SECURITY POLICIES AND SECURITY MODELS

J. A. Goguen and J. Meseguer
SRI International
Menlo Park CA 94025

1 Introduction

We assume that the reader is familiar with the ubiquity of information in the modern world and is sympathetic with the need for restricting rights to read, add, modify, or delete information in specific contexts. This need is particularly acute for systems having computers as significant components.

This paper motivates and outlines a new approach to secure systems with the following novel properties:

- It introduces a simple and general automaton theoretic approach to modelling secure systems.
- It shows how to use abstract capabilities to model the dynamic security aspects of such systems.
- The approach can be applied not only to computer operating systems, but also to secure message systems, and to data base systems; it is not limited to systems which are entirely computer based, but applies just as well to systems that contain manual components, and even to entirely manual systems.
- It introduces a general concept of security policy, such that allowed policies include multi-level security (MLS), capability passing, confinement, compartmentation, discretionary access, multi-user/multi key access, automatic distribution and authorization chains, and downgrading.
- It does without the notion of "trusted processes."[1]
- It provides a formalism for the specification of security policies.
- It supports a rigorous treatment of intransitive information flow.
- It provides techniques for proving that a given system satisfies a given policy.
- It supports use of a hierarchy of models at various levels of detail, thus making it easier to prove security properties by factoring the difficulties, and by letting them appear at their proper level of abstraction.

However, our approach does not address the problems of user authentication, of security breaches arising through inference, either logical or statistical, of unauthorized information from information which is authorized (the so-called aggregation problem), or of fault-tolerant secure computing.

The literature on computer security provides many different "security models," without saying what a security model actually is. We propose to distinguish sharply between a security policy, which defines the security requirements for a given system, and the system itself, which may be represented by a model, for example, a high level specification or an abstract machine description of what the system does[2]. In general, security policies are very simple, and should be easy to state in an appropriate formalism. We provide a very simple requirement language for stating security policies, based on the concept of noninterference, where

> one group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

In this approach, security verification consists of showing that a given policy is satisfied by a given model. Taking this abstract view considerably simplifies many aspects of the problem.

[1]A "trusted process" is generally taken to be a subsystem that is permitted to violate some global security policy, usually MLS. However, it seems to us unnecessarily dangerous to admit subsystems that are permitted to perform arbitrary actions upon system resources; rather, one should state precise policies for the interaction of these subsystems with the whole system, and then verify that those policies are satisfied.

[2]Even though such a model describes how some system intends to achieve security, it probably should not be called a "security model" unless it has been proved to satisfy some security policy; in particular, we should also be able to consider models of insecure systems, in order to be able to explore their flaws.

Information flow techniques attempt to analyze what users (or processes, or variables) can potentially interfere with others, whereas we begin oppositely, by saying what users (or processes, or variables) must not interfere with others in order that some security policy hold. Information flow techniques generally proceed to form the transitive closure of a "potentially flows" relation, and thus end up with a large number of cases that actually cannot occur, but nonetheless must be analyzed. We hope to avoid this bog by using a more refined analysis based on noninterference and containing explicit information about the operations invoked by users.

Our work was in part inspired by the approach of [Feiertag 80] and [Feiertag, Levitt & Robinson 77]. Some differences are that we are more explicit and rigorous about our framework and assumptions, we treat the general dynamical case in which capabilities may be passed between users, and we consider arbitrary security policies, rather than just MLS. Our approach is related to that of [Rushby 81a] which also uses an automaton model. Differences are that [Rushby 81a] only considers the static case (no passing of capabilities) of a policy that separates processes by permitting them to communicate only through specified channels.

The reader who wants more background information on computer security should consult [Denning 82], or [Landwehr 81], [Rushby 81b] or [Turn 81].

1.1 The Problem of Defining Security

The purpose of a so-called "security model" is to provide a basis for determining whether or not a system is secure, and if not, for detecting its flaws. Most of the models given in the literature are not mathematically rigorous enough to support meaningful determinations of the kind needed; some do not support a sufficiently general view of security (for example, they may fail to handle breaches of security by cooperating multiple users); and some are restricted to specific kinds of system, or even to just one system. These models are often very complex, and thus obscure the fundamental intuition which is usually very simple.

Many of the most complicated security problems arise in connection with so-called capability systems; for example, the passing of capabilities among users can lead to situations in which it is difficult to determine whether or not security can be violated. The rigorous mathematical verification of non-trivial security policies for such systems seems not to have been previously studied.

One assumption behind this paper is that security is fundamentally a requirement for certain systems[3], where we use the word "requirement" to refer to the social context of a system, rather than to some pre-existing mathematical model. Most work in computer security has ignored the social contexts in which systems are actually used. However, different organizations have different security needs, and use their systems in different ways; in general, they have different security policies. Providing a model without understanding the needs of the community involved is unlikely to yield the most useful results. What is ultimately necessary is that the actual community of users should be satisfied that the system they are using is sufficiently secure in a sense which is appropriate for their particular purposes. Once the needs of a community have been understood, it may be possible to formalize those needs and to model their information processing system; at this point it will be meaningful and useful to provide proofs. In this, we disagree with [DeMillo, Lipton & Perlis 77], who seem to believe that the social process in itself can be adequate for security verification; see [Dijkstra 78] for related dialectics.

Thus, we envision a four stage approach: first, determine the security needs of a given community; second, express those needs as a formal requirement; third, model the system which that community is (or will be) using; and last, verify that this model satisfies the requirement. Only the last of these steps is purely mathematical, although the other steps have mathematical aspects; the remainder of this paper concentrates on such aspects.

1.2 The Problem of Verifying Security

A great deal has been written about the verification of allegedly secure systems. A basic point is that it is necessary to verify not only that some high level design specification, such as might be implemented in PSOS [Neumann, Boyer, Feiertag, Levitt & Robinson 80], satisfies some security policy, such as MLS, but one must also verify that the code for the system actually satisfies the design specification. In particular, it will be necessary to check at some level that such features as interrupts and memory maps are handled correctly. It is here that work on security kernels becomes relevant, as attempts to simplify these verification problems.

In general, work on security kernels has not addressed issues of security policy. For example, [Popek & Farber 78] presents an automaton-like model for a mechanism to enforce access control in their kernel, but they do not attempt to show that it satisfies any particular constraints on the flow of information. Rather, they provide suggestions on how to prove that some code satisfies their model. Their model falls under our heading of "dynamic capability systems," because they have a "policy manager" process which can change the protection data of the system. Unfortunately, they fail to state any restrictions on this process.

[Rushby 81a] discusses the notion of a "separation kernel," which it seems might be useful for modelling and verifying the lower levels of an abstract machine hierarchy, because it does permit discussion of interrupts, memory maps, etc., and also simplifies system structure by postulating separation and channel control policies for component processes.

[3]Although we will often speak as if concerned only with computer systems, in fact our aproach, including general policies, capability models, and verification can also be applied directly to manual or mixed manual-computer systems, and our use of the word "system" should be understood in this sense.

1.3 Some Highlights of the Approach

In order to treat a sufficiently wide variety of systems, we need a rather abstract notion of system. This paper gives a set theoretic model which is a sort of generalized automaton, called a "capability system." It has an ordinary state machine component, and also a capability machine component which keeps track of what actions are permitted to what users. Systems which do not use capabilities can be modelled with this notion, simply by omitting the capability component.

We define what it means for one set of users to be "noninterfering with" another; this formalizes the notion of information not flowing where it shouldn't, without assuming that information flow is necessarily transitive. We next provide a general definition of "security policy" and discuss what it means for a given capability system to satisfy a given security policy, thus defining the security verification problem. Finally, we suggest some methods for actually carrying out such verifications.

Recent work on abstract data types suggests that it may be unnecessary and even harmful to separate a data structure from the commands which create, access, and modify it [Goguen, Thatcher & Wagner 78, Guttag 75]. Furthermore, experience with applications suggests that it is necessary to take account of less conventional commands, for example, commands that "execute" data, or that "summarize" it statistically. For this reason, we consider "abilities," which are simply sets of commands.

1.4 Acknowledgements

We would very much like to thank Peter Neumann, Karl Levitt, Rob Shostak, and Rich Feiertag for their valuable comments on this work as it has evolved through several drafts and lectures. We also thank Dorothy Denning, Carl Landwehr, Roger Schell, Ken Shotting, and Gene Wiatrowski for their valuable comments.

2 Security Models

This section presents our technique for modelling secure systems, using standard structures from automaton theory in a way that incorporates capabilities.

2.1 Static Systems

We first discuss the classic case in which what users are permitted to do does not change over time. We may assume that all the information about what users are permitted to do is encoded in a single abstract "capability table"[4].

The system will also have information which is not concerned with what is permitted; this will include users' programs, data, messages, etc. We will call a complete characterization of all such information a <u>state</u> of the system, and we let S denote the set of <u>all</u> such states. The system will provide commands that change these states; their effect can be described by a function

$$do: S \times U \times C \rightarrow S$$

where C is the set of state changing commands and U is the set of users. It might be that if u is not permitted to perform a command c, then do(s,u,c) = s; such security restrictions can be implemented by consulting the capability table, and are here simply assumed to be built into the function do.

We will also assume that for a given state and user, we know what output (if any) is sent to that user. This aspect of the system can be described by a function

$$out: S \times U \rightarrow Out \ ,$$

where Out is the set of all possible outputs (e.g., screen display states, listings, etc.). We will assume that all information given to users by the system is encoded in this function. (Presumably users can also get information in other ways outside the system, but we do not attempt to describe that.) Note that the function out may also consult the capability table in determining its value.

Putting all this together, and adding an initial state, we get a simplified version of our general model to be given later, which is just an ordinary automaton[5] A significant advantage of having our model based on a standard notion like the automaton is that an extensive literature and well developed intuition become immediately applicable to our problem domain.

Definition 1: A <u>state machine</u> M consists of the following:

- A set U whose elements are called "users."
- A set S whose elements are called "states."
- A set C whose elements are called "state commands."
- A set Out whose elements are called "outputs."

together with

[4]Of course, in an actual implementation, this information may be distributed among a number of tables and algorithms. Exactly how this is done does not concern us in this model; we only need an abstract summary of what is permitted and what is not. In this respect, our "capability tables" are much like the "access matrices" of [Lampson 74]. Thus, our abstract capabilities should be carefully distinguished from "concrete capabilities," which might be, for example, very long machine words used to actually implement abstract capabilities.

[5]Note that there is no assumption here that any of the sets involved are finite. This convenient fiction permits, for example, the state set S to include a pushdown store, and allows the storage of arbitrarily large integers and arbitrarily long character strings.

- A function out: S x U -> Out which "tells what a given user sees when the machine is in a given state," called the output function.

- A function do: S x U x C -> S which "tells how states are updated by commands," called the state transition function.

- A constant s0, the initial machine state, an element of S.

[]

The connection with the standard form of the definition of state machine is to take U x C to be the set of inputs.

What we have called "users" could also be taken to be "subjects" in the more general way in which that word is sometimes used in operating systems theory. Processes can be handled in this way.

2.2 Capability Systems

In order to handle the dynamic case, in which what users are permitted to do can change with time, we will assume that in addition to the state machine features there are also "capability commands" that can change the capability table. The effects of such commands can be described by a function

cdo: Capt x U x CC -> Capt ,

where Capt is the set of all possible capability tables, U is the set of all users, and CC is the set of capability commands. (If a user u is not allowed to perform the capability command c on the table t, then cdo(t,u,c) may be just t again; as before, this is determined by consulting the capability table.) In order to distinguish capability commands from state commands, in the following we will denote the set of all state commands by SC. So that the state transitions and the outputs can be checked for security against the capability table, we will add a capability table component to the state transition function and to the output function.

Adding all this to the definition of a state machine, and also adding an initial capability table, we get the following as our basic concept:

Definition 2: A capability system M consists of the following:

- A set U whose elements are called "users."

- A set S whose elements are called "states."

- A set SC whose elements are called "state commands."

- A set Out whose elements are called "outputs."

- A set Capt whose elements are called "capability tables."

- A set CC whose elements are called "capability commands."

together with

- A function out: S x Capt x U -> Out which "tells what a given user sees when the machine, including its capability component, is in a given state," called the output function.

- A function do: S x Capt x U x SC -> S which "tells how states are updated by commands," called the state transition function.

- A function cdo: Capt x U x CC -> Capt which "tells how capability tables are updated," called the capability transtion function.

- Constants t0 and s0, the "initial capability table" and the "initial machine state," respectively elements of Capt and of S.

[]

For convenience in the following, let C = CC U SC, the set of all commands.

As we have formulated capability systems, there are no commands that change both the state and the capability table; however, if it is desired to accommodate such commands for some application, our definition can be easily changed to do so.

The capability component of a capability system is itself a state machine, with state set Capt and input set U x CC; it models the way in which the capability table is updated, and includes such possibilities as passing and creating capabilities. We will call it the capability sub-machine of the capability system. The entire capability system is a cascade connection (in the sense of automaton theory, e.g. what is called "serial connection" in [Hartmanis & Stearns 66]) of this machine with another that models the processing of all the non-capability information, such as user files. In the following figure illustrating this cascade connection, the function Check returns the information from the capability table needed to determine whether or not a given command is authorized for a given user.

Let us call a subset of C = SC U CC an ability, and let Ab denote the set of all such subsets. Then it would make sense, for example, to let Capt = [U -> Ab], the set of all functions from U to Ab, so that a capability table would tell us exactly what ability each user actually has. However, our abstract model does not require any such particular representation. We will discuss later on some

```
                    State Commands
             ---------------------------
            |                           |
            |                           V
            |    -------             -------
   Commands |   |       |   CHECK   |       |   Out
   ---------->  | Capt  |---------->|   S   |-------->
                |       |           |       |
                 -------             -------
```

Figure 1: Cascade Connection

particular models which use representations like this to implement specific security policies.

Given a capability system M, we can define a system transition function which describes the effect of commands on the combined system state space, which is S x Capt. This function

$$\text{csdo}: S \times Capt \times U \times C \to S \times Capt$$

is defined, for s in S, t in Capt and u in U by

$$\text{csdo}(s,t,u,c) = (\text{do}(s,t,u,c),t) \text{ if c is in SC}$$

and

$$\text{csdo}(s,t,u,c) = (s,\text{cdo}(t,u,c)) \text{ if c is in CC.}$$

We can now view a capability system as a state machine, with state space S x Capt, input space (U x C) and output space Out. We can therefore extend the system transition function in the classical way to strings of inputs, by defining

$$\text{csdo}: S \times Capt \times (U \times C)^* \to S \times Capt$$

with the equations

$$\text{csdo}(s,t,NIL) = (s,t)$$

and

$$\text{csdo}(s,t,w.(u,c)) = \text{csdo}(\text{csdo}(s,t,w),u,c)$$

for s in S, t in Capt, u in U, c in C, and w in (U x C)*, where NIL denotes the empty string and the "dot" denotes string concatenation. We will use the notation

$$[[\ w\ ]] = \text{csdo}(t0,c0,w)$$

for the "denotation" or effect of the input string w on states, starting from the initial state of the whole system.

We now recall one more concept from classical automaton theory, and then specialize it to the case at hand. A state s of a state machine M is reachable iff there is some w in C* such that [[w]] = s. But notice that the set of reachable states of a capability system will not in general be a Cartesian product of a set of (ordinary) states and a set of capability tables. However, the set of reachable states will always be the state set of a reachable submachine, the reachable capability subsystem of the given capability system.

3 Security Policies

Whereas the previous section presented our approach to modelling secure systems, this section presents our approach to defining security policies. The purpose of a security policy is to declare which information flows are not to be permitted. Giving such a security policy can be reduced to giving a set of noninterference assertions. Each noninterference assertion says that

> **what one group of users does using a certain ability has no effect on what some other group of users sees.**

Section 3.1 first discusses static security policies, which give a set of noninterference assertions that are to hold independently of any changes in the capability table. After that, Section 3.1 considers dynamic security policies, whose noninterference assetions may take account of the state of the capability table. It should be noted that one may wish to impose static security policies (such as MLS) even for systems where capabilities may be passed, and that it is possible to consider both static and dynamic policies for a single system.

3.1 Static Policies

Let us begin with some auxiliary notation. Given a state machine M, let w be an element of (U x C)* and let u be a user. We define $[[\ w\ ]]_u$ to be the "output to u after doing w on M," i.e.,

$$[[\ w\ ]]_u = \text{out}([[\ w\ ]],u)\ .$$

It is the "denotation of w from the point of view of user u." Note that M may have additional structure as a capability system. In that case, the state space has the form S x Capt, the set C of commands is a disjoint union CC U SC of state and capability table changing commands, and the state transition function is csdo. Thus, all the general definitions given below also apply to capability systems. However, it is simpler to state them for a arbitrary state transition system.

Our second auxiliary notation has to do with the selection of subsequences of user-command pairs.

Definition 3: Let $G \subseteq U$ be a "group" of users, let $A \subseteq C$ be an ability, and let w be in (U x C)*. Then we let $P_G(w)$ denote the subsequence of w obtained by eliminating those pairs (u,c) with u in G. Similarly, we let $P_A(w)$ denote the subsequence of w obtained by eliminating those pairs (u,c) with c in A. Combining these two, we let $P_{G,A}(w)$ denote the subsequence of w obtained by eliminating the pairs (u,c) with u in G and c in A. []

For example, if G = {u,v} and A = {c1,c2}, then

$$P_{G,A}(\ (u',c1)(u,c3)(u,c2)(v',c1)\ ) =$$
$$= (u',c1)(u,c3)(v',c1)\ ,$$

where u',v' are other users and c3 is another command.

Now we are ready for the basic technical concept of noninterference, which we give in three different forms (we will later generalize all this to conditional noninterference).

Definition 4: Given a state machine M and sets G, G' of users, we say that G does not interfere with (or is non-interfering with) G', written G :| G', iff for all w in (U x C)* and all u in G',

$$[[\ w\ ]]_u = [[\ p_G(w)\ ]]_u\ .$$

Similarly, given an ability A and a group G of users, we say that A does not interfere with G, written A :| G, iff for all w in (U x C)* and u in G,

$$[[\ w\ ]]_u = [[\ p_A(w)\ ]]_u\ .$$

More generally, users in G with ability A do not interfere with users in G' written A,G :| G' iff for all w in (U x C)* and u in G,

$$[[\ w\ ]]_u = [[\ p_{G,A}(w)\ ]]_u\ .$$

[]

Although we have stated these definitions for state machines, they apply immediately to capability systems because we have shown that capability systems are also state machines. In the following we will generally be applying these definitions to the case of dynamically changing capabilities as covered by capability systems.

Example 1: It follows from the above definition that if A :| {u}, then the commands in A have no effect whatsoever on the output seen by u. For example, if A is the ability to create, write, modify or delete a file F, then "A noninterfering with u" means that the information read from F by u cannot be changed by any commands in A. In particular, if F did not originally exist, then u will always be told that F doesn't exist, independently of what commands in A may actually have done. []

Notice that neither the "noninterfering with" relation, nor its complement relation, "potentially interfering with," are assumed to be transitive. This means that we are able to consider the fully general case of intransitive information flow.

This concept of noninterfering is similar to concepts given in [Feiertag, Levitt & Robinson 77] and [Feiertag 80], in that both consider isolating the effects of sequences of commands; however, the concept captured here is more general, as we can treat arbitrary policies (not just MLS) and our capability machine approach allows us to treat dynamically changing capability tables.

We are now ready for the major concept of the paper.

Definition 5: A security policy is a set of noninterference assertions. []

This definition seems to be fully general for stating restrictions, or the lack of restrictions, on information flow, once the generalization to conditional noninterference has been given. In particular, it can handle downgrading, multi-user/multi-key access, compartmentation, and channel control; it can easily and naturally describe situations where passing information across boundaries is permitted, for example when information is declassified, or when it is controlled by discretionary access.

Example 2: Multilevel Security. Let L be a simply ordered set of "security levels," such as {unclassified, secret, top-secret}, with ordering relation $\leq$, as in [Bell & LaPadula 74]. Assume that we are given a function Level: U -> L. We now need a bit more notation. For x in L, let

$$U[-\infty,x] = \{\ u \text{ in } U \mid \text{Level}(u) \leq x\ \}$$

$$U[x,+\infty] = \{\ u \text{ in } U \mid \text{Level}(u) \geq x\ \}$$

where we do not assume that there actually are minimum and maximum elements ($-\infty$ and $+\infty$) in L. Then a capability system M is multilevel secure with respect to the given level function iff, for all x > x' in L, the noninterference assertion

$$U[x,+\infty]\ :|\ U[-\infty,x']$$

holds for M.

The function Level might be stored in the capability component of the system, and it is not assumed to be necessarily constant. Or the capability component might be much more complex, containing information on who is allowed to change classifications, or to pass capabilities to change classifications. This will be discussed below.

Let us call a group G of users invisible (relative to other users) iff G :| -G ("invisible" seems appropriate for these users because they can see without being seen). It is very easy to express MLS using this notion:

for every level x, $U[x,+\infty]$ is invisible.

MLS easily generalizes to a partially ordered set L of security levels, just by replacing "x > x'" by "x is not $\leq$ x'." This generalization permits MLS to encompass compartments: if c is a level such that for all x in L, neither x > c nor c > x, then c is completely isolated from all other levels, and the users of level c are a "compartment." It is also easy to express this generalization in terms of invisibility:

for every level x, $U - U[-\infty,x]$ is invisible,

where "-" denotes set difference. []

Example 3: Security Officer. Suppose that the set A consists of exactly those commands that can change the capability table, and suppose that our desired policy is that there is just one designated user, "seco," the Security Officer, whose use of those commands will have any effect. This policy is simply expressed by the single noninterference assertion

$$A,-\{\text{seco}\}\ :|\ U\ .$$

[]

Example 4: Isolation. A group G of users is isolated (or is separated, or is a compartment) iff G :| -G and -G :| G. The isolation policy says that nothing can flow in or out of the group G. A system is completely isolated if every user is isolated. This is a policy that [Rushby 81a] wishes to prove for his separation kernels. []

Example 5: Channel Control. A very general notion of channel is just a set of commands, i.e., an ability $A \subseteq O$. Let G and G' be groups of users. Then the policy saying that G and G' can communicate only through the channel A is

-A,G :| G' and -A,G' :| G .

(This is our formalization of a concept in [Rushby 81b]). []

Example 6: Information Flow. In order to show how information flow can be included under our more general concept of noninterference, let us suppose that a, b, c and d are "processes," and that A1, A2 and A3 are "channels," such that a, b, c, and d can communicate only as indicated in the figure below, in which information can flow only to the right

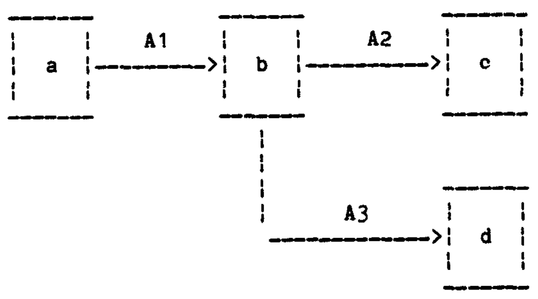

Figure 2: An Information Flow Diagram

The constraints implied by this picture include the following noninterference assertions:

```
{b,c,d} :| {a}    -A1,{a} :| {b,c,d}
  {c,d} :| {b}    -A2,{b} :| {c}
    {c} :| {d}    -A3,{b} :| {d}
    {d} :| {c}
```

We will show in a future paper how to formally produce a complete set of of noninterference assertions from an information flow diagram like that of Figure 2. For the moment, let us note that the left group of assertions are purely topological, while the right group encode information about the specific abilities mentioned in the graph. []

3.2 Dynamic Policies

Dynamic policies, i.e., policies which depend on the state of the capability component of a system, can be handled using conditional noninterference assertions. For static policies, no conditioning of the noninterference assertions is needed, because the assertions are supposed to hold always. But for dynamic policies, whether or not a given user u can interfere with another user v, by using an operation c may vary with time. The conditions that we attach to noninterference assertions will be predicates defined over the sequences of operations used to reach the current state. The basic definition follows. It gives only the general case "mixed" noninterference, with both users and operations involved. Further motivation is provided with Examples 7 and 8 below.

Definition 6: Let G and G' be sets of users, let A be a set of commands, and let P be a predicate defined over (U x C)*. Then G using A is noninterfering with G' under condition P, written

G,A :| G' if P

iff for all u' in G' and for all w in (U x C)*,

$$[[\ w\ ]]_{u'} = [[\ p(w)\ ]]_{u'}\ ,$$

where p is defined by

$$p(\lambda) = \lambda\ ,$$

where λ is the empty string, and

$$p(o1...on) = o'1...o'n$$

where

$o'_i = \lambda$ if $P(o'_1 ... o'_{i-1})$ and $o_i = (u,a)$

with u in G and a in A ,

and

$o'_i = o_i$ otherwise .

The reason for giving such a complex definition of the projection function p is to take account of the fact that there may be some subsequences in a long sequence of commands that are noninterfering, while others may be interfering.

The examples below give several conditional noninterference assertions and show how they apply to command sequences.

Example 7: Discretionary Access. In this example, we assume the existence of a function CHECK(w,u,c), which looks at the capability table in state [[w]] to see whether or not u is authorized to do command c; it returns true if he is, and false if not. We can then regard CHECK(u,c) as a predicate on command sequences w. One general policy that we wish to enforce for all users u and all commands c is

(*) {u},{c} :| U if not CHECK(u,c) ,

where U is the set of all users. This just says that u cannot interfere using c if he does not have the capability to use c in that state.

Now let us consider the case where for each user u and command c, there is another command, denoted pass(u,c), which says to pass to u the ability to do c; of course, the user issuing this command may or may not be authorized to do so. We need a bit more notation. If w is a command sequence of the form w'.o, let previous(w) = w' and let last(w) = o. Then let us write CHECK(previous,u,c) for the predicate CHECK(previous(w),u,c) of w. Now the policy that we wish to enforce regarding use of the pass command is

$$(**)\ \{u\},\{c\} :| U \ \underline{if}[\text{not CHECK(previous,u,c)}] \text{ and } [\text{if CHECK(previous,u}',\text{pass(u,c)) then not last} = (u',\text{pass}(u,c))]$$

This says that u using c cannot interfere if in the previous state he didn't have the capability to use c, unless some user u' having the capability in the previous state to pass u the ability to use c, in fact did so.

The corresponding assertion for the revocation operation, which we shall denote unpass(u,c), is

$$\{u\},\{c\} :| U \ \underline{if}\text{CHECK(previous,u}',\text{unpass(u,c)) and last} = (u',\text{unpass}(u,c))$$

This says that u can't interfere using c if in the state previous to trying to use c, some user u' who had the capability to revoke u's capability to use c in fact did so.

Let us see how Definition 6 applies to a particular sequence of commands and the assertion (**). Let us suppose that user u' has capability to use the command pass(u,c), i.e., that for all strings w of commands,

$$\text{CHECK}(w,u',\text{pass}(u,c)) = T$$

and

$$\text{CHECK}(w.(u',\text{pass}(u,c)),u,c) = T\ .$$

Further suppose that u does not initially have capability for c, and that (u'',d) is noninterfering with the capability of u to use c, i.e.,

$$\text{CHECK(NIL},u,c) = T$$

and for each w,

$$\text{CHECK}(w.(u'',d),u,c) = \text{CHECK}(w,u,c)\ .$$

Then, for example, (**) says that for any user v,

$$[[\ (u,c)(u',\text{pass}(u,c))(u'',d)(u,c)\ ]]_v =$$

$$[[\ (u',\text{pass}(u,c))(u'',d)(u,c)\ ]]_v\ ,$$

i.e., that the first instance of (u,c) in the command sequence has no observable effect. []

Example 8: Bailout Function. We now express a dynamic policy having one function that differs from standard MLS; the purpose of this example is of course to illustrate the use of our conditional noninterference assertion formalism, rather to argue for or against any particular security policy. In this policy, there is a command B that when executed changes the level of the user to the lowest security level in an irrevocable manner. Thus, we assume a simply ordered set L of security levels, with bottom level "Unc" say, and with a function Level from users to levels. Then the policy is stated with the following noninterference assertions for each user u

$$\{u\} :| U[-\infty,\text{Level}(u))\ \underline{if}\ \text{CHECK}(u,B)\ ,$$

where $U[-\infty, x) = \{\ u' \mid \text{Level}(u') < x\ \}$, and

$$U(\text{Unc},+\infty] :| \{u\}\ \underline{if}\ \text{not CHECK}(u,B)\ ,$$

where $U(\text{Unc},+\infty] = \{\ u' \mid \text{Level}(u') > \text{Unc}\ \}$, assuming the following about the function CHECK, for every sequence w of commands, user u, and operation o,

$$\text{CHECK}(w.o,u,B) = \text{CHECK}(w,u,B)\ \underline{if}\ \text{not } o=(u,B)$$

$$\text{CHECK}(w.(u,B),u,B) = \text{false}\ ,$$

and

$$\text{CHECK}(\lambda,u,B) = \text{true}\ .$$

3.3 Security Policy Definition Language

Our definition of security policy suggests that a specialized requirement language can be given for stating security policies, such that the basic statements of this language are noninterference assertions. Notice that each such assertion can be seen as an infinite set of equalities of sequences of commands; this set can also be expressed as one equation having one second order quantifier over command sequences. We believe that the very simple form that these assertions have will permit us to construct a special purpose verification tool, rather like the MLS tool of [Feiertag 80], but applicable to any policy that can be formulated in the language. Two steps in the operation of such a tool are to eliminate all explicit induction, and then to translate into the simpler logical formalism of a mechanical theorem prover. We hope to discuss this in future work.

4 Less Abstract Models of Capability Systems

Specific systems, such as PSOS, can be modelled in our framework, by instantiating the various sets and functions involved in Definition 2. This can be done in many different specification formalisms, including the state machine approach of SPECIAL [Levitt, Robinson & Silverberg 79]; the first order logic decision procedure approach of STP [Shostak, Schwartz & Melliar-Smith 81]; the inductive definition over lists and numbers approach of [Boyer & Moore 80]; the parameterized procedure approach of CLEAR [Burstall & Goguen 77] or in a more usable form, ORDINARY [Goguen & Burstall 80]; and the executable abstract data type/rewrite rule approach of OBJ [Goguen & Tardo 79].

Another possibility is the formalism surveyed in [Snyder 81], one of the few which considers the passing of capabilities. We observe that this is a graph grammar formalism in the sense of [Ehrig, Kreowski, Rosen & Winkowski 78][6]. Unfortunately, it seems to be difficult to verify policies in this formalism.

5 Verification of Security Policies

How can we verify that a security policy P is satisfied by a capability system M? For example, how can we verify that MLS has been correctly implemented in some version of PSOS? From a general point of view, this is a matter of verifying that the noninterference assertions in the policy are true of some particular abstract machine. This can be done by induction over the commands of the system. We have some hope that a great deal can be accomplished with purely syntactic checking of specifications for the operating system, as with the Feiertag MLS tool [Feiertag 80], because of the simple form of the assertions occurring in the security policy definitions. More detailed discussion of this will be the subject of a future report.

But, you may ask, how can we verify that some given code running on a given machine actually satisfies some policy? One approach is just to verify the policy for a high level specification, and then to verify that a lower level machine correctly implements the specification, perhaps through a sequence of intermediate abstract machines; this is essentially the approach of HDM [Levitt, Robinson & Silverberg 79].

6 Summary

This paper has described an approach to security which is based on:

1. Modelling the information processing system by an automaton of the form that we call a capability system.
2. Defining security policies as sets of noninterference assertions.
3. Verifying that a given system satisfies a given policy.

[6]This observation seems to be new, and might be of some use for the further development of the formalism described in [Snyder 81], because of the existence of a considerable body of results on graph grammars, including pumping lemmas, decidability results, and normal forms.

REFERENCES

[Bell & LaPadula 74]
Bell, D. E. and LaPadula, L. J.
Secure Computer Systems: Mathematical Foundations and Model.
Technical Report, MITRE Corporation, 1974.
Bedford, MA.

[Boyer & Moore 80]
Boyer, R. and Moore, J. S.
A Computational Logic.
Academic Press, 1980.

[Burstall & Goguen 77]
Burstall, R. M. and Goguen, J. A.
Putting Theories together to Make Specifications.
Proceedings, Fifth International Joint Conference on Artificial Intelligence 5:1045-1058, 1977.

[DeMillo, Lipton & Perlis 77]
DeMillo, R. A., Lipton, R. J. and Perlis, A. J.
Social Processes and Proofs of Theorems and Programs.
In Proceedings, Fourth ACM Symposium on Principles of Programming Languages, pages 206-214. ACM, 1977.

[Denning 82] Denning, D.
Cryptography and Data Security.
Addison-Wesley, 1982.

[Dijkstra 78] Dijkstra, E. W.
On a Political Pamphlet from the Middle Ages.
Software Engineering Notes 3(2):14-16, 1978.

[Ehrig, Kreowski, Rosen & Winkowski 78]
Ehrig, E., Kreowski, H.-J., Rosen, B. K. and Winkowski, J.
Deriving Structures from Structures.
In Proceedings, Mathematical Foundations of Computer Science, . Springer-Verlag, Zakopane, Poland, 1978.
Also appeared as technical report RC7046 from IBM Watson Research Center, Computer Sciences Dept.

[Feiertag 80] Feiertag, R.
A Technique for Proving Specifications are Multilevel Secure.
Technical Report, SRI Report CSL-109, 1980.

[Feiertag, Levitt & Robinson 77]
Feiertag, R. J., Levitt, K. N. and Robinson, L.
Proving Multilevel Security of a System Design.
In Proceedings, Sixth ACM Symposium on Operating Systems Principles, pages 57-65. , 1977.

[Goguen & Burstall 80]
Goguen, J. A. and Burstall, R. M.
An Ordinary Design.
Technical Report, SRI International, 1980.
Draft report.

[Goguen & Tardo 79]
Goguen, J. A. and Tardo, J.
An Introduction to OBJ: A Language for Writing and Testing Software Specifications.
In Specification of Reliable Software, pages 170-189. , 1979.

[Goguen, Thatcher & Wagner 78]
Goguen, J. A., Thatcher, J. W. and Wagner, E.
An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.
In R. Yeh (editor), Current Trends in Programming Methodology, . Prentice-Hall, 1978.
Originally published as IBM T. J. Watson Research Center Report RC 6487, October 1976.

[Guttag 75] Guttag, J. V.
The Specification and Application to Programming of Abstract Data Types.
PhD thesis, University of Toronto, 1975.
Computer Science Department, Report CSRG-59.

[Hartmanis & Stearns 66]
Hartmanis, J. and Stearns, R. E.
Algebraic Structure Theory of Sequential Machines.
Prentice-Hall, 1966.

[Lampson 74] Lampson, B. W.
Protection.
Operating Systems Review 8(1):18-24, 1974.

[Landwehr 81] Landwehr, C. E.
A Survey of Formal Models for Computer Security.
Technical Report, Naval Research Laboratory, 1981.
NRL Report 8489.

[Levitt, Robinson & Silverberg 79]
Levitt, K., Robinson, L. and Silverberg, B.
The HDM Handbook.
Technical Report, SRI, International, Computer Science Lab, 1979.
Volumes I, II, III.

[Neumann, Boyer, Feiertag, Levitt & Robinson 80]
Neumann, P. G., Boyer, R.S., Feiertag, R.S., Levitt, K. N. and Robinson, R.S.
A Provably Secure Operating System: The System, its Applications, and Proofs. 2nd. ed.
Technical Report, SRI International, Computer Science Laboratory, 1980.

[Popek & Farber 78]
Popek, G. J. and Farber, David A.
A Model for Verification of Data Security in Operating Systems.
Communications of the Association for Computing Machinery 21(9):737-749, 1978.

[Rushby 81a] Rushby, J. M.
Proof of Separability: A Verification Technique for a Class of Security Kernels.
Technical Report, Computing Laboratory, University of Newcastle upon Tyne, 1981.

[Rushby 81b] Rushby, J. M.
Verification of Secure Systems.
Technical Report, University of Newcastle upon Tyne, 1981.

[Shostak, Schwartz & Melliar-Smith 81]
Shostak, Schwartz & Melliar-Smith.
STP: A Mechanized Logic for Specification and Verification.
Technical Report, Computer Science Lab, SRI International, 1981.

[Snyder 81] Snyder, L.
Formal Models of Capability-Based Protection Systems.
IEEE Transactions on Computers C-30(3):172-181, 1981.

[Turn 81] Turn, Rein.
Advances in Computer System Security.
Artech House, 1981.