ON PROTECTION IN OPERATING SYSTEMS*

by

Michael A. Harrison
Walter L. Ruzzo
University of California at Berkeley

and

Jeffrey D. Ullman+
Princeton University

Abstract

A model of protection mechanisms in computing systems is presented and its appropriateness is demonstrated. The "safety" problem for protection systems under our model is to determine in a given situation whether a subject can acquire a particular right to an object. In restricted cases, one can show that this problem is decidable, i. e., there is an algorithm to determine whether a system in a particular configuration is safe. In general, and under surprisingly weak assumptions, one cannot decide if a situation is safe. Various implications of this fact are discussed.

Keywords and Phrases: Protection, Protection system, Operating system, Decidability, Turing machine.

CR Category Numbers: 4.30, 4.31, 5.24

I Introduction

One of the key aspects of modern computing systems is the ability to allow many users to share the same facilities. These facilities may be memory, processors, data bases or software, such as compilers or subroutines. When diverse users share common items, one is naturally concerned with protecting various objects from damage or from missapropriation by unauthorized users. In recent years, a great deal of attention has been focussed on the problem. Papers [3-5,7-12,14] are but a sample of the work that has been done. In particular, Saltzer [14] has formulated a hierarchy of protection levels, and current systems are only halfway up the hierarchy.

The schemes which have been proposed to achieve these levels are quite diverse, involving a mixture of hardware and software. When such diversity exists, it is often fruitful to abstract the essential features of such systems and to create a formal model of protection systems.

The first attempts at modeling protection systems, such as [3,5,9] were really abstract formulations of the reference monitors and protected objects of particular protection systems. It was thus impossible to ask questions along the lines of "which protection system best suits my needs." A true model of protection systems was created in [7], which could express a variety of policies and which contained the "models" of [3,5,9] as special cases. However, no attempt to prove results was made in [7], and the model was not completely formalized.

On the other hand, there have been models in which attempts were made to prove results [1,2,12]. In [1], which is similar to [7] but independent of it, theorems are proven. However, the model is informal and it uses programs whose semantics (particularly side effects, traps, etc.) are not specified formally.

*Research sponsored by the National Science Foundation Grants GJ-474 and GJ-43332.
+Work done while on leave at the University of California, Berkeley.

In the present paper, we shall offer a model of protection systems. The model will be sufficiently formal that one can rigorously prove meaningful theorems. Only the protection aspects of the system will be considered, so it will not be necessary to deal with the semantics of programs or with general models of computation. Our model is similar to that of [5,9], where it was argued that the model is capable of describing most protection systems currently in use.

Section II describes the motivation for looking at decidability issues in protection systems. Section III presents the formal model with examples. In Section IV we introduce the question of safety in protection systems. Basically, safety means that an unreliable subject cannot pass a right to someone who did not already have it. We then consider a restricted family of protection systems and show that safety is decidable for these systems. In Section V we obtain a surprising result, that there is no algorithm which can decide the safety question for arbitrary protection systems. The proof uses simple ideas, so it can be extended directly to more elaborate protection models.

II Significance of the Results

To see what the significance for the operating system designer of our results might be, let us consider an analogy with the known fact that ambiguity of a context free grammar is undecidable (see [6], e.g.). The implication of the latter undecidability result is that proving a particular grammar unambiguous might be hard, although it is possible to write down a particular grammar, for ALGOL say, and prove that it is unambiguous. In analogy, one might desire to show that in a particular protection system a particular situation is safe, in the sense that a certain right cannot be given to an unreliable subject. Our result on general undecidability does not rule out the possibility that one could decide safety for a particular situation in a particular protection system. Indeed, we have not ruled out the possibility of giving algorithms to decide safety for all possible situations of a given protection system, or even for whole classes of systems. In fact we provide an algorithm of this nature.

In analogy with context free grammars, once again, if we grant that it is desirable to be able to tell whether a grammar is ambiguous, then it makes sense to look for algorithms that decide the question for large and useful classes of grammars, even though we can never find one algorithm to work for all grammars. A good example of such an algorithm is the LR(k) test (see [6], e.g.). There, one tests a grammar for LR(k)-ness, and if found to possess the property, we know the grammar is unambiguous. If it is not LR(k) for a fixed k, it still may be unambiguous, but we are not sure. It is quite fortunate that most programming languages have LR(k) grammars, so we can prove their grammars unambiguous.

It would be nice if we could provide for protection systems an algorithm which decided safety for a wide class of systems, especially if it included all or most of the systems that people seriously contemplate. Unfortunately, our one result along these lines involves a class of systems called "mono-operational," which are not terribly realistic. Our attempts to extend these results have not succeeded, and the problem of giving a decision algorithm for a class of protection systems as useful as the LR(k) class is to grammar theory appears very hard.

III A Formal Model of Protection Systems

We are about to introduce a formal protection system model. Because protection is but one small part of a modern computing system, our model will be quite primitive. No general purpose computation is included, as we are only concerned with protection - that is, who has what access to which objects.

Definition: A protection system consists of the following parts.

(1) A finite set of generic rights $R = r_1,\ldots,r_n$.

(2) A finite set of initial subjects S_o and a finite set of initial objects O_o, where $S_o \subseteq O_o$.

(3) A finite set of commands C of the form $\alpha(X_1,\ldots,X_k)$ where α is a name and $X_1,\ldots,X_k$ are formal parameters which denote objects.

(4) An interpretation I for commands, so that I maps C into sequences of primitive operations. The primitive operations are:

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

where r is a generic right, s is a subject name, and o is an object name.

(5) Conditions for commands. A condition **c** is a map from the set of commands into a finite set of rights. A right is a triple (r,s,o), where $r \in R$, and s and o are formal parameters which are subjects and objects, respectively. Note the distinction between a "right" and a "generic right." A "right" is the privilege that a given subject s has to exercise generic right r on object o.

Before proceeding, let us interpret the parts of a protection system. In practice, typical subjects might be processes [3] and typical objects (other than those objects which are subjects) might be files. A common generic right is read, i.e., a process has the right to read a certain file. The commands mentioned in item (3) above are meant to be formal procedures. Since we wish to model only protection aspects of a sytem, we wish to avoid embedding into the model unrestricted computing power. The command procedures are therefore interpreted (item 4, above) to be sequences of specific, primitive operations. To understand the meanings of the primitive operations, imagine a matrix whose rows represent subjects and whose columns represent objects. The (s,o) entry is the set of rights which subject s may exercise over object o. For example, the first primitive operation:

enter r into (s,o)

enters r into the matrix at positon (s,o), if it is not already there.

It is best to think of item (5), the "conditions," in the following way. Associate with every command a finite set of conditions (or predicates) of the form:

$$r_1 \in (s_1,o_1)$$
$$\vdots$$
$$r_m \in (s_m,o_m)$$

The intention is that if each of the m conditions is satisfied, then the command may be executed. If one or more fail, then the command is not executed.

Next we define a configuration, or instantaneous description of a protection system.

Definition: A configuration of a protection system is a triple (S,O,P), where S is the set of current subjects, O is the set of current objects, S $\subseteq$ O, and P is an access matrix, with a row for every subject in S and a column for every object in O. P[s,o] is a subset of R, the generic rights. P[s,o] gives the rights to object o possessed by subject s. The access matrix can be pictured as in Figure 1. Note that row s of the matrix in Figure 1 is like a "capability list" [3] for subject s, while column o is similar to an "access list" for object o.

Example 1: Let us consider what is perhaps the simplest discipline under which sharing is possible. We assume that each subject is a process and that the objects other than subjects are files. Each file is owned by a process, and we shall model this notion by saying that the owner of a file has generic right own to that file. The other generic rights are read, write and execute, although the exact nature of the generic rights other than own is unimportant here. The actions affecting the access matrix which processes may perform are:

(1) Create a new file. The process creating the file has ownership of that file. We represent this action by command CREATE(s,o). The interpretation of CREATE is the sequence of primitive operations:

create object o
enter own into (s,o)

There are no conditions for the CREATE command.

(2) The owner of a file may confer any right to that file, other than own, on any subject (including the owner himself). We thus have three commands $\text{CONFER}_r(s_1,s_2,o)$ for r = read, write or execute, with interpretations:

enter r into (s_2,o)

each with the single condition:

own $\in (s_1,o)$

(3) Similarly, we have three commands $\text{REMOVE}_r(s_1,s_2,o)$, which remove r from the (s_2,o) entry of the access matrix, where r

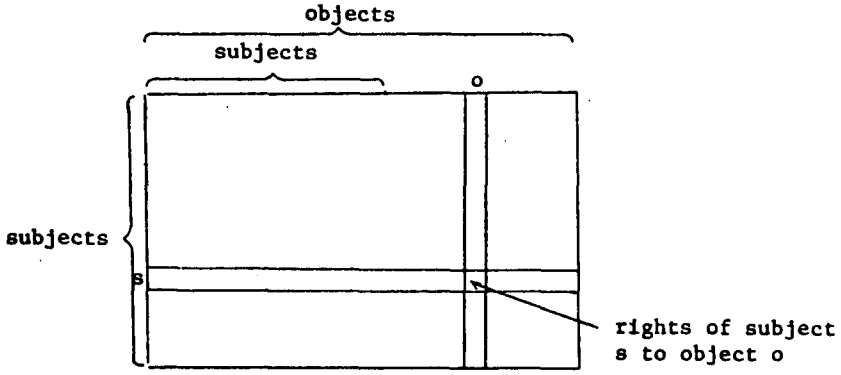

Fig. 1 Access Matrix

= read, write or execute. The interpretation of $REMOVE_r$ is:

delete r from (s_1,o)

and the conditions are:

own $\in$ (s_1,o)
$r \in (s_2,o)$*

This completes the specification of most of the example protection system. We shall expand this example after learning how such systems "compute."

To formally describe the effects of commands, we must give rules for changing the state of the access matrix.

Definition: Let (S,O,P) and (S′,O′,P′) be configurations of a protection system, and let c be a primitive operation. We say that:

$$(S,O,P) \Rightarrow_c (S',O',P')$$

[read (S,O,P) yields (S′,O′,P′) under c] if either:

(1) c = enter r into (s,o) and S = S′, O = O′, $s \in S$, $o \in O$, $P'[s_1,o_1] = P[s_1,o_1]$ if $(s_1,o_1) \neq (s,o)$ and $P'[s,o] = P[s,o] \cup \{r\}$, or

(2) c = delete r from (s,o) and S = S′, O = O′, $s \in S$, $o \in O$, $P'[s_1,o_1] = P[s_1,o_1]$ if $(s_1,o_1) \neq (s,o)$ and $P'[s,o] = P[s,o] - \{r\}$.

(3) c = create subject s′, s′ is a new symbol not in O, $S' = S \cup \{s'\}$, $O' = O \cup \{s'\}$, $P'[s,o] = P[s,o]$ for all (s,o) in SxO, $P'[s',o] = \emptyset$** for all $o \in O'$, and $P[s,s'] = \emptyset$ for all $s \in S'$.

(4) c = create object o′, o′ is a new symbol not in O, S′ = S, $O' = O \cup \{o'\}$, $P'[s,o] = P[s,o]$ for all (s,o) in SxO and $P'[s,o'] = \emptyset$ for all $s \in S$.

(5) c = destroy subject s′, where $s' \in S$, $S' = S - \{s'\}$, $O' = O - \{s'\}$, and $P'[s,o] = P[s,o]$ for all $(s,o) \in S' \times O'$.

(6) c = destroy object o′, where $o' \in O-S$, S′ = S, $O' = O - \{o'\}$, and $P'[s,o] = P[s,o]$ for all $(s,o) \in S' \times O'$.

The quantification in the previous definition is quite important. For example, a primitive operation

enter r into (s,o)

requires that s be the name of a subject which now exists, and similarly for o. If these conditions are not satisfied, then the command is not executed. The primitive operation

create subject s′

requires that s′ is not a current object name. Thus there can never be duplicate names of objects.

Next we see haw a protection system executes a command.

Definition: Let (S,O,P) and (S′,O′,P′) be configurations of a protection system. Let C be a command. Then we say

$$(S,O,P) \vdash_C (S',O',P')$$

if

(1) for each $(r,s,o) \in C(C)$, we have $r \in P[s,o]$ and

(2) if we let $I(C) = c_1 \ldots c_m$, where the c_i's are primitive operations, then there exists $m \geq 0$, and there exist configurations (S_i,O_i,P_i) such that:

(a) $(S,O,P) = (S_0,O_0,P_0)$

(b) $(S_{i-1},O_{i-1},P_{i-1}) \Rightarrow_{c_i} (S_i,O_i,P_i)$ for $0 \leq i \leq m$, and

(c) $(S_m,O_m,P_m) = (S',O',P')$

We write $(S,O,P) \vdash (S',O',P')$ if there is some command C such that $(S,O,P) \vdash_C (S',O',P')$. It is convenient to write $(S,O,P) \vdash^* (S',O',P')$, where $\vdash^*$ is the reflexive and transitive closure of $\vdash$, that is, $\vdash^*$ represents zero or more applications of $\vdash$.

There are a number of points involved in our use of parameters which should be emphasized. Note that every command (except the empty one) has parameters. Each command is given in terms of formal parameters. At execution time, the formal parameters are replaced by actual parameters which are object names. Although the same symbols are often used in this exposition for formal and actual parameters, this should not cause confusion. The "type checking" involved in determining that a command may be executed takes place with respect to actual parameters. For example, consider the command $C(s_1,s_2,o)$, which consists of:

enter r_1 into (s_1,s_1)
destroy subject s_1
enter r_2 into (s_2,o)

There can never be a pair of configurations (S,O,P) and S′,O′,P′) such that

$$(S,O,P) \vdash (S',O',P')$$

*This condition need not be present, since delete r from (s_2,o) will have no effect if r is not there.

**$\emptyset$ denotes the empty set.

under the command $C(s_1,s_1,o)$, since the third primitive operation enter r_2 into (s_1,s_1) will occur at a point where subject s_1 does not exist.

Note that we do not place an a priori bound on the number of subjects and objects which may exist in these "computations." Such bounds lead to a trivialization of this theory as it does with models for digital computers.

Example 2: Let us consider the protection system whose commands were outlined in Example 1. Suppose initially there are two processes Sam and Joe, and no files created. Suppose that neither process has any rights to itself or to the other process (there is nothing in the model that prohibits a process from having rights to itself). The initial access matrix is:

	Sam	Joe
Sam	ø	ø
Joe	ø	ø

Now, Sam creates two files named Code and Data, and gives Joe the right to execute Code and Read Data. The sequence of commands whereby this takes place is:

CREATE(Sam,Code)
CREATE(Sam,Data)
$CONFER_{execute}$(Sam,Joe,Code)
$CONFER_{read}$(Sam,Joe,Data)

To see the effect of these commands on configurations, note that the configuration (S,O,P) can be represented by drawing P, and labeling its rows by elements of S and its columns by elements of O, as we have done for the initial configuration. The first command, CREATE(Sam,Code) may certainly be executed in the initial configuration, since CREATE has no conditions. Its interpretation consists of two primitive operations, create object Code and enter own into (Sam,Code). Then, using the => notation, we may show the effect of the two primitive operations as

	Sam	Joe
Sam	ø	ø
Joe	ø	ø

==========> create object Code

	Sam	Joe	Code
Sam	ø	ø	ø
Joe	ø	ø	ø

==========> enter own into (Sam,Code)

	Sam	Joe	Code
Sam	ø	ø	{own}
Joe	ø	ø	ø

Thus, using the |-- notation for complete commands we can say that:

	Sam	Joe
Sam	ø	ø
Joe	ø	ø

|---------- CREATE(Sam,Code)

	Sam	Joe	Code
Sam	ø	ø	{own}
Joe	ø	ø	ø

The effect on the initial configuration of the four commands listed above is:

	Sam	Joe
Sam	ø	ø
Joe	ø	ø

|--

	Sam	Joe	Code
Sam	ø	ø	{own}
Joe	ø	ø	ø

|--

	Sam	Joe	Code	Data
Sam	ø	ø	{own}	{own}
Joe	ø	ø	ø	ø

|--

	Sam	Joe	Code	Data
Sam	ø	ø	{own}	{own}
Joe	ø	ø	{execute}	ø

|--

	Sam	Joe	Code	Data
Sam	ø	ø	{own}	{own}
Joe	ø	ø	{execute}	{read}

We may thus say:

	Sam	Joe
Sam	ø	ø
Joe	ø	ø

|--*

	Sam	Joe	Code	Data
Sam	ø	ø	{own}	{own}
Joe	ø	ø	{execute}	{read}

It should be clear that in protection systems, the order in which commands are executed is not prescribed in advance. The nondeterminacy is important in modeling real systems in which accesses and changes in privileges occur in an unpredictable order.

We should note that our naming convention for new objects has not been fully specified. For example, in a primitive operation like create subject s, we did not specify where s comes from. There are obvious ways to formalize the generation of new names, and we shall continue to ignore this point, because the formalization lends no new insight.

It is our contention that the model we have presented has sufficient generality that it allows one to specify almost all of the protection schemes that have been proposed. Cf. [5] for many examples of this flexibility. It is of interest to note that it is immaterial whether hardware or software is used to implement the primitive operations of our model. The important issue is what one can say about systems we are able to model. We shall develop the theory of protection systems using our model in the next two sections. We close this section with two additional examples of the power of our model to reflect common protection ideas.

Example 3: A mode of access called "indirect" is discussed in [5]. Subject s_1 may access object o indirectly if there is some subject s_2 with that access right to o, and s_1 has the "indirect" right to s_2. Formally, we could model an indirect read by postulating the generic rights read and iread, and a command IREAD(s_1,s_2,o) with interpretation:

enter read into (s_1,o)
delete read from (s_1,o)

and with conditions:
read $\in$ (s_2,o)
iread $\in$ (s_1,s_2)

It should be noted that the command in Example 3 has both multiple conditions and an interpretation consisting of more than one primitive operation, the first example we have seen of such a situation. In fact, since the REMOVE commands of Example 1 did not really need two conditions, we have our first example where multiple conditions are needed at all.

We should also point out that the interpretation of IREAD in Example 3 should not be taken to be null, even though the interpretation actually chosen has no effect on the access matrix. The reason for this will become clearer when we discuss the safety issue in the next section. Intuitively, we want the interpretation of IREAD to show that s_1 temporarily has the read right to o, even though it must give up the right.

Example 4: The UNIX operating system [13] uses a simple protection mechanism, where each file has one owner. The owner may specify his own privileges (read, write and execute) and the privileges of all other users, as a group.* Thus the system makes no distinction between subjects except for the owner-nonowner distinction.

This situation cannot be modeled in our formalism as easily as could the situations of the previous examples. It is clear that the generic right own is important, and that the rights of a subject s to a file o which s owns could be placed in the (s,o) entry of the access matrix. However, when we create a file o, it is not possible in our formalism to express a command such as "give all subjects the right to read o," since there is no a priori bound on the number of subjects.

The solution we propose actually reflects the software implementation of protection in UNIX quite well. We associate the rights to a file o with the (o,o) entry in the access matrix. This decision means that files must be treated as special kinds of subjects, but there is no logical reason why we cannot do so. Then a subject s can read (or write or execute) a file o if either:

(1) own is in (s,o), i.e., s owns o, and the entry "owner can read" is in (o,o), or

(2) the entry "anyone can read" is in (o,o).

Now we see one more problem. The conditions under which a read may occur is not the logical conjunction of rights, but rather the disjunction of two such conjuncts, namely

(1) own $\in$ P[s,o] and oread $\in$ P[o,o]
or
(2) aread $\in$ P[o,o]

where oread stands for "owner may read," and aread for "anyone may read." For simplicity we did not allow disjunctions in conditions. However, we can simulate a condition consisting of several lists of rights, where all rights in some one list must be satisfied in order for execution to be permissible. We simply use several commands whose interpretations are identi-

*We ignore the role of the "superuser" in the following discussion.

cal. That is, for each list of rights there will be one command with that list as its condition. Thus any set of commands with the more general, disjunctive kind of condition is equivalent to one in which all conditions are as we defined them originally. We shall, in this example, use commands with two lists of rights as a condition.

We can now model these aspects of UNIX protection as follows. Since write and execute are handled exactly as read, we shall treat only read. The set of generic rights is thus own, oread, aread and read. The first three of these have already been explained. read is symbolic only, and it will be entered temporarily into (s,o) by a READ command, representing the fact that s can actually read o. read will never appear in the access matrix between commands and in fact is not reflected directly in the protection mechanism of UNIX. The list of commands is shown in Figure 2.

IV Safety

We shall now consider one important family of questions that could be asked about a protection system, those concerning safety. When we say a specific protection system is "safe," we undoubtedly mean that access to files without the concurrence of the owner is impossible. However, protection mechanisms are often used in such a way that the owner gives away certain rights to his objects. Example 4 illustrates this phenomenon. In that sense, no protection system is "safe," so we must consider a weaker condition that says, in effect, that a particular system enables one to keep one's own objects "under control."

Since we cannot expect that a given system will be safe in the strictest sense, we suggest that the minumum tolerable situation is that the user should be able to tell whether what he is about to do (give away a right, presumably) can lead to the further leakage of that right to truly unauthorized subjects. As we shall see, there are protection systems under our model for which even that property is too much to expect. That is, it is in general undecidable whether, given an initial access matrix, there is some sequence of commands in which a particular generic right is entered at some place in the matrix where it did not exist before.

This question, whether a generic right can be "leaked" is itself insufficiently general. For example, suppose subject s plans to give subject s' generic right r to object o. The natural question is whether the current access matrix, with r entered into (s',o), is such that generic right r could subsequently be entered somewhere new. To avoid a trivial "unsafe" answer because s himself can confer generic right r, we should in most circumstances delete s itself from the matrix. It might also make sense to delete from the matrix any other "reliable" subjects who could grant r, but whom s "trusts" will not do so. It is only by using the hypothetical safety test in this manner, with "reliable" subjects deleted, that the ability to test whether a right can be leaked has a useful meaning in terms of whether it is safe to grant a right to a subject

Another common notion of the term "safety" is that one be assured it is impossible to leak right r to a particular object o_1. We can use our more general definition of safety to simulate this one. To test whether in some situation right r to object o_1 can be leaked, create two new generic rights, r' and r". Put r' in (o_1,o_1), but do nothing yet with r". Then add a new command DUMMY(s,o) with conditions:

$$r' \leftarrow (o,o)$$
$$r \leftarrow (s,o)$$

and interpretation:

	command	interpretation	condition
(1)	CREATEFILE(s_1,s_2)	create subject s_2 enter own into (s_2,s_2)	
(2)	OREAD(s_1,s_2)	enter oread into (s_2,s_2)	own $\leftarrow$ (s_1,s_2)
(3)	AREAD(s_1,s_2)	enter aread into (s_2,s_2)	own $\leftarrow$ (s_1,s_2)
(4)	READ(s_1,s_2)	enter read into (s_1,s_2) delete read from (s_1,s_2)	either own $\leftarrow$ (s_1,s_2) oread $\leftarrow$ (s_2,s_2) or aread $\leftarrow$ (s_2,s_2)

Figure 2. UNIX type Protection Mechanism

enter r" into (o,o)

Then, since there is only one instance of generic right r', o must be o_1 in command DUMMY. Thus, leaking r" to anybody is equivalent to leaking generic right r to object o_1 specifically.

We shall now give a formal definiton of the safety question for protection systems.

Definition: Given a protection system, we say command $\alpha(X_1,\ldots,X_n)$ potentially leaks (or just leaks) generic right r if its interpretation has a primitive operation of the form enter r into (s,o) for some s and o.

Note that we detect a leak of r even if command α behaves like the "neat burglar," who removes r from (s,o) at the conclusion of the burglary. Commands IREAD in Example 3 and READ in Example 4 are typical of commands which leak a right and then immediately remove it, leaving no "trace." In fact, we defined these commands purposely to symbolize the fact that the right was temporarily granted, even though no permanent change in the access matrix occurred.

Definition: Given a particular protection system and right r, we say that initial configuration (S_0,O_0,P_0) is safe for r in this system if there does not exist configuration (S,O,P) such that:

$$(S_0,O_0,P_0) \vdash\!\!-^* (S,O,P)$$

and there is a command $\alpha(X_1,\ldots,X_n)$ whose conditions are satisfied in (S,O,P), and for some actual parameters, α potentially leaks r via primitive operation enter r into (s,o), for some subject s and object o which exist at the execution time of the enter command, and for which r is not in P[s,o].

Example 5: Let us reconsider the simple example of a command $C(s_1,s_2,o)$ which immediately preceeds Example 2. Suppose C were the only command in the system. Then the system can be considered to leak r_1, but this protection system is safe for r_1 if the initial configuration has $O = \emptyset$.

There is a special case for which we can show it is decidable whether a given right is potentially leaked in any given initial configuration. The result we are about to present is not significant in itself, since most interesting examples do not meet the condition. However, it is suggestive of stronger results that might be proved - results which would enable the designer of a protection system to be sure that an algorithm to decide safety, in the sense we have used the term here, existed for his system.

Definition: A protection system is mono-operational if each command's interpretation is a single primitive operation.

Example 4, based on UNIX, is not mono-operational because the interpretation of CREATE has length two.

It turns out that we can decide safety in mono-operational protection systems, and this fact is our first theorem.

Theorem 1: There is an algorithm which given a mono-operational protection system, a generic right r and an initial configuration (S_0,O_0,P_0) determines whether or not (S_0,O_0,P_0) is safe for r in this protection system.

Proof: Suppose we are given a sequence of configurations

$$Q_0 \vdash\!\!-_{C_1} Q_1 \vdash\!\!-_{C_2} \cdots \vdash\!\!-_{C_m} Q_m$$

where $Q_0 = (S_0,O_0,P_0)$, and command $\alpha(X_1,\ldots,X_n)$, which leaks r, has its conditions met in Q_m. Assume without loss of generality that this sequence is as short as possible. Then we may make the following inferences.

Observation 1: For all i, $1 \le i \le m$, C_i is not destroy subject or destroy object.*

Proof: Suppose C_j is the first destroy command, which we may suppose to be destroy subject s. Then we may remove command C_j from the sequence. If s is ever created again, change the created object to a new name other than s. Since objects are created without any entries in their row and column of the access matrix, should the original sequence leak r, then the new one will as well. But the new sequence is shorter than the old, a contradiction.

Observation 2: There do not exist i and j, with $i < j$, such that both C_i and C_j are create subject commands, or both are create object commands.

Proof: Suppose for specificity that C_i is create subject s_1 and C_j is create subject s_2. We create a new shorter sequence of configurations leaking r, by identifying s_1 and s_2, letting s_1 play the role of both. That is, we delete C_j from the sequence of commands, and change all subsequent references to s_2 into references to s_1. By Observation 1, we may assume neither s_1 nor s_2 are ever destroyed, so all references to these subjects refer to the ones created at the i^{th} and j^{th} steps.

*Since the system is mono-operational, we can identify the command by the type of primitive operation and its interpretation.

Formally, we create a new sequence of configurations:

$$Q_o \vdash_{C_1} \cdots \vdash_{C_{j-1}} Q_{j-1} \vdash_{C_{j+1}}$$

$$R_{j+1} \vdash_{C_{j+2}} \cdots \vdash_{C_m} R_m$$

where configuration R_k is obtained from Q_k by merging s_1 and s_2. That is, if T_k and P_k are the access matrices in R_k and Q_k, respectively, then $T_k[s,o]=$

(1) $P_k[s,o]$ if $s \neq s_1$ and $o \neq s_1$

(2) $P_k[s_1,o] \cup P_k[s_2,o]$ if $s = s_1$ and $o \neq s_1$.

(3) $P_k[s,s_1] \cup P_k[s,s_2]$ if $s \neq s_1$ and $o = s_1$.

(4) $P_k[s_1,s_2] \cup P_k[s_2,s_1] \cup P_k[s_1,s_1] \cup P_k[s_2,s_2]$ if $s = o = s_1$.

Thus assuming two create subject commands allows us to create a shorter sequence showing Q_o not to be safe for r, contrary to the hypothesis that $Q_o \vdash Q_1 \vdash \ldots \vdash Q_m$ was the shortest such sequence. A similar argument rules out two create object commands.

Observation 3: $Q_i \neq Q_j$ if $i \neq j$.

Proof: If $Q_i = Q_j$ and $i < j$, then $Q_o \vdash_{C_1} Q_1 \vdash_{C_2}$ unsafety for r.

Observation 4: Let there be g generic rights in the protection system. Then*

$$m < 4x2^{g(|S_o|+1)(|O_o|+2)}.$$

Proof: By Observation 3, all Q_i's are distinct. By Observation 2, at most one subject and two objects (including the subject) are created. Thus, each access matrix can be represented by determining whether the new subject and/or object has been created and by selecting one of the 2^g subsets of rights for each of the at most $(|S_o|+1)(|O_o|+2)$ entries. By Observation 1 the names of subjects and objects do not change, so two access matrices with the same entries represent the same configuration. Thus there are are at most

$$4x(2^g)^{(|S_o|+1)(|O_o|+2)}$$

configurations in the sequence $Q_o \vdash Q_1 \vdash \ldots \vdash Q_m$, and the result follows!

An obvious decision algorithm for safety now presents itself. Consider all legal sequences of configurations satisfying Observations 1, 2 and 3. By Observation 4 there are only a finite number of these. We inspect each configuration to see if a leak may occur. We have thus completed the proof of the theorem.

V Undecidability of the Safety Problem

We are now going to prove that the general safety problem is not decidable. We assume the reader is familiar with the notion of a Turing machine (see [6], e.g.). Each Turing machine T consists of a finite set of states K and a distinct finite set of tape symbols Γ. One of the tape symbols is the blank B, which initially appears on each cell of a tape which is infinite to the right only (that is, the tape cells are numbered 1,2,...,i,...). There is a tape head which is always scanning (located at) some cell of the tape.

The moves of T are specified by a function δ from $K \times \Gamma$ to $K \times \Gamma \times \{L,R\}$. If $\delta(q,X) = (p,Y,R)$ for states p and q and tape symbols X and Y, then should the Turing machine T find itself in state q, with its tape head scanning a cell holding symbol X, then T enters state p, erases X and prints Y on the tape cell scanned and moves its tape head one cell to the right. If $\delta(q,X) = (p,Y,L)$, the the same thing happens, but the tape head moves one cell left (but never off the left end of the tape at cell 1).

Initially, T is is state q_0, the initial state, with its head at cell 1. Each tape cell holds the blank. There is a particular state q_f, known as the final state, and it is a fact that it is undecidable whether started as above, an arbitrary Turing machine T will eventually enter state q_f.

Theorem 2: It is undecidable whether a given configuration of a given protection system is safe for a given generic right.

Proof: We shall show that safety is undecidable by showing that a protection system, as we have defined the term, can simulate the behavior of an arbitrary Turing machine, with leakage of a right corresponding to the Turing machine entering a final state, a condition we know to be undecidable. The set of generic rights of our protection system will include the states and tape symbols of the Turing machine. At any time, the Turing machine will have some finite initial prefix of its tape cells, say 1,2,...k, which it has ever scanned. This situation will be represented by a sequence of k subjects, $s_1,s_2,\ldots,s_k$, such that s_i "owns" s_{i+1} for $1 \leq i < k$. Thus we use the ownership relation to order subjects into a linear list representing the tape of the Turing machine. Subject s_i represents cell i,

*|A| stands for the number of members in set A.

and the fact that cell i now holds tape symbol X is represented by giving s_i generic right X to itself. The fact that q is the current state and that the tape head is scanning the j^{th} cell is represented by giving s_j generic right q to itself. Note that we have assumed the states distinct from the tape symbols, so no confusion can result.

There is a special generic right end, which marks the last subject, s_k. That is, s_k has generic right end to itself, indicating that we have not yet created the subject s_{k+1} which s_k is to own. The generic right own completes the set of generic rights. An example showing how a tape whose first four cells hold WXYZ, with the tape head at the second cell and the machine in state q, is shown in Figure 3.

	s_1	s_2	s_3	s_4
s_1	{W}	{own}		
s_2		{X,q}	{own}	
s_3			{Y}	{own}
s_4				{Z,end}

Figure 3. Representing a Tape

The moves of the Turing machine are reflected in commands as follows. First, if $\delta(q,X) = (p,Y,L)$, then there is a command $C_{qX}(s,s')$ with conditions:

own $\in$ (s,s')
q $\in$ (s',s')
X $\in$ (s',s')

That is, s and s' must represent two consecutive cells of the tape, with the machine in state q, scanning the cell represented by s', and with symbol X written in s'. The interpretation of command C_{qX} is:

delete q from (s',s')
delete X from (s',s')
enter p into (s,s)
enter Y into (s',s')

For example, Figure 3 becomes Figure 4 when command C_{qX} is applied.

If $\delta(q,X) = (p,Y,R)$, that is, the tape head moves right, then we have two commands, depending whether or not the head passes the current end of the tape, i.e., the end right. There is command $C_{qX}(s,s')$ with conditions

	s_1	s_2	s_3	s_4
s_1	{W,p}	{own}		
s_2		{Y}	{own}	
s_3			{Y}	{own}
s_4				{Z,end}

Figure 4. Representing a Move

own $\in$ (s,s')
q $\in$ (s,s)
X $\in$ (s,s)

and interpretation:

delete q from (s,s)
delete X from (s,s)
enter p into (s',s')
enter Y into (s,s)

There is also a command $D_{qX}(s,s')$ with conditions:

end $\in$ (s,s)
q $\in$ (s,s)
X $\in$ (s,s)

and interpretation:

delete q from (s,s)
delete X from (s,s)
create subject s'
enter B into (s',s')
enter p into (s',s')
enter Y into (s,s)
delete end from (s,s)
enter end into (s',s')

If we begin with the initial matrix having one subject s_1, with rights q_0, B (blank) and own to itself, then the access matrix will always have exactly one generic right that is a state. This follows because each command deletes a state known by the conditions of that command to exist. Each command also enters one state into the matrix. Also, no entry in the access matrix can have more than one generic right that is a tape symbol by a similar argument. Likewise, end appears in only one entry of the matrix, the diagonal entry for the last created subject.

Thus, in each configuration of the protection system reachable from the initial configuration, there is at most one command applicable. That follows from the fact that the Turing machine has at most one applicable move in any situation, and the fact that C_{qX} and D_{qX} can never be simulaneously applicable. The protection system must therefore exactly simulate the Turing machine using the representation we have described. If the Turing machine enters state q_f, then the protection system can leak generic right q_f, otherwise, it is safe for q_f. Since it is undecid-

able whether the Turing machine enters q_f, it must be undecidable whether the protection system is safe for q_f.

We can prove a result similar to Theorem 2 which is in a sense a strengthening of it. By simulating a universal Turing machine on arbitrary input, we can exhibit a particular protection system for which it is undecidable whether a given initial configuration is safe for a given right. Thus, although we can give different algorithms to decide safety for different classes of systems, we can never hope even to cover all systems with a finite, or even infinite, collection of algorithms.

VI Conclusions and Open Questions

A very simple model for protection systems has been presented in which most protection issues can be represented. In this model, it has been shown that no algorithm can decide the safety of an arbitrary configuration of an arbitrary protection system. To avoid misunderstanding of this result, we shall list some implications of the result explicitly.

First, there is no hope of finding an algorithm which can certify the safety of an arbitrary configuration of an arbitrary protection system, or of all configurations for a given system. This result should not dampen the spirits of those working on operating systems verification. It only means they must consider restricted cases (or individual cases), and undoubtedly they have realized this already.

In a similar vein, the positive result of Section IV should not be a cause for celebration. In particular, the result is of no use unless it can be strengthened along the lines of the models in [7].

Our model does provide an interesting framework for investigating these questions. It offers a natural classification of certain features of protection systems. Which features cause a system to slip over the line and have an undecidable safety problem? Are there natural restrictions to place on a protection system which make it have a solvable safety question?

Acknowledgement

The authors thank one of the referees for simplifying the proof of Theorem 2.

References

(1) G. R. Andrews, "COPS - A Protection Mechanism for Computer Systems," Ph. D. Thesis and Technical Report 74-07-12, Computer Science Program, Univ. of Washington, Seattle, Wash., July, 1974.

(2) D. E. Bell and L. J. LaPadula, "Secure Computer Systems, Vol. I. Mathematical Foundations and Vol. II. A Mathematical Model," MITRE Corp. Technical Report MTR-2547, 1973.

(3) J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," CACM, Vol. 9, pp 143-155, 1966.

(4) R. M. Graham, "Protection in an Information Processing Utility," CACM, Vol. 11, pp. 365-369, 1968.

(5) G. S. Graham and P. J. Denning, "Protection - Principles and Practice," Proc. 1972 SJCC, Vol. 40, pp. 417-429, AFIPS Press, 1972.

(6) J. E. Hopcroft and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison Wesley, 1969.

(7) A. K. Jones, "Protection in Programmed Systems," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., June 1973.

(8) A. K. Jones and W. Wulf, "Towards the Design of Secure Systems," in Protection in Operating Systems, Colloques IRIA, Rocquencourt, France, pp. 121-136, 1974.

(9) B. W. Lampson, "Protection," Proc. Fifth Princeton Symp. on Information Sciences and Systems, Princeton University, March 1971, pp. 437-443. Reprinted in Operating Systems Review, Vol. 8, No.1, pp. 18-24, January 1974.

(10) B. W. Lampson, "A Note on the Confinement Problem," CACM, Vol. 16, pp. 613-615, 1973.

(11) R. M. Needham, "Protection Systems and Protection Implementations," Proc. 1972 FJCC, Vol. 41, pp. 571-578, AFIPS Press, 1972.

(12) G. J. Popek, "Correctness in Access Control," Proc. ACM National Computer Conference, pp. 236-241, 1974.

(13) D. M. Ritchie and K. Thompson, "The UNIX Time Sharing System," CACM, Vol. 17, pp. 365-375,1974.

(14) J. H. Saltzer, "Protection and the Control of Information Sharing in MULTICS," CACM, Vol. 17, pp. 388-402, 1974.