (Access-Rules(delete-data, current_process, file_name)));
           THEN
             error-exit;
           ELSE
             [* truncate the file *];
             [* open the file *];
             OPEN(current_process, file_name) = OPEN(current_process,file_name)
                  SET-UNION {mode};
             set-attributes;
             normal-exit;

         CASE OFF
           IF
             (mode == "read" AND
             Access-Rules(read-open, current_process, file_name))
             OR
             (mode == "write" AND
             Access-Rules(write-open, current_process, file_name))
             OR
             (mode == "read&write" AND
             Access-Rules(read&write-open, current_process, file_name))
           THEN
             [* open the file *];
             set-attributes;
             OPEN(current_process, file_name) = OPEN(current_process, file_name)
                  SET-UNION {mode};
             normal-exit;
           ELSE
             error-exit;

     CASE STATUS(file_name) == "unused" (* the directory search was valid and the
          file does not exist *)
```

```
    SELECT CASE create_option
     CASE create_option == ON
      (* create the file and open it for the type of access specified by the mode argument *)
      [* create the attribute set for the file and set the basic values, such as object-identifier *]
      IF
       Access-Rules(create, current_process, file_name);
      THEN
       set-attributes;
       [* create the file *]
       (* check whether current_process may open the file *)
       IF
        (mode == "read" AND
        Access-Rules(read-open, current_process, file_name))
        OR
        (mode ==  "write" AND
        Access-Rules(write-open, current_process, file_name))
        OR
        (mode == "read&write" AND
        Access-Rules(read&write-open, current_process, file_name));
       THEN
        set-attributes;
        [* open the file *]
        OPEN(current_process, file_name) = OPEN(current_process, file_name)
             SET-UNION {mode};
        normal-exit;
       ELSE
        error-exit;
      ELSE
       error-exit;
     CASE create_option == OFF
      error-exit;
    CASE STATUS(file_name) == "inaccessible" (* the directory search failed – e.g.,
         permission denied, directory non-existent, etc. *)
     error-exit;
   END-SELECT
  ELSE
   error-exit;
```

4.2.2. Read

Read (file_descriptor, buffer, size):

The read system call causes a specified number of bytes (size) to be moved from an open file (file_descriptor) to a data structure (buffer) in the requesting process. The read starts at the next byte after the last byte transferred by a read call so that successive reads of a file deliver the file data in sequence.

```
IF
  "read" is in OPEN(current_process, object[identified by file_descriptor])
  AND
  Access-Rules(read, current_process, object)
THEN
  set-attributes;
  [* read the file *];
  normal-exit;
ELSE
  error-exit;
```

4.2.3. *Fork*

Fork ( ):

The fork system call enables a process to create a new process. The created process, called the child process, is identical to the process that creates it, the parent process, except for their process identifiers. Also, some process-internal variable(s) of the child are set by the kernel so that the child process can recognize itself as the child when it runs, presumably so that it can do something different from its parent.

```
[* create the new process if resources are available *];
IF
  Access-Rules(clone, current_process, new_process);
THEN
  set-attributes;
  Open(new_process, object) = Open(current_process, object) for all objects in the system;
  (* the new process inherits access to all the objects the current process can access *)
  [* complete the fork operation *]
  normal-exit;
ELSE
  error-exit;
```

4.2.4. *Kill*

kill (process-identifier, signal):

The kill system call enables a process to send one of a number of signals to another process. The SIGKILL signal causes the kernel to terminate the target process if appropriate authorizations are satisfied.[4] If the signal is any of the other valid signals, then the process(es) receiving the signal will process the signal in accordance with the specification established by its (their) signal system call(s) or with the default specification for the signal.

```
IF
  (* the process-identifier and signal arguments are valid *)
THEN
  SELECT CASE signal
  CASE signal is SIGKILL (* the sending process is attempting to kill a
        process or group of processes *)
    FOR-EACH process (* specified by the process-identifier argument*):
      (* terminate the process *)
      OPEN(process, file_name) = {}5 for every file_name;
      Access-Rules(terminate6, process);
    END-FOR-EACH;
    normal-exit;

  CASE ELSE
  FOR-EACH process (* specified by the process-identifier argument *):
    IF
      Access-Rules(send-signal, current_process, process) == YES
    THEN
      set-attributes;
      (* send the specified signal to the process *)
    ELSE
      error-exit;
  END-FOR-EACH;
    normal-exit;
  ELSE
   error-exit;
```

4.2.5. Unlink

unlink (file_name):

The unlink system call removes a directory entry for a file. In general, a number of directory entries may exist for a given file, created via the link system call. A file is not deleted until all its names (links) have been removed.

```
IF
  Access-Rules(search, directory_name[current directory or directory from
        specified pathname]
THEN
  SELECT CASE STATUS(file_name)
    CASE STATUS(file_name) == "active" (* the directory search was valid and
          the file exists *)
      IF
        (* unlinking the file will delete the file itself *)
      THEN
        IF
          Access-Rules (delete, current_process, file_name)
```

```
        THEN
          (* delete the file – remove directory entry and return file space
               to system pool *);
          STATUS(file_name) = "unused";
          normal-exit;
        ELSE
          error-exit;
      ELSE
        (* unlink the file – remove directory entry *);
        normal-exit;
    CASE STATUS(file_name) == "unused" (* the directory search was valid and
          the file does not exist *)
      error-exit;
ELSE
  error-exit;
```

4.3. Additional Remarks

The modeler has choices to make. A fundamental question is "Will the rules of operation map one-to-one or many-to-one to system calls?" Successful modeling can be done either way. If the rules of operation are one-to-one with the system calls, they include a wealth of detail and make subsequent assurance efforts easier. If the rules of operation map many-to-one to the system calls, the rules can be simpler and the model will then be easier to understand and analyze in its own right. The modeler must decide how to approach this issue, based on an understanding of the modeled class of systems and the purposes of the modeling.

Having decided that issue, the modeler can then examine the expected or actual TCB interface of the system. For the model in this article, this means examining each UNIX System V system call. The modeler must figure out what each system call or equivalent will do to the state of the system and whether it relates to the system's security policies. Every system call potentially has relevance to some policy defined by the Rule Set Model. Some system calls, for example, may have nothing to do with mandatory access control but significant relevance to the Clark-Wilson Integrity Policy. Looking at the system calls in this way attracts attention to needed constraints in one or more policies that the modeler might otherwise overlook. An example of this kind of analysis can be found in my detailed report on rule-based modeling (LaPadula, 1991). The modeler must ensure that the rules of operation needed to model the system calls are included in the State Machine Model. Finally, the modeler should decide how each rule of operation will employ the elements of the interface definition to invoke the Rule Set Model.

This approach gives high confidence that a system's implementation clearly derives from the elements of the formal model instead of additionally depending on many design and policy decisions not addressed in the model.

5. Rule Set Model

5.1. Introduction

This section describes a Rule Set Model for a trusted system that implements four policies:

- A mandatory access control (MAC) policy
- A Clark-Wilson integrity (CWI) policy
- A functional control (FC) policy
- A security information modification (SIM) policy

The MAC policy represents the MAC policy of American Telephone and Telegraph's (AT&T) System V/MLS (Flink, 1988), Release 1.2.1. Inclusion of this MAC policy shows that other policies can be integrated with traditional non-disclosure security requirements. The MAC policy uses a lattice of security levels as the basis for its access decisions. The CWI policy provides control over modification of information by regulating the transactions that users can apply to files of information. This policy employs roles and types and execute-control lists (the Clark-Wilson triples). Its inclusion shows how the commercial data processing requirements described by Clark and Wilson (Clark, 1987) can be modeled and integrated with the MAC policy for a UNIX system. The functional control (FC) policy implements a general role and type policy in terms of system-roles of users and categories of objects. This policy allows the roles **system administrator, security officer,** and **user** and uses the categories **general, security,** and **system**. The security information modification (SIM) policy is based on types of system data and system-roles of users. This policy allows only the security officer to change the system's security information.

This article focuses on the MAC and CWI policies but includes the FC and SIM policies for completeness. A useful trusted system must provide the kinds of access control defined by FC and SIM, but formal models typically have not included such policies.

I will present the four policies of this model in detail in this section, giving the MAC and CWI policies completely but presenting only skeletal forms of

the FC and SIM policies. Again, my purpose is to describe the approach, not to provide a complete formal model. Before giving the policies, though, I need to address the issue of why the Rule Set Model does not include the discretionary access control (DAC) and identity-based access control (IBAC) of UNIX System V.

5.2. System V DAC and IBAC

UNIX System V controls access to resources through its Discretionary Access Control (DAC) and Identity-Based Access Control (IBAC). The DAC capability enables the system and its users to decide who may access the files they own and in what manner. It uses the familiar read, write, and execute privileges. The UNIX kernel ensures that the permissions defined by the users and the system will be honored. In addition, the kernel incorporates a non-discretionary policy based on super-user privileges and the several types of user identifiers (real, effective, saved) that it uses. This identity-based access control (IBAC) policy and the user-defined DAC policy are the access control of UNIX System V.[7]

The model in this article includes neither of these policies. The IBAC policy of UNIX is typically not modeled although it is the kind of policy that should be of interest to the modeler. A more elaborate IBAC policy can replace the UNIX super-user approach in trusted systems to provide better separation of duty. Instead of modeling the super-user-based IBAC of UNIX, this Rule Set Model has a functional control policy that has better separation of duty and is also far less complicated than the IBAC of UNIX.

Formal models often do include the DAC policy. This formal model does not because, in short, it is not an interesting policy. It should certainly be included in the state machine representation of a UNIX system, assuming its level of detail is appropriate to the model. But it does not belong in the Rule Set Model because, other than the primitive "policy" of enforcing the permissions assigned by users, UNIX DAC is not a predefined policy for the system to enforce. The underlying "permissions policy" provides a mechanism by which users attempt to impose their own sharing policy on the resources they own. But, there's no assurance that they will succeed. A Trojan Horse, for example, can easily defeat a user's non-disclosure objectives since the DAC mechanism provides no way for a user to prohibit copying a file he has allowed to be read. Nor does the DAC mechanism adequately support integrity objectives because it provides no way for a user to specify *how* others might modify objects that he owns. In summary, the DAC mechanism does not strongly support any known, well-conceived policy objective in which users are likely to have an interest.

This model assumes that a favorable DAC check, when appropriate, precedes each invocation of the Rule Set Model by the State Machine Model. Any of the four policies we have included in the Rule Set Model can override a favorable DAC decision.

5.3. Policies of the Rule Set Model

5.3.1. Mandatory Access Control Policy

Mandatory access control is based on security levels of the processes, users, and objects of the system and the request of the process. This policy affects access of processes to objects—for example, reading, writing, and deleting files, directories, and message queues—and other aspects of processing—for example, forking a process and sending signals to other processes. The policy depends on the security-level attribute of processes and objects and on the object-type attribute of objects. The object types defined for this policy are **file, directory, ipc,** and **scd**. **file** and **directory** have their obvious UNIX meanings. **ipc** means "inter-process communication"; the message queue and shared memory in UNIX map to this type. **scd** means "system control data"—data the system uses to control its operations; the inode in UNIX is of this type.

In the next several tables the letter "P" stands for the security level of the process making the request for access, the letter "O" stands for the security level of the referenced object, "≥" indicates the usual dominates relation between levels, and "=" indicates equality between levels. Tables 1, 2, 3 and 4 define the policy for controlling access of a process to objects of type **file**, **directory**, **ipe** and **scd** respectively.

Table 1. MAC Policy for Objects of Type **file**.

If the request is	then access is allowed if
create	O is set equal to P
delete	P = O
delete-data	P = O
execute	P ≥ O
read	no condition[8]
read-open	P ≥ O
read&write-open	P = O
write	no condition (see footnote on read above)
write-open	P = O

Table 2. MAC Policy for Objects of Type **directory**.

If the request is	then access is allowed if
create	O is set equal to P
delete	P = O
read	$P \geq O$
search	$P \geq O$
write[9]	P = O

Table 3. MAC Policy for Objects of Type **ipc**.

If the request is	then access is allowed if
alter	P = O
create	O is set equal to P
delete	P = O
read	no condition (footnote on read in table 1 applies)
read&write-open	P = O
write	no condition (footnote on read in table 1 applies)

Table 4. MAC Policy for Objects of Type **scd**.

If the request is	then access is allowed if
change-owner	P = O
create	O is set equal to P
delete	P = O
get-permissions-data	$P \geq O$
get-status-data	$P \geq O$
modify-access-data	P = O
modify-permissions-data	P = O

The requests get-permissions-data and get-status-data could be modeled as undistinguished reads of the system control data. Similarly the change-owner, modify-access-data, and modify-permissions-data could be modeled as undistinguished writes. This model has separate requests to allow access control decisions based on the distinctions they make. The CWI policy uses these distinctions.

Table 5 defines the MAC policy governing process management.

Table 5. MAC Policy for Process Management.

If the request is	then access is allowed if
clone	no condition—P2 is set equal to P1
send-signal	P1 = P2

The mandatory access control policy will be referred to as the MAC (Mandatory Access Control) policy.

5.3.2. Integrity Control Policy

The integrity policy comes directly from the Clark-Wilson Integrity (CWI) policy. (Clark, 1987). It reflects not only the intent of their policy but also the specific details of their approach. In addition, I have included the ancillary policy that appears to me necessary to support their intentions. Although I assume the interested reader will have good familiarity with the CWI model, a summary of the model's certification and enforcement rules is given next for the reader's convenience.

The certification and enforcement rules of the Clark-Wilson model (Clark, 1987) are as follows:

Certification Rule 1: All Integrity Verification Procedures (IVPs) must properly ensure that all Constrained Data Items (CDIs) are in a valid state at the time the IVP is run.

Certification Rule 2: All Transformation Procedures (TPs) must be certified to be valid. That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a "relation," which defines that execution. A relation is thus of the form: (TPi, (CDIa, CDIb, CDIc, . . .)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.

Enforcement Rule 1: The system must maintain the list of relations specified in Certification Rule 2 and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.

Enforcement Rule 2: The system must maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, . . .)), which relates a user, a TP, and the data objects that TP may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

Certification Rule 3: The list of relations in Enforcement Rule 2 must be certified to meet the separation of duty requirement.

Enforcement Rule 3: The system must authenticate the identity of each user attempting to execute a TP.

Certification Rule 4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.

Certification Rule 5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.

Enforcement Rule 4: Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, those associated with a TP. An agent that can certify an entity may not (i.e., must not) have any execute rights with respect to that entity.

Clark-Wilson Integrity policy provides for both external and internal consistency of data. Measures for external consistency, such as their Integrity Verification Procedures (IVPs), ensure that the data stored in the computer system correctly models the state of the real-world systems to which it relates. Measures for internal consistency ensure that modification of data results in a valid state. Some of the CWI rules deal with the relationship between internal and external consistency of data. The integrity control policy in this article focuses on the rules for internal consistency and also supports the capability to ensure external consistency. Some of the Clark-Wilson Integrity rules that deal with external consistency are, naturally, beyond the scope of this internal system model.

Integrity control is based on the following:

- integrity-controlled programs called Transformation Procedures (TPs) and Integrity Verification Procedures (IVPs)
- integrity-controlled objects called Constrained Data Items (CDIs)
- user permissions to apply certain TPs to specified CDIs and permission to apply an IVP to a CDI

Users and objects in the computer system have the following attributes to support integrity control:

- The object attribute "program-type" may have the following values:

TP	means that the object is a CWI TP
IVP	means that the object is a CWI IVP

TPICD	means that the object is a special TP that operates on integrity control data
NIL	means that the object is not an integrity-controlled object

The use of these attribute values for controlling execution of integrity-related programs is discussed later.

- The object attribute "data-type" may have the following values:

CDI	means that the object is a CWI CDI
CDIIC	means that the object is a CWI CDI used for integrity control[10]
NIL	means that the object is an Unconstrained Data Item (UDI)—that is, not integrity-controlled

- The user attribute "integrity-role" may have the following values:

TP-user	means that the user is authorized to execute TPs
TP-manager	means that the user is authorized to manage (create, delete, and modify) certain integrity objects specified below
IVP-user	means that the user is authorized to execute IVPs
IVP-manager	means that the user is authorized to manage IVPs, as specified below
NIL	means that the user has no integrity role

These roles are static during system operation. The intent is that a system administrator will assign integrity-roles to users, one role per user, in consonance with the organization's policy.

The authorizations of a user with an integrity role are described in Table 6.

Table 6. CWI Policy for Execute, Create, Delete, and Modify.

A user in integrity-role	may execute	may create/delete	may modify
TP-user	TPs		
TP-manager	TPICDs	TPs, TPICDs, CDIICs	CDIIC[11]
IVP-user	IVPs		
IVP-manager		IVPs, CDIs	

Clark and Wilson (Clark, 1987) require that the system " . . . maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, . . .)), which relates a user, a TP, and the data objects that TP may reference on behalf of that

user." Further, the system " . . . must ensure that only executions described in one of the relations are performed." The Rule Set Model given in this article has a User-Transformation Procedures Associations (UTPA) table for representing the relations. It consists of ordered triples of the form (user-identifier, TP, list of CDIs). The triples impose no constraints on modes or order of access since the CWI model does not, but one can imagine systems in which such constraints are useful. However, other policies may constrain the TP with respect to mode of access. For example, when a TP attempts to open a CDI for writing, the TP must be allowed to write the CDI by the MAC policy of the system.

The UTPA satisfies the requirement to maintain a list of relations. One can envision several ways to ensure that only executions defined by the UTPA are carried out in the system. For example, define a new system call— apply(TP,list_of_CDIs). "Apply" operates like the exec system call but has an additional argument. The second argument shows on which CDIs the requesting process wishes the TP to operate. The kernel passes the arguments of this system call to the rule set. The rule set then checks the UTPA to see if the list of CDIs is valid for the owner of the requesting process. The difficulty is that the "enforcement" provided by this approach is weak. Lacking any further access checks during its operation, the TP could access some CDI for which the user is not authorized. One may argue that the TP has been certified to operate correctly so that it should only carry out correct procedures. This is acceptable if all correct and authorized executions are built into the TP and certified. Clark and Wilson suggest, however, that " . . . an important research goal must be to shift as much of the security burden as possible from certification to enforcement . . . " since " . . . the certification process is complex, prone to error, and must be repeated after each program change."

Therefore, in this model of CWI, the initial request of the process is only a request to operate the TP. In the UNIX environment this is an exec system call in which the process names the object to execute. When the named object is a TP, its program-type attribute is TP. When the TP subsequently makes requests for access to CDIs, those requests are adjudicated by the rule set in the usual manner. In addition, the rule set keeps a record of the CDIs being accessed, ensuring at each request that the requested access is allowed by one of the triples defined in the UTPA. This idea needs further elaboration, provided in the following paragraphs.

When a process executes a TP, one of the rules for integrity control will add the process-identifier of the process to all triples in the UTPA having the user-identifier of the owner of the process and specifying the named TP. This marks all candidate executions of the named TP by this process. Note that the same user may already have other executions of this TP in progress. When the TP (now a process having the process-identifier of the process that executed it) attempts to

access an object that is a CDI, one of the integrity-control rules will remove the process-identifier of the process from all triples in the UTPA currently marked with this process-identifier but not having the named CDI listed. This reduces the set of candidate executions of the named TP by this process. If after taking this action there are no entries in the UTPA marked with this process-identifier, then the attempted access by the TP on behalf of the user is not valid and the request will be denied. The next sequence of tables illustrates this method.

Suppose user A is allowed to apply TP1 in any of the following ways:

- to CDIs 1 and 2
- to CDIs 1 and 3
- to CDIs 2 and 3

as illustrated in Table 7:

Table 7. User-Transformation Procedures Associations (UTPA).

User	TP	CDIs	Processes
User A	TP1	CDI-1, CDI-2	
User A	TP1	CDI-1, CDI-3	
User A	TP1	CDI-2, CDI-3	
User Z	etc.	etc.	

Suppose a process having process-identifier PID requests execution of TP1 on behalf of user A. Assuming the requested action is authorized, the UTPA table is marked as follows.

User A	TP1	CDI-1, CDI-2	PID
User A	TP1	CDI-1, CDI-3	PID
User A	TP1	CDI-2, CDI-3	PID

The process is now known as a TP-type process. If the process requests access to CDI-2, the table is modified, with the following result.

User A	TP1	CDI-1, CDI-2	PID
User A	TP1	CDI-1, CDI-3	
User A	TP1	CDI-2, CDI-3	PID

If the process now requests access to CDI-3, the table is modified, with the following result.

User	TP	CDIs	Processes
User A	TP1	CDI-1, CDI-2	
User A	TP1	CDI-1, CDI-3	
User A	TP1	CDI-2, CDI-3	PID

If the process now requests access to CDI-1, the request is invalid.

This approach to enforcing the Clark-Wilson triples has the advantage that it does not require a new system call or data structure in the UNIX environment. Note, though, that it allows a TP to access any UDI in the normal manner for access to a file by a process, subject to the constraints of the other policies implemented by the rule set. The certification process must ensure that the TP accesses only those UDIs it should access for a particular execution. But, this is not in keeping with the spirit of moving as much as possible from certification to enforcement, as suggested by Clark and Wilson. One possibility for changing this is to add the names of the allowed UDIs for a particular TP to the triples or, perhaps better, to the TP-CDIs relation. For the latter, the TP-CDIs relation must be added to the model. It is then no longer redundant with the triples.

The scheme just outlined describes the situation in which a process executes a single TP. The integrity policy must also cover the cases where a TP-type process attempts to execute another file (UNIX exec) or attempts to clone itself (UNIX fork). If a TP-type process were to fork a child, the child would be identical to the parent with respect to executable code and open files (e.g., the CDIs being worked on). However, it makes no sense for a TP-type child to continue processing with the executable code of its parent since to do so would require unwarranted complex coordination between parent and child to preserve integrity. It really only makes sense to consider the case that the child executes new code—that is, a new TP. Allowing a TP-type process to spawn another TP-type process in this way adds to the complexity of the certification of the original TP code but adds no functional capability. According to Clark and Wilson, the certification task should be kept as simple as possible by having the system enforce as much of the integrity policy as possible. The needed functionality, enforced by the system in the scheme above, is achieved by an ordinary process cloning a process that changes itself into a TP-type process by executing a TP-type object. In short, it is neither desirable nor necessary for a TP-type process to clone itself.

Thus, to carry out the intent of the Clark-Wilson integrity policy without significantly modifying the System V system calls, the ability of a process to execute

(exec system call) and clone (fork system call) must be constrained in the following ways:

- When an ordinary[12] process executes an object of type TP, IVP, or TPICD, the process executing the object becomes the type of the object. That is, its process-type attribute takes on the value of the program-type attribute of the object. When an ordinary process executes a TP-type object, the UTPA table is updated as described above. A TP-, IVP-, or TPICD-type process is allowed to execute only an object of its own type. When a TP-type process executes a TP-type object, no changes are made to the UTPA. Allowing the original TP to execute a TP-type object is a convenience related to how a TP is organized into units of executable code.
- A TP-, IVP-, or TPICD-type process is not allowed to clone (UNIX fork).

The following additional constraints[13] are needed to support the intent of the CWI policy in the UNIX System V/MLS environment:

- Changing ownership of TPs, IVPs, TPICDs, and CDIs is not allowed by CWI policy.
- Aliasing (via the link system call in UNIX) of file names is not a good practice under the CWI Policy. Through aliasing an ordinary user can defeat the attempt of an authorized user (i.e., TP-manager) to remove a TP from the system.[14] The policy on aliasing integrity-controlled objects is defined by Table 8.

Table 8. CWI Policy for Alias.

If the user has the integrity role	then the user may alias
TP-manager	TPs, TPICDs, and CDIs
IVP-manager	IVPs and CDIICs

- Tracing (via the ptrace system call in UNIX) of TPs, IVPs, or TPICDs should not be allowed under the CWI policy since tracing would enable modification of a TP or IVP during its execution.
- Only authorized users may acquire or modify status information about integrity-controlled objects, as defined in Table 9.

Table 9. CWI Policy on Status Information.

If the user has the integrity role	then the user's process may read/write status information about
TP-user	—
TP-manager	TPs, TPICDs, CDIs
IVP-user	—
IVP-manager	IVPs, CDIICs

The integrity policy of this model allows a TP-type process to receive a signal, via the kill system call in UNIX System V, from a non-TP-type process, presumably the parent process that spawned the TP-type process. The danger here is that a TP-process will be killed (terminated) at such a time that the CDIs on which it is operating are left in an inconsistent state. A justification for allowing this is that it preserves functionality provided by UNIX and the TP can be designed to take appropriate action on the CDIs before exiting. This puts the burden on the certification of the TP to ensure that the TP handles signals appropriately.

This integrity control policy will be referred to as the CWI (Clark-Wilson Integrity) policy.

5.3.3. Functional Control Policy

Functional control (FC) uses system-roles of users and categories of objects. The system-roles are **user, security officer,** and **administrator**. The categories are **general, security,** and **system**. A process whose owner has system-role **R**, requesting access to an object having object-category **C**, shall be allowed the access only if **R** *is compatible with* **C**. The role-category compatibilities are checked in Table 10.

Table 10. Definition of the Compatibility Relation.

	general	system	security
user	√		
administrator	√	√	
security officer	√		√

This functional control policy requires that a process assume only one of the possible roles of its user. Whether a user can be assigned more than one role is outside the scope of this policy. But, a functional control policy should

be more elaborate than the one just described. For example, it should specify who can change what attributes of what entities. It's the logical place to define control of trusted subjects, such as daemons of the UNIX system. The simple policy given above satisfies the goals of this article but is not adequate for a real system.

This policy will be referred to as the **FC** (Functional Control) policy.

5.3.4. Policy for Modifying Security Information

The policy for modification of security information uses types of data and system-roles of users. The data-type needed for this policy is **si**. The value **si** means the object contains security information. In UNIX, for example, the /etc/password file would have this value for its data-type attribute. **NIL** means the object contains ordinary user or system data. The data-attribute may have other values as well, such as those the CWI Policy uses. But the SIM Policy treats all values other than **si** the same as **NIL**.

When a process requests access to data of type **si** in a mode that enables modification of the information, this policy permits the access only if the system-role of the owner of the process (i.e., the user) is **security officer**. As with the FC Policy, the SIM Policy could encompass more elaborate rules of operation, but the simple form given here suffices for the purposes of this article.

This policy will be referred to as the **SIM** (Security Information Modification) policy.

5.4. Access Control Information

To support the policies just described, the following attributes, in three groups of access control information (ACI) as displayed in Tables 11, 12, and 13, are needed. Attributes that are new in this model have been explained in one of the preceding policy descriptions.

Table 11. Attributes of User ACI.

USER-ACI	VALUES
user-identifier (CWI)	a user identifier
access-approvals (MAC)	a security level
system-role (FC & SIM)	user, security officer, or administrator
integrity-role (CWI)	NIL, TP-user, TP-manager, IVP-user, or IVP-manager

Table 12. Attributes of Process ACI.

PROCESS-ACI	VALUES
owner (pointer to USER-ACI)	–
security-level	a security level
process-identifier	a process identifier
process-type	NIL, TP, IVP, or TPICD

Table 13. Attributes of Object ACI.

OBJECT-ACI	VALUES
security-level (MAC)	a security level
object-identifier (CWI)	an object identifier
object-category (FC)	general, security, or system
object-type (MAC)	file, directory, ipc, or scd
program-type (CWI)	NIL, TP, IVP, or TPICD
data-type (SIM & CWI)	NIL, CDI, CDIIC, or si

In addition, the access control context information (ACC) displayed in Table 14 is defined for the use of the rule set.

Table 14. Entities of the Access Control Context.

Access Control Context (ACC) Information		
Name of Entity	Structure of Entity	Comment
User-Transformation Procedures Associations (UTPA)	set of ordered 4-tuples (user-identifier, TP, list of CDIs (by object-identifier), list of process-identifiers)	This set gives the CWI "triples" for all users in the system that are allowed to apply TPs; the fourth element of the 4-tuple is used for access control as explained in the discussion of the Integrity Control Policy.

5.5. Rules of the Rule Set Model

Four groups of rules define the policies described above:

- Mandatory Access Control (MAC) Rules

- Clark-Wilson Integrity (CWI) Rules
- Functional Control (FC) Rules
- Security Information Modification (SIM) Rules

Each policy of the Rule Set Model is implemented as one or more rules. When combined, as described later, the rules constitute the Access-Rules function.

Each rule is an expression having one of four values.

- YES: This value means that the request of the State Machine Model has been evaluated by the rule and the request may be granted.
- NO: This value means that the request of the State Machine Model has been evaluated by the rule and the request may not be granted.
- DC: This value means that the request of the State Machine Model has been recognized by the rule, but the rule's policy does not require any checks of attribute values and/or relations among attribute values. The rule is tolerant of the request in the sense that the policy "doesn't care" (DC). DC is similar to YES but provides additional information useful for analysis of a rule set.
- UNDEFINED: This value means that the request of the State Machine Model has not been recognized by the rule. UNDEFINED is different from NO and DC: both NO and DC indicate that the Rule Set Model is cognizant of the request, while UNDEFINED indicates the opposite. This not only provides useful information for analysis of a rule set, but in a system implementation it might serve to detect improper configurations of the system.

In addition, a rule may specify an effect that should occur if the request of the process will ultimately be acted on by the State Machine Model. Since all effects are changes to attribute values, an effect is specified in the form

set-attribute(attribute name, attribute value).

For example, when a file is to be created, the MAC rule specifies an effect that sets the sensitivity level of the file to the value of the sensitivity level of the process creating the file.

To define the Access-Rules function, we need the binary operator ∔ (pronounced "and-plus") defined in Table 15.

Table 15. Definition of the Binary Operator ✣

A	B	A ✣ B
YES	YES	YES
YES	NO	NO
YES	DC	YES
YES	UNDEFINED	UNDEFINED
NO	YES	NO
NO	NO	NO
NO	DC	NO
NO	UNDEFINED	UNDEFINED
DC	YES	YES
DC	NO	NO
DC	DC	DC
DC	UNDEFINED	UNDEFINED
UNDEFINED	YES	UNDEFINED
UNDEFINED	NO	UNDEFINED
UNDEFINED	DC	UNDEFINED
UNDEFINED	UNDEFINED	UNDEFINED

The Access-Rules function is defined as follows:

```
Access-Rules(request(input argument), process/object(input argument), ...,
      process/object(input argument))):
  function-value = MAC (✣) CWI (✣) FC (✣) SIM;
  IF
    function-value is UNDEFINED;
  THEN
    system-error;
  ELSE
    return (function-value);
```

Each of the rules will be expressed in the following general form:

POLICY ← POLICY Rule

```
POLICY Rule:
SELECT CASE request
  CASE request, request, ... , request
    statement-block
  CASE request, request, ... , request
    statement-block
      *
      *
      *
```

```
  CASE request, request, ... , request
    statement-block
END SELECT
```

The notation "POLICY ← POLICY Rule" means that the variable POLICY should be set to the value of the expression in parentheses. For each POLICY (MAC, CWI, FC, and SIM), representative CASEs are given to illustrate the modeling approach. These representative CASEs were extracted from a complete rule set, although not one that had been verified, to ensure that they made sense.

5.5.1. Mandatory Access Control (MAC) Rules

The following logical operator is needed:

dominates(level1, level2) has the value TRUE if level1 dominates[15] level2, FALSE otherwise.

MAC ← (MAC Rule 1)

MAC Rule 1:

```
SELECT CASE request

  CASE alias
    return(DC);

  CASE alter
    SELECT CASE object-type[object]
      CASE ipc
        IF
          security-level[process] equals security-level[object];
        THEN
          return(YES);
        ELSE
          return(NO);
      CASE ELSE
        return(UNDEFINED);

  CASE clone
    return(set-attribute(security-level[process2], security-level[process1]);
    YES);

  CASE create
    return(set-attribute(security-level[object], security-level[process]); YES);

  CASE execute
    SELECT CASE object-type[object]
      CASE file
```

```
      IF
        security-level[process] dominates security-level[object];
      THEN
        return(YES);
      ELSE
        return(NO);
    CASE ELSE
      return(UNDEFINED);

CASE modify-attribute (*arguments are process, qualifier, attribute,
  value*)
  SELECT CASE qualifier[input argument]
    CASE user
      IF
        security-level[process] equals access-approvals[user pointed to by
        qualifier];
      THEN
        SELECT CASE attribute[input argument]
          CASE access-approvals
            IF
              system-role[user pointed to by owner[process]] equals
              security officer;
            THEN
              return(YES);
            ELSE
              return(NO);
            CASE ELSE
              return(YES);
          ELSE
            return(NO);
        CASE process
          SELECT CASE attribute[input argument]
            CASE security-level
              return(NO);
            CASE ELSE
              IF
                security-level[process] equals security-level[process pointed to
                by qualifier];
              THEN
                return(YES);
              ELSE
                return(NO);
        CASE object
          IF
            security-level[process] equals security-level[object pointed to by
            qualifier];
```

```
     THEN
       SELECT CASE attribute[input argument]
         CASE access-approvals
           IF
             system-role[user pointed to by owner[process]] equals
             security officer;
           THEN
             return(YES);
           ELSE
             return(NO);
         CASE ELSE
           return(YES);
       ELSE
         return(NO);
     CASE ELSE
       return(UNDEFINED);

CASE read
  SELECT CASE object-type[object]
    CASE directory
      IF
        security-level[process] dominates security-level[object];
      THEN
        return(YES);
      ELSE
        return(NO);
    CASE file, ipc
      return(DC);
    CASE ELSE
      return(UNDEFINED);

CASE read-open
  SELECT CASE object-type[object]
    CASE file
      IF
        security-level[process] dominates security-level[object];
      THEN
        return(YES);
      ELSE
        return(NO);
    CASE ELSE
      return (UNDEFINED);

CASE read&write-open
  SELECT CASE object-type[object]
    CASE file, ipc
```

```
      IF
        security-level[process] equals security-level[object];
      THEN
        return(YES);
      ELSE
        return(NO);
    CASE ELSE
      return(UNDEFINED);

CASE write-open
  SELECT CASE object-type[object]
    CASE file
      IF
        (security-level[object] equals security-level[process]);
      THEN
        return(YES);
      ELSE
        return(NO);
    CASE ELSE
      return(UNDEFINED);
CASE change-owner, change-role, delete, delete-data, get-permissions-
     data, get-status-data, modify-access-data, modify-permissions-
     data, read-attribute, search, send-signal, terminate, trace, write
     <omitted: note that these cases would be grouped into several
     cases or treated as separate cases if specified in this
     model>
CASE ELSE
  return(UNDEFINED);


   END SELECT
```

5.5.2. *Clark-Wilson Integrity (CWI) Rules*

To specify the rules for the Clark-Wilson Integrity Policy the following functions are needed.

- The function Allowed-Access has the value TRUE or FALSE. It performs the search and modify of the UTPA as described earlier under Integrity Policy, returning TRUE if at least one triple remains in the UTPA as a candidate execution of the TP, FALSE otherwise.
- The function Allowed-Execute has the value TRUE or FALSE. It performs a search of the UTPA as described earlier under Integrity Policy, returning TRUE if there is some triple in the UTPA containing the ordered pair given as arguments to Allowed-Execute, FALSE otherwise.

- The function Mark-Candidates-in-UTPA places the process-identifier in each 4-tuple of the UTPA containing the ordered pair (user-identifier, object-identifier) given as its first two arguments.

CWI ← (CWI Rule 1)

CWI Rule 1:

```
SELECT CASE request

 CASE alias, get-status-data, modify-access-data
  IF
   data-type[object] is not CDI AND data-type[object] is not CDIIC AND
   program-type[object] is not TP AND program-type[object] is not IVP
   AND program-type[object] is not TPICD
  THEN
   return(DC);
  ELSE
   IF
    ((data-type[object] is CDI OR program-type[object] is TP OR
    program-type[object] is TPICD) AND integrity-role[user identified
    by owner[process]] is TP-manager)
    OR
    ((program-type[object] is IVP OR data-type[object] is CDIIC) AND
    integrity-role[user identified by owner[process]] is IVP-manager)
   THEN
    return(YES);
   ELSE
    return(NO);

 CASE alter, get-permissions-data, modify-permissions-data, read, write,
  search, send-signal, terminate
  return(DC);

 CASE create, delete
  IF
   data-type[object] is not CDI AND data-type[object] is not CDIIC AND
   program-type[object] is not TP AND program-type[object] is not IVP
   AND program-type[object] is not TPICD
  THEN
   return(DC);
  ELSE
   IF
    ((data-type[object] is CDIIC OR program-type[object] is TP OR
    program-type[object] is TPICD) AND integrity-role[user identified
    by owner[process]] is TP-manager)
    OR
```

```
  ((program-type[object] is IVP OR data-type[object] is CDI) AND
  integrity-role[user identified by owner[process]] is IVP-manager)
 THEN
  return(YES);
 ELSE
  return(NO);

CASE execute
 SELECT CASE process-type[process]
  CASE NIL
   SELECT CASE program-type[object]
    CASE TP
     IF
      integrity-role[user identified by owner[process]] is TP-user
      AND Allowed-Execute
                (user-identifier[user identified by
                owner[process]],
                object-identifier[object]);
     THEN
      Mark-Candidates-in-UTPA
                (user-identifier[user identified by
                owner[process]],
                object-identifier[object],
                process-identifier[process]);
        return(set-attribute(process-type[process], program-
        type[object]);
        YES);
     ELSE
      return(NO);

    CASE IVP
     IF
      integrity-role[user identified by owner[process]] is IVP-user
     THEN
      return(set-attribute(process-type[process],program-
      type[object]); YES);
     ELSE
      return(NO);

    CASE TPICD
     IF
      integrity-role[user identified by owner[process]] is TP-
      manager
     THEN
      return(set-attribute(process-type[process],program-
      type[object]); YES);
```

```
          ELSE
            return(NO);

        CASE ELSE
          return(DC);

      CASE ELSE
        IF
          process-type[process] is not TP AND program-type[object] is not
          TP AND
          process-type[process] is not IVP AND program-type[object] is
          not IVP AND
          process-type[process] is not TPICD AND program-type[object] is
          not TPICD;
        THEN
          return(DC);
        ELSE
          IF
            process-type[process] is TP AND program-type[object] is TP
            OR
            process-type[process] is IVP AND program-type[object] is IVP
            OR
            process-type[process] is TPICD AND program-type[object] is
            TPICD
          THEN
            return(YES);
          ELSE
            return(NO);

CASE read-open, write-open
  SELECT CASE data-type[object]
    CASE CDI
      SELECT CASE object-type[object]
        CASE file
          IF
            process-type[process] is TP AND
            Allowed-Access(process-identifier[process], object-
            identifier[object])
            OR
            process-type[process] is IVP
          THEN
            return(YES);
          ELSE
            return(NO);
        CASE ELSE
          return(UNDEFINED);
```

```
CASE CDIIC
  SELECT CASE object-type[object]
    CASE file
      IF
        process-type[process] is TPICD
      THEN
        return(YES);
      ELSE
        return(NO);
   CASE ELSE
     return(UNDEFINED);
CASE ELSE
  return(DC);
CASE change-owner, change-role, clone, delete-data, modify-attribute,
     read-attribute, read&write-open, trace
     <omitted: note that these cases would be grouped into several
     cases or treated as separate cases if specified in this
        model>
CASE ELSE
  return(UNDEFINED);

END SELECT
```

5.5.3. *Functional Control (FC) Rules*

FC ← (FC Rule 1)
    FC Rule 1:

```
SELECT CASE request
  CASE alias, alter, change-owner, create, delete, delete-data, execute, get-
       permissions-data, get-status-data, modify-access-data, modify-
       permissions-data, read, read&write-open, read-open, search,
       write, write-open
    IF
      (system-role[user pointed to by owner[process]] is user AND
      object-category[object] is general)
      OR
      (system-role[user pointed to by owner[process]] is administrator AND
      object-category[object] is system or general)
      OR
      (system-role[user pointed to by owner[process]] is security officer
      AND
      object-category[object] is  security or general)
      OR
      (system-role[user pointed to by owner[process]] is daemon AND
      object-category[object] is system or general);
```

```
  THEN
   return(YES);
  ELSE
   return(NO);

 CASE clone, read-attribute, send-signal, terminate, trace
  return(YES);

 CASE change-role
  <omitted>

 CASE modify-attribute
  <omitted>

 CASE ELSE
  return(UNDEFINED);

END SELECT
```

5.5.4. *Security Information Modification (SIM) Rules*

SIM ← (SIM Rule 1)

SIM Rule 1:

```
SELECT CASE request

 CASE alias, alter, change-owner, create, delete, delete-data,
      modify-access-data, modify-permissions-data, write, write-open,
      read&write-open
  SELECT CASE system-data-type[object]:
   CASE system-data-type[object] is si:
    IF:
     system-role[user pointed to by owner[process]] is security officer;
    THEN:
     return(YES);
    ELSE:
     return(NO);
   CASE ELSE:
    return(DC);
 CASE change-role
  <omitted>
 CASE modify-attribute
  <omitted>
 CASE clone, execute, get-permissions-data, get-status-data, read, read-
```

```
    attribute, read-open, search, send-signal, terminate, trace
  return(DC);

 CASE ELSE
  return(UNDEFINED);

END SELECT
```

6. Conclusion

We can look at the rule-set approach described in this article in light of other models of trusted systems. The scheme displayed in Table 16 will enhance our view. This table, based on a taxonomy developed by Williams (Williams, 1990), shows several stages in the development of security requirements for a trusted system. Each succeeding stage has more detailed elaboration of a trust policy.

Table 16. Stages of Elaboration of Security Requirements.

Stage of Elaboration	Examples
1 Trust Objectives	TCSEC mandatory security objective (NCSC, 1985); CWI: integrity objectives (Clark, 1987)
2 External Models	Noninterference (Goguen, 1982); SMMSM: user's view of SMMS operation and the security assumptions[16] (Landwehr, 1984)
3 Internal Models	BLM: *-property[17] (Bell, 1976); SMMSM: security assertions[18] (Landwehr, 1984); CWI: certification and enforcement rules[19] (Clark, 1987); CMWM: maccessible expression[20] (Millen, 1990)
4 Rules of Operation	BLM: Open-file access checks (Bell, 1976); CMWM: Read-file label float (Millen, 1990)
5 Functional Designs	Functional specification of UNIX open system call (Bach, 1986)

We can characterize the five stages as follows.

- Trust Objective. A trust objective specifies what is to be achieved by proper design and use of the computing system. It characterizes the desired conditions for information that should be maintained by the system. A non-disclosure objective, for example, states that there should be no unauthorized viewing of classified data. An integrity objective might state that there should be no unauthorized modification of sensitive data.

- External Model. An external model describes the trust objectives for the system in a formal, abstract manner, in terms of real-world entities such as people, their roles, types and groupings of information, and operations on information. It may, for example, describe authorizations for people to access information of various kinds. (It should be understood that the people are potential users of the target system and the information will be managed by the target system.)

- Internal Model. An internal model describes, in a formal, abstract manner, how the goals of the external model are met within the system. It may do this by specifying constraints on the relationships among system components and, in the case of a TCB-oriented design, among controlled entities.

- Rules of Operation. Rules of operation explain how the internal requirements developed in the internal model are enforced. They do this by specifying the access checks and related behaviors that guarantee satisfaction of the internal requirements.

- Functional Design. Like the rules of operation, the functional design specifies behavior of system components and controlled entities, but it is a complete functional description. A functional design may, for example, consist of functional specifications of the system calls or commands that will be implemented in the system.

The traditional Bell-LaPadula Model (BLM) (Bell, 1976) addresses the third and fourth levels of elaboration. The simple security property and the *-property are two axioms of the model that express the mandatory access control policy as constraints on a trusted system's operation. The BLM defines these properties as internal requirements at the third stage of elaboration. Its rules of operation elaborate the behavior of the trusted system at the more detailed level four. The proof of security in the BLM consists in showing that the rules of operation developed in stage four are a correct elaboration of the internal requirements developed in stage three. No specific provisions are made for demonstrating that the stage 3 requirements are a correct interpretation of

a stage 2 model. Although some aspects of the work reported in the referenced work are oriented toward the Multics system, the BLM itself is quite general. Thus, it can have wide application to systems, but its generality also means it provides no significant guidance in the development of a descriptive top-level specification, a requirement for B2 or higher classes of the TCSEC (NCSC, 1985).

The more recent Compartmented Mode Workstation model (CMWM) (Millen, 1990), which essentially addresses the same stages as the BLM, is more heavily oriented toward a particular class of computer system, in this case UNIX. Thus, the developer finds more guidance in the CMWM for the development of functional specifications. But, as with the BLM, the modeling approach employed does not take into account the desirability of being able to show correspondence with some external model.

The Clark-Wilson informal model (Clark, 1987) directly addresses external consistency issues. Although it does not explicitly articulate an external model, it describes an internal model (third stage) that appears capable of supporting a class of external models. The class is exemplified by an accounting enterprise in the Clark and Wilson paper (Clark, 1987).

The Secure Military Message System model (SMMSM) explicitly gives both an external model and an internal model. The external model is informal but is clearly reflected in the formal internal model. In the original formulation of the model, the Security Assertions were developed as an external model of the computer system, in that they define the properties that the computer hardware and software must ensure at the user (external) interface. In the terms defined here, however, the Security Assertions can be viewed as an internal model. This internal model, in contrast to the BLM and CMWM, defines a set of secure transforms rather than a set of secure rules of operation. The latter approach enhances the model's ability to avoid choosing implementation strategies, an avowed goal of the model's developers (Landwehr, 1984).

The modeling approach described in this paper addresses levels three, four, and five. The description of the ORGCON policy gives the level-three requirements. The rule set of the Policy Model gives the level-four security requirements. And, the rules of operation of the State Machine Model, because they correspond to system calls, provide a basis for a complete level-five elaboration of requirements. The rule set provides a foundation for showing consistency of the internal model with some external model. Because the rule set is separate from the rules of operation, the task of proving assertions about the security policies modeled should be easier than in the traditional approach. The separate rule set constitutes a specification of policy in a formal language. Thus, it can be analyzed with automated tools. The model described in this paper, although similar in one respect or

another to other models, has the following significant characteristics not shared by the other models:

- The model draws heavily from the functional design of a UNIX System V system for the specification of its rules of operation. A one-to-one correspondence exists between UNIX system calls and rules of operation.
- The rules of operation reflect far more functional design than the other models but do not include the access checks and other operations that guarantee satisfaction of the internal requirements. Instead, the rules of operation appeal to the rule set, which implements the internal requirements. The rules of operation also specify the appropriate places in the system's functional description for the rule set to be invoked. This can be viewed as a generalization of the use of the maccessible Boolean in the rules of operation of the CMWM.
- Internal requirements at stage 3 are implemented by a model in its own right, the Rule Set Model. The Rule Set Model consists of a set of rules that express the security policies of the trusted system. The rules of this model play a role similar to that of the CMWM maccessible expression and the BLM *-property, but they are far more extensive and express policies in addition to mandatory access control. The rules have more in common with the secure transforms of the SMMSM in that they express constraints on the functioning of the secure system.

The model in this article includes far more detail than is usual. (Bell, 1976; Landwehr, 1984; Clark, 1987; Millen, 1990). While recognizing that less detail gives advantages of simplicity, easy comprehensibility, and availability of existing automated tools, I believe we need detailed system models and that our collective knowledge base of ideas and techniques supports this approach. Moreover, I believe that a model should clearly communicate its meaning to a wide audience, not just to mathematicians. For this reason, the formal language of the rule set has the form of pseudo-code.

Model building is a process of abstraction in which certain details are suppressed to focus on those issues the model builder considers important. But what may seem unimportant to the model builder may be of great import during subsequent phases of requirements definition, when more and more detail must be considered. Thus, the model builder may be wrong in selecting a level of abstraction and important details may be abstracted away. Once this situation is recognized, the level of detail needs to be increased to include the important details in the model. In traditional models of the BLM/CMWM variety, the access rules, built on the notion of access by subjects to undifferentiated objects,

are normally so abstract that they treat the opening of a file and the opening of an inter-process communication object as the same thing. This is a convenience for the modeler, but a burden for all who follow in the development, implementation, and evaluation of a trusted system. At these later stages of requirements definition there may be important policy differences between opening a file and opening a message queue.[21] The SMMSM model takes advantage of its intended application area by defining the type of object "message" in addition to distinguishing between "objects" and "containers" as information entities.

The model builder of 20 years ago could not draw on experience with the technology of trusted systems to permit modeling at the level of detail I am advocating. There were many difficult and important problems to solve at high levels of abstraction, where few details of any real system could be considered. Not all of those problems have been solved, but the developing security technology of today can support modeling at a level useful for guiding the detailed design of a trusted system.

The ultimate goal of applying formal methods to developing trusted systems is to provide traceability from requirements and specifications all the way to implementation and use. Formal methods are employed today at several high-level points in this spectrum. The detail included in the formal model in this article is a step toward the goal of bringing formal methods closer to the final stages of implementation—complete functional design and coding.

Acknowledgments

In alphabetical order, I thank the following persons:

Dr. Marshall Abrams of The MITRE Corporation for his basic insights on access control that led to this modeling effort and for his encouragement during the writing of this article.

Edward G. Amoroso of AT&T Bell Laboratories for identifying significant inconsistencies, errors, and ambiguities in an earlier version of this paper and for his many helpful suggestions for improving it.

Charles W. Flink II of AT&T Bell Laboratories for his patient and comprehensive explanations of many of the design aspects and system calls of System V/MLS.

Carl Landwehr of Naval Research Laboratory for his suggestions on improving Table 16 (stages of elaboration of requirements) and his assistance in characterizing the Secure Military Message System in this regard.

Professor Ravi Sandhu of George Mason University, Editorial Staff of the JCS, for his thorough, helpful review of an earlier version of this paper, particularly his diligent efforts to improve the paper's organization.

Dr. James Williams of The MITRE Corporation for his contribution of a view of the stages of elaboration of requirements for trusted systems and for numerous interactions with me on various issues of formal modeling.

This work was supported in part by The MITRE Corporation as MITRE-Sponsored Research and in part by the U.S. Army as Mission-Oriented Investigation and Experimentation under contract DAAB07-91-C-N751. Technical direction for the research was provided by the National Security Agency.

References

1. Abrams, Marshall D., K. E. Eggers, L. J. LaPadula, and I. M. Olson, A Generalized Framework for Access Control: An Informal Description, *Proceedings of the 1990 National Computer Security Conference*, 13(1):135–143, October 1990.
2. Bach, Maurice J., *The Design of the UNIX® Operating System*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986
3. Bell, David E. and L. J. LaPadula, Secure Computer Systems: Unified Exposition and Multics Interpretation, *MITRE Technical Report 2997*, The MITRE Corporation, Bedford, Massachusetts, March 1976; available from NTIS, reference AD A023 588.
4. Clark, David D. and D. R. Wilson, A Comparison of Commercial and Military Computer Security Policies, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 184–194, April 1987.
5. Flink, Charles W. and J. D. Weiss, System V/MLS Labeling and Mandatory Policy Alternatives, *AT&T Technical Journal*, May/June 1988.
6. Goguen, J. A. and J. Meseguer, Security Policies and Security Models, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 11–20, April 1982.
7. International Standards Organization (ISO), Working Draft on Access Control Framework, ISO/IEC JTC 1/SC 21 N5045, July 1990.
8. Landwehr, Carl E., C. L. Heitmeyer, and J. McLean, A Security Model for Military Message Systems, *ACM Transactions on Computer Systems*, 2(3):198–222, August 1984.
9. LaPadula, Leonard J., Formal Modeling in a Generalized Framework for Access Control, *Proceedings of the Computer Security Foundations Workshop III*, 100–109, June 1990.
10. LaPadula, Leonard J., A Rule-Base Approach to Formal Modeling of a Trusted Computer System, MITRE Paper M91-021, The MITRE Corporation, Bedford, Massachusetts, August 1991.

11. Millen, Jonathan K. and D. J. Bodeau, A Dual-Label Model for the Compartmented Mode Workstation, MITRE Paper M90-51, The MITRE Corporation, Bedford, Massachusetts, August 1990.
12. National Computer Security Center (NCSC), Department of Defense Trusted Computer System Evaluation Criteria, DOD Standard 5200.28-STD, December 1985.
13. Page, J., J. Heaney, M. Adkins, and G. Dolsen, Evaluation of Security Model Rule Bases, Proceedings of the 1989 National Computer Security Conference, 98-111, 1989.
14. Williams, James G., Stages of Elaboration of Security Requirements for a Trusted Computer System, private communication, December 1990.

Appendix: Model Language and Constructs

A.1. Language for Expressing Rules

The method for expressing the model's rules departs from the traditional use of mathematical notation. A mixture of programming language statements and limited mathematical notation creates a specification language that is intuitively understandable to a broad audience.

Both rules of operation and rules of the rule set are defined in a language that looks like a programming language. Two basic language constructs are used to organize statements and show their interrelationships: SELECT CASE and IF THEN ELSE.

The SELECT CASE statement has the following syntax:

```
SELECT CASE attribute
  CASE attribute-value1
    statement-block-1
  CASE attribute-value2
    statement-block-2
 .
 .
 .

  CASE ELSE
    statement-block-n
 END SELECT
```

A statement-block is one or more statements. Individual statements are terminated by a semicolon. The value of the SELECT CASE statement is the value of

the statement-block following the CASE identified by the current value of the selected attribute. For example, the next SELECT CASE has the value of statement-block-2 when the "amount" is $200.

```
SELECT CASE amount
  CASE $100
    statement-block-1
  CASE $200
    statement-block-2
  CASE ELSE
    statement-block-n
END SELECT
```

If the current value of the selected attribute is not identified by one of the CASEs given, then the value of the SELECT CASE statement is the value of the CASE ELSE statement-block.

A final word on the SELECT CASE statement. The END SELECT part of the statement will be omitted when no ambiguity results – the use of indentation will make clear the scope of a SELECT CASE.

The IF THEN ELSE statement has the following syntax:

```
IF
  Boolean-expression
THEN
  statement-block
ELSE
  statement-block
```

The IF THEN ELSE statement has its usual meaning. A Boolean expression is an expression consisting of attributes and relational or logical operations and having a value of TRUE or FALSE.

A FOR-EACH statement is also useful. Its syntax is:

```
FOR-EACH process:
  statement-block
END-FOR-EACH
```

Because attributes may apply to more than one kind of entity, the language clarifies an ambiguous reference to an attribute by qualifying each attribute with the name of the entity to which the attribute belongs. For example, the attribute "security-level" applies to processes and several kinds of objects; "security-level(process)" refers to the security level of the process.

Rules of operation use the form "[* . . . *]" to identify a system operation. For example, the Open rule uses the statement [* truncate the file *] to stand for the UNIX operation that deletes the data in a file. Rules may use the form "(* . . . *)" to enclose a comment, such as (* the directory search was valid and the file exists *) appearing in the Open rule.

Boolean expressions and all statements except the SELECT CASE, the IF-THEN-ELSE, and the FOR EACH end with a semicolon. Boolean expressions use the usual inequality operators "<" and ">" and use "==" for expressing equality. Logical operators such as AND and OR are used in obvious ways.

Rules use the specifications "set-attribute" and "set-attributes" to manage the values of attributes. The rules of the Rule Set Model use "set-attribute" to designate the value that an attribute should have if the current request is granted. The syntax for this use is

set-attribute(attribute_name, attribute_value)

The rules of the State Machine Model use "set-attributes" to indicate that they are carrying out the set-attribute specifications given by the rules of the Rule Set Model. Suppose, for example, the State Machine Model invokes the Rule Set Model with a create-file request. Suppose that the rules of the Rule Set Model approve the request and give two set-attribute specifications:

set-attribute(security-level(file), **SECRET**)
set-attribute(object-category(file), **general**)

Then, the portion of the create rule that carries out the create request will include a set-attribute statement. The meaning of the statement is that the security-level of the file is set to the value **SECRET** and the object-category of the file is set to the value **general**.

A.2. Constructs of the State Machine Model

A.2.1. Types

A type is a class that is defined by the common attributes possessed by all its members. The name of each type suggests a useful interpretation for the class. The model uses the following types:

request: {alias, alter, change-owner, change-role, clone, create, delete, delete-data, execute, get-permissions-data, get-status-data, modify-access-data, modify-attribute, modify-permissions-data, read, read-attribute, read&write-open, read-open, search, send-signal, terminate, trace, write,

write-open}
process
file
directory
ipc
scd
signal
object: [a file, directory, ipc, or scd]
phase: {"active", "unused", "inaccessible"}
flag: {ON, OFF}
mode: {"read", "write", "read&write"}

A.2.2. Variables

A variable is an alterable entity. The variables of the State Machine Model define the system states. We can think of variables as functions whose domains are types. Just as naturally, we can regard them as records of information containing one or more items of data. The model uses the following variables:

current_process: process
new_process: process
file_name: file
directory_name: directory
truncate_option: flag
create_option: flag
STATUS(object): phase
OPEN(process, object): set(mode)

A.2.3. Constants

TRUE
FALSE
ON
OFF

A.2.4. Expressions

Access-Rules(request, process/object, process/object): Extended-Boolean

A.2.5. Effects

An effect is an action of the state machine. The model uses the following effects:

normal-exit
error-exit
set-attributes
save
restore

Endnotes

[1] UNIX is a registered trademark of UNIX System Laboratories.

[2] The arguments used in the system calls are similar to those defined by Bach (Bach, 1986) but I have changed the names, invented some new ones, and sometimes rearranged them for clarity.

[3] I show the first argument to the open call (and others) as file_name. This may actually be a pathname involving one or more directories. When file_name is used as an argument to the Access-Rules, the reader should understand that the name of the file itself, not the full pathname, is intended.

[4] The real or effective user ID of the sending process must match the effective or saved effective user ID of the receiving process, unless the effective user ID of the sending process is super-user. Recall that we do not model super-user access controls or controls based on user IDs.

[5] {} denotes the empty set.

[6] Recall from the Interface Definition that the terminate message means that the system has terminated the process. The State Machine Model gives this request to the Policy Model for information only. It enables the Policy Model to update its information base, if necessary.

[7] Note that the system in this context is System V, not System V/MLS. AT&T's System V/MLS additionally incorporates the mandatory access control (MAC) policy.

[8] A distinction is made between read and read-open in this model. Read-open functionally enables the process to read the object; read actually causes the transfer of data from the object that has been opened into the memory space of the process doing the reading. The MAC policy for controlling read access is, therefore, applied at the read-open and no MAC policy applies to the actual reading of the data. However, other models can be conceived in which a floating label policy for the sensitivity label of the object might be applied at the read, similar to the floating information label policy of the CMW model (Millen, 1990).

[9] "Write" to a directory includes addition, modification, and deletion of entries in the directory.

[10] An example is the User-Transformation Procedures Associations (UTPA) table of this model, described subsequently.

[11] In this model, the integrity data is contained in the User-Transformation Procedures Association (UTPA) table. Thus, only the TP-manager is allowed to modify the UTPA.

[12] In this context, an ordinary process is one whose process-type equals NIL.

[13] Obviously a certain amount of interpretation is involved here; the constraints I have specified seem reasonable to me, but I recognize that other interpretations of the intent of Clark-Wilson integrity are possible.

[14] The UNIX unlink operation actually deletes a file only when all links to it have been deleted.

[15] The definition of (NCSC, 1985) is assumed: security level level1 is said to dominate security level level2 if the hierarchical classification of level1 is greater than or equal to that of level2 and the non-hierarchical categories of level1 include all those of level2 as a subset.

[16] The security assumptions part of the SMMSM reflect security constraints on the behavior of users.

[17] The BLM *-property specifies part of the mandatory access control policy of the model. It is defined as follows: The *-property places restrictions on current access triples (subject, object, attribute) based on the value of current-level(subject):

if attribute is read, current-level(subject) dominates level(object);
if attribute is append, current-level(subject) is dominated by level(object);
if attribute is write, current-level(subject) equals level(object);
if attribute is execute, current-level(subject) and level(object) have no required relation.

[18] The SMMSM has 10 security assertions. The classification hierarchy security assertion is: The classification of any container is always at least as high as the maximum of the classifications of the entities it contains.

[19] The CWI model has five certification rules and four enforcement rules. Enforcement rule 3 is: The system must authenticate the identity of each user attempting to execute a TP.

[20] The referenced expression specifies the mandatory access control policy for the Compartmented Mode Workstation. It is defined as follows: Maccessible(s: Subject, o: Object, m: Mode): Boolean = m = "read" and "mac_override_read" ∈ Privs(s) or m = "write" and "mac_override_write" ∈ Privs(s) or m = "read" and Sens_label(s) >= Sens_label(o) or m = "write" and Sens_label(o) >= Sens_label(s) and Max_level(Owner(s)) >= Sens_label(o).

[21] For example, in System V/MLS a process can open a file for reading only but can open a message queue (via the msgget system call) only for reading and writing. Thus, the rules for opening a file differ from the rules for opening a message queue: the former allows the subject's sensitivity label to dominate the file's sensitivity label while the latter requires that the two labels be equal.