Protection—Principles and practice*

by G. SCOTT GRAHAM and PETER J. DENNING

Princeton University
Princeton, New Jersey

INTRODUCTION

The protection mechanisms of computer systems control the access to objects, especially information objects. The range of responsibilities of these mechanisms includes at one extreme completely isolating executing programs from each other, and at the other extreme permitting complete cooperation and shared access among executing programs. Within this range one can identify at least seven levels at which protection mechanisms can be conceived as being required, each level being more difficult than its predecessor to implement:

1. No sharing at all (complete isolation).
2. Sharing copies of programs or data files.
3. Sharing originals of programs or data files.
4. Sharing programming systems or subsystems.
5. Permitting the cooperation of mutually suspicious subsystems—e.g., as with debugging or proprietary subsystems.
6. Providing "memoryless" subsystems—i.e., systems which, having performed their tasks, are guaranteed to have kept no secret record of the task performed (an income-tax computing service, for example, must be allowed to keep billing information on its use by customers but not to store information secretly on customers' incomes).
7. Providing "certified" subsystems—i.e., those whose correctness has been completely validated and is guaranteed *a priori.*

We shall consider here the protection mechanisms required for level 5, but not those required for levels 6 or 7. We do this because we are interested in specifying the structure of protection mechanisms that work irrespective of considerations of internal program structure; levels 6 and 7 in general require control over internal program structure. Moreover, little is known about the mechanisms for levels 6 and 7, whereas much is known about the mechanisms for levels 1 to 5.

Much has been written about the legal and social implications of protection and privacy (see, for example Reference 10). We deal exclusively with the technical aspects of protection—i.e., the procedures governing the access of executing programs (processes) to various resources in the system. This is because little has been written about the technical side of the problem. It is also because the technical approach, when balanced with the legal approach, is useful in clarifying the inadequacies of both approaches and in understanding how they can complement one another. For example, without a technically sound protection system, a user might find it possible to perform malicious acts undetected, rendering certain laws unenforceable. Even with a sound protection system, however, certain aspects of protection can be dealt with only by laws, e.g., the problem of false identification.

The first step in the design of a protection system is the specification of the main parties and their interactions. That is, the entities to be protected and the entities to be protected against are identified, and rules by which the latter may access the former are formulated. For this, common sense and examples of existing protection systems can be used to guide our thinking in the development of the elements and principles of a protection system. We formalize these principles as a model (theory) of protection, based on one developed by Lampson.*[16] Perhaps the most important aspect of

* Work reported herein was supported in part by NASA Grant NGR-31-001-170 and by National Research Council of Canada Grant A7146.

* Approximately half our development here is an exposition and reformulation of the elements of Lampson's model. The remainder extends Lampson's in several important ways. We have investigated the problems of establishing the correctness of a protection system, of creating and deleting objects, of implementing cooperation among mutually suspicious subsystems, and of identifying the existence or absence of the elements of the model in contemporary systems.

the model is the notion that each process has a unique identification number which is attached by the system to each access attempted by the process. As will be seen, this will make it impossible for a process to disguise its identity and will allow us to establish the correctness of the model with respect to implementing protection at level 5.

The implications of the protection model with respect to operating system structure and hardware architecture will be considered. Although the model is abstracted from examples of existing systems, we shall see that few current commercial systems meet the requirements of a complete (secure) protection system. We shall identify a few existing systems which do meet the completeness requirement of our model.

A THEORY OF PROTECTION

In current systems, protection takes many forms. The IBM System/360 uses a "key-lock" approach to memory protection.[12] A "ring" protection mechanism is used in Multics.[9,18] The Hitac 5020 Time-Sharing System uses a key-lock-ring mechanism.[17] There are many other examples. As Lampson has pointed out, the myriad of implementations forces us, as observers, to take an abstract approach to the subject of protection;[16] otherwise, we might never make sense out of the seemingly endless variety of solutions to protection problems.

Elements of the model

A protection system comprises three parts, which will be reflected as components of our model. The first component is a set of *objects*, an object being an entity to which access must be controlled. Examples of objects are pages of main memory, files, programs, auxiliary memory devices, and instructions. Associated with each object X is a unique *identification number* which is assinged to the object when it is created in the operating system. Identification numbers might, for example, be derived from a clock which measures time in microseconds since the system was first built; the time-of-day clock on the IBM System/370, being a 52-bit counter, could allow a new object name every microsecond for approximately 143 years.[13] We shall use the symbol X interchangeably for an object's name and number.

The second component of the protection model is a set of *subjects*, a subject being an active entity whose access to objects must be controlled. A subject may be regarded as a pair (process, domain), in which a process is a program in execution and a domain is the protection environment (context) in which the process is operating. Examples of domains include supervisor/problem-program states in IBM System/360,[12] the 15 user environments in IBM's OS/360-MVT,[11] or the file directories and rings in Multics.[1] Other terms which have been used to denote the idea of domain are "sphere of protection" [4] and "ring." [9,18] Since subjects must be protected, they too are objects, and each has a unique identification number.

The third component of a protection system is the rules which govern the accessing of objects by subjects. We shall describe a particular choice of rules shortly. These rules are the heart of the protection system. The rules must be simple, allowing users to gain an immediate understanding of their scope and use. They must be complete, not allowing a subject to gain unauthorized access to an object. They must be flexible, providing mechanisms which easily allow the desired degree of authorized sharing of objects among subjects.

An important property of our model is: each and every attempted access by a subject to an object is validated. This is necessary in order to permit cooperation among mutually suspicious subjects, for it cannot otherwise be guaranteed that a previously well-behaved subject which suddenly turned malicious or went awry will be denied access when authorization is lacking.

The choice of the subjects, objects, and rules of a protection system is at the discretion of the system designer and will be made to meet the protection and sharing requirements of the given system. We return later to the problems of choosing subjects and objects.

It is convenient to regard all the information specifying the types of access subjects have to objects as constituting a "protection state" of the system. There are three distinct problems to be solved: representing the protection state, causing subjects to access objects only as permitted by the protection state, and allowing

		OBJECTS						
		subjects			files		devices	
		S_1	S_2	S_3	F_1	F_2	D_1	D_2
SUBJECTS	S_1		block wakeup		read write		seek	
	S_2			stop		update		seek
	S_3				delete	execute		

Figure 1—Portion of an access matrix

subjects to alter the protection state in certain ways. With respect to the first, we may represent the protection state of the system as an *access matrix* A, with subjects identifying the rows and objects the columns (see Figure 1). The entry $A[S, X]$ contains strings, called *access attributes*, specifying the access privileges held by subject S to object X. If string α appears in $A[S, X]$, we say "S has α access to X." For example, in Figure 1, subject S_1 may read file F_1, since 'read' appears in $A[S_1, F_1]$; or, S_2 may stop S_3. Figure 2 shows that an alternative representation of the protection state is a directed graph, which is in one-to-one correspondence with the access matrix.

Associated with each type of object is a *monitor*, through which all access to objects of that type must pass to be validated. Examples of monitors are the file system for files, the hardware for instruction execution and memory addressing, and the protection system for subjects. An access proceeds as follows:

1. S initiates access to X in manner α.
2. The computer system supplies the triple (S, α, X) to the monitor of X.
3. The monitor of X interrogates the access matrix to determine if α is in $A[S, X]$; if it is, access is permitted, otherwise, it is denied and a protection violation occurs.

Note that access attributes are interpreted by object

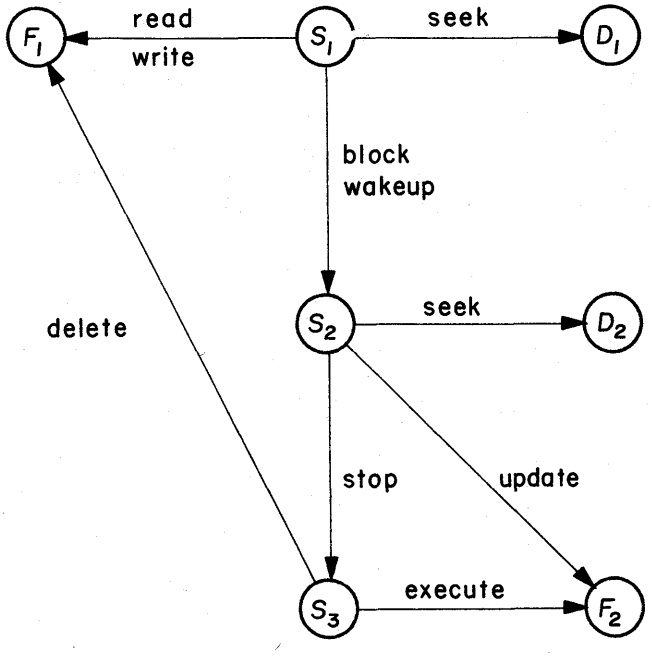

Figure 2—Access diagram

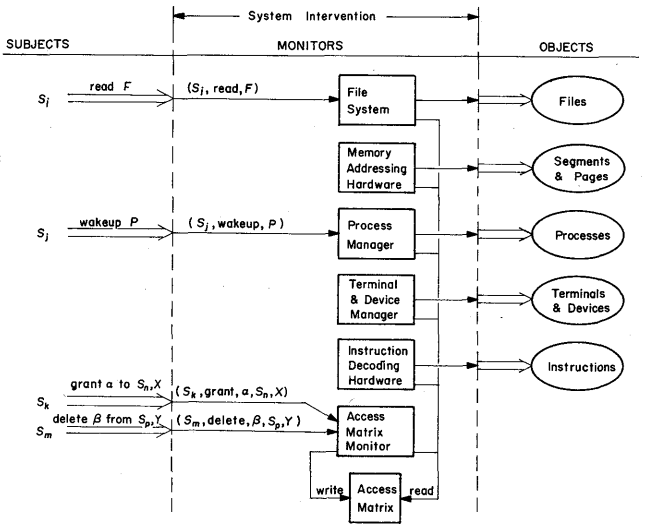

Figure 3—Organization of protection system

monitors at the times accesses are attempted. Figure 3 shows the organization of the protection system. The mechanisms between the dashed lines of that diagram are invisible to subjects—subjects direct their references to objects, these references being intercepted and validated by the monitors of the protection system.

It is important to note that the identification number of a subject is system-provided in rule 2 above, and cannot be forged by S. That is, even if every subject knows the identification number of every other subject, there is no way for S to alter the fact that its identification number is presented to the monitor in rule 2; hence the monitor cannot be "tricked" into interrogating the wrong entry of A. Since no subject may access A, it is on this basis that we can develop proofs concerning the correctness of the protection system. The correctness of the protection system can be reduced to the correctness of the monitors.

The foregoing rules govern the use, by monitors, of the access matrix, once it has been specified. We require rules for changing the access matrix itself. These rules will be enforced by the monitor of the access matrix. Unlike the monitors of system objects, the access matrix monitor may modify the access matrix. In particular, it may transfer, grant, or delete access attributes on command of subjects and only on appropriate authorization. For this purpose, we introduce the access attributes 'owner' and 'control,' and the notion of a *copy flag* (denoted by asterisk), and the rules R1-R3 of Table I to be implemented by the access matrix monitor.

TABLE I—Protection System Commands

Rule	Command (by S_o)	Authorization	Operation
R1	**transfer** $\left\{\begin{matrix}\alpha^* \\ \alpha\end{matrix}\right\}$ **to** S, X	'α^*' in $A[S_o, X]$	store $\left\{\begin{matrix}\alpha^* \\ \alpha\end{matrix}\right\}$ in $A[S, X]$
R2	**grant** $\left\{\begin{matrix}\alpha^* \\ \alpha\end{matrix}\right\}$ **to** S, X	'owner' in $A[S_o, X]$	store $\left\{\begin{matrix}\alpha^* \\ \alpha\end{matrix}\right\}$ in $A[S, X]$
R3	**delete** α **from** S, X	'control' in $A[S_o, S]$ or 'owner' in $A[S_o, X]$	delete α from $A[S, X]$
R4	$\omega :=$ **read** S, X	'control' in $A[S_o, S]$ or 'owner' in $A[S_o, X]$	copy $A[S, X]$ into ω
R5	**create object** X	none	add column for X to A; store 'owner' in $A[S_o, X]$
R6	**destroy object** X	'owner' in $A[S_o, X]$	delete column for X from A
R7	**create subject** S	none	add row for S to A; execute **create object** S; store 'control' in $A[S, S]$
R8	**destroy subject** S	'owner' in $A[S_o, S]$	delete row for S from A; execute **destroy object** S

Rule 1 permits a subject to transfer any access attribute it holds for an object to any other subject, provided the copy flag of the attribute is set, and it may specify whether the copy flag of the transferred attribute is to be set; in Figure 4, for example, S_1 may place 'read*' in $A[S_2, F_1]$ or 'write' in $A[S_3, F_1]$, but it may not transfer its ability to block S_2 to any other subject. The purpose of the copy flag is preventing an untrustworthy subject from wantonly giving away access to objects. Rule R2 permits a subject to grant to any subject access attributes for an object it owns; in Figure 4, for example, S_1 can grant any type of access for S_2 to any subject, or any type of access to device D_2 to any subject. Rule R3 permits a subject to delete any access attribute (or copy flag) from the column of an object it owns, or the row of a subject it controls; in Figure 4, for example, S_1 can delete any entry from columns S_2 or S_3, or from row S_3. In order to facilitate these rules, we require that 'control' be in $A[S, S]$ for every subject S and we include rule R4, which permits a subject to read that portion of the access matrix which it owns or controls. Rules R5-R8, which govern the creation and deletion of subjects and objects, will be discussed shortly.

It should be noted that a subject may hold 'owner' access to any object, but 'control' access only to subjects. For reasons to be discussed shortly, we shall assume each subject is owned or controlled by at most one other subject, though other multiply-owned non-subject objects are allowable. If a subject has 'owner' access to another subject, the former can grant itself 'control' access to the latter, so that 'control' is implied by 'owner.' It is undesirable for a subject to be 'owner' of itself, for then (by Rule R3) it can delete other subjects' access to itself.

The rules R1-R4 and access attributes shown in Figures 1 and 4 should be interpreted as examples of rules which can be provided for the purposes intended. One might, for example, introduce a "transfer-only"

	S_1	S_2	S_3	F_1	F_2	D_1	D_2
S_1	control	owner block wakeup	owner control	read* write*		seek	owner
S_2		control	stop	owner	update	owner	seek*
S_3			control	delete	owner execute		

Figure 4—Extended access matrix

mode by postulating a "transfer-only copy flag"; if this flag is denoted by the symbol •, then in R1 the command to transfer α (or $\alpha\bullet$) from S_0 to S for object X would be authorized if $\alpha\bullet$ were in $A[S_0, X]$ and would cause $\alpha\bullet$ to be deleted from $A[S_0, X]$ and α (or $\alpha\bullet$) be placed in $A[S, X]$. This is useful if one wants to limit the number of outstanding access attributes to a given object—e.g., if it is required that each subject be owned by at most one other subject, or that a given object be accessible to a limited number of subjects. One can also define "limited-use" access attributes, of the form (α, k); access in manner α is permitted only if $k>0$, and each use of this attribute decreases k by one.

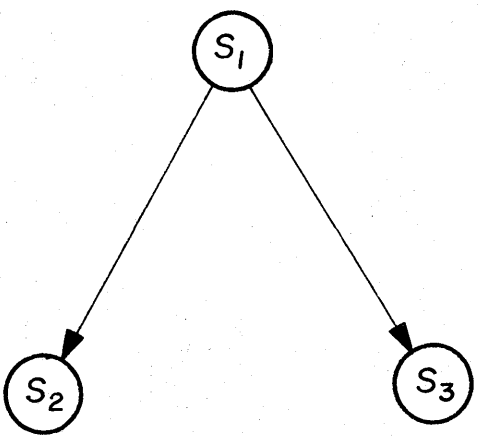

Figure 5—Ownership diagram

Creation and deletion of subjects and objects

The creation of a non-subject object, e.g., a file, is straightforward, consisting of adding a new column to the access matrix. The creator subject executes a command (rule R5 of Table I) and is given 'owner' access to the newly created object; it then may grant access attributes to other subjects for the object according to rule R2. The destruction of an object, permitted only to its owner, corresponds to deleting the column from the access matrix (rule R6).

Creating a subject consists of creating a row and column for the new subject in the access matrix, giving the creator 'owner' access to the new subject, and giving the new subject 'control' access to itself (rule R7). The destruction of a subject, permitted only to its 'owner,' corresponds to deleting both the row and the column from the access matrix (rule R8).

According to our definition of subject as a pair (process, domain), there are two ways a subject can come into existence: a given process switches from one domain to another, or a new process is created in some domain. (Similarly, there are two ways a subject can go out of existence.) With respect to the former problem, a process must have authorization to switch domains since domain-switching entails a gain or loss of access attributes. Specifically, if process P wishes to switch from domain D_1 to domain D_2, subjects $S_1=(P, D_1)$ and $S_2=(P, D_2)$ would exist in the system; the switch by P from D_1 to D_2 is permitted only if authorization (e.g., access attribute 'switch') is in $A[S_1, S_2]$. In practice, it would be more efficient to regard a subject as being dormant (rather than nonexistent) when its process is operating in another domain—i.e., if P switches from D_1 to D_2, subject (P, D_1) becomes dormant and subject (P, D_2) becomes active, but the switch neither creates a new subject nor destroys an old one. (The idea is very much like that of transferring control between coroutines.) In practice, therefore, rules R7 and R8 would be invoked only when a process is created or destroyed.

We suppose that the system enforces the requirement that each subject be owned by at most one other subject—i.e., an owner may not grant 'owner,' and 'owner' is either untransferable or is transfer-only. In this case the relation 'owner' defines naturally a tree hierarchy of subjects. Figure 5 shows the subject tree for the access matrix of Figure 4. If S_1 is 'owner' of S_2, S_2 is *subordinate* to S_1. By rule R1, a subject can transfer only a subset of its access attributes to a subordinate; by rule R2, it can grant any access for a subordinate to any other subject; by rule R3, it can delete any access attribute from a subordinate; and by rule R4, it can discover what access attributes a subordinate has. We have carefully avoided including any provision for a subject to acquire for itself (copy) an access attribute it does not have but which a subordinate has obtained. This can be incorporated, if desired, as an extension to the model.

It is not overly restrictive to require that subjects be members of a hierarchy, the utility of hierarchies having been demonstrated adequately in practice. This mechanism can be used to ensure that a subordinate, upon creation, has no more privileges than its creator. It can be used to define a protocol for reporting protection violations (and indeed any other fault condition), for the violation can be reported to the immediate superior of the subject generating it. It can be used as an aid in resource allocation and accounting, for a subordinate can be granted only a subset of the resources held by its creator. Considerations such as these motivated the design of the RC4000 system.[2] Our subject

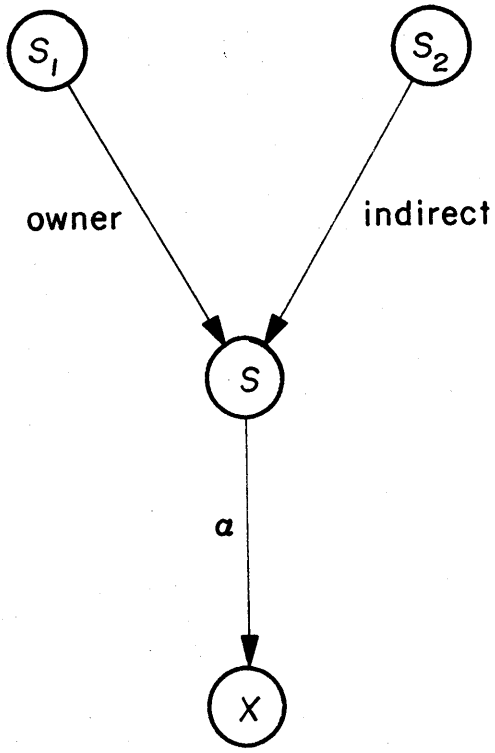

Figure 6—Sharing an untrustworthy subsystem

hierarchy is in fact the same as the sphere-of-protection hierarchy proposed by Dennis and Van Horn,[4] or the domain hierarchy used by Lampson.[15] If one regards a process operating in the context of a file directory as a subject, then the directory hierarchy of Multics defines another example of a subject hierarchy.[1] As will be seen, the rings of Multics also define a subject hierarchy.[9,18]

A subject in a hierarchy without an owner is called a *universal subject*. For convenience, we assume there is only one universal subject, and it has every possible access attribute to every object. When a user wishes to sign on the system, he communicates with the universal subject. Having convinced itself of the user's identity, the universal subject will create a subordinate subject for the user, and will initialize the row of the access matrix corresponding to that subject with attributes derived from some data file private to the universal subject.

Sharing untrustworthy subsystems

It is clear from the above that, if subject S_1 wishes to share an object X with subject S_2, it may do so without risk of S_2 acquiring access to any other of its objects. But if X is a subject, we must ensure that S_2 is allowed access to objects X can access and that X, if an active entity, cannot access objects of S_2 without authorization.

Suppose subject S_1 owns a subsystem S which it wishes to share with subject S_2. However, S_2 does not trust S, and S_1 may wish to revoke S_2's access to S at any time. We introduce the access attribute 'indirect' for use among subjects: if one subject has 'indirect' access to another, the former may access objects in the same manner as the latter, and it may read but not acquire for itself access attributes of the latter; the subject which owns the latter may (rule R3) revoke the former's 'indirect' access at any time. The sharing of S among S_1 and S_2 according to this idea is illustrated in Figure 6. The rules of operation of object monitors must be modified slightly to allow for indirect access:

1. S_2 initiates indirect access to X through S in manner α;
2. The system supplies $(S_2, \alpha, S\text{-}X)$ to the monitor of X;
3. The monitor of X interrogates the access matrix to determine if 'indirect' is in $A[S_2, S]$ and α is in $A[S, X]$; if so access is permitted, otherwise it is denied and a protection violation occurs.

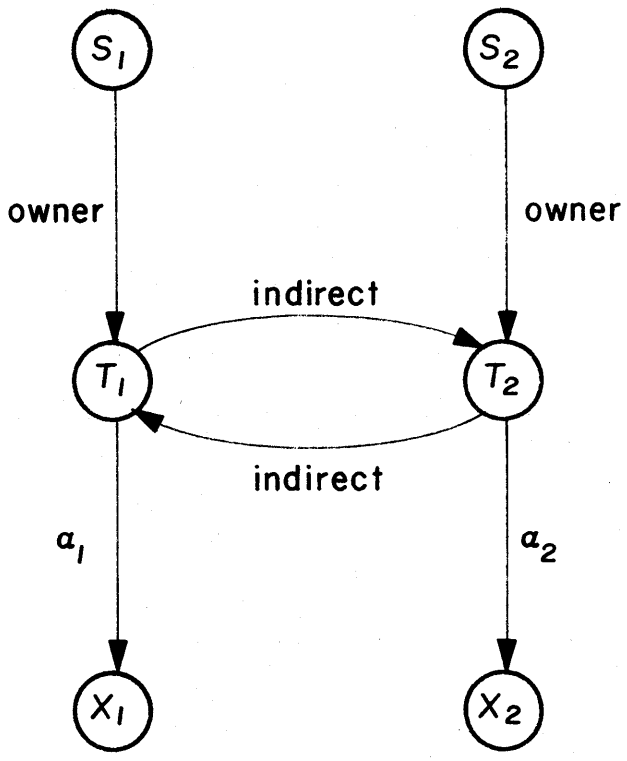

Figure 7—Cooperation between mutually suspicious subsystems

An example of the type of sharing in Figure 6 is the "debugging problem," in which S_1 wishes S_2 to debug S. Here S_2 needs complete access to all objects accessible to S, but S must be denied access to S_2 or its other objects. Another example of the same type of sharing is the "grading program" problem, where S_1 corresponds to a student who is submitting a program S to an instructor-program S_2 for grading.

A more complicated example involves mutually suspicious subsystems. Suppose S_1 and S_2 own respectively subsystems T_1 and T_2; T_1 and T_2 are to cooperate, but neither trusts the other. For example, T_1 might be a proprietary subroutine which S_1 wants executed, but not read, by others; and T_2 might be a data management system which accesses certain files owned by S_2. Figure 7 shows how the cooperation can be arranged. Observe that T_1 may access only the objects of S_2 (such as X_2) which are accessible to T_2, but no others, and that S_2 may revoke T_1's access to T_2 at any time. (The converse is true for T_2 and S_1.) Observe that T_1 can only use, but not acquire for itself, access attributes of T_2.

We have now developed the necessary basis for a protection theory, allowing us to view the protection state of a system as being defined dynamically by its access matrix. We have shown how this model allows for protection at the fifth level discussed in the introduction. The utility of the abstractions, both in understanding current protection systems and in formulating future "ideal" protection systems will be discussed shortly, after we have considered the notion of correctness of a protection system.

CORRECTNESS, SHARING, AND TRUST

To prove that a protection model, or an implementation of it, is correct, one must show that a subject can never access an object except in an authorized manner. Two things must be proved: any action by a subject which does not change the protection state cannot be an unauthorized access; and any action by a subject which does change the protection state cannot lead to a new protection state in which some subject has unauthorized access to some object.

With respect to the first, given the correctness of each monitor, it follows that the attachment by the system of the identification number of a calling subject to each reference makes it impossible for a subject to gain unauthorized access to an object. The most important requirement in the argument is thus the assumption that the system attaches a nonforgeable identification number to each attempted access. It is necessary to prove that the operating system attaches the identification number correctly, the monitors interrogate the correct entry in the access matrix, and no monitor (except the access matrix monitor) alters the contents of the access matrix.

With respect to the second, it is clear that each application of rules R1-R8 produces a new protection state in the manner specified, assuming the correctness of the access matrix monitor. There are, however, various aspects of trust implicit in rules R1-R8. Specifically, if the owner of a given object is not careful, it is possible for untrustworthy subjects, acting singly or in collusion, to grant some subject access to that object, access not intended by the object's owner. Since the model cannot deal satisfactorily with problems relating to trust, this will demonstrate the need for external regulations or laws to complement the technical solution implemented by the model. We shall explore this point in the paragraphs following.

Consider the transferring and granting rules (R1 and R2). Suppose subject S_1 has created an object X and granted various types of access for X to subjects $S_2, \ldots, S_n$. Suppose further that S_1 intends that, under no circumstances, should S_0 gain access to X. He can do this by avoiding granting any access attributes to S_0, but he must also grant attributes with the copy flags off to any subject among $S_2, \ldots, S_n$ who he feels might violate trust and grant access for X to S_0 indirectly. In other words, an understanding must exist between S_1 and subjects receiving access to X that S_0 is not to receive, either directly or indirectly, any access to X. Only when these considerations are understood and the requisite trust is present should S_1 be willing to pass a copyable attribute. With this, rules R1 and R2 are correct.

On the basis of the foregoing argument, we conclude that the protection system is correct and will operate exactly as intended among *trustworthy* subjects. Untrustworthy subjects cannot be dealt with completely by mechanisms of the protection system. External regulation, together with a system for detecting and reporting violations, is required.

Consider the subject creation rule (R7). The correctness of the environment of the created subject is founded on the assumption that the creator's environment is correct, since a subordinate subject cannot be initialized with any access attribute not held by its creator. Thus the correctness of any subject's environment derives ultimately from the universal subject's having created a correct environment for a user when he signs on the system. Here is where the problem of false identification enters. An arbitrarily complicated question-and-answer procedure can be developed to

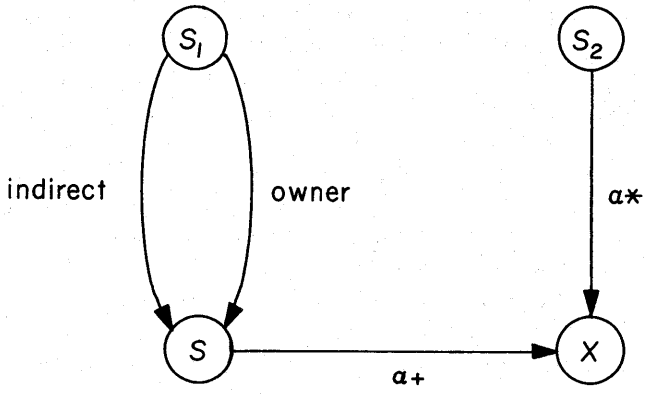

Figure 8—Preventing unwanted indirect access

establish the identity of a user to within any reasonable doubt, but it cannot be done absolutely. Again, a system for detecting and reporting violations of this type is required, and external regulations must deal with offenders.

As further illustration of the role of trust in the model, we consider four instances of trust's being implicit in the interpretation of access attributes themselves. First, consider the 'read' attribute. Reading is an unusually powerful operation, as it implies, for example, the ability to read and copy a file. It is then possible for one subject to distribute access attributes for the copy freely to other subjects, even to subjects who were not authorized to read the original. In this sense, the 'read' attribute is equivalent to the 'read*' attribute.

Second, consider the 'indirect' attribute, and refer to Figure 8. Subject S_2 has α^* access to X and wishes to transfer α to S. However, the owner S_1 of S may grant itself 'indirect' access to S and so gain α access to X, even though this may not have been intended by S_2. There are several ways to solve this:

(1) Define an access mode, denoted by the symbol '+,' which cannot be used in an indirect manner; thus $\alpha+$ is not only nontransferable, but it is usable only by the subject to whom it was granted.† This is shown in Figure 8.
(2) Do not allow an owner to grant itself 'indirect' access to a subordinate subject.
(3) Make α a "limited-use" attribute so that the exposure of X to the owner of S is of limited duration. It is not clear that (2) or (3) are viable solutions.

Third, consider the 'owner' attribute. A subject creating an object is given 'owner' access to that object, indicating that it may grant access attributes for that object to other subjects. It is possible in our model for the 'owner' attribute of a nonsubject object to be transferred, and thus multiple ownership may arise. Prior arrangements must exist among the owners, else contradictory actions may result—e.g., one owner grants access the others do not want granted. Such difficulties can be avoided by allowing only one owner, e.g., 'owner' is either untransferable or transfer-only but it is not clear whether this is a desirable or useful solution.

Fourth, consider rule R3, which permits the owner of an object to revoke arbitrarily any access held by any nonsubordinate subject to that object. This ability is open to question. Vanderbilt argues that no such ability is warranted as, in effect, the owner of an object and those to whom access has been granted have entered a contract[19]—i.e., the nonowners presumably have used the object in their programs and depend on its presence. Lampson, on the other hand, argues that, for the proper implementation of cooperation among mutually suspicious subsystems, absolute power of revocation is warranted.[16] The latter view is reflected in our model.

These considerations illustrate that trust is a distinctly nontechnical concept. They reemphasize an earlier point, viz., the complete solution to the protection problem must strike a balance between technical and nontechnical issues. More research is required on both issues, especially on methods of detecting and reporting violations and on coordinating external regulations with internal protection mechanisms.

Two other ways of breaching the security of protection systems exist, but do not fall in the province of the model. First is the "wire-tapping" issue. We consider this a solved problem, since coding techniques exist that render it essentially impossible for anyone tapping a data line to decode what he finds there. Second is the "system-error" issue. What if a system error changes the access matrix or the identification number of a subject or other object? Can a subject gain access it did not have before the error? The problem can be solved by appropriate use of coding techniques for subject and object identification numbers and for access attributes; in particular, it is possible to use error-correcting codes, and to arrange that the probability of an identification number (or access attribute) being changed to another valid identification number (or access attribute) is arbitrarily small.

† We have defined altogether four modes for access attributes: α^*, $\alpha\cdot$, α, and α^+, where α^+ is the most restricted mode of α and α^* the least.

SYSTEM IMPLICATIONS

The protection model presented above does not resemble any particular system with which the reader is familiar, although it likely contains concepts with which he is. By considering implementations of the access matrix, one can show that many protection systems are practical examples of the theory.

Storing the access matrix

Since the access matrix is likely to be sparse, it would be impractical to implement it directly as a two-dimensional matrix. A more efficient implementation could store the access matrix as a table of triples $(S, X, A[S, X])$—i.e., the nonempty entries of A. Since most subjects and objects would not be active at any given time, it would be unnecessary to store all the triples in fast memory at once. Moreover, it is often necessary in practice to be able to determine what object a given subject can access (or what subject can access a given object); a simple table of triples has too little structure to permit efficient search procedures to be implemented. Therefore, this method is unlikely to be practical, especially when the number of subjects and objects is large.

There are, however, at least three practical implementations. The first uses the idea of storing the access matrix A by rows, i.e., with each subject S is associated a list of pairs $(X, A[S, X])$, each pair being called a *capability* and the list a *capability list* (C-list).[4] A C-list represents all the objects that a subject can access, together with the authorized modes of access. Following this approach, one may regard a C-list as defining the environment (domain) of a process, a subject as a pair (process, C-list), and the operation of switching domains as switching to a new C-list.[4,16] Because the protection system allows only authorized operations on C-lists, possession of a capability by a process is *prima facie* proof that the process has access to an object.[15]

A second approach uses the idea of storing the access matrix by columns, i.e., with each object X is associated a list of pairs $(S, A[S, X])$. In Multics, such a list is called an *access control list* when the objects are segments;[1,18] in the Cambridge multiple-access system, it is called an *authority list*.[20]

A third approach represents a compromise between the C-list and the access-control list implementations. Suppose a set of subjects desires access to a certain set of objects. In the C-list of each subject will appear entries (X, K) where X is an object name and K a *key*. Associated with the set of objects is a *lock list* containing entries of the form (L, α) where L is a *lock* and α an access attribute. The monitor of the set of objects, upon detecting that S is attempting to access X in manner β, will obtain an (X, K)-pair from the C-list of X; it then will search the lock list and permit the access only if it finds an (L, α)-pair for which $K=L$ and $\alpha=\beta$ (i.e., the key fits a lock that, when opened, releases the desired access attribute). In this system, the owner of an object X can distribute α access to X simply by granting (X, K) pairs to various subjects and placing (K, α) in the lock list. He can revoke this instance of α access to X simply by deleting (K, α) from the lock list (i.e., changing the lock); in this case the outstanding (X, K) pairs will be invalidated. It is possible for an owner to create various keys and locks for the same access attribute α—e.g., he can grant (X, K_1) to some subjects and (X, K_2) to others, placing both (K_1, α) and (K_2, α) in the lock list. This resembles the system of "storage keys" used in IBM System/360[12] with one important difference—viz., there is no concept of a C-list and it is possible for certain subjects to forge storage keys.

Efficiency

The viability of the model in practice will depend on the efficiency of its implementation. With respect to this, a few observations can be made. (1) Access attributes would not be represented as strings. An entry $A[S, X]$ of the access matrix might be represented as a binary word whose ith bit is 1 if and only if the ith access attribute is present. A corresponding binary word could be used to store the copy flags. (2) According to the rules, there is nothing to prevent one subject from granting another unwanted or unusable access attributes. Although this does not alter the correctness of the model, a practical implementation may require some mechanism to limit this effect. (3) The identification number of a subject can be stored in a protected processor register where it is quickly accessible to object monitors. (4) The implementation of the rules for creating and deleting columns and rows of the access matrix must be tailored to the manner in which the matrix is stored. In the C-list implementation (storage by rows), for example, deleting a row corresponds to deleting a C-list, but deleting a column would imply a search of all C-lists. In this case, it would be simpler to invalidate the name of the object whose column is deleted and use a garbage collection scheme to remove invalid capabilities from C-lists. (5) The C-list implementation is inherently more efficient than the access control list implementation because a C-list must be

stored in fast-access memory whenever a subject is active whereas an object may not be active often enough to warrant keeping its access control list in fast-access memory at all times.

It is worth a special note that the mapping tables of a virtual memory system are a highly efficient form of the C-list implementation for memory protection. A separate table is associated with each process, and each entry of such a table is of the form (X, Y, α) where X is the name of a segment or page, Y is its location in main memory, and α specifies the types of access permitted to X (typically, α is some combination of 'read,' 'write' and 'execute'). In fact, the C-list proposal by Dennis and Van Horn[4] was motivated by generalizing the idea of mapping tables to include entries for objects other than segments. The point is: the same techniques used to make virtual memory efficient can be generalized to make the C-list implementation efficient. The requisite hardware has been described by Fabry[6] and Wilkes.[20] It is interesting to note that Multics, even though it centers on the access control list implementation, uses a form of the C-list implementation (the "descriptor segment") as well, the appropriate entries of the access control list being copied into the descriptor segment as needed.[1]

Choosing subjects and objects

The model assumes that an access attribute applies uniformly to an object and all its components, but allows components of objects to be objects themselves. It is possible, for example, to regard a file as an object and some of its records as objects too. Thus the designer has great flexibility in the choice of objects.

By taking a subject to be a (process, domain) pair, the model forces all components of a process to have identical access privileges to objects. Thus, whenever it is necessary to grant components of a process differing access privileges, it is necessary to define different subjects for each component. To illustrate this point, suppose a process P operating in domain D consists of subprocesses $P_1, \ldots, P_n$ which must be accorded differing access powers. Since the domain defines the access powers of a process, the subjects $(P_1, D), \ldots, (P_n, D)$ would have identical access privileges. Instead, the system would have to allow the definition of subjects $(P, D_1), \ldots, (P, D_n)$ where domain D_i is effective while subprocess P_i is executing; only then can the subprocess P_i of P be accorded differing access privileges.

As further illustration of the latter problem, consider a system implementing segmentation, in which a process P is uniquely and permanently associated with a name space described by a segment table T; i.e., each subject is a pair (P, T). Suppose $T = [(s_1, \alpha_1), \ldots, (s_n, \alpha_n)]$ where s_i is the ith segment and α_i defines the access P has to s_i. If now we want to debug a new segment s_{n+1}, there is no way to change P's access power when it enters segment s_{n+1}. In terms of Figure 6, it is not possible in this system to put the segment being debugged at a subordinate level. One solution would be to place s_{n+1} in its own name space with segment table T' and process P', so that the subject (P', T') being debugged is subordinated to the debugger (P, T). This cannot be done very efficiently, as software-implemented message-transmitting facilities would have to be used for communicating between the processes P and P'. The problem could have been avoided in the first place if it were possible to define a series of domains $T_i = [(s_i, \alpha_i)]$ and subjects $S_i = (P, T_i)$ so that S_{n+1} can be made subordinate to $S_1, \ldots, S_n$. This is the approach suggested by Evans and Leclerc.[5]

Existing systems

We examine now several important systems in the light of the model. For each system we will identify the elements of the model, points of inefficiency, and points of insecurity (if they exist).

One of the most common systems in use is IBM's OS/360.[11] In this system, objects typically are files, tasks, and instructions. The domains correspond to pairs (mode, block) where "mode" refers to the supervisor/problem-program machine states and "block" refers to a block of the multiprogramming partition of main memory. A subject is then a triple (task, mode, block). One of the principal elements of the model is not present, viz., the idea of attaching a subject identifier to each attempted access; the storage-key mechanism in effect does this for access to objects in memory, but there is no such notion associated with attempts to switch domains or to reference files. It is possible for a task to guess a supervisor call number and issue a supervisor call (SVC) with that number, thereby gaining unauthorized access to the supervisor domain. It is also possible for a user to guess a file name and access it through the job control language.

OS/360 is representative of many current commercial systems in the lack of an adequate protection system and underlying protection theory. The limitations above indicate the OS/360 does not even provide protection at the lowest level mentioned in the Introduction, i.e., it cannot guarantee absolutely that each user is protected from the ravages of other users. The two systems described below (RC4000 and Multics) do embrace

the concepts of the model, concepts which must be embraced widely in the commercial systems before they will be capable of providing protection at the level required for sharing mutually suspicious subsystems. There are, of course, other systems which embrace the concepts of the model, but which cannot be discussed here for lack of space; these include the CAL system at Berkeley,[14] the capability hardware proposed by Fabry at Chicago[6] and Wilkes,[20] Project SUE at the University of Toronto,[8] and the IDA system.[7]

The RC4000 system is one example of a commercial system having a protection system incorporating all the elements of the model.[2] The objects of this system include the usual repertoire of files, devices, instructions, and processes. In this system, each process defines a domain (i.e., subjects are identical to processes), and a "father-son" relation defines a hierarchy of processes. Each object is associated with exactly one domain; therefore each process has exclusive control over certain objects, and it must request other processes to access other objects on its behalf. (In this sense, each process is the monitor of its objects.) For this purpose, the system implements a message-transmission facility and associates with each process a message buffer in which are collected all messages from other processes. A process P wishing to send message α to process P' uses a system call

send message (α, P'),

the effect of which is to place the pair (P, α) in the message buffer of P'. Note that this is precisely the notion of attaching the identification number of a subject to an attempted access, if we take the view that each process is an object monitor and a message is an attempted access to some object controlled by the monitor. This system provides a great deal of flexibility: each process plays the role of a monitor; the number of monitors is variable; the definition of an object can be dynamic since the programming of a process can be used to implement "virtual objects" which are accessible to other processes via the message-transmission facility; and each process can determine the authenticity of each message it finds in its message buffer. The limitations of this system include: the message facility, being implemented in software, is of limited efficiency; there is no way to stop a runaway process, as the system cannot force a process to look in its message buffer; and a possibly elaborate system of conventions may need to be adopted in order that receivers can learn the identification numbers of senders. Further discussions of these points are found in References 3 and 15.

The Multics system at MIT is an example of a non-commercial system which implements the elements of the protection model. Typical objects are segments and processes. The virtual memory is the monitor of the segments and the traffic-controller the monitor of the processes. The domains of this system are defined by pairs (ring, base) where "ring" refers to integers 0, 1, 2, ... and "base" is a pointer to the descriptor segment (segment table) of the name space. Each process is associated with exactly one name space so the base can be used as its identification number. Each segment s has associated with it a ring number n_s; if process P is executing instructions from segment s in name space with base b, the subject is the triple (P, n_s, b). Observe that a control transfer from segment s to segment t for which $n_s \neq n_t$ defines a change of subject. The access privileges of a subject are considered to diminish as its ring number increases; in other words, for given P and b, the ring numbers 0, 1, 2, ... define a linear hierarchy of subjects $(P, 0, b)$, $(P, 1, b)$, $(P, 2, b)$, ... Associated with segment s are three pairs of integers

(r_1, r_2)—the read bracket, $r_1 \leq r_2$

(w_1, w_2)—the write bracket, $w_1 \leq w_2$

(e_1, e_2)—the execute bracket, $e_1 \leq e_2$

(These numbers can be stored in the segment table along with flags indicating whether each type of access is permitted at all. In Multics, only three of the seven integers associated with a segment are distinct, so only three integers need be stored to represent the three access brackets and ring number.) In terms of a subject hierarchy, the notion of an access bracket for access attribute α of an object means simply, for a certain range of levels above and below the owner of the given object, a superior or subordinate subject within the range is automatically granted α access to the object. It is an extension of the concept of access attribute. The access matrix is stored statically in the form of access control lists associated with objects (segments). This information is copied dynamically into the descriptor segment as segments are referenced for the first time, so that the descriptor segment is a form of a capability list. A subject (P, n_s, b) may access in any manner any segment t listed in the descriptor segment when $n_t > n_s$; furthermore, if $n_t \leq n_s$ it may read, write, or execute (transfer control to) t if n_s falls in the proper attribute access bracket of segment t. If it wishes to transfer control to a segment t whose execute bracket is (e_1, e_2) but $e_2 < n_s$, it may do so by transferring to a "gate" (protected entry point) of segment t; the attempt to do so generates an interrupt which invokes a "gatekeeper" that permits transfers only to gates. A

full description of the mechanism as outlined above can be found in References 9 and 18.

CONCLUSIONS

We have carefully tried to limit the complexity of the model, omitting from its basic structure any feature which is either controversial or not completely understood. It is possible to extend the model in any or all of the following ways. (1) A subject which creates an object can specify that the object is permanent, so that the universal subject can make that object available for future reincarnations of that subject. Alternatively, a subject can specify that, if it dies, ownership of objects it owns can revert to a superior subject. (2) In a subject hierarchy, superior subjects can be regarded as dormant when their processes are not executing, so that subordinates can send messages which, for example, request additional access attributes. (3) Since a subordinate subject may acquire access attributes not held by its superiors, rules may be required specifying when superiors can acquire such attributes for themselves; Lampson allows for this.[15] (4) Given the possibility of multiply-owned objects, rules may be required to permit owners to guard their access attributes from deletion by other owners. (5) If one adopts the view that a subject and all its subordinates constitute a "family," [2] one can make a case for allowing a subject to transfer an access attribute to a subordinate even if the copy flag is not set.

The considerations of the section on System Implications indicate that most current commercial systems do not implement a complete protection mechanism, even at the lowest level described in the Introduction. The most serious defect is the lack of completeness when a process changes protection environments. Although the incompleteness of these systems could be rectified by (probably extensive) software modifications, a complete protection system cannot hope to be efficient unless the basic elements of the model are implemented in hardware. The requisite hardware has been described in many forms—by Fabry,[6] by Schroeder and Saltzer,[18] and by Wilkes[20]—and is in fact quite straightforward.

It can be noted that the "average" user, who employs only system compilers and library routines, is unlikely to break the security of the system if only because compilers and library routines will not cause the loopholes to be exercised. This does not mean attention should not be paid to structuring the hardware so that even the most enterprising and malicious user cannot break security. It takes only one such user to compromise another's privacy which, once lost, may be irrecoverable. On this basis, the cost of hardware and software development required to achieve efficient implementations of protection is of the utmost value.

We began the paper with the statement that the technical approach to protection had not been treated adequately to date in the literature. Hopefully, the discussion here has shown that it is possible to state the problem on an abstract level where the elements of protection can be isolated, where methods of proving the correctness of a protection system can be formulated, where drastically different physical implementations of the one model can be compared and evaluated, and where the nontechnical issues required to complement the technical ones can be identified. The discussion here has been intended as an example of what is possible in a model; we are under no delusions that this is the only model or that this is the best model. Our preliminary work has indicated that the abstractions formed in the modeling process are useful in themselves and that the model provides a framework in which to formulate precisely previously vague questions. We hope that this discussion will motivate others to undertake additional research in this area. Much needs to be done.

ACKNOWLEDGMENTS

We are indebted to Michael Schroeder for pointing out the seven levels of sharing mentioned in the Introduction, and to Richard Holt, Butler Lampson, Richard Muntz, and Dennis Tsichritzis for many helpful discussions and insights.

REFERENCES

1 A BENSOUSSAN C T CLINGEN R C DALEY
The MULTICS virtual memory
Proc 2nd ACM Symposium on Operating Systems Principles Oct 1969 pp 30-42

2 P BRINCH-HANSEN (Ed)
RC-4000 software multiprogramming system
A/S Regnecentralen Copenhagen April 1969

3 P J DENNING
Third generation computer systems
Computing Surveys 3 4 Dec 1971

4 J B DENNIS E C VAN HORN
Programming semantics for multiprogrammed computations
Comm ACM 9 3 March 1966 pp 143-155

5 D C EVANS J Y LECLERC
Address mapping and the control of access in an interactive computer
AFIPS Conf Proc 30 Spring Joint Computer Conference 1967 pp 23-30

6 R S FABRY
Preliminary description of a supervisor for a machine oriented around capabilities
ICR Quarterly Report 18 University of Chicago August 1968 Section I pp 1-97

7 R S GAINES
An operating system based on the concept of a supervisory computer
Comm ACM 15 3 March 1972

8 G S GRAHAM
Protection structures in operating systems
MSc Thesis Department of Computer Science University of Toronto August 1971

9 R M GRAHAM
Protection in an information processing utility
Comm ACM 11 5 May 1968 pp 365-369

10 L J HOFFMAN
Computers and privacy: a survey
Computing Surveys 1 2 June 1969 pp 85-104

11 *IBM System/360 operating system concepts and facilities*
IBM Report No GC28-6535 November 1968

12 *IBM System/360 principles of operation*
IBM Report No GA22-6821 September 1968

13 *IBM System/370 principles of operation*
IBM Report No GA22-7000 June 1970

14 B W LAMPSON
On reliable and extendable operating systems
Techniques in software engineering NATO Science Committee Working Material Vol II September 1969

15 B W LAMPSON
Dynamic protection structures
AFIPS Conf Proc 35 Fall Joint Computer Conference 1969 pp 27-38

16 B W LAMPSON
Protection
Proc Fifth Annual Princeton Conference on Information Sciences and Systems Department of Electrical Engineering Princeton University Princeton New Jersey 08540 March 1971 pp 437-443

17 S MOTOBAYASHI T MASUDA N TAKAHASHI
The Hitac 5020 time sharing system
Proc 24th ACM Nat'l Conference 1969 pp 419-429

18 M D SCHROEDER J H SALTZER
A hardware architecture for implementing protection rings
Comm ACM 15 3 March 1972

19 D H VANDERBILT
Controlled information sharing in a computer utility
MIT Project MAC report MAC-TR-67 October 1969

20 M V WILKES
Time sharing computer systems
American Elsevier 1968