USE OF A FORMALISM FOR MODELING THE
PROTECTION ASPECTS OF OPERATING SYSTEMS*

BY

F.B. Schneider
E.A. Akkoyunlu
State University of New York at Stony Brook
Department of Computer Science
Stony Brook, New York 11794

TECHNICAL REPORT #74
JULY 1977

* This work was partially support by NSF Grant MCS 76 04828
1.0 Introduction

A large body of literature exists that is relevant to the problem of protection. Much work has been done in the design and implementation of secure systems (SPI73, SAL74, WUL74, DEN66, HOA74, to name a few). Except for ad hoc verification techniques and exhaustive penetration studies (ATT76) no general methods exist for the proof of the protection properties of real systems.

The protection formalism as defined by Harrison, Ruzzo, and Ullman (HAR75) provides a simple way to describe the protection properties of a system. Harrison et al show the formalism to be so powerful that system safety, a property that secure systems exhibit, is recursively undecidable for the general protection system, though decidable for a subclass of protection systems called mono-operational.

While the work of Harrison et al is essentially uninterpreted, we present an interpretation of the formalism in terms of operating system concepts. Modifications are made to the formalism to make it useful for modeling the protection properties of real operating systems. The formalism is discussed in light of hierarchical system design, and the types of systems it can model. Finally, a definition of security that corresponds to the intuitive notion of security is made in terms of the formalism.
2.0 The Protection Formalism

Harrison et al (HAR75) define a protection system, \( \sigma \), in terms of a 6-tuple:

\[
\sigma = (R, S_0, O_0, C, I, K)
\]

where:

- \( R \) is a finite set of generic rights. Examples of generic rights are "read", "write", "execute", etc.
- \( S_0 \) is a finite set of initial subjects. Subjects correspond to users, processes, or domains.
- \( O_0 \) is a finite set of initial objects. An object is the smallest protectable entity. These include both real objects (memory, tape drives, etc.) and virtual objects (semaphores, files, etc.). Note that \( S_0 \subseteq O_0 \).
- \( C \) is a finite set of commands. Each command roughly corresponds to a subroutine.

A command invocation is denoted by:

\[
c_i(x_1, \ldots, x_n)
\]

where \( c_i \in C \) and \( x_1, \ldots, x_n \in O \).

The \( x_i \)'s are actual parameters and denote elements in the object set. When a command is invoked, the actual values (those in the parenthesized list following the command name) are substituted for the corresponding formal parameters as defined in the first line of the command. This is essentially an ALGOL call by name subroutine call.

\( I \) is a finite set of interpretations of the commands in \( C \). Each command has exactly one interpretation. An interpretation consists of a
a finite sequence of primitive operations. A primitive operation may be one of the following:

- enter \( r \) into \((s, o)\)
- delete \( r \) from \((s, o)\)
- create subject \( s \)
- delete subject \( s \)
- create object \( o \)
- delete object \( o \)

The primitives are executed in the order in which they appear in the interpretation.

\( K \) is a finite set of conditionals for the commands in \( C \). Each command, \( c_i \), has a finite set of conditionals associated with it. The primitives (i.e., interpretation) of a particular command are performed if and only if all the conditionals associated with the command are true. A conditional is of the form:

\[ r_i \in (s_j, o_k) \quad \text{where} \quad r_i \in R, \quad s_j \in S, \quad o_k \in O. \]

The conditional is said to be true only if subject \( s_j \) has generic right \( r_i \) to object \( o_k \).

An actual right is a triplet: \( (r_i, s_j, o_k) \). Actual rights refer to particular instances of generic rights. Thus,

\[ r_i \in (s_j, o_k) \text{ true } \leftrightarrow (r_i, s_j, o_k) \]

Examples of commands are as follows:

\[ \text{SAMPLE}(s_i, o_j) \]

create subject \( s_i \)
enter read into \((s_i, o_j)\)

is a command with two primitives in its interpretation, and no conditionals. An invocation of the command is:

\[ \text{SIMPLE}(s_r) \]

\( \text{own} \in (s_r, s_r) \)
delete subject \( s_r \)
Only if the actual right \((\text{own}, s_r, s_r)\) exists for some \(s_r \in S\), will the conditional(s) be satisfied, and the interpretation performed. (In this case \(s_r\) is deleted.)

A configuration, \(Q\), which completely defines the state of the system with respect to protection, consists of a triplet:

\[
Q = (S, O, P) \quad \text{where} \quad S = \text{the current subjects set} \\notag \\
O = \text{the current objects set} \\notag \\
P = \text{a protection matrix} \notag
\]

The sets \(S, O, P\) are all finite. As before \(S \subseteq O\). \(P\), the protection matrix is defined in terms of the conditionals that may be satisfied by it.

\[
P = \{(r_i, s_j, o_k) \mid r_i \in R, s_j \in S, o_k \in O, r_i \in (s_j, o_k)\} \notag
\]

Clearly, \(P(s, o) \subseteq R\). The protection matrix, \(P\), corresponds to the access matrix of Lampson. (LAM71)

A command invocation is said to leak a right, \(r_i\), to an object, \(o_k\), if in the interpretation of the command there is a primitive of the form:

\[
\text{enter } r_i \text{ into } (s_j, o_k). \notag
\]

It is not necessary that the command leave this right in the configuration when it is finished executing. For example, the following command leaks right LEKRITE to \(o_k\), even though its execution may not alter the initial configuration:

\[
\text{DAN\_SCHORR}(s_i, o_k) \notag \\
\text{enter LEKRITE into } (s_i, o_k) \notag \\
\text{delete LEKRITE from } (s_i, o_k) \notag
\]

At times we may refer to a command leaking a right \(r_i\). In that case we mean that an invocation of that command will leak \(r_i\) to some object \(o_k\).
A configuration, Q, is said to be unsafe for a particular right, r₁, with respect to an object, oₖ, if:

1. there exists a command, cᵢ, that leaks r₁ to oₖ
2. the conditionals of cᵢ are satisfied by P in Q

A configuration is said to be unsafe for a particular right, r₁, if it is unsafe for that right with respect to any object, oₖ. A configuration is said to be unsafe if it is unsafe for any of a particular set of rights that are of interest. A system, S, is said to be unsafe if there exists a sequence of commands which, when executed, result in an unsafe configuration.

A configuration, Q₁, is said to be derived from a previous configuration, Q₀, if and only if:

\[ Q₀ = (S₀, O₀, P₀) \]
\[ Q₁ = (S₁, O₁, P₁) \]

\[ Q₀ \rightarrow Q₁ \]

where \( \pi₁ \) is a primitive operation.

and

\( \pi₁ \) is "enter r into (s,o)" then:

\[ S₁ = S₀ \quad O₁ = O₀ \]
\[ P₁(sᵢ,oⱼ) = P₀(sᵢ,oⱼ) \quad \text{for all } sᵢ \in S, oⱼ \in O \]
\[ P₁(s,o) = P₀(s,o) \cup \{r\} \]

\( \pi₁ \) is "delete r from (s,o)" then:

\[ S₁ = S₀ \quad O₁ = O₀ \]
\[ P₁(sᵢ,oⱼ) = P₀(sᵢ,oⱼ) \quad \text{for all } sᵢ \in S, sᵢ \notin S, oⱼ \in O, oⱼ \neq 0 \]
\[ P₁(s,o) = P₀(s,o) \setminus \{r\} \]
\[ \pi_1 \] is "create subject \ s" (s \neq 0) then:

\[
S_1 = S_0 \cup \{s\} \quad 0_1 = 0_0 \cup \{s\}
\]
\[
P_1(s_i, o_j) = P_0(s_i, o_j) \quad \text{for all } s_i \in S, o_j \in 0
\]

\[ \pi_1 \] is "Create object \ o" (o \neq 0) then:

\[
S_1 = S_0 \quad 0_1 = 0_0 \cup \{o\}
\]
\[
P_1(s_i, o_j) = P_0(s_i, o_j) \quad \text{for all } s_i \in S, o_j \in 0
\]
\[
P(s, o) = \emptyset \quad \text{for all } s \in S
\]

\[ \pi_1 \] is "destroy subject \ s" then:

\[
S_1 = S_0 - \{s\} \quad 0_1 = 0_0 - \{s\}
\]
\[
P_1(s_i, o_j) = P_0(s_i, o_j) \quad \text{for all } s_i \in S, o_j \in 0
\]

\[ \pi_1 \] is "destroy object \ o" then:

\[
S_1 = S_0 \quad 0_1 = 0_0 - \{o\}
\]
\[
P_1(s_i, o_j) = P_0(s_i, o_j) \quad \text{for all } s_i \in S, o_j \in 0
\]

In general, commands will be used to derive configurations from other configurations. In that case, \( Q_1 \) is derived from \( Q_0 \) by command, \( c_1 \), if:

\[
Q_0 = (S_0, 0_0, P_0)
\]
\[
Q_1 = (S_1, 0_1, P_1)
\]
\[
c_1 \in C
\]
\[
K_1 = \{(r_i, s_j, o_k) \mid r_i \in R, s_j \in S_0, o_k \in 0_0, r_i \in (s_j, o_k) \}
\]

a conditional for command, \( c_1 \)
Then if for each $K \in K_1$, an actual right exists in $P$ (i.e., all the conditionals are true) and the interpretation of $c_1$ is:

$$\pi_1, \pi_2, \ldots, \pi_n$$

and:

$$(S, O, P) \xrightarrow{\pi_1} (S', O', P') \xrightarrow{\pi_2} (S'', O'', P'') \xrightarrow{\pi_3} \cdots \xrightarrow{\pi_n} (S_1, O_1, P_1)$$

The transitive closure of command derivatives will be denoted, as in formal language theory (HOP69) by an asterisk:

$$Q_0 \xrightarrow{*} Q'$$

There exists $c_1, c_2, \ldots, c_r \in C$, and

$$Q_0 \xrightarrow{c_1} Q_1 \xrightarrow{c_2} Q_2 \xrightarrow{c_3} \cdots \xrightarrow{c_r} Q'$$
3.0 Previous Results

In addition to defining a formalism, Harrison et al (HAR75) prove two interesting results about the power of the formalism. They show that, in general, it is undecidable whether a protection system is safe for a given generic right, and that there exists a subclass of protection systems, called mono-operational systems, for which the safety question is decidable.

As is pointed out in their paper (HAR75), safety is too strict a requirement to impose on a system. Consider the case where some subject that owns or controls an object wishes to grant an access right to that object. Thus, the system would be unsafe, by definition. Harrison et al are concerned with whether granting a right to some subject may eventually cause other subjects to receive that right without "authorization." Thus, as a safety test, all reliable subjects are deleted from the access matrix, P, and then an attempt to determine if an unsafe configuration is derivable is made. This modified form of safety also is, in general, undecidable, though for mono-operational systems it is decidable.

A mono-operational system is one in which the interpretation of each command contains at most one primitive operation. Harrison et al show that there is an algorithm which, given a mono-operational system, a generic right $r$, and an initial configuration $Q_0$, determines if $Q_0$ is safe for $r$ (HAR75). The reader is referred to the original paper for a formal proof.

As it is pointed out by Harrison et al, mono-operational systems are of very limited practical value in modeling real operating systems. It also shown that whether a given configuration of a given general protection system is safe for a given generic right is recursively undecidable. Their proof
consists of defining a correspondence between an arbitrary Turing machine and a protection system leaks a right precisely when the Turing machine halts.

Lipton and Snyder (LIP76) show that there exists a linear-time algorithm for deciding subject security in a graph-theory based formalism. Their formalism can be easily translated to the one which is discussed in this paper. The resultant system is not mono-operational, but somewhat more complex. In addition, the system modeled is one that has been discussed in the literature as a viable operating system protection mechanism (DEN72).
4.0 New Results

The work of Harrison et al (HAR75) deals with the formalism as an uninterpreted system. In actually applying the formalism to the modeling of operating systems certain problems arise that require additions to the formalism. In addition, precise interpretation of the elements of the formalism in operating system terms must be made. The modifications and commentary to the formalism that follow are based on our attempts to model actual operating systems.

4.1 Implementation of Protection Systems

Many recent operating systems have been designed and implemented using the level structured approach of Dijkstra (DIJ68). In this light, it is reasonable to consider the design of an operating system which is both poly-operational, (i.e., a general protection system), and mono-operational, depending on which level the system is viewed from. Would the safety of such a system be decidable?

Consider an operating system which is poly-operational at some level, say k. Assume that some lower level, say k-1, is a "protection command interpreter." That is, it simulates the execution of the interpretation of a poly-operational command by issuing a sequence of mono-operational commands. Some even lower level, say k-2, interprets the mono-operational commands, interrogates the conditionals, and actually executes the primitive operations on the access matrix, P. Thus, it appears that there is a way to circumvent the undecidability result previously discussed by using a mono-operational simulation of a general protection system. This, however, is not the case in general. To show this, we first define the implementation of a protection system by another protection system.
Def.: System A implements system B if every derivable configuration in system B is also derivable in system A from the same initial configuration.

Notice that the implementor (i.e., system A) may have more generic rights that there are in the implemented system. It is only stipulated that there exist a derivation of a configuration in the implementor for every derivable configuration in the implemented system that is identical to it (i.e., does not contain these additional generic rights).

The following lemma will be required:

Lemma: Let \( Q = \{ Q \mid Q \rightarrow^* Q' \} \), for some system \( S \)
\[ Q_m = \{ Q \mid Q \rightarrow^* Q', \text{ for some system } S' \} \]

Then, \( Q_m \leq Q_R \Rightarrow (S \text{ safe} \Rightarrow S' \text{ safe}) \).

Proof: Assume \( S \) is safe. Then, for all \( Q_i \in Q_R \), \( Q_i \) is a safe configuration. Each \( Q_j \in Q_m \) is also an element of \( Q_R \), hence a safe configuration. Thus, by definition, \( S' \) is safe.

Q.E.D.

We now state:

Theorem: a) There exists a uniform effective procedure to obtain a mono-operational implementaiton of a general protection system.

b) This mono-operational implementation is such that if it is safe, then the system being simulated is safe also.

Proof: a) Let \( \sigma = (R, S_0, Q_0, C, I, K) \) be a general protection system. Then, for all \( c_i \in C \), without loss of generality, assume \( c_i \) is of the form:
\[
\text{cond}_1 \\
\text{cond}_2 \\
\vdots \\
\text{cond}_r \\
\pi_1 \\
\pi_2 \\
\vdots \\
\pi_n
\]

where \(\text{cond}_1\) are conditionals and \(\pi_i\) are primitive operations

A number of mono-operational commands will be constructed to "implement" \(c_i\). First, if all the conditionals that correspond to \(c_i\) are true, a new subject will be created and given the "OK" right to itself. Then for each primitive operation in the interpretation of \(c_i\), a command with that primitive alone is constructed with checks if the "OK" right exists. This indicates that the conditionals for \(c_i\) were satisfied prior to execution. Finally, the new subject is erased.

Create \(n+3\) commands as follows:

\[
\begin{align*}
c_{i,0} & : \quad \begin{array}{l}
\text{create subject } i
\end{array} \\
c_{i,1} & : \quad \begin{array}{l}
\text{cond}_1 \\
\text{cond}_2 \\
\vdots \\
\text{cond}_r \\
\text{enter } \text{OK into } (i,i)
\end{array} \\
c_{i,2} & : \quad \begin{array}{l}
\text{OK } \epsilon (i,i) \\
\pi_1
\end{array} \\
c_{i,3} & : \quad \begin{array}{l}
\text{OK } \epsilon (i,i) \\
\pi_2
\end{array} \\
\vdots \\
c_{i,n+1} & : \quad \begin{array}{l}
\text{OK } \epsilon (i,i) \\
\pi_n
\end{array} \\
c_{i,n+2} & : \quad \begin{array}{l}
\text{OK } \epsilon (i,i) \\
delete \text{subject } i
\end{array}
\end{align*}
\]
Clearly, if $\sigma' = (R \cup \{OK\}, S_0, O_0, C_m, I_m, K_m)$ where $C$, $I$, and $K$ are generated as above, the resultant system is monoperational.

Consider $c_1 \in C$, and $c_{i,0} c_{i,1} c_{i,2} \cdots c_{i,n+2} \in C_m$ as defined previously. If $Q_0 \xrightarrow{c_{i}} Q_1$ in $\sigma$, then in $\sigma'$

$\sigma' \xrightarrow{c_{i,0}} Q_0 \xrightarrow{c_{i,1}} Q_0' \xrightarrow{c_{i,2}} \cdots \xrightarrow{c_{i,n+2}} Q_1$. Thus, $\sigma'$ implements $\sigma$ in the sense defined.

Q.E.D.

b) Consider $\{Q_0', Q_0'', Q_0''', \ldots \}$ as shown above. These configurations are not derivable in $\sigma$. They are, however, as are all derivable configurations of $\sigma$, derivable in $\sigma'$. Hence, by previous lemma, $\sigma'$ safe $\implies \sigma$ safe.

Q.E.D.

Thus, in cases where a mono-operational system implements a general protection system, if the resultant mono-operational system is not safe, it does not necessarily imply that the general system is unsafe. In fact, the unsafe configurations may only be derivable in the mono-operational system because of the extra "freedom" provided with respect to the order in which mono-operational commands may be executed. So, a general protection system is decidable only if the mono-operational simulation is safe. Otherwise, the safety of a general protection system is undecidable.

4.2 The RUNNING Pseudo-Right

Since access to objects in the Harrison et al (HAR75) model is granted based on the subject that executed the command, and the access matrix, $P$,
it would be expected that there should be a way to ascertain the name of the executing subject. In most operating systems this would correspond to the primitive that returns the name of the executing process.

Consider the following command:

\[ \text{GRANT}(S_{a}, S_{b}, 0_{a}) \]
\[ \text{own} \in (S_{a}, 0_{a}) \]
\[ \text{grant read to } (S_{b}, 0_{a}) \]

If some subject, say S10, invokes the command as \text{GRANT}(S10, S37, FILE1), then if S10 owns FILE1, S37 will receive read access to that file. Alternately, if S37 invokes the command as \text{GRANT}(S10, S37, FILE1), then S37 will also receive read access to FILE1 (if S10 owns FILE1). This is illustrated as follows:

<table>
<thead>
<tr>
<th>FILE1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S10</td>
<td>own</td>
<td>GRANT(S10, S37, FILE1)</td>
</tr>
<tr>
<td>S37</td>
<td></td>
<td>either S10 or S37 invokes</td>
</tr>
</tbody>
</table>

In one case, S10 gave the read right to S37; in the other case it was taken. This apparent violation of security would not be detected by the safety test proposed by Harrison et al. In their safety test, S10 would be deemed a "reliable" subject with respect to FILE1 and the read right. Consequently, S10 would be deleted from the access matrix, P, and no derivation would exist by which an arbitrary subject (like S37) could receive read access to FILE1. This potential "leak" would not be discovered as illustrated below:
Thus, it would be desirable to be able to identify the subject that executed a command, while performing the interpretation of that command. For this purpose, the "running" pseudo-right is added to the formalism. The "running" generic right will be granted to the subject executing a command, in the $P(\text{subject}, \text{subject})$ set, and deleted at the end of execution. We call the "running" right a pseudo-right because we will not permit any command to explicitly grant or delete this right. Thus, the GRANT command could be re-written as follows:

$$\text{GRANT}(S_a, S_b, O_a)$$

running $\in (S_a, S_a)$

own $\in (S_a, O_a)$

grant read to $(S_b, O_a)$

Now only the owner of $O_a$ may execute GRANT to give away the read right.

The introduction of the "running" right does, however, restrict us to modeling uni-processor systems. This is due to the tacit assumption that the subject executing a command be the only one with the running right. If more than one subject has the running right, then a problem similar to the original one occurs. Consider the case where $S10$ and $S37$ are both executing in a di-processor system. Then, as before, either $S10$ or $S37$ can invoke $\text{GRANT}(S10, S37, \text{FILE1})$, and $S37$ will receive the read
right to FILE1, if S10 has the own right for it. A more general solution (based on process IDs) will have to be developed for multi-processor systems.

4.3 Adding Interpretation to the Formalism

In this section, the elements of the formalism will be discussed in terms of the operating system concepts they represent or model. In addition, the semantic consequences of the primitive operations will be noted.

4.3.1 Subjects

From the definition of the formalism, a subject is clearly the smallest entity for which access may be authorized. Depending on the object for which access is to be regulated, a process may not satisfy this definition. In some computer systems, the set of valid instructions varies according to the state that the processor is in. In others, memory access depends on an address translation scheme, and hence on some base-bounds registers or some address translation tables. This information will be referred to as the protection state.

Def.: protection state – that information which is required to determine access rights to objects in the system, and changes as a consequence of process execution, but not command execution.

Thus, the protect key and problem/supervisor state bits in the 360 architecture are examples of components of a protection state. Protection state is essentially the process state in an operating system, less the information not relevant to access authorization. Notice that protection
state is defined in terms of objects, and not solely in terms of process state.

A domain, \( d_i \), the smallest entity for which access may be authorized, and hence the "general" element of the subject set, is now defined. **Def.**: domain = a (process, protection state) pair.

Thus, \( S \), the subject set, is the cartesian product of the process names and protection states. As another example, consider the MULTICS operating system. In that case, the protection state would include the current ring of execution and the address of the current instruction (segment number). (SAL74)

4.3.2 Objects

Operating systems, in general, besides managing real resources (objects) also create virtual resources (objects). Which, if any, of these objects should appear in the object set, \( O \)? At any time, certain objects are known about by the various processes, while others are not. As an example of this, consider a hierarchical operating system implementation. At each level, there is a set of objects which are known, and hence accessible to that level. In many cases, a level may shield the existence of an object it manipulates by creating a virtual object which is known about at higher levels. A memory management level is an example of this. The memory manager deals with real memory (objects), and may create a set of virtual address spaces (VSSs). At higher levels, real memory is no longer a valid object, while virtual memory is.
Consequently, a hierarchical system, or any operating system where the existence of all objects is not known to all domains, must be modeled by a series of protection systems. Each protection system would correspond to a level where a given set of objects is known. To prove such a system safe, all levels (hence, each of the protection systems) must be proved safe, starting from the lowest level and continuing to the highest. At each level, the objects and domains will undoubtedly be different, as the various layers change the "virtual machine" that an executing process sees. For example, in many systems, at the highest level, a user is not aware that there exists a virtual memory; hence, this would not be part of the protection state at that level.

4.3.3 Primitives

Primitive operations were discussed in terms of their effect on the access matrix, P. Though the delete right and enter right primitive operations have no semantic consequences in terms of operating system concepts, except altering the access matrix, this is not true of the create subject/object and delete subject/object primitive operations. Both instances of the create command add named entities to the system. Thus, control blocks must be created and tables up-dated as part of the implementation of these primitives. Alternately, the delete primitive operations require that besides deleting control blocks and freeing the resource (it is not expected that "deleting a disk drive" will result in its absolute destruction as implied by the command name), any information that may have been saved about, or on the object must be deleted. Thus, when a file is destroyed, the contents of the file must be "zeroed."
Otherwise, a subject may subsequently create an object which coincides physically with the actual storage space that was once a file, and ascertain what information was stored in that file.

Thus, though the uninterpreted formalism provides a straightforward way to verify the safety of a system, it is necessary that the system being modeled conform to the interpreted formalism as well as the uninterpreted formalism.

4.4 Security and Safety

Thus far, the discussion of the formalism has been oriented toward the safety of a system. Safety deals with the ability of a domain (subject) to obtain a right to an object. As has been shown, no useful system will be safe, as it is expected that rights to objects will be given by one subject to another in the course of normal operation. Security, on the other hand, is concerned with the ability of a domain to access an object, or information, for which it does not have authorization, and the ability to obtain authorization illicitly. The definition of security in terms of the formalism is difficult because a way must be found to indicate which subjects "have authorization" to obtain a right or access an object or information. In the following discussion, an attempt is made to define a secure system in terms of the formalism.

Earlier, it was shown that primitive operations have certain semantic consequences that are not expressed in the formalism. The same is true for both rights and commands in the formalism as will be shown.

Consider any command that "leaks" a right, r. This command will only leak r if the conditionals that correspond to the command are true. The
logical combination of these actual rights will be called a privilege right for r. In commands that leak a right (say r) that have only one conditional associated with them, the privilege right for r will be a generic right. For example, in

\[
\text{LEAK}(S_a, S_b, O_a)
\]

\[
\text{own} \in (S_a, O_a)
\]

\[
\text{grant} \text{ } r \text{ } \text{to} \text{ } (S_b, O_a)
\]

"own" is a privilege right for the generic right r. Alternately, privilege rights need not be generic rights as can be seen in the following:

\[
\text{LEAK}(S_a, S_b, O_a)
\]

\[
\text{own} \in (S_a, O_a)
\]

\[
\text{control} \in (S_a, O_a)
\]

\[
\text{grant} \text{ } r \text{ } \text{to} \text{ } (S_b, O_a)
\]

In this case, \((\text{own}, S_a, O_a) \land (\text{control}, S_a, O_a)\) is the privilege right for r. It will not matter that the privilege right does not belong to the generic rights set.

In some cases, commands may exist that correspond to actual operating system primitives, but which have in interpretation in terms of the formalism's primitive operations. For example, a "read" primitive does not alter the access matrix, but nonetheless should be modeled. Consequently, the interpretation of such a read command will not have any primitive operation, but is rich in operations system semantic consequences. In the case of a read operation an information transfer occurs. We will be concerned with information transfer from an object by a domain. The former will be referred to as a put consequence, the latter as a get.
consequence. As was the case for privilege rights, the conditional section of a command that has a get or put consequence will be referred to as a get or put right (depending on the consequence, of course).

Privilege rights, get rights, and put rights provide the necessary characterization to define security in terms of the formalism. Clearly, generic rights are concerned with the access of objects by domains. Get rights are concerned with the access of information by domains. Privilege rights are concerned with a domain obtaining other rights. This is almost sufficient to formalize the original definition of security. The "transitivity" of information flow must also be considered. Consider a situation (the confinement problem [LAM73]) where some I/O domain has get access to all files, and put access to a buffer pool. In addition, user domains have get access to the memory area where the buffers are. Then, theoretically, it is possible for a user domain to read the contents of any file as it is transferred to the buffer area which the user domain has access to. Thus, certain configurations of get and put rights may result in an information flow that is not intended. Now, to formalize the notion that in a secure system no subject may access an object or information for which it does not have authorization, nor may authorization be obtained illicitly, the following is proposed.

Def.: **secure system** - A protection system is secure if the following are true:

1. Any commands that can leak a right, r, must interrogate the privilege rights for r, and also insure that these privilege rights are with respect to the domain executing the command (by use of the "running" right).
(2) The system is safe for each right with respect to each object when the following modified access matrices are employed: For the generic right under consideration, delete all privilege rights for that right in all entries of the matrix.

(3) The following command, when applied to any derivable configuration of the system, will not alter the configuration.

\[ S_a \in S, S_b \in S, O_a \in O, O_b \in O \]

\[ r_p \text{ a put right, } r_g \text{ a get right} \]

\[ \text{TEST}(S_a, S_b, O_a, O_b) \]

\[ r_g \in (S_a, O_b) \]

\[ r_p \in (S_a, O_a) \]

\[ r_g \in (S_b, O_b) \]

grant \( r_g \) to \((S_b, O_a)\)

Conditions (1) and (2) correspond to the access of objects, and obtaining rights aspects of the definition. Condition (3) corresponds to the information flow restriction.

4.5 Systems that are Representable in the Formalism

In the other portions of this paper it is assumed that many operating systems can in fact be modeled by the formalism. The following result shows this.

Def.: centralized system - a system in which there exists an entity which always has a correct global view of the system. (This would not be the case in many network computing systems.)
Theorem: The protection aspects of a centralized uni-processor system can be modeled by the formalism as defined, provided:

a) an uncircumventable system primitive which provides the name of the domain executing exists

b) only the objects accessible to the domains are in the object set, $O$

c) the only domains that may execute are in the subject set, $S$

d) the only way for a domain to alter the configuration of the system (hence, the access matrix, $P$) is via a command

e) the only way for a domain to interact with an object is via a command

Proof: At any time, the system can be described by the set of objects accessible, the domains that may access these objects, and the types of access these domains have to these objects. This is just the information contained in a configuration. The set of primitive operations provide a set of operations on the protection matrix, $P$, to derive any configuration from any other configuration. The protection aspects of the execution of a system are therefore just a sequence of configurations. This, however, is a sequence of commands. Naturally it is assumed that the semantic consequences of the execution of a primitive operation are reflected in the system being modeled. Q.E.D.
5.0 Conclusion

The formalism has been shown to be a flexible tool potentially useful for modeling operating systems. Investigations into what characterizes a safe mono-operational system should be made. The development of proof methodologies or techniques that an operating system designer could employ to verify the security of a particular system would also be useful.

Acknowledgments

The authors would like to thank S. Kamin, K. Ekanadham, and M. McCormick for useful discussions in the preparation of this paper.
References


HOP69  John E. Hopcroft and Jeffrey D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, 1969.


LIP76  R.J. Lipton and L. Snyder, "A Linear True Algorithm for Deciding Subject Security", Yale University Department of Computer Science Research Report #72, (also appears in Proc 17th Annual Symposium on Foundations of Computer Science, pp. 33-41).

