

# Rule-Set Modeling of a Trusted Computer System

Leonard J. LaPadula

---

This essay describes a new approach to formal modeling of a trusted computer system. A finite-state machine models the access operations of the trusted computer 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. We will show how this approach to modeling relates to general ideas of access control, as you might expect. We will also relate 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. The trust policies we demonstrate in the rule set of the model include 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.

The modeling approach we will discuss grew out of several ideas developed in the Generalized Framework for Access Control (GFAC) project directed by Abrams [ABRA90]. The vision of that project was to gain greater utility in our trusted computer systems by creating a technology for putting a rich set of various access control policies into a single trusted computer system. Our modeling approach responds to the challenge of that vision, as expressed in these objectives:

- Make it easy to state, formalize, and analyze access control policies besides traditional mandatory access control (MAC) and discretionary access control (DAC), to increase the availability of diverse, assured security policies.
- Make it feasible to configure a system with security policies chosen from a vendor-provided set of options with confidence that the re-

sulting system's security policy makes sense and will be properly enforced.

- Construct the model in a manner that allows one to show that it satisfies an accepted definition of each security policy it represents.

The remainder of this essay has three parts:

- First we discuss the Generalized Framework for Access Control view of a trusted system. GFAC motivates the approach we have taken to modeling.
- Next we describe the modeling approach.
- Finally we illustrate elements<sup>1</sup>of the state-machine model and the rule-set model — the two components of the complete model.

## **Overview of the Generalized Framework for Access Control**

The Generalized Framework for Access Control thesis asserts that all access control is based on a small set of fundamental concepts [ABRA90]. Borrowing some of its terminology and concepts from the ISO “Working Draft on Access Control Framework” [ISO90], GFAC starts with the premise that all access control policies can be viewed as rules expressed 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 certain attributes of controlled resources.

**Attributes:** Attributes describe characteristics or properties of subjects and objects. The computer system will base its decisions about access control on the attributes of the subjects and objects it controls. Examples of attributes are:

- security classification
- type of object
- domain of process
- date and time of last modification
- owner identification

---

<sup>1</sup>The reader will find additional analysis and a complete policy model in my report [LAPA91].

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

The generalized framework explicitly recognizes two parts of access control — adjudication and enforcement. We use the term access control decision facility (ADF) to denote the agent that adjudicates access control requests, and the term 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.

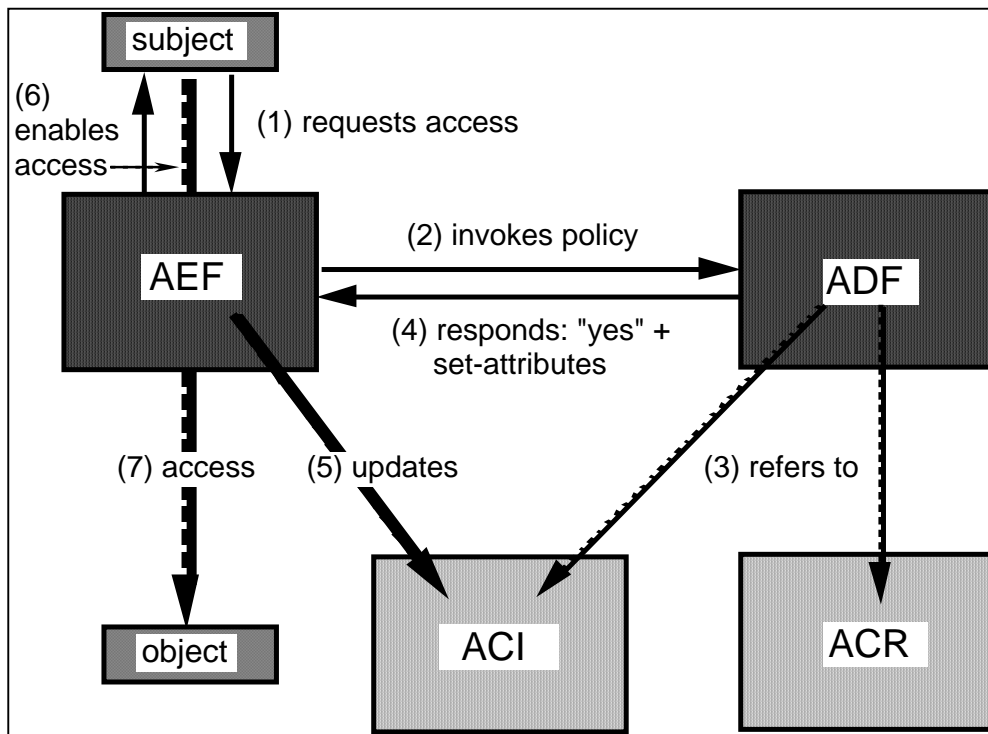


Figure 1. Overview of the Generalized Framework for Access Control.

## Formal modeling approach

**Background.** The GFAC goals translate into these objectives for our formal model:

- Develop a modeling technology in which it is easy to express various policies besides traditional MAC and DAC.
- Fashion the modeling technology to enable the selection of a desired set of security policies from some preevaluated set without having to reevaluate the resulting collection of policies.
- Provide for showing that a model satisfies an accepted definition of each security policy enforced by the model.

Our rule-set modeling has much in common with traditional state machine modeling. It differs significantly, though, in the way it sets up access rules. Models like the Bell-LaPadula model [BELL76] and the recent Compartmented Mode Workstation model [MILL90] include access control rules in their rules of operation. In these models, an Open File rule describes both access policy and system behavior. The rule uses built-in criteria to decide whether to permit the Open File request, and the rule describes the behavior of the modeled system as a state transition. A typical nondisclosure criterion is whether the security level of the subject making the Open File request dominates 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.

Our approach separates the decision criteria from the state transition descriptions. A rule set embodies the security policies of the modeled system while a finite state machine model describes the behavior of the system. Thus we have partitioned the system function (Open File) into two operations:

1. Decide if the request should be granted. (Is the subject allowed to open the referenced object?)
2. Grant the request (open the file) or not (return an error indication).

Figure 2 depicts this rule-set approach.

**Structure of the Model.** A trusted computer system built according to our modeling plan has two major parts inside its Trusted Computing Base (TCB). In the ISO terminology we mentioned earlier, our TCB has an access enforcement facility (AEF) and an access decision facility (ADF). In this

scheme, the AEF owns and operates the system operations that are available to computer programs that it runs. The ADF keeps the rule set that expresses the access policies of the system. When a computer program attempts to execute some system operation, the AEF appeals to the ADF for an access decision. As we pointed out earlier, the rule set of the ADF expresses the access policies of the system. Thus, the ADF will evaluate the rule set each time the AEF appeals to it for an access decision. Naturally, the AEF will provide some set of arguments to the ADF when it invokes the ADF. These arguments provide or define whatever access control information (ACI) the ADF needs for decision making.

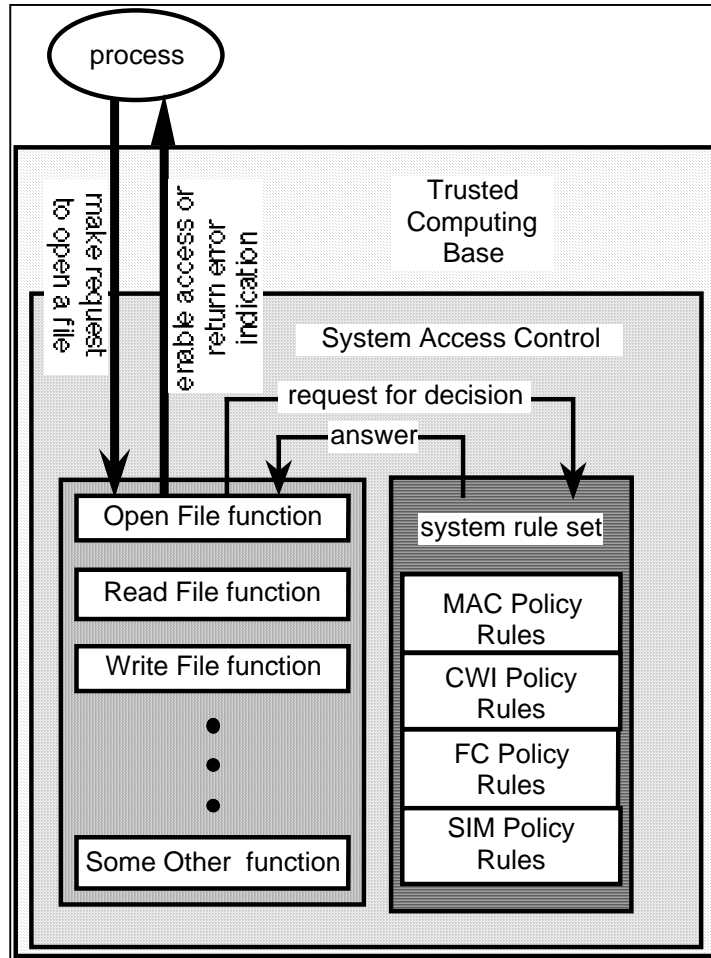


Figure 2. Rule-Set Modeling

Applying an AEF-ADF partitioning to our modeling gives us:

- A model of the system operations, constituting an AEF. We treat this model as a state machine and call it the State Machine Model. The State Machine Model abstractly describes the interface of computer PROCESSES to the system's TCB and defines the relationship of the system operations to the system's trust policies.
- A model of the security policy, viewed as an ADF. We define this model as a rule set and call it the Rule Set Model. It defines the security policies of the trusted computer system.

The State Machine and Rule Set Models need an interface between them. This interface allows the state machine to invoke the rule set for adjudication of a process's request. The design of the interface depends on several critical factors:

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

We have decided these issues for this essay as follows:

- The State Machine Model targets the class of UNIX™ System V systems.
- The state machine has a transition rule for each UNIX™System V system call. Each transition rule is an abstraction of its corresponding system call, but the abstraction preserves the essential functionality of the system call.
- The rule set addresses essentially the same level of detail as the state machine. But, to the extent that our art of modeling allows, we generalize it to make it useful with other state machines. You will see the form this generalization has as we discuss the interface between the state machine and the rule set in the next section.

**Interface Between the State Machine and the Rule Set.** Our modeling mirrors a trusted computer system in which the TCB has two parts. The access control enforcement (AEF) part affords system services to processes and determines the behavior of the system. The access control decision facility (ADF) part remembers the system's security policies and decides whether processes' requests satisfy those policies. The AEF part communicates with the ADF part through some appropriate interface. Our modeling approach has the same structure:

- The State Machine Model (enforcement agent) corresponds to the AEF part of the trusted computer system's TCB.
- The Rule Set Model (judicial agent) corresponds to the ADF part of the TCB.

The State Machine Model has rules of operation. Each rule of operation abstractly describes some behavior of the modeled system. The Rule Set Model has rules of access. Each rule of access describes some security policy of the modeled system.

We relate the models to each other with an interface. The interface permits communication between the state machine and the rule set. Briefly, the communication occurs through messages. We will see the formal structure of these messages later.

Let's consider an example, using a real system as an analogue of our model's structure. The real system for this example is UNIX System V. We imagine that its kernel has 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 must contain or reference 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, or returning an error message if not.

Our model operates in much the same manner. The State Machine Model corresponds to the AEF-kernel. Corresponding to each system call supported by the AEF-kernel we have a rule of operation in the State Machine Model. The Rule Set Model corresponds to the ADF-kernel. The Rule Set Model expresses the security policies of the modeled system in its rules of access. Each rule of operation in the State Machine Model "invokes" the Rule Set Model with a function we call "Access-Rules." The arguments of the function Access-Rules correspond to the messages exchanged between the AEF-kernel and ADF-kernel.

Again, let's consider a simple example based on the notion of an Open system call. 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 desired action of the process and a set of relevant attributes (access control information). So, we can define the needed interface by a set of requests with appropriate access control information. We will use several terms for the needed information. We explain those terms now to show their specific meanings for the interface definition.

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 queues or semaphores.
scd	a set of attributes associated with a UNIX system object that stores system control data (therefore the acronym "scd"); the 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. We include them to help exposition of various security policies later in this essay. Then their meaning and use should become clear.

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 also may be used for communication of 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." Modelers should choose whatever requests make sense for the system they wish to model. We describe requests here that seem appropriate for detailed modeling of systems like UNIX System V. You should notice these features:

- For each system call of UNIX System V, as defined by Bach [BACH86], 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. For example, we have several requests that 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.

Here is the list of requests:

- ALIAS (process, file). The process is attempting to create an alias 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, ipc12). 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<sup>13</sup>. 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 argu-

ment "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 signal14 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 Rule Set Model for information only. This request enables 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.

## Examples of the Model Components

**State Machine Model.** The UNIX System V system as described by Bach [BACH86] shapes the nature of our State Machine Model. We will not develop a complete State Machine Model in this essay since we want to explain the modeling approach, not build a trusted computer system. On the other hand, we have tried to achieve the breadth and depth of coverage needed to make obvious what a modeler should do. To this end, this section contains rules of operation that abstractly describe many of the key system calls of UNIX System V. The selected rules adequately demonstrate the modeling approach we have described.

*Introduction* The State Machine Model is a state-transition machine. Its rules of operation define the valid transitions for the modeled system. Recall we said earlier 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. Doing so 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 demands a one-to-many mapping of system call to rules of operation by the designer or evaluator of the system.

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
STATUS(file_name) = "active"
AND
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 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 above form of the open rule does not reflect these options. In the following form<sup>2</sup> we show not only these options but additional details of the system call such as directory searching.

*Rules of Operation.* This section gives the State Machine model specification for the following rules of operation:

- Open
- Read
- Fork
- Create
- Execute
- Kill
- Unlink

- **Open** (file\_name, mode, truncate\_option, create\_option<sup>3</sup>). The open system call is the first operation a process performs to access data in 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

---

<sup>2</sup>The reader with computer systems experience should have no trouble understanding the language that expresses the rules. The reader who may not be familiar with computer programming languages or who may want to verify the meaning of a language form should see appendix 1, which gives a description of the language as well as the modeling constructs employed.

<sup>3</sup>The arguments used in this essay 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 of the rules.

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 pathname4]
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;
```

---

<sup>4</sup>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 a parameter to the Access-Rules, the reader should understand that the name of the file itself, not the full pathname, is intended.

```

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;

```

- **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;
```

- **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 ob-
  jects 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;
```

- **creat (file\_name, mode)**. The creat (create) system call creates a new file in the system. But if the file already exists, the kernel opens it and truncates it if permissible.

```
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 *)
(* check if it is permissible to delete the data in the file and write-
    open it *)
save (* attribute values of current_process and file_name *);
IF
Access-Rules (delete-data, current_process, file_name)
THEN
    set-attributes;
IF
    Access-Rules (write-open, current_process, file_name)
THEN
    set-attributes;
    (* clear the file and write-open it *);
    OPEN(current_process, file_name) = OPEN(current_process,
        file_name) SET-UNION {mode};
    normal-exit;
ELSE
    restore (* restore the attribute values of current_process and
        file_name *);
    error-exit;
ELSE
    error-exit;
CASE STATUS(file_name) == "unused" (* the directory search was
    valid and the file does not exist *)
(* check if it is permissible to create the file, write the relevant di-
    rectory, and write-open the file *)
save (* save the attribute values of current_process, file_name, and
    directory *);
IF
    Access-Rules (create, current_process, file_name)
THEN
    set-attributes;
IF
    Access-Rules (write, current_process, directory (* directory is
        found by the namei kernel subroutine in UNIX *))
THEN
    set-attributes;
IF
    Access-Rules (write-open, current_process, file_name)
THEN
    set-attributes;
    (* create the file, with appropriate directory entry, and write-
        open it *);
ELSE

```



```

        restore (* restore the attribute values of current_process,
                file_name, and directory *);
        error-exit;
    ELSE
        restore (* attribute values of current_process, file_name, and
                directory *);
        error-exit;
    ELSE
        error-exit;
ELSE
    error-exit;

```

- **exec** (file\_name). The exec system call causes process execution to continue with the code contained in the named file. This should not be confused with creating a new process, which is the normal result of a fork system call. The execution of the code specified by the exec system call is part of the process that invoked the exec system call. Thus, the open files of the process are still open after a successful exec.

```

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
            Access-Rules(execute, current_process, file_name)
        THEN
            set-attributes;
            (* carry out the "exec"ing of the filename *);
            normal-exit;
        ELSE
            error-exit;
    CASE STATUS(file_name) == "unused" (* the directory search was
        valid and the file does not exist *)
        error-exit;
ELSE
    error-exit;

```

- **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<sup>5</sup>. 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) = {}6 for every file_name;
      Access-Rules(terminate7, 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;
```

- **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

---

<sup>5</sup>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.

<sup>6</sup>{ } denotes the empty set.

<sup>7</sup>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.

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;
```

*Additional remarks.* The modeler has choices to make. A fundamental decision is whether the rules of operation of the state machine model will map one-to-one to system calls or not. Successful modeling can be done either way. The differences between the two approaches can be characterized as tradeoffs, as you might expect. 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. 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 the basic approach, the modeler should examine each operation the system provides for processes. In UNIX System V these operations are system calls. The modeler should figure out what each system call does 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. What we mean here are the policies defined by the Rule Set Model. Some system calls have no relevance to traditional mandatory access control policy 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.

The modeler should decide if the rules of operation needed to model the system calls exist in the state machine model. If not, they should, of course, be added to the model. The modeler also should define how each rule uses the interface definition to invoke the Rule Set Model.

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

**Rule Set Model.** This section describes a Rule Set Model for a trusted computer system that implements the following 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, Release 1.2.1. This system<sup>8</sup> received a B1 rating from the National Computer Security Center in 1989. The MAC policy shows the traditional security policy for a trusted computer system. Its inclusion shows that other policies can be integrated with the traditional nondisclosure security requirements. This 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 (CLAR87) 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 admin-

---

<sup>8</sup>The evaluated product was System V/MLS, Version 1.1.2, running with UNIX System V Release 3.1.1 on the AT&T 3B2/500 or AT&T 3B2/600 minicomputers. Through the rating maintenance program (RAMP) of the NCSC, the rating was extended in September, 1990 to System V/MLS Release 1.2.0 and 630/MLS Release 1.2.0 running with UNIX System V Release 3.1.1 on the AT&T 3b2/500 and AT&T3B2/600 minicomputers and the AT&T 630 MTG terminal.

istrator, security officer, and user and has 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 essay focuses mainly on the MAC and CWI policies but includes the FC and SIM policies for completeness. A useful trusted computer 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. But I will not give complete policies because the purpose is to describe the approach not to provide a complete formal model. Before describing the policies modeled in this essay, we 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.

*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 supports the ability of 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. What the kernel ensures is that the permissions defined by the users and the system will be enforced. But, in addition, the kernel incorporates a non-discretionary policy based on superuser 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<sup>9</sup>.

Our model 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 superuser approach in trusted computer systems to provide better separation of duty. Instead of modeling the superuser-based IBAC of UNIX, this Rule Set Model has a functional control policy which 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 really doesn't belong in the Rule Set Model because DAC is not a policy the system can enforce — DAC is 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. For example, a Trojan Horse can easily defeat a user's nondisclosure objectives since the DAC mechanism provides no

---

<sup>9</sup>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.

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 that users are likely to have an interest in.

In this formal model, we assume 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.

*Access Control Information.* To support the policies described herein, the following attributes in three groups of access control information (ACI), are needed. Each attribute shown here is explained in one of the policy descriptions that follow.

<b>USER-ACI</b>	<b>VALUES</b>
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

<b>PROCESS-ACI</b>	<b>VALUES</b>
owner (pointer to USER-ACI)	—
security-level	a security level
process-identifier	a process identifier
process-type	<b>NIL, TP, IVP, or TPICD</b>

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

*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;
- other aspects of processing — for example, spawning a child process and sending a signal to another process.

The policy described here represents the MAC policy of UNIX System V/MLS, Release 1.2.1 (FLIN88) and uses the types of objects defined for that implementation. System V/MLS is a multi-user, multi-tasking operating system that maintains UNIX System V application compatibility. In addition to using the traditional protection mechanisms of the UNIX operating system to provide DAC, System V/MLS provides MAC to limit the distribution of information to authorized users. The MAC policy is consistent with the Bell-LaPadula model (NCSC85) and satisfies DOD policy.

Basically, the MAC policy depends on the security-level attribute of processes and objects and on the object-type attribute of objects.

The object-type values defined for this policy<sup>10</sup> 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"; the inode in UNIX maps to this type. Generally, system control data refers to data the system uses to control its operations.

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.

Table 1 through 4 define, respectively, the policies for controlling access of a process to an object of types file, directory, tpc, and scd.

---

<sup>10</sup>We could model additional types as well. We might, for example, wish to include a type "communications-device", realized as a socket in some UNIX systems, if we wanted to have intercomputer communications reflected in our model.

**Table 1. MAC policy for objects of type file.**

<b>If the request is</b>	<b>then access is allowed if</b>
create	O is set equal to P
delete	P = O
delete-data	P = O
execute	P >= O
read	no condition <sup>11</sup>
read-open	P >= O
read&write-open	P = O
write	no condition (see footnote for read above)
write-open	P = O

**Table 2. MAC policy for objects of type directory.**

<b>If the request is</b>	<b>then access is allowed if</b>
create	O is set equal to P
delete	P = O
read	P >= O
search	P >= O
write <sup>12</sup>	P = O

<sup>11</sup>A distinction is made between read and read-open and between write and write-open in this model. Read-open (write-open) enables the process to read (write) the object, read (write) actually transfers data from (to) the open object into (from) the memory space of the process. The MAC policy for controlling read (write) access applies at the read-open (write-open) and no MAC policy applies to the transfer of the data. Other models can be conceived in which a floating label policy for the security level of the object might be applied when a read or write (an actual transfer of data) occurs, similar to the floating information label policy of the CMW model (MILL90).

<sup>12</sup>"Write" to a directory includes addition, modification, and deletion of entries in the directory.



**Table 3. MAC policy for objects of type ipc.**

<b>If the request is</b>	<b>then access is allowed if</b>
alter	$P = O$
create	$O$ is set equal to $P$
delete	$P = O$
read	no condition <sup>13</sup>
read&write-open	$P = O$
write	no condition

**Table 4. MAC policy for objects of type scd.**

<b>If the request is</b>	<b>then access is allowed if</b>
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. In this model, the separate requests have been used to illustrate the possibility of making access control decisions on the basis of the distinctions represented by these requests.

Table 5 defines the MAC policy governing process management.

**Table 5. MAC policy for process management.**

<b>If the request is</b>	<b>then access is allowed if</b>
clone	$P2$ is set equal to $P1$
send-signal	$P1 = P2$

<sup>13</sup>The distinction between read and read-open and between write and write-open for files applies also to **ipc** objects in this model.

*Clark-Wilson integrity policy* The integrity policy comes directly from the Clark-Wilson Integrity (CWI) policy (CLAR87). I have attempted to model their policy in as straightforward a manner as possible, using their concepts, terminology, and point of view as literally as possible. In addition, I have included the ancillary policy that appears to me necessary to support their intentions. For the convenience of the reader, a summary of the Clark-Wilson model's certification and enforcement rules is given in Appendix B. But the reader is encouraged to read their paper (CLAR87) for an understanding of their motivation and a greater appreciation of their model.

The CWI 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 it relates to. Measures for internal consistency ensure that modification of data results in a valid state. And, some CWI rules deal with the relationship between the internal and external consistency of data. The integrity control policy in this model focuses on the rules for internal consistency and also supports the capability to ensure external consistency. Some of the CWI rules that deal with external consistency are, naturally, not reflected in this "internal" system model.

Integrity control in the model is based on

- 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:

<b>TP:</b>	means that the object is a CWI TP
<b>IVP:</b>	means that the object is a CWI IVP
<b>TPICD:</b>	means that the object is a special TP that operates on CWI integrity control data
<b>NIL</b> <sup>14</sup> :	means that the object is not an integrity-controlled object.

---

<sup>14</sup>NIL has its obvious meaning -- nothing -- which, in this context amounts to saying that the object is not any of the other types defined. The attribute "program-type" could be

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 that is used for integrity control<sup>15</sup>
  - NIL:** means that the data is not integrity-controlled, that is, in the terminology of Clark and Wilson it is an Unconstrained Data Item (UDI).
- The user attribute "integrity-role" may have the following values:
  - TP-user:** means that the user is authorized to execute Transformation Procedures (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 Integrity Verification Procedures (IVPs)
  - IVP-manager:** means that the user is authorized to manage IVPs, as specified below
  - NIL:** means that the user has no integrity role.
- The authorizations of a user with an integrity role are described in Table 6.

---

used for other policies as well, in which case other values might be defined for it. In this model, this attribute is used only for enforcing the integrity policy.

<sup>15</sup>An example is the User-Transformation Procedures Associations (UTPA) table of this model, described subsequently.

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

<b>User in integrity-role</b>	<b>may execute</b>	<b>may create or delete</b>	<b>may modify</b>
TP-user	TPs		
TP-manager	TPICD	TPs, TPICDs, CDIICs	CDIIC
IVP-user	IVPs		
IVP-manager		IVPs, CDIs	

Clark and Wilson require that the system “maintain a list of relations of the form: (UserID, TP<sub>i</sub>, (CDI<sub>a</sub>, CDI<sub>b</sub>, CDI<sub>c</sub>, ...)), 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.” This model uses a User-Transformation Procedures Associations (UTPA) table to capture the Clark-Wilson triples. Each entry in the table is an ordered triple of the form

(user-identifier, TP, list of CDIs)

No constraint is put on modes or order of access since Clark and Wilson do not require it, although one can conceive a system in which such constraints would serve a useful purpose. Still, other policies in the system 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 CWI requirement to maintain a list of relations. One can imagine many designs for ensuring that only executions described in the UTPA are performed. For example, we could define a new system call, say `apply(TP,list_of_CDIs)`. “apply” would be like the `exec` system call, but having the additional second argument. This argument shows which CDIs the requesting process wishes the TP to operate on. The kernel would pass the arguments of this system call to the rule set. The rule set would check the UTPA to see if the list of CDIs was valid for the owner of the requesting process. The difficulty is that the “enforcement” provided by this approach is weak. Lacking any other access checks, 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 would, possibly, be acceptable if all correct and authorized executions were 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." Also, using "apply" effectively rules out interaction between the user and the TP. Ideally, the user should be able to provide input arguments specifying which CDIs to operate on.

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 has the value **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<sub>22</sub>, 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, or
- to CDIs 2 and 3

This is illustrated in Table 7, the user-transformation procedures associations (UTPA) table:

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

<b>user</b>	<b>TP</b>	<b>CDIs</b>	<b>processes</b>
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 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. The scheme just outlined describes the situation when a process executes just 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 as I understand it, 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<sup>16</sup> 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<sup>17</sup> 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. By aliasing, an ordinary user could defeat the attempt of an authorized user (that is, TP-

---

<sup>16</sup>In this context, an ordinary process is one whose process-type equals NULL.

<sup>17</sup>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.

manager) to remove a TP from the system<sup>18</sup>. Our policy, then, is that only individual's with an appropriate role can give an integrity-controlled object an alias. Specifically, a user in the role TP-manager may alias TPs, TPICDs, and CDIs while a user in the role IVP-manager may alias 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. Specifically, a user in the role TP-manager may read/modify status information about TPs, TPICDs, and CDIs. A user in the IVP-manager role may read/modify status information about IVPs and CDIICs.
- Our integrity policy 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 the TP-process could be killed (terminated) at such a time that the CDIs on which it was operating would be left in an undefined state. A justification for allowing the signaling anyway is that it preserves functionality and the TP can be designed to take appropriate action on the CDIs before exiting. This preserves functionality but does put the burden on the certification of the TP to ensure that the TP handles signals appropriately.

*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**. When a process whose owner has system-role R, requests access to an object having object-category C, the FC policy allows the access only if R is compatible with C. We define *is compatible with* as follows:

- **user** is compatible with **general**
- **administrator** is compatible with **general** and **system**
- **security officer** is compatible with **general** and **security**.

A functional control policy can and probably should be more elaborate than the one just described. It can, for example, specify who can change what attributes of what entities. And, it's the logical place to have the policy that governs trusted subjects, such as daemons of the UNIX system. But the simple policy just given satisfies the goals of this essay so we develop it no further here.

---

<sup>18</sup>The UNIX unlink operation actually deletes a file only when all links to it have been deleted.



*Security information modification (SIM) policy.* 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, the SIM Policy allows 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 essay.

*Rules of the rule-set model.* Four groups of rules define the policies described above:

- mandatory aAccess control (MAC) Rules
- Clark-Wilson integrity (CWI) rules
- functional control (FC) Rules
- security iInformation modification (SIM) rules.

Each policy of the Rule Set Model is implemented as one or more rules. When combined as described below, 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 result is that the request may be granted according to the rule's policy.
- **NO.** This value means that the request of the State Machine Model has been evaluated by the rule and the result is that the request may not be granted according to the rule's policy.
- **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's policy 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 in that both NO and DC indicate that the Rule Set Model is cognizant of the request. UNDEFINED not only

provides useful information for analysis of a rule set, but in a system implementation might serve to detect improper configurations of the system.

===== here

-

## Original text resumes

formal methods closer to the final stages of implementation — complete functional design and coding.

### Appendix A: Model language and constructs

**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 the attribute belongs to. 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** and the **IF THEN ELSE** 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**.

### **Constructs of the state-machine model**

*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, mod-
ify-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"}
```

*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)

#### *Constants*

TRUE  
FALSE  
ON  
OFF

#### *Expressions*

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

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

normal-exit  
error-exit  
set-attributes  
save  
restore

## **Appendix B: Summary of the Clark-Wilson integrity model**

Certification Rule 1: All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.

Certification Rule 2: All 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: (TP<sub>i</sub>, (CDI<sub>a</sub>, CDI<sub>b</sub>, CDI<sub>c</sub>, ...)), 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, TP<sub>i</sub>, (CDI<sub>a</sub>, CDI<sub>b</sub>, CDI<sub>c</sub>, ...)), 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. [Note to the reader: My model of Clark-Wilson integrity allows a TP to access any UDI in the normal manner for access to an object by a process in this system, subject to the constraints of the other (than integrity) policies implemented by the ADF. It is up to the certification process to 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 approach is to add the names of the allowed UDIs for a particular TP to the triples or, perhaps better, to the TP-CDIs relation, which would have to be added to the model since it is currently not included. Doing so would mean that the TP-CDI relation is no longer redundant with the triples.]

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 who can certify an entity may not (that is, must not) have any execute rights with respect to that entity.

## **Acknowledgments**

I thank Marshall Abrams for his fundamental insights on access control that led to the modeling approach described in this essay and for his encouragement during the writing of this essay. I thank James Williams of the MITRE Corporation for sharing with me his views on the stages of elaboration of requirements for trusted systems and for many conversations about formal modeling. I thank Charles W. Flink II of AT&T Bell Laboratories for his patient and comprehensive explanations of many design aspects and system calls of System V/MLS, Release 1.2.1.