Operating Systems
R.S. Gaines Editor

# Protection in Operating Systems

Michael A. Harrison and Walter L. Ruzzo
University of California, Berkeley
Jeffrey D. Ullman
Princeton University

**A model of protection mechanisms in computing systems is presented and its appropriateness is argued. The "safety" problem for protection systems under this model is to determine in a given situation whether a subject can acquire a particular right to an object. In restricted cases, it can be shown 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, it cannot be decided if a situation is safe. Various implications of this fact are discussed.**

**Key Words and Phrases: protection, protection system, operating system, decidability, Turing machine**
**CR Categories: 4.30, 4.31, 5.24**



Research was sponsored by NSF Grants GJ-474 and GJ-43332. Work was performed while the third author was on leave at the University of California at Berkeley.

Author's addresses: M.A. Harrison and W. L. Ruzzo, Department of Computer Science, University of California, Berkeley, CA 94720; J.D. Ullman, Department of Electrical Engineering, Computer Science Laboratory, Princeton University, Princeton, NJ 08540.

## 1. 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, databases, 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 misappropriation by unauthorized users. In recent years, a great deal of attention has been focussed on the problem. Papers [4–6, 8–13, 15] are but a sample of the work that has been done. In particular, Saltzer [15] 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 [4, 6, 10] 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 more complete model of protection systems was created in [8], which could express a variety of policies and which contained the "models" of [4, 6, 10] as special cases. However, no attempt to prove global properties of protection systems was made in [8], and the model was not completely formalized.

On the other hand, there have been models in which attempts were made to prove results [2, 3, 13]. In [2], which is similar to [8] 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.

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 [6, 10], where it was argued that the model is capable of describing most protection systems currently in use.

Section 2 describes the motivation for looking at decidability issues in protection systems. Section 3 presents the formal model with examples. In Section 4 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 can be decided for these systems. In Section 5 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.

## 2. 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 [7], e.g.). The implication of the latter undecidability result is that proving a particular grammar unambiguous might be difficult, although it is possible to write down a particular grammar, for Algol, say, and prove that it is unambiguous. By 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.

By 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 [7], e.g.). There, one tests a grammar for $LR(k)$-ness, and if it is 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 difficult.

## 3. 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* $R$,

(2) a finite set $C$ of *commands* of the form:

**command** $\alpha(X_1, X_2, \ldots, X_k)$
  **if** $r_1$ in $(X_{s_1}, X_{o_1})$ and
    $r_2$ in $(X_{s_2}, X_{o_2})$ and
      ...
    $r_m$ in $(X_{s_m}, X_{o_m})$
  **then**
    $op_1$
    $op_2$
    ...
    $op_n$
**end**

or if $m$ is zero, simply

**command** $\alpha(X_1, \ldots, X_k)$
  $op_1$
  ...
  $op_n$
**end**

Here, $\alpha$ is a name, and $X_1, \ldots, X_k$ are formal parameters. Each $op_i$ is one of the *primitive operations*

**enter** $r$ **into** $(X_s, X_o)$
**delete** $r$ **from** $(X_s, X_o)$
**create subject** $X_s$
**create object** $X_o$
**destroy subject** $X_s$
**destroy object** $X_o$

Also, $r, r_1, \ldots, r_m$ are generic rights and $s, s_1, \ldots, s_m$ and $o, o_1, \ldots, o_m$ are integers between 1 and $k$. We may call the predicate following **if** the *conditions* of $\alpha$ and the sequence of operations $op_1, \ldots, op_n$ the *body* of $\alpha$.

Before explaining the significance of the commands we need to 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" [4] for subject $s$, while column $o$ is similar to an "access list" for object $o$.

Now let us interpret the parts of a protection system. In practice, typical subjects might be processes [4], 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 (2) above are meant to be formal procedures.

Since we wish to model only the protection aspects

of an operating system, we wish to avoid embedding into the model unrestricted computing power. The commands therefore, are required to have a very simple structure. Each command may specify a test for the presence of certain rights in certain positions of the current access matrix. These conditions can be used to verify that the action to be performed by the command is authorized. For example, "**if** $r$ in $(X_s, X_o)$ **then** . . ." indicates that the subject $x_s$ needs right $r$ to object $x_o$, where $x_s$ and $x_o$ are the actual parameters corresponding to formal parameters $X_s$ and $X_o$. If the conditions are not satisfied, the body of the command is not executed. The command body is simply straight line code, a sequence of primitive operations containing no conditional or unconditional branches, no loops, and no procedure calls.

Each primitive operation specifies some modification which is to be made to the access matrix. For example, **enter** $r$ **into** $(X_s, X_o)$ will enter right $r$ into the matrix at position $(x_s, x_o)$, where $x_s$ and $x_o$ are the actual parameters corresponding to the formals $X_s$ and $X_o$. That is, subject $x_s$ is granted generic right $r$ to object $x_o$. The effect of a command will be defined more formally after an example.

*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 as follows.

(1) A process may create a new file. The process creating the file has ownership of that file. We represent this action by

```
command CREATE(process, file)
  create object file
  enter own into (process, file)
end
```

(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 of the form

```
command CONFER_r (owner, friend, file)
  if own in (owner, file)
  then enter r into (friend, file)
end
```

where $r$ is **read**, **write**, or **execute**. Technically the $r$ here is not a parameter (our model allows only objects as parameters). Rather, this is an abbreviation for the three commands $CONFER_{\mathbf{read}}$, etc.

(3) Similarly, we have three commands by which the owner of a file may revoke another subject's access rights to the file.

```
command REMOVE_r (owner, exfriend, file)
  if own in (owner, file) and
    r in (exfriend, file)[1]
  then delete r from (exfriend, file)
end
```

where $r$ is **read**, **write**, or **execute**.

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.* The six primitive commands mean exactly what their names imply. Formally, we state their effect on access matrices as follows. Let $(S, O, P)$ and $(S', O', P')$ be configurations of a protection system, and let $op$ be a primitive operation. We say that:

$$(S, O, P) \Rightarrow_{op} (S', O', P')$$

[read $(S, O, P)$ yields $(S', O', P')$ under $op$] if either:

(1) $op =$ **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) $op =$ **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 $P'[s, o] = P[s, o] - \{r\}$.

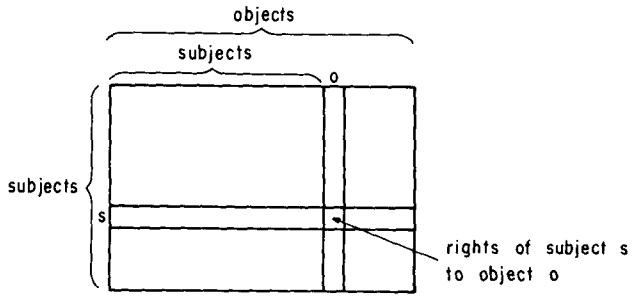
(3) $op =$ **create subject** $s'$, where $s'$ is a new symbol not in $O$, $S' = S \cup \{s'\}$, $O' = O \cup \{s'\}$, $P'[s, o] = P[s, o]$ for all $(s, o)$ in $S \times O$, $P'[s', o] = \varnothing$[2] for all $o \in O'$, and $P[s, s'] = \varnothing$ for all $s \in S'$.

(4) $op =$ **create object** $o'$, where $o'$ is a new symbol not in $O$, $S' = S$, $O' = O \cup \{o'\}$, $P'[s, o] = P[s, o]$ for all $(s, o)$ in $S \times O$ and $P'[s, o'] = \varnothing$ for all $s \in S$.

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

Fig. 1. Access matrix.




[1] This condition need not be present, since **delete** $r$ **from** (exfriend, file) will have no effect if $r$ is not there.

[2] $\varnothing$ denotes the empty set.

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 primitive operation 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 how a protection system executes a command.

*Definition.* Let $Q = (S, O, P)$ be a configuration of a protection system containing:

**command** $\alpha(X_1, \ldots, X_k)$
  **if** $r_1$ in $(X_{s_1}, X_{o_1})$ and
  ...
  $r_m$ in $(X_{s_m}, X_{o_m})$
  **then** $op_1, \ldots, op_n$
**end**

Then we say

$$Q \vdash_{\alpha(x_1,\ldots,x_k)} Q'$$

where $Q'$ is the configuration defined as follows:

(1) If $\alpha$'s conditions are not satisfied, i.e. if there is some $1 \leq i \leq m$ such that $r_i$ is not in $P[x_{s_i}, x_{o_i}]$, then $Q = Q'$.

(2) Otherwise, i.e. if for all $i$ between 1 and $m$, $r_i \in P[x_{s_i}, x_{o_i}]$, then let there exist configurations $Q_0, Q_1, \ldots, Q_n$ such that

$$Q = Q_0 \Rightarrow_{op_1^*} Q_1 \Rightarrow_{op_2^*} \cdots \Rightarrow_{op_n^*} Q_n$$

where $op_i^*$ denotes the primitive operation $op_i$ with the actual parameters $x_1, \ldots, x_k$ replacing all occurrences of the formal parameters $X_1, \ldots, X_k$, respectively. Then $Q'$ is $Q_n$.

We say that $Q \vdash_\alpha Q'$ if there exist parameters $x_1, \ldots, x_k$ such that $Q \vdash_{\alpha(x_1,\ldots,x_k)} Q'$; we say $Q \vdash Q'$ if there exists a command $\alpha$ such that $Q \vdash_\alpha Q'$.

It is also convenient to write $Q \vdash^* Q'$, 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

**command** $\alpha(X, Y, Z)$
  **enter** $r_1$ **into** $(X, X)$
  **destroy subject** $X$
  **enter** $r_2$ **into** $(Y, Z)$
**end**

There can never be a pair of configurations $Q$ and $Q'$ such that

$$Q \vdash_{\alpha(x,x,z)} Q'$$

since the third primitive operation **enter** $r_2$ **into** $(x, z)$ will occur at a point where no subject named $x$ exists.

*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 | ∅ | ∅ |
| Joe | ∅ | ∅ |

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*$_{\text{execute}}$(Sam, Joe, Code)
*CONFER*$_{\text{read}}$(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 body consists of two primitive operations, **create object** Code and **enter own into** (Sam, Code). Then, using the $\Rightarrow$ notation, we may show the effect of the two primitive operations as:

| | Sam | Joe |
|---|---|---|
| Sam | ∅ | ∅ |
| Joe | ∅ | ∅ |

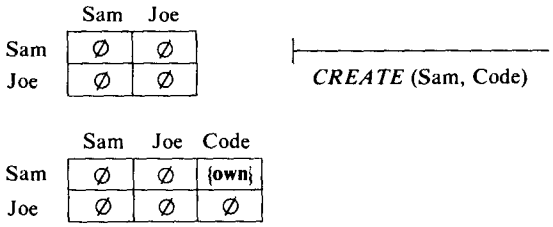
⟹ **create object** Code

| | Sam | Joe | Code |
|---|---|---|---|
| Sam | ∅ | ∅ | ∅ |
| Joe | ∅ | ∅ | ∅ |

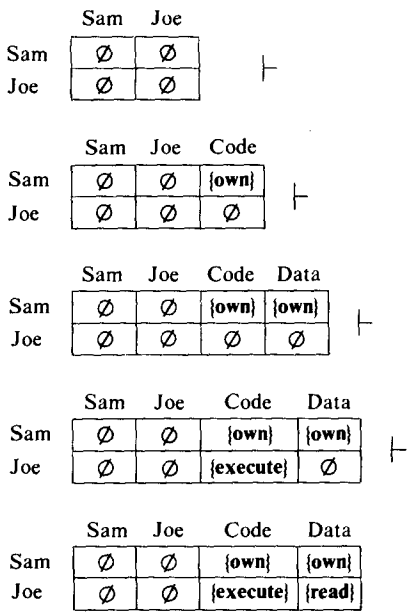
⟹ **enter own into** (Sam, Code)

| | Sam | Joe | Code |
|---|---|---|---|
| Sam | ∅ | ∅ | {own} |
| Joe | ∅ | ∅ | ∅ |

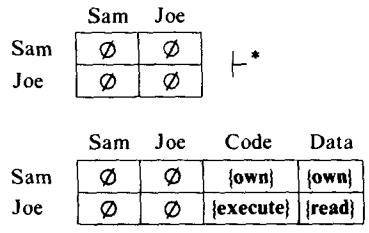
Thus, using the $\vdash$ notation for complete commands we can say that:




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




We may thus say:




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.

It is our contention that the model we have presented has sufficient generality to allow one to specify almost all of the protection schemes that have been proposed: cf. [6] 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. In the next two sections, we shall develop the theory of protection systems using our model. 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 [6]. 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

**command** *IREAD*($s_1$, $s_2$, $o$)
  **if**
    **read** in ($s_2$, $o$) and
    **iread** in ($s_1$, $s_2$)
  **then**
    **enter read into** ($s_1$, $o$)
    **delete read from** ($s_1$, $o$)
**end**

It should be noted that the command in Example 3 has both multiple conditions and a body 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 command actually has no net 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 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 [14] 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.[3] 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 a generic right **own** is needed, and that the rights of a user $u$ to a file $f$ which $u$ owns could be placed in the $(u,f)$ entry of the access matrix. However, when we create a file $f$, it is not possible in our formalism to express a command such as "give all subjects the right to read $f$," 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 $f$ with the $(f,f)$ 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 user $u$ can read (or write or execute) a file $f$ if either:

(1) **own** is in $(u, f)$, i.e., $u$ owns $f$, and the entry "owner can read" is in $(f, f)$, or
(2) the entry "anyone can read" is in $(f, f)$.

[3] We ignore the role of the "superuser" in the following discussion.

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** $\in P[u,f]$ and **oread** $\in P[f,f]$ or
(2) **aread** $\in P[f,f]$

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 identical. 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 $(u,f)$ by a *READ* command, representing the fact that $s$ can actually read $f$. **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.

Fig. 2. UNIX type protection mechanism.

```
command CREATEFILE(u, f)
  create subject f
  enter own into (u, f)
end
command LETOREAD(u, f)
  if own in (u, f)
  then enter oread into (f, f)
end
command LETAREAD(u, f)
  if own in (u, f)
  then enter aread into (f, f)
end
command READ(u, f)
  if either
       own in (u, f) and
       oread in (f, f)
    or
       aread in (f, f)
  then
    enter read into (u, f)
    delete read from (u, f)
end
```

## 4. 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 minimum 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. Furthermore, in some restricted cases where safety is decidable, the decision procedures are probably too slow to be of practical utility.

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

```
command DUMMY(s, o)
  if
     r in (s, o) and
     r' in (o, o)
  then
     enter r'' into (o, o)
end
```

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 definition of the safety question for protection systems.

*Definition.* Given a protection system, we say command $\alpha(X_1, \ldots, X_k)$ *leaks generic right r from configuration* $Q = (S, O, P)$ if $\alpha$, when run on $Q$, can execute a primitive operation which enters $r$ into a cell of the access matrix which did not previously contain $r$. More formally, there is some assignment of actual parameters $x_1, \ldots, x_k$ such that

(1) $\alpha(x_1, \ldots, x_k)$ has its conditions satisfied in $Q$, i.e. for each clause "$r$ in $(X_i, X_j)$" in $\alpha$'s conditions we have $r \in P[x_i, x_j]$, and

(2) if $\alpha$'s body is $op_1, \ldots, op_n$, then there exists an $m$, $1 \leq m \leq n$, and configurations $Q = Q_0, Q_1, \ldots Q_{m-1} = (S', O', P')$, and $Q_m = (S'', O'', P'')$, such that

$$Q_0 \Rightarrow_{op_1^*} Q_1 \Rightarrow_{op_2^*} \cdots Q_{m-1} \Rightarrow_{op_m^*} Q_m$$

where $op_i^*$ denotes $op_i$ after substitution of $x_1, \ldots, x_k$ for $X_1, \ldots, X_k$ and moreover, there exist some $s$ and $o$ such that

$$r \notin P'[s, o] \quad \text{but} \quad r \in P''[s, o]$$

(Of course, $op_m$ must be **enter** $r$ **into** $(s, o)$).

Notice that given $Q$, $\alpha$ and $r$, it is easy to check whether $\alpha$ leaks $r$ from $Q$. Also note that $\alpha$ leaks $r$ from $Q$ even if $\alpha$ deletes $r$ after entering it. Commands *IREAD* in Example 3 and *READ* in Example 4 are typical of commands which enter a right and then immediately delete it. In a real system we would expect a procedure called "*READ*" to contain code between the enter and delete operations which passes data from the file read to some other file or process. Although we do not model directly the presence of such code, the temporary presence of the "read" right in the access matrix pinpoints this data transfer, thus identifying the potential leak.

We should emphasize that "leaks" are not necessarily "bad." Any interesting system will have commands which can enter some rights (i.e. be able to leak those rights). The term assumes its usual negative significance only when applied to some configuration, most likely modified to eliminate "reliable" subjects as discussed in the beginning of this section, and to some right which we hope cannot be passed around.

*Definition.* Given a particular protection system and generic right $r$, we say that the initial configuration $Q_0$ is *unsafe* for $r$ (or *leaks* $r$) if there is a configuration $Q$ and a command $\alpha$ such that

(1) $Q_0 \vdash^* Q$, and

(2) $\alpha$ leaks $r$ from $Q$.

We say $Q_0$ is *safe* for $r$ if $Q_0$ is not unsafe for $r$.

*Example 5.* Let us reconsider the simple example of a command $\alpha(X, Y, Z)$ which immediately precedes Example 2. Suppose $\alpha$ were the only command in the system. If the initial configuration has exactly one subject and no other objects, then it is safe for $r_2$ but not for $r_1$.

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. Decidability in this special case is not significant in itself, since it is much too restricted to model interesting systems. 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 CREATEFILE has length two.

THEOREM 1. *There is an algorithm which decides whether or not a given mono-operational protection system and initial configuration is unsafe for a given generic right r.*

PROOF. The proof hinges on two simple observations. First, commands can test for the presence of rights, but not for the absence of rights or objects. This allows delete and destroy commands[4] to be removed from computations leading to a leak. Second, a command can only identify objects by the rights in their row and column of the access matrix. No mono-operational command can both create an object and enter rights, so multiple creates can be removed from computations, leaving the creation of only one subject. This allows the length of the shortest "leaky" computation to be bounded.

Suppose

$$(*) \quad Q_0 \vdash_{c_1} Q_1 \vdash_{c_2} \cdots \vdash_{c_m} Q_m$$

is a minimal length computation reaching some configuration $Q_m$ for which there is a command $\alpha$ leaking $r$. Let $Q_i = (S_i, O_i, P_i)$. Now we claim that $C_i$, $2 \leq i \leq m$ is an enter command, and $C_1$ is either an enter or create subject command. Suppose not, and let $C_n$ be the last non-enter command in the sequence $(*)$. Then we could form a shorter computation

$$Q_0 \vdash_{c_1} Q_1 \vdash \cdots Q_{n-1} \vdash_{c'_{n+1}} Q'_{n+1} \vdash \cdots \vdash_{c'_m} Q_m'$$

as follows.

(a) if $C_n$ is a delete or destroy command, let $C_i' = C_i$ and $Q_i' = Q_i$ plus the right, subject or object which would have been deleted or destroyed by $C_n$. By the first observation above, $C_i$ cannot distinguish $Q_{i-1}$ from $Q'_{i-1}$, so $Q'_{i-1} \vdash_{c_i'} Q_i'$ holds. Likewise, $\alpha$ leaks $r$ from $Q_m'$ since it did so from $Q_m$.

[4] Since the system is mono-operational, we can identify the command by the type of primitive operation.

(b) Suppose $C_n$ is a create subject command and[5] $|S_{n-1}| \geq 1$ or $C_n$ is a create object command. Note that $\alpha$ leaks $r$ from $Q_m$ by assumption, so $\alpha$ is an enter command. Further, we must have $|S_m| \geq 1$ and

$$|S_m| = |S_{m-1}| = \ldots = |S_n| \geq 1$$

($C_m, \ldots, C_{n+1}$ are enter commands by assumption). Thus $|S_{n-1}| \geq 1$ even if $C_n$ is a create object command. Let $s \in S_{n-1}$. Let $o$ be the name of the object created by $C_n$. Now we can let $C_i' = C_i$ with $s$ replacing all occurrences of $o$, and $Q_i' = Q_i$ with $s$ and $o$ merged. For example, if $o \in O_n - S_n$ we would have

$$S_i' = S_i,$$
$$O_i' = O_i - \{o\},$$
$$P_i'[x, y] = \begin{cases} P_i[x, y] & \text{if } y \neq s \\ P_i[x, s] \cup P_i[x, o] & \text{if } y = s. \end{cases}$$

Clearly,

$$P_i[x, o] \subseteq P_i'[x, s],$$

so for any condition in $C_i$ satisfied by $o$, the corresponding condition in $C_i'$ is satisfied by $s$. Likewise for the conditions of $\alpha$.

(c) Otherwise, we have $|S_{n-1}| = 0$, $C_n$ is a create subject command, and $n \geq 2$. The construction in this case is slightly different—the create subject command cannot be deleted (subsequent "enters" would have no place to enter into). However, the commands preceding $C_n$ can be skipped (provided that the names of objects created by them are replaced), giving

$$Q_0 \vdash_{c_n} Q_n' \vdash_{c_{n+1}'} Q_{n+1}' \vdash \ldots \vdash_{c_m'} Q_m'$$

where, if $S_n = \{s\}$, we have $C_i'$ is $C_i$ with $s$ replacing the names of all objects in $O_{n-1}$, and $Q_i'$ is $Q_i$ with $s$ merged with all $o \in O_{n-1}$.

In each of these cases we have created a shorter "leaky" computation, contradicting the supposed minimality of (*). Now we note that no $C_i$ enters a right $r$ into a cell of the access matrix already containing $r$, else we could get a shorter sequence by deleting $C_i$. Thus we have an upper bound on $m$:

$$m \leq g(|S_0| + 1)(|O_0| + 1) + 1$$

where $g$ is the number of generic rights.

An obvious decision procedure now presents itself—try all sequences of enter commands, optionally starting with a create subject command, of length up to the bound given above. This algorithm is exponential in the matrix size. However, by using the technique of "dynamic programming" (see [1], e.g.), an algorithm polynomial in the size of the initial matrix can easily be devised for any given protection system.

It is worth noting that if we wish a decision procedure for all mono-operational systems, where the commands are a parameter of the problem, then the decision problem is "*NP*-complete." To say that a problem is *NP*-complete intuitively means that if the problem could be shown to be solvable in polynomial time, this would be a major result in that a large number of other problems could be solved efficiently. The best known such problem is probably the "traveling salesperson problem." Thus the above problem is almost certainly of exponential time complexity in the size of the matrix. Cf. [1] for a more thorough discussion of these matters.

For those familiar with the technical definitions needed, the argument will be sketched. (All these definitions may be found in [1].) We can prove the result by reducing the $k$-clique problem to the problem: given a mono-operational system, a right $r$ and an initial access matrix, determine if that matrix is safe for $r$. Given a graph and an integer $k$, produce a protection system whose initial access matrix is the adjacency matrix for the graph and having one command. This command's conditions test its $k$ parameters to see if they form a $k$-clique, and its body enters some right $r$ somewhere. The matrix will be unsafe for $r$ in this system if and only if the graph has a $k$-clique. The above is a polynomial reduction of the known *NP*-complete clique problem to our problem, so our problem is at best *NP*-complete. It is easy to find a nondeterministic polynomial time algorithm to test safety, so our problem is in fact *NP*-complete and no worse.

One obvious corollary of the above is that any family of protection systems which includes the mono-operational systems must have a general decision problem which is at least as difficult as the *NP*-complete problems, although individual members of the family could have easier decision problems.

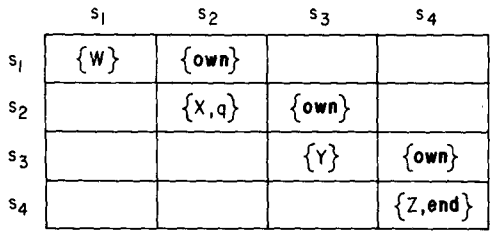
Another unproven but probably true characteristic of *NP*-complete problems has interesting implications concerning proofs of safety. We can give a "short," i.e. polynomial length, proof that a given matrix for a mono-operational system is *not* safe [just list the computation (*)], although such a proof may be difficult to find. However, it is probable that there is no proof system in which we can guarantee the existence of, let alone find, short proofs that an initial matrix for an arbitrary mono-operational system *is* safe.

## 5. 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 [7], e.g.). Each Turing machine $T$ consists of a finite set of *states* $K$ and a distinct finite set of *tape symbols* $\Gamma$. 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 $1, 2, \ldots, i, \ldots$). There is a *tape head* which is always *scanning* (located at) some cell of the tape.

[5] $|A|$ stands for the number of members in set $A$.

Fig. 3. Representing a tape.




The *moves* of $T$ are specified by a function $\delta$ 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 $\sigma(q, X) = (p, Y, L)$, 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 in 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 $1, 2, \ldots k$, 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 $i$, 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.

The moves of the Turing machine are reflected in commands as follows. First, if

$$\delta(q, X) = (p, Y, L),$$

then there is

```
command C_qX(s, s')
  if
    own in (s, s') and
    q in (s', s') and
    X in (s', s')
  then
    delete q from (s', s')
    delete X from (s', s')
    enter p into (s, s)
    enter Y into (s', s')
end
```

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 the symbol $X$ written in $s'$. The body of the command changes $X$ to $Y$ and moves the head left, changing the state from $q$ to $p$. 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, that is, the **end** right. There is

```
command C_qX(s, s')
  if
    own in (s, s') and
    q in (s, s) and
    X in (s, s)
  then
    delete q from (s, s)
    delete X from (s, s)
    enter p into (s', s')
    enter Y into (s, s)
end
```

To handle the case where the Turing machine moves into new territory, there is also

```
command D_qX(s, s')
  if
    end in (s, s) and
    q in (s, s) and
    X in (s, s)
  then
    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')
    enter own into (s, s')
end
```

If we begin with the initial matrix having one sub-

ject $s_1$, with rights $q_o$, $B$ (blank) and **end** 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. This 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 simultaneously 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 undecidable 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. Theorem 2 says that there is no single algorithm which can decide safety for all protection systems. One might hope that for each protection system, one could find a particular algorithm to decide safety. We can easily show that this is not possible. By simulating a universal Turing machine [7] on an 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.

Two other facts are easily seen. First, since we know that there are arbitrarily complex computable functions, there must be special cases of protection systems where safety is decidable but arbitrarily difficult. Second, although any real system must place a bound on the number of objects which can be created, this bound will not make the decision of the safety question easy. While the finiteness of real resources does make safety decidable, we can show the following.

THEOREM 3. *The question of safety for protection systems without create commands is complete in polynomial space.*[6]

PROOF. A construction similar to that of Theorem 2 proves that any polynomial space bounded Turing machine can be reduced in polynomial time to an initial access matrix whose size is polynomial in the length of the Turing machine input.

Fig. 4. After one move.

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

## 6. 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 4 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 [8].

Our model offers a natural classification of certain features of protection systems and provides an interesting framework for investigating the following questions: 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?

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

Received November 1974; revised December 1975

[6] This probably implies that that decision problem requires exponential time: cf. [1].

**References**

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. Andrews, G.R. COPS—A protection mechanism for computer Systems. Ph.D. Th. and Tech. Rep. 74-07-12, Computer Sci. Program, U. of Washington, Seattle, Wash., July, 1974.
3. Bell, D.E., and LaPadula, L.J. Secure Computer Systems, Vol. I: Mathematical Foundations and Vol. II: A Mathematical Model. MITRE Corp. Tech. Rep. MTR-2547, 1973.
4. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (March 1966), 143–155.
5. Graham, R.M. Protection in an information processing utility. *Comm. ACM 11*, 5 (May 1968), 365–369.
6. Graham, G.S., and Denning, P.J. Protection—principles and practice. AFIPS Conf. Proc., 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., 1972, pp. 417–429.
7. Hopcroft, J.E., and Ullman, J.D. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Mass. 1969.
8. Jones, A.K. Protection in programmed systems. Ph.D. Th., Dep. of Computer Sci., Carnegie-Mellon U., Pittsburgh, Pa., June 1973.
9. Jones, A.K., and Wulf, W. Towards the design of secure systems. In *Protection in Operating Systems*, Colloques IRIA, Rocquencourt, France, 1974, pp. 121–136.
10. Lampson, B.W. Protection, Proc. Fifth Princeton Symp. on Information Sciences and Systems, Princeton University, March 1971, pp. 437–443. Reprinted in *Operating Systems Rev. 8*, 1 (Jan. 1974), 18–24.
11. Lampson, B.W. A note on the confinement problem. *Comm. ACM 16*, 10 (Oct. 1973), 613–615.
12. Needham, R.M. Protection systems and protection implementations. AFIPS Conf. Proc., 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., 1972, pp. 571–578.
13. Popek, G.J. Correctness in access control. Proc. ACM Nat. Computer Conf., 1974, pp. 236–241.
14. Ritchie, D.M., and Thompson, K. The UNIX time sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
15. Saltzer, J.H. Protection and the control of information sharing in MULTICS. *Comm. ACM 17*, 7 (July 1974), 388–402.

Programming Techniques

C. Manacher, S. L. Graham Editors

# An Insertion Technique for One-Sided Height-Balanced Trees

D. S. Hirschberg
Princeton University

**A restriction on height-balanced binary trees is presented. It is seen that this restriction reduces the extra memory requirements by half (from two extra bits per node to one) and maintains fast search capabilities at a cost of increased time requirements for inserting new nodes.**

**Key Words and Phrases: balanced, binary, search, trees**

**CR Categories: 3.73, 3.74, 4.34, 5.25, 5.31**

Binary search trees are a data structure in common use. To keep search time relatively small, the method of balancing binary trees was introduced by Adel'son-Vel'skii and Landis [1]. These height-balanced binary trees (also called AVL trees) require two extra bits per node and require only $O(\log N)$ operations to search and/or insert an item, where $N$ is the number of nodes in the tree. Each node in the tree has a *height* which is defined to be the length of the longest path from that node down the tree. The heights of the two sons of a node may differ by at most one.

Knuth [2] suggests considering the case where the tree is further restricted so that the right son never has smaller height than the left son. We call such a tree a *one-sided height-balanced* (OSHB) *tree*. In this case, only one extra bit is required per node. The saving of one bit per node is significant if that bit would have re-



Research supported by an IBM fellowship. Author's present address: Department of Electrical Engineering, Rice University, Houston, TX. 77001.