The Protection of Information in Computer Systems

JEROME H. SALTZER, SENIOR MEMBER, IEEE, AND MICHAEL D. SCHROEDER, MEMBER, IEEE

Invited Paper

Authorize

To grant a principal access to certain information.

Capability

To perform a particular function, such as reading or writing data.

Certify

To validate that a principal has the authority to access a particular object.

Complete isolation

A protection system that prevents a principal from gaining access to objects that are not directly related to the principal's authorized access.

Confinement

The restriction of a principal's access to only those objects that are necessary for the principal's authorized tasks.

Descriptor

A data structure that contains information about an object, such as its name, type, and access controls.

Discretionary

Controls that are based on the individual preferences of a principal, rather than on predefined rules.

Domain

A set of objects that are logically related and are managed by a single protection system.

Encipherment

The process of converting data into an unrecognizable form to protect its confidentiality.

Grant

To give a principal permission to access or modify certain objects.

Hierarchical control

A protection system that organizes objects into a hierarchical structure, with each object having access rights relative to its position in the hierarchy.

Glossary

Access

The ability to make use of information stored in a computer system. Access is usually measured in terms of read, write, execute, or modify privileges.

Access control list

A list of principals that are authorized to have access to a particular object.

Authenticate

The process of verifying the identity of a principal or other entity before granting access to protected resources.

The following glossary provides, for reference, brief definitions for several terms as used in this paper in the context of protecting information in computers.

The author is with Project MAC and the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. 02139.
each authorization is controlled by an
other authorization, resulting in a hier-
archical tree of authorizations.

Used to describe a protection system in
which each protected object has a list of
authorized principals.

User

Used imprudently to refer to the individ-
ual who is accountable for some identi-
fiable set of activities in a computer
system.

I. BASIC PRINCIPLES OF INFORMATION PROTECTION

A. Considerations Surrounding the Study of Protection

1) General Observations: As computers become better understood and more economical, every day brings new ap-
plications. Many of these new applications involve the stor-
ing and simultaneous use by several individuals.

The key concern in this paper is multiple use. For those ap-
plications in which all users should not have identical author-
ity, some scheme is needed to ensure that the computer sys-

tem implements the desired authority structure.

For example, in an airline seat reservation system, a reserva-
tion agent might have authority to make reservations and to
cancel reservations for people whose names he can supply. A
flight-boarding agent might have the additional authority to
print out the list of all passengers who hold reservations on
the flights for which he is responsible. The airline might wish
to withhold from the reservation agent the authority to print out
a list of reservations, so as to be sure that a request for a pas-
senger list from a law enforcement agency is reviewed by the
correct level of management.

The airline example is one of protection of corporate infor-
mation for corporate self-protection (or public interest, de-
pending on one's view). A different kind of example is an on-
line warehouse inventory management system that generates
reports about the current status of the inventory. These re-
ports not only represent corporate information that must be
protected from release outside the company, but also may
indicate the quality of the job being done by the warehouse
manager. In order to preserve his personal privacy, it may be
appropriate to restrict the access to such reports, even within
the company, to those who have a legitimate reason to be
judging the quality of the warehouse manager's work.

Many other examples of systems requiring protection of
information are encountered every day: credit bureau data on
banks; law enforcement information systems; time-sharing
service bureau; on-line medical information systems; and
government social service data processing systems. These
examples span a wide range of needs for organizational and
personal privacy. All have in common controlled sharing of
information among multiple users. All, therefore, require
some plan to ensure that the computer system helps imple-
ment the correct authority structure. Of course, in some
applications no special provisions in the computer system are
necessary. It may be, for instance, that an externally admin-
istered case of ethics or a lack of knowledge about computers ade-
quately protects the stored information. Al-
though there are situations in which the computer need pro-
vide no aid to ensure protection of information, often it is
appropriate to have the computer enforce a desired authority
structure.

The words "privacy," "security," and "protection" are
frequently used in connection with information-holding sys-
tems. Not all authors use these terms in the same way. This
paper uses definitions commonly encountered in computer
literature.

The term "privacy" denotes a socially defined ability of an
individual (or organization) to determine whether, when, and
to whom personal (or organizational) information is to be released.

This paper will not be explicitly concerned with privacy, but instead with the mechanisms used to help achieve it. The term "security" describes techniques that control who may use or modify the computer or the information contained in it. Security specialists (e.g., Anderson [6]) have found it useful to place potential security violations in three categories:

1) Unauthorized information release: An unauthorized person is able to read and take advantage of information stored in the computer. This category of concern sometimes extends to "traffic analysis," in which the intruder observes only the patterns of information use and from those patterns can infer some information content. It also includes unauthorized use of a proprietary program.

2) Unauthorized information modification: An unauthorized person is able to make changes in stored information—a form of sabotage. Note that this kind of violation does not require that the intruder see the information he has changed.

3) Unauthorized denial of use: An intruder can prevent an authorized user from referring to or modifying information, even though the intruder may not be able to refer to or modify the information. Causing a system "crash," disrupting a scheduling algorithm, or firing a bullet into a computer are examples of denial of use. This is another form of sabotage.

The term "unauthorized" in the three categories listed above means that release, modification, or denial of use is not contrary to the desires of the person who controls the information, possibly even contrary to the constraints supposedly enforced by the system. The biggest complication in a general-purpose remote-access computer system is that the "intruder" in these definitions may be an otherwise legitimate user of the computer system. Examples of security techniques sometimes applied to computer systems are the following:

1) labeling files with lists of authorized users,
2) verifying the identity of a prospective user by demanding a password,
3) shielding the computer to prevent interception and subversion of electromagnetic radiation,
4) enquiring information sent over telephone lines,
5) locking the room containing the computer,
6) controlling who is allowed to make changes to the computer system (both its hardware and software),
7) using redundant circuits or programmed cross-checks that maintain security in the face of hardware or software failures,
8) certifying that the hardware and software are actually implemented as intended.

It is apparent that a wide range of considerations are pertinent to the engineering of security of information. Historically, the literature of computer systems has more narrowly defined the term protection to be just those security techniques that control the access of executing processes to stored information. As an example of a protection technique is labeling of computer-stored files with lists of authorized users. Similarly, the fine authentication is used for those security techniques that verify the identity of a person (or other external agent) making a request of a computer system. An example of an authentication technique is demanding a password. This password conceives on protection and authentication mechanisms, with only occasional reference to the other equally necessary security mechanisms. One should recognize that concentration on protection and authentication mechanisms provides a narrow view of information security, and that a narrow view is dangerous. The objective of a secure system is to prevent unauthorized use of information, a negative kind of requirement. It is hard to prove that this negative requirement has been achieved, for one must demonstrate that every possible threat has been anticipated. Thus an expansive view of the problem is most appropriate to help ensure that no gaps remain in the system. The strategy. In contrast, a narrow concern with protection mechanisms, especially those logically impossible to defeat, may lead to false confidence in the system as a whole.

2) Functional Levels of Information Protection:

Many different designs have been proposed and mechanisms implemented for protecting information in computer systems. One reason for differences among protection schemes is their different functional properties—the kinds of access control that can be expressed naturally and enforced. It is convenient to divide protection schemes according to their functional properties. A rough categorization is as follows:

a) Unprotected systems: Some systems have no provision for preventing a determined user from having access to a piece of information stored in the system. Although these systems are not directly of interest here, they are worth mentioning since, as of 1975, many of the most widely used, commercial-available batch processing systems fall into the unprotected category—for example, the Disk Operating System for the IBM System 370 [9]. Our definition of protection, which excludes features usable only for mistake prevention, is important here because it is common for unprotected systems to contain a variety of mistake-prevention features. These may provide just enough control that an aware user of control is likely to be the result of a deliberate act rather than an accident. Nevertheless,

Some authors have widened the scope of the term "protection" to include mechanisms designed to limit the consequences of accidental mistakes in programming or in applying programs. With such definitions, even computer systems used to a single user might be called "protection mechanisms." The effect of a broader definition of "protection" would be to include in our study mechanisms that are deliberately bypassed by the user, on the basis that useful protection mechanisms are used to be as small and as detailed as possible. Such mechanisms might be considered "protection" if they were adoptable to apply to a situation in which a systematic attack by another user is to be anticipated. We will not insist on the narrow definition. Protection mechanisms are very useful in preventing mistakes, but mistake-prevention mechanisms that are deliberately bypassed have little value in providing protection. Another commonly coined term, not used in this paper, is "stability". We are using the term "security" to mean the security or reliability of information systems and computer services despite accidental failures of hardware, software, or programs. This, we think, is a much more appropriate term than "stability" because it is not clear what an inherently "stable" computer system is. We are using the term "security" to indicate that we are not considering the reliability of information systems and computer services despite accidental failures of hardware, software, or programs.

The broad view, encompassing all the considerations mentioned here and more, is taken in several current books [41-43].
It would be a mistake to claim that such systems provide any security.

b) Deceptive- or nothing-systems: These are systems that provide deception of users, sometimes moderated by total sharing of some pieces of information. If only Isians is provided, the user of such a system might just as well be using his own pen and computer, as far as protection and sharing of information is concerned. More commonly, such systems also have public sub-systems to which every user may have access. In some cases the public library mechanism may be extended to accept user contributions, but still on the basis that all users have equal access to the system. For the first generation of commercial computer sharing systems, provide a protection scheme with this level of function. Examples include the Dartmouth Time-Sharing System (DTSS) [10] and IBM's VM/CMS system [11]. There are unnumerable others.

c) Controlled sharing: Significantly more complex machinery is required to control explicitly who may access each data item stored in the system. For example, such a system might provide each file with a list of authorized users and allow an owner to distinguish several common patterns of use, such as reading, writing, or executing the contents of the file as a program. Although conceptually straightforward, actual implementation is surprisingly intricate, and only a few complete examples exist. These include M.I.T.'s Compatible Time-Sharing System (CTSS) [12], Digital Equipment Corporation's DECsystem-10 [13], System Development Corporation's Advanced Development Prototype (ADEPT) system [14], and Bolt, Beranek, and Newman's TENEX [15].

d) User-programmed sharing controls: A user may wish to restrict access to a file in a way not provided in the standard facility for controlling sharing. For example, he may wish to permit access only on weekdays between 9:00 A.M. and 6:00 P.M. Possibly, he may wish to permit access to only the named value of the data in a file. Maybe he wishes to require that the file be modified only if two users agree. For such cases, and in a myriad of others, a general purpose add-on is required to provide for user-defined protected objects and sub-systems. A protected sub-

system is a collection of programs and data with the property that only the programs of the subsystem have direct access to the data (that is, the protected objects). Access to those programs is limited to calling specific entry points. Thus the programs of the subsystem completely control the operation defined on the data. By constructing a protected subsystem, a user may develop any programmable form of access control to the objects he creates. Only a few of the more ad

vanced system designs have tried to permit user-specified pro-

tected subsystems. These include Honeywell's Multics [16], the University of California's CAL system [17], Bell Labora-

tories' UNIX system [18], the Berkeley Computer Corpora-

tion's BCC-500 [19], and two systems currently under con-

struction at Cambridge University [20], University of Califor-

nia, and HYDRA system of Carnegie-Mellon University [21]. Ex-

ploring alternative mechanisms for implementing protected subsystems is a current research topic. A specialized use of protected subsystems is the implementation of protection controls based on data content. For example, in a file of salaries, one may wish to permit access to all salaries under $15,000. Another example is permitting access to certain statistical aggregations of data but not to any individual data item. This area of protection raises questions about the possibility of discerning information by statistical tests and by examining indexes, without having direct access to the data itself. Protection based on content is the subject of a variety of recent or current research projects [22]-[23] and will not be explored in this tutorial.

e) Putting strings on information: The foregoing three levels have been concerned with establishing conditions for the release of information to an executing program. The fourth level of capability is to maintain some control over the user of the information even after it has been released. Such control is desired, for example, in releasing incoming information to a tax advisor, constraints should prevent him from passing the information to a firm which prepares mailing lists. The printed labels on classified military information declaring a document to be "Top Secret" are another example of a constraining of information after its release to a person authorized to receive it. One may not (without risking severe penalties) release such information to others, and the label serves as a notice of the restriction. Computer systems that implement such strings on information are rare and the mechanisms are incomplete. For example, the ADEPT system [14] keeps track of the classification level of all input data used to create a file; all output data are automatically labeled with the highest classification encountered during execution. There is a tendency that cats across all levels of functional capability: the dynamics of use. This refers to how one establishes and changes the specification of who may access what. At any of the levels it is relatively easy to enfor-
cement (and assign) systems that statically express a particular protection intent. But the need to change access authorization dynamically and the need for such changes to be re-
quested by executing programs introduces much complexity into protection systems. For a given functional level, most existing protection systems differ primarily in the way they handle protection dynamics. To gain some insight into the complexity introduced by program-directed changes to access authorization, consider the question "Is there any way that O'Hara could access file X?" One should check to see not only of O'Hara has access to file X, but whether or not O'Hara may change the specification of file X's accessibility. The next step is to see if O'Hara can change the specification of who may change the specification of file X's accessibility, etc. Another problem of dynamics arises when the owner revokes a user's access to a file while that file is being used. Letting the previously authorized user continue until he is "finished" with the information may not be acceptable, if the owner but suddenly realized that the file contains sensitive data. On the other hand, immediate withdrawal of authorization may severely disrupt the user. It should be apparent that provisions for the dynamics of use are at least as important as those for static specification of protection intent.

In many cases, it is not necessary to meet the protection needs of the person responsible for the information stored in the computer entirely through computer-aided enforcement. External mechanisms such as controls, ignorance, or brut
wire fences may provide some of the required functional capability. This discussion, however, is focused on the internal method of a set of protection mechanisms depending on the quality of a system to prevent security violations. In practice, producing a system at any level of functionality (except level one) that actually does prevent all unauthorized acts or processes to be extremely difficult. So- phisticated users or most systems are aware of at least one way to crash the system, denying other users authorized access to stored information. Penetration exercises involving a large number of different general-purpose systems as shown have shown that users can construct programs that can obtain unautho- rized access to information stored within. Even if various designers and implemented with security as an important ob- jective, design and implementation flaws provide paths that permit the intended access constraints. Design and construc- tion techniques that systematically exclude flaws are the topic of much research activity, but no complete method applicable to the construction of large general-purpose systems exists yet. This difficulty is related to the negative quality of the requirement to prevent unauthorized actions.

In the absence of such methodical techniques, experience has provided some useful principles that can guide the design and contribute to an implementation without security flaws. Here are eight examples of design principles that appear particularly to protection mechanisms.

1) Economy of mechanism: Keep the design as simple and small as possible. This well-known principle applies to any system but is especially important for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (but normal use usually does not include attempts to exercise improper access paths. As a result, tech- niques such as line-by-line inspection of software and physical examination of hardware that implements protection mecha- nisms are necessary. For such techniques to be successful, a small and simple design is essential.

2) Failure defaults: lease access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965, means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure sys- tem design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately con- sidered, so a default of lack of permission is safer. A design or implementation mistake is a mechanism that gives explicit permission tends to fail by refusing permission, a safe situa-

Design principles 2), 4), 5), and 7) are revised versions of material originally published in Communications of the ACM [24, p. 396]. © Copyright 1976. Association for Computing Machinery, Inc. Reprinted by permission.

In this paper we have attempted to identify original sources when- ever possible. Many of the seminal ideas, however, were widely spread by word of mouth or internal memo-form rather than by journal publication, and historical accuracy is sometimes difficult to obtain. In addition, some ideas related to protection were originally conceived in other contexts. In such cases, we have attempted to credit the re- searcher who first noticed their applicability to protection in computer systems, rather than the original inventor.

It may be an unknown fact that a simple design may be more vulnerable to the risk of massing large than a complex design is to the risk of massing small.

The principle of least privileges provides a rationale for the "need-to-know" concept of security, which is the concept that all information is an asset which must be protected. The concept that a system is only as strong as its weakest link is well established.

For example, if a system has been found to be vulnerable to a particular attack, the system administrator should take steps to prevent such an attack. This may involve changing passwords, restricting access to certain resources, or implementing additional security measures.

In addition, the principle of least privileges also helps to prevent information overload. If a user is given access to only the information that they need to perform their job, they will be less likely to misuse the information.

However, it is important to note that the principle of least privileges does not guarantee that a system is completely secure. It is only a tool to help organizations make informed decisions about what information to share and who should have access to it.
SALTER AND SCHROEDER: PROTECTION OF COMPUTER INFORMATION

1283

3. Summary of Considerations Surounding Protection: Briefly, then, we may outline our discussion to this point. The application of computers to information handling problems produces a need for a variety of security mechanisms. We are focusing on one aspect, computer protection mechanisms—the mechanisms that control access to information by executing programs. At least four levels of functional goals for a protection system can be identified: all-or-nothing systems, controlled sharing, user-programmed sharing controls, and putting strings on information. But at all levels, the provisions for dynamic changes to authorization for access are a severe complication. Since no one know how to build a system without flaws, we alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws: Economy of mechanism, fail-safe defaults, compile mediaion, open design, separation of privileges least privileges, common mechanism, and psychological acceptability. Finally, some protection designs can be evaluated by computing the resources of a potential attacker with the work factor required to defeat the system, and compromise record may be a useful strategy.

B. Technical Underpinnings

1) The Development Plan: At this point we begin a development of the technical basis of information protection in modern computer systems. There are two ways to approach the subject: from the top down, emphasizing the abstract concepts involved; or from the bottom up, identifying insights by studying example systems. We shall follow the bottom-up approach, introducing a series of models of systems as they are (or could be) built in real life. The reader should understand that on this point the authors' judgment differs from that of some of their colleagues. The top-down approach can be very satisfactory when a subject is coherent and self-contained, but for a topic still containing ed hoc strategies and competing world views, the bottom-up approach seems safer.

Our first model is of a multiuser system that completely isolates its users from one another. We shall then see how the logically perfect walls of this system can be lowered in a controlled way to allow limited sharing of information between users. Section II of this paper generalizes the mechanics of sharing, and shows two different models: the capability system and the access control list system. It then extends these two models to handle the dynamic situation in which authorizations
can change under control of the programs running inside the system. Further extensions to the models control the dy-
amic. The final model (only slightly explored) is of pro-
tected objects and protected subsystems, which allow arbi-
trary modes of sharing that are unanticipated by the system-
designer. These models are not yet fully developed enough to explain the precise details of interaction protection.

Our emphasis throughout the presentation is on direct
access to information (for example, using LOAD and STORE
instructions) rather than acquiring information indirectly (as
when calling a data base management system to request the
average value of a set of numbers supposedly not directly
accessible). Control of such access is the function of the pro-
tected subsystems developed near the end of the paper. Herein
lies perhaps the chief defect of the bottom-up approach, since
conceptually there seems to be no reason to distinguish direct
and indirect access, yet the detailed mechanisms are typically
quite different. The findings of a top-down approach based
on a message model that avoids distinguishing between direct
and indirect information access may be found in a paper by
Lampson [30].

2) The Essentials of Information Protection: For purposes of
discussing protection, the information stored in a com-
puter system is not a single object. When one is considering
direct access, the information is divided into mutually ex-
clusive partitions, as specified by various creators. Each
partition contains a collection of information, all of which is
intended to be protected uniformly. The uniformity of pro-
tection is the same kind of uniformity that applies to all of the
diamonds scored in the vault; any person who has a copy of
the combination can obtain any of the diamonds. Thus the
collections of information in the partitions are the funda-
mental objects to be protected.

Conceptually, then, it is necessary to build an impenetrable
wall around each distinct object that warrants separate protec-
tion, construct a door in the wall through which access can be
obtained, and post a guard at the door to control its use. Con-
trol of such access is necessary for two reasons: the guard must have some way of knowing which users are authorized to have access, and that
each user have some reliable way of identifying himself to the
guard. This authority check is usually implemented by having
the guard demand a match between something he knows and
something the prospective user possesses. Both protection and
authentication mechanisms can be viewed in terms of this
model.

Before extending the model, we pause to consider two con-
crete examples. The multiplexing of a single computer system
among several users and the authentication of a user's claimed
identity. These initial examples are complete isolation systems—no sharing of information can happen. Later we will
extend our model of guards and walls in the discussion of
shared information.

3) An Isolated Virtual Machine: A typical computer con-
sists of a processor, a linearly addressed memory system,
and some collection of input/output devices associated with the
processor. It is relatively easy to use a single computer to
simulate several, each of which is completely unaware of the
existence of the others, except that each runs more slowly than
usual. Such a simulation is of interest, de novo during the
interval when one of the simulated (commonly called virtual)
processors is waiting for an input or output operation to finish,
another virtual processor may be able to process at its normal
rate. Thus a single processor may be able to take the place of
several. Such a scheme is the essence of a multiprogramming
system.

To allow each virtual processor to be unaware of the existence
of the others, it is essential that some isolation mechanism is
provided. One such mechanism is a special hardware register called a descriptor register, as in Fig. 1. In this figure, all
memory references by the processor are channeled by a special
piece of hardware that is interposed in the path to the memory.
The descriptor register controls exactly which part of memory is
accessible. The descriptor register contains two components
a hard value and a bound value. The base is the lowest num-
bered address the program may use, and the bound is the num-
ber of locations beyond the base that may be used.11 We will
call the value in the descriptor register a descriptor, as it
describes an object (in this case, one program) stored in mem-
ory. The program controlling the processor has full access to
everything in the base-bound range, by virtue of possession of
its one descriptor. As we go on, we shall embellish the con-
cept of a descriptor: it is central to most implementations of
protection and of sharing of information.

So far, we have not provided for the dynamics of a complete protec-
tion scheme: we have not discussed how tasks the descriptor
register. If any running program could lead it into any
arbitrary treasury, there would be no protection. The in-
11 In most implementations, addresses are also relocated by adding to
them the value of the base. This relocation implies that for an initial
A to be legal, it must lie in the range 0 <= A <= base.

12 The concepts of 'non-maskable' and 'read-only' hardware-interrupt
descriptors appeared, apparently independently, between 1957 and
1959 on three projects with diverse goals. At MIT, J. McCarthy et
al. invented the "page" and the "block" abstraction, and the necessary
mechanism to make time-sharing feasible. IBM independently de-
veloped the concept of a non-maskable interrupt as a mechanism to prevent
multiprogramming of the Stretch (1960) computer system [31]. At
Burrington, M. Burton suggested that hardware-supported descriptors
would provide direct support for the running system of rules if higher level
languages in the BS/000 computer system [32].

Fig. 1. Use of a descriptor register to simulate multiple virtual ma-
cines. Program C bliss control of the processor. The privileged state bit
has value zero, indicating that program C is a user program. When
program C is running, the privileged state has its value one. In the
(bottom) figure, lower addresses are nearer the bottom of the figure.

13 The concepts of 'non-maskable' and 'read-only' hardware-interrupt
descriptors appeared, apparently independently, between 1957 and
1959 on three projects with diverse goals. At MIT, J. McCarthy et
al. invented the "page" and the "block" abstraction, and the necessary
mechanism to make time-sharing feasible. IBM independently de-
veloped the concept of a non-maskable interrupt as a mechanism to prevent
multiprogramming of the Stretch (1960) computer system [31]. At
Burrington, M. Burton suggested that hardware-supported descriptors
would provide direct support for the running system of rules if higher level
languages in the BS/000 computer system [32].
The supervisor program is part of all three protection mechanisms, for it is responsible for maintaining the integrity of the identification manifest in the descriptor register and the privileged state. The supervisor does not do its job correctly, virtual processors could become labeled with the wrong base and bound values, or user programs could become labeled with a privileged state but that is on. The supervisor protects itself from the user programs with the same isolation hardware that separates users, an example of the "economy of mechanism" design principle.

With an appropriately sophisticated and careful supervisor program, we now have an example of a system that completely isolates its users from one another. Similarly isolated peripherals storage can be added to such a system by attaching some long-term storage devices (e.g., magnetic disks) and developing a similar descriptor scheme for its use. Since long-term storage is accessed less frequently than primary memory, it is common to implement its descriptor scheme with the supervisor program rather than hardware, but the principle is the same. Data streams to input or output devices can be controlled similarly. The combination of a virtual processor, a memory area, some data streams, and an isolated region of long-term storage is known as a virtual machine. Long-term storage does, however, force us to face one further issue. Suppose that the virtual machines communicate with its user through a typewriter terminal. If a new user approaches a previously unused terminal and requests to use a virtual machine, which virtual machine (and, therefore, which set of long-term stored information) should he be allowed to use? We may solve this problem outside the system, by having the supervisor permanently associate a single virtual machine and its long-term storage area with a single terminal. Then, for example, if a user is at the terminal. If, on the other hand, a more flexible system is desired, the supervisor program must be prepared to associate any terminal with any virtual machine and, as a result, must be able to verify the identity of the user at a terminal. Schemes for performing this authentication are the subject of our next example.

4) Authentication Mechanisms: Our second example is of an authentication mechanism: a system that verifies a user's claimed identity. The mechanics of this authentication mechanism differ from those of the protection mechanisms for implementing virtual machines mainly because not all of the components of the system are under uniform physical control. In particular, the user himself and the communication system connecting his terminal to the computer are components to be viewed with suspicion. Consequently, the user needs to verify that he is in communication with the expected computer system and the intended virtual machine. Such systems follow our abstract model of a guard who demands a match between something he knows and something the requester possesses. The objects being protected by the authentication mechanism are the virtual machines. In this case, however, the requester is a computer system user rather than an executing program, and because of the lack of physical control over the user and the communication system, the security of the computer system must depend on either the security of the identification or the unforgeability of the user's identification.
The primary weakness of such schemes is that the hard-to-fabricate object, after being examined by the specialized input device, is reduced to a stream of bits to be transmitted to the computer. Unless the terminal, its object reader, and its communication lines to the computer are physically secured against tampering, it is relatively easy for an intruder to modify the terminal to transmit any sequence of bits he chooses. It may be necessary to make the acceptable bit sequences happen after all. On the other hand, the scheme is inconvenient, resists casual misuse, and provides a conventional form of accountability through the physical objects used as keys.

A problem common to both the password and the unforgeable object approach is that they are "one-way" authentication schemes. They authenticate the user to the computer system, but not vice versa. An easy way for an intruder to penetrate a password system, for example, is to intercept all communications to and from the terminal and direct them to another computer—one that is under the intruder's control. This computer can be programmed to "masquerade," that is, to act just like the system the caller intended to use, up to the point of requesting him to type his password. After receiving the password, the masquerader gracefully terminates the conversation with some unsuspicious error message, and the caller may be unaware that his password has been stolen. The same attack can be used on the unforgeable object system as well.

A more powerful authentication technique is sometimes used to protect against masquerading. Suppose that a remote terminal is equipped with enciphering circuitry, such as the LUCIFER system [38], that scrambles all signals from the terminal. Such devices normally are designed so that the exact encipherment is determined by the value of a key known as the encryption or transformation key. For example, the transformation key might consist of a sequence of 1000 binary digits read from a magnetically striped plastic card. In order that recipient of such an enciphered signal may comprehend it, he must have a deciphering circuit princed with an exact copy of the transformation key, or else he must cryptoanalytically scramble the stream to try to discover the key. The strategy of encipherment/decipherment is usually involved for the purpose of providing communications security on an otherwise unprotected communications system. However, it can simultaneously be used for authentication, using the following technique, first published in the unclassified literature by Pointz [39]. The user, at a terminal, begins bypassing the enciphering equipment. He types his name. This name passes, unenciphered, through the communication system to the computer. The computer looks up the name, just as with the password system. Associated with each name, instead of a secret password, is a secret transformation key. The computer loads the transformation key into its enciphering mechanism, turns it on, and attempts to communicate with the user. Meanwhile, the user has loaded his copy of the transformation key into his enciphering mechanism and turned it on. Now, if the keys are identical, exchange of some standard hand-shaking sequence will succeed. If they are not identical, the exchange will fail and both the user and the computer system will encounter an unintelligible stream of bits. If the exchange succeeds, the complete system is certain of the identity of the user, and the user is certain of the identity of the computer. The secret used for authentication—the transformation key—has not been revealed.
SALTER AND SCHROEDER: PROTECTION OF COMPUTER INFORMATION

are used by the communication system. If communication
is used by the user is unauthorized, the system has been re-
placed by a masquerade, or an error occurred, each party to
the transmission has immediate warning of a problem.16

Relatively complex elaborations of these various strategies
have been implemented, differing both in economics and in
security philosophy of the prospective user. For example, Bratseth [40] explored in detail strategies of authoriza-
tion in multimode computer networks. Such elaborate
measures, though fascinating to study and analyze, are diversion-
ary to our main topic of protection mechanisms.

3) "Security Information. The virtual machines of the earlier
system were totally independent, as far as information ac-
ceptability was concerned. Each user might just as well have
his own private computer system. With the steadily declining
prices of computer manufacture there are few technical reasons
not to use a private computer. On the other hand, for many
applications some sharing of information among users is use-
able, or even essential. For example, there may be a library of
commonly used, reliable programs. Some users may create new
programs which other users would like to use. Users may
wish to be able to update a common data base, such as a file of
airline seat reservations or a collection of programs that imple-
mant a biomedical statistics system. By these means, virtual
machines are inadequate, because of the total isolation of their
users from one another. Before extending the virtual machine
concept any further, let us turn to our abstract discussion of
guards and guards.

Implementations of protection mechanisms that permit sharing
among the various categories described by Wilkes [37].

a) "List-oriented" implementations, in which the guard
holds a list of identifiers of authorized users, and the user
checks a unique unforgeable identifier that must appear on
the list for access to be permitted. A store clerk checking
a list of credit customers is an example of a list-oriented imple-
mentation. The individual might use his driver's license as a
unique unforgeable identifier.

b) "Ticket-oriented" implementations, in which the guard
holds the description of a single identifier, and each user has
a collection of unforgeable identifiers, or tickets,17 correspond-
ing to the objects to which he has been authorized access.
A ticket decoder that opens with a key is probably the most com-
mon example of a ticket-oriented mechanism; the guard is im-
plemented at the hardware of the lock, and the matching key
is the (presumably) unforgeable authorizing identifier.

An important distinction between access to some object,
a different in these two schemes. Is a list-oriented system, a
system is authorized to use an object by having his name placed
on the guard's list for that object. In a ticket-oriented system,
a user is authorized by giving him a ticket for the object.

We can also note a crucial mechanical difference between the
two kinds of implementations. The list-oriented mecha-
nism requires that the guard examine his list at the time access
is requested, which means that some kind of associative search
must accompany the access. On the other hand, the ticket-
oriented mechanism passes on the user the burden of choosing
which ticket to present, a task he can combine with deciding
which information to access. The guard only needs compare
the presented ticket with his own expectation before allowing
the physical memory access. Because associative matching
tends to be either slower or more costly than simple com-
parison, list-oriented mechanisms are not often used in applica-
tions where traffic is high. On the other hand, ticket-oriented
mechanisms typically require considerable technology to con-
trast forgery of tickets and to control passing tickets around
from one user to another. As a rule, most real systems contain
both kinds of sharing implementations—a list-oriented system
at the human interface and a ticket-oriented system in the
underlying hardware implementation. This kind of arrange-
ment is accomplished by providing, at the higher level, a list-
oriented guard18 whose only purpose is to hand out temporary
tickets which the lower level (ticket-oriented) guards will
honor. Some added complexity arises from the need to keep
unauthorized, as represented in the two systems, synchro-
nized with each other. Computer protection systems differ
mostly in the extent to which the architecture of the underly-
ing ticket-oriented system is visible to the user.

Finally, let us consider the degenerate cases of list- and
ticket-oriented systems. In a list-oriented system, if each
guard's list of authorized users can contain only one entry, we
have a "complete isolation" kind of protection system, in
which no sharing of information among users can take place.
Similarly, in a ticket-oriented system, if there can be only one
ticket for each object in the system, we have a "com-
plete isolation" kind of protection system. Thus the "com-
plete isolation" protection system turns out to be a particular
degenerate case of both the list-oriented and the ticket-oriented
protection implementations. These observations are important
in examining real systems, which usually consist of interesting
protection mechanisms, some of which are list-oriented, some of
which are ticket-oriented, some of which provide complete
isolation and therefore may be implemented as degenerate examples of either of the two, depending
on local circumstances.

We should understand the relationship of a user to these
transactions. We are concerned with protection of informa-
tion from programs that are executing. The user is the in-
dividual who assumes accountability for the actions of an
executing program. Inside the computer system, a program is
evoked as a virtual processor, so one or more virtual proces-
sors can be identified with the activities directed by the user.19

16 Actually, there is still one untoward possibility: a masquerade
would exactly recall the encoded bits in one communication, and
transmit a message that appeared to be from the user, but deceptively
defining the user or the computer system. The general countermeasure
is to require a check on each message, with something that is not
yet predictable, such as the current date and time. By examina-
tion of the message, one can determine if the decoded message is not a
replicated copy of an old one. Various techniques are employed in detail by Smith
and Brandt [40].

17 As shown later, in a computer system, descriptions can be used on
the tickets.

18 Called an agency by Bratseth [40]. The advantage of delegation at
the various sessions of a cooperation is frequently controlled by an
agent—upon presentation or proof of identity, the agency issues a
wallet that will be honored by guards at each station. The agent
is an entity in the computer system that is independent of the
individuals who use it (who ignore the names printed on the wallet) i.e.
ticket-oriented.

19 The terms "process", "execution unit" and "task" are sometimes
used for the abstraction or any similar cases. We will use the term
"virtual processor" for all self-evident operational definitions, following
a suggestion by Wilkes.
In a list-oriented system it is the guard's business to know whose virtual processor is attempting to make an access. The virtual processor has been marked with an unforgeable label identifying the user accountable for its actions, and the guard inspects this label when making access decisions. In a ticket-oriented system, however, the guard cares only that a virtual processor present the appropriate unforgeable ticket when attempting an access. The connection to an accountable user is more diffuse, since the guard does not know or care how the virtual processor acquired the tickets. In either case, we conclude that in addition to the information inside the impenetrable wall, there are other things that must be protected: the guard's authorization information, and the association between a user and the unforgeable label or set of tickets associated with his virtual processors.

Since an association with some user is essential for establishing accountability for the actions of a virtual processor, it is useful to introduce an abbreviation for this accountability—the \textit{principal}. A principal is, by definition, the entity accountable for the actions of a virtual processor.\footnote{The word “principal,” suggested by Dennis and Van Horn \cite{Dennis Van Horn 1972}, is used in this discussion because of its association with the legal concepts of authority, accountability, liability, and responsibility. The detailed relationship among these four concepts was an interesting study, but inside the computer system, accountability is the only one usually measurable. For purposes of the discussion of accountability, we are restricting our attention to the individual guiding the course of the computation. We are avoiding the complication that responsibility for any specific action of a process may actually be shared among the user, the programmer, and the maintenance of the program being executed among others.}

In the situations discussed so far, the principal corresponds to the user outside the system. However, there are situations in which one-to-one correspondence of individuals with principals is not adequate. For example, a user may be accountable for some very valuable information and authorized to use it. On the other hand, on some occasions he may wish to use the computer for some purpose unrelated to the valuable information. In this case there is a need for two different principals corresponding to the same user.

Some mechanisms envision a data base that is to be modified only by a committee agrees. There might be an authorized principal that cannot be used by any single individual; all of the committee members must agree upon its use simultaneously. Because the principal represents accountability, we shall use the term in a system dynamic authorization of sharing that authorizing access is done in terms of principals. That is, if one wishes friends have access to a file, the authorization is done by naming a principal only that friend can use.

For each principal we may identify all the objects in the system which the principal has been authorized to use. We will name that set of objects the domain of that principal.

Summarizing, then, a principal is the unforgeable identifier attached to a virtual processor in list-oriented system. When a user first approaches the computer system, that user must identify the principal to be used. Some authentication mechanism, such as a request for a secret password, establishes the user's right to use that principal. The authorization mechanism itself may be either list- or ticket-oriented or of the complete isolation type. Then a computation is begun in which all the virtual processors of the computation are labeled with the identifier of this principal, which is considered accountable for all further actions of these virtual processors. The authentication mechanism has allowed the virtual processor to utilize the domain of that principal. This situation makes apparent the importance of the authorization mechanism. Clearly, one must carefully control the conditions under which a virtual processor enters another system.

Finally, we should note that in a ticket-oriented system there is no mechanism needed to associate an unforgeable identifier for a virtual processor, since the tickets themselves are assumed unforgeable. Nevertheless, a collection of tickets can be considered to be a domain, and therefore correspond to some principal, even though there may be no obvious identity for that principal. Thus accountability is ticket-oriented systems is difficult to implement.

Now we shall return to our example system and extend it to include sharing. Consider for a moment the problem of sharing a library program—say, a mathematical function sub-routine. We could place a copy of the math routine in the long-term storage area of each virtual machine that had a use for it. This scheme, although workable, has several defects. Most obvious, the multiple copies require multiple storage spaces. More subtle, the scheme does not respond well to changes. If a newer, better math routine is written, upgrading the multiple copies requires effort proportional to the number of users. These two observations suggest that one would like to have some scheme to allow different users access to a single \textit{master} copy of the program. The storage space will be smaller, and the communication of updated versions will be faster.

In terms of the virtual machine model of our earlier example, we can share a single copy of the math routine by adding to the real processor a second descriptor register, as in Fig. 2, setting the new register some place in memory by itself and placing a descriptor file it in the second descriptor register. Following the previous strategy, we assume that the virtual state bit assures that the supervisor program is the only one permitted to load other descriptor register. In addition, some scheme must be provided in the architecture of the processor to permit a choice of which descriptor register to be used for each address generated by the processor. A simple scheme would be to let the high-order address bit select the descriptor register. Thus, in Fig. 2, all addressing in the lower half of the address range would be interpreted relative to descriptor register 1, and addressing in the upper half of the address range would be relative to descriptor register 2. An alternate scheme, suggested by Dennis \cite{Dennis 1972}, is to add explicitly to the format of instruction words a field that selects the descriptor register to be used with the address in that instruction. The use of descriptor for sharing information is intimately related to the addressing architecture of the processor, a relation that can cause considerable confusion. The reason why descriptor set of interest for sharing appears by comparing parts \textit{a} and \textit{b} of Fig. 2. When program \textit{A} in control, it can have access only to itself and the math routine; similarly, when program \textit{B} in control, it can have access only to itself and the math routine. Some neither program the power to change the descriptor register, sharing of the math routine has been accomplished while maintaining isolation of program \textit{A} from program \textit{B}.

The effect of sharing is shown more graphically in Fig. 3, which is Fig. 2 redrawn with two virtual processors, one executing program \textit{A} and the other executing program \textit{B}.
Although the basic mechanism to permit information sharing is now in place, a remarkable variety of implications that follow from its introduction require further mechanisms. These implications include the following.

1) If virtual processor P1 can overwrite the shared math routine, then it could disrupt the work of virtual processor P2.

2) The shared math routine must be careful about making modifications to itself and about when in memory it writes temporary results, since it is to be used by independent computations, perhaps simultaneously.

3) The scheme needs to be expanded and generalized to cover the possibility that more than one program or data base is to be shared.

4) The supervisory needs to be informed about which principals are authorized to use the shared math routine (unless it happens to be completely public with no restrictions).

Let us consider these four implications in order. If the stored area of memory is a procedure, then to avoid the possibility that virtual processor P1 will maliciously overwrite it, we can restrict the methods of access. Virtual processor P1 needs to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants embedded in the procedure, but it has no need to write into any part of the shared procedure. We may accomplish this restriction by extending the descriptor schemes and the descriptors themselves to include accessing permission, an idea introduced for different reasons in the original Bunovsky BS5000 design [32]. For example, we may add two bits, one controlling permission to read and the other permission to write to the storage area defined by each descriptor, as in Fig. 4. In virtual processor P1 of Fig. 3, descriptor 1 would have both permissions granted, while descriptor 2 would permit only reading.
of data and execution of instructions. An alternative scheme would be to attach the permission bits directly to the storage areas containing the shared program or data. Such a scheme is attractive because, unlike the descriptor structure, it would be effective when the program is far removed from the data. However, the use of dedicated bits appears to imply a more centralized control over the shared resources than is needed. The need for a more flexible and decentralized control is evident in the need for multiple users to access the same data or program simultaneously. This is particularly important in a distributed system where a program might be shared by multiple users or processes. The concept of a shared memory system is relevant here, as it allows for multiple processes to access and modify the same data simultaneously. This can be achieved through the use of synchronization mechanisms, such as semaphores or monitors, which ensure that access to shared resources is controlled and that conflicts are avoided.

As for the third impetus, the need for expansion, we could generalize our example to permit several distinct shared items merely by increasing the number of descriptor registers and informing the supervising processor which shared objects should be addressable by each virtual processor. However, there are two substantially different forms of this generalization—capability systems and access control list systems. In terms of the earlier discussion, capability systems are ticket-oriented, while access control list systems are list-oriented. The combination of these two forms, the capability system for speed and an access control list system for the human interface, along with the fourth implication, authorization, more groundwork must be laid.

In Section II, the development of protection continues with a series of successively more sophisticated models. The initial model, of a capability system, explores the use of encapsulated but sharable objects as tickets to provide a flexible authorization scheme. In this context we establish the general rule that communication external to the computer must preclude dynamic authorization of sharing. The limitations of capability systems—primarily lack of accountability for the use—lead to analysis of protection and the observation that in the absence of accountability for the use of data, the same or identical objects as to provide detailed control of authorization.

The use of access control lists leads to a discussion of controlling changes of authorization, of the use being more attractive. This will provide a more sophisticated form of authorization. The advantage of access control lists is that it is easier to maintain a formal record of access to data than it is to maintain an effective protection policy. This can be seen by comparing the number of accesses to the same object that must be tracked in an access control list system with the number of accesses to different objects that must be tracked in a capability system. In the former, all accesses to a particular object must be recorded, while in the latter, only accesses to different objects need to be tracked. This makes the access control list system more efficient for large numbers of identical objects, while the capability system is more efficient for large numbers of different objects.
SALTZER AND SCHROEDER: PROTECTION OF COMPUTER INFORMATION

universal. Any single processor address space, on the other hand, is defined by the particular protection descriptors associated with the processor and therefore is local. If the processor switches control of a real processor from one virtual processor to another, it would first reload the protection descriptors; the processor address space thus is different for different users, while the address space remains the same for all users.

With the addressing function separated architecturally from the protection function, we may now examine the two generalized forms of protection systems: the capability system and the access control list system.

B. The Capability System

1. The Concept of Capability: The simplest generalization is the capability system suggested by Dennis and Van Horn [41], and first partially implemented on an M.I.T. PDP-1 computer [48]. There are many different detailed implementations for capability systems: we illustrate with a specific example. Recall that we introduced the privileged state bit to control who may load values into the protection descriptor registers. Another way to maintain the integrity of these registers would be to allow any program to load the protection descriptor registers, but only from locations in memory that have been certified to contain acceptable protection descriptor values.

Suppose, for example, that every location in memory were tagged with an extra bit, if the bit is ON, the word in that location is an ordinary data or instruction word. If the bit is OFF, the word is taken to contain a value suitable for loading into a protection descriptor register. The instruction that loads the protection descriptor register will operate only if its operand address leads to a location in memory that has the tag bit ON. To complete the scheme, we should provide an instruction that stores the contents of a protection descriptor register in memory and turns the corresponding tag bit OFF, and we must arrange that all other core instructions set the tag bit OFF in any memory location they write into. This gives us two kinds of objects stored in memory: protection descriptor values and ordinary data values. There are also two sets of instructions, separate registers for manipulating the two kinds of objects, and, effectively, a wall that prevents values that are subject to general computational manipulation from ever being used as protection descriptor values. This kind of scheme is a particular example of what is called a tagged architecture.

This particular tagged architecture is known as a capability system, that is, the user place protection descriptor values in memory addresses that are convenient to him. A memory word that contains a protection descriptor value (in ordinary memory, the system, that is, the machine tag bit ON) is known as a capability.

11 A detailed analysis of the resulting architectural applications was made by Fair and Yegnas [42]. The capability system is a close relative of a modern organization of the Rice Research Computer [50], but Dennis and Van Horn seem to be the first to have noticed the applicability of that organization to interactive protection.

12 Tagged architectures were invented for a variety of motivations other than protection. The Burroughs B550 and its ancestors, and the Rice Research Computer [50], are examples of architectures that use multiple tags to separately identify instructions, descriptors, and several different types of data. All examples of tagged architecture seem to return back to suggestions made by J. Babbage. A thorough discussion of the concept is given by Fejesz [51].
Fig. 6. A simple capability system. Program $A$ is in control of the processor. Note that there is no way for the processor to address Smith's catalog or data base $Y$. On the other hand, data base $X$ could be accessed by loading capability $C_1$ into a protection descriptor register. Capability $C_1$ is local because it is stored in a segment that can be reached from a capability already loaded in protection descriptor register $R_2$. Note also that the former function of the privileged state has been accomplished by protecting the capabilities. The privileged state has also lost useful services and will be re-introduced later.

To see how capabilities can be used to generalize our basic sharing strategy, suppose that each processor has several (say four) protection descriptor registers, and that program $A$ is in control of a processor, as in Fig. 6. (For clarity, this and future figures omit the addressing descriptors of the segmented memory.) The first two protection descriptor registers have already been loaded with values permitting access to two segments, program $A$ and a segment we have labeled "Catalog for Doe." In our example, this latter segment contains two locations with tags indicating that they are capabilities $C_2$ and $C_3$. Program $A$ may direct the processor to load the capability at location $C_2$ into one of the protection descriptor registers, and then the processor may address the shared math routine. Similarly, either program $A$ or the shared math routine may direct the loading of the capability at location $C_3$ into a protection descriptor register, and then the processor may address the segment labeled "Private Data Base $Z$." By a similar chain of reasoning, another processor starting with a capability for the segment labeled "Catalog for Smiths" can address both the shared math routine and the segment "Private Data Base $Z$." We can now arrange for any desired static pattern of sharing of segments. For example, for each user, we can provide one segment for use as a catalog and place in that catalog a capability for every segment he is authorized to use. Each capability contains separate read and write permission bits, so that some users may receive capabilities that permit reading and writing some segment, while other receive capabilities permitting only reading from that same segment. The catalog segment actually might consist of pairs: a character-string name for some segment and the associated capability that permits addressing that segment. A user would create a new segment by citing the supervisor. The supervisor by conversion might set some protection descriptor to contain a capability for the new segment. The user could then file his new segment by storing this new capability in his catalog along with a name for the segment. The user then could have an authentication mechanism to request a new virtual processor and starting it executing in a supervisor program that initially has a capability for a user identification table, as in Fig. 7. If the user identifies himself as "Doe" and supplies a password, the supervisor program can look up his identification in the user identification table. If it verifies the password and finds a protection descriptor register the capability for the catalog associated with Doe's entry in the user identification table, and then running some program in Doe's directory, say program $A$. Program $A$ can extend its capability to any segment to which a capability exists in Doe's catalog. Formally, after verifying the claimed identity of the user, the authentication system has allowed the virtual processor to enter Doe's domain, starting in procedure $4$.

By providing for authentication we have actually tied together two protection systems: 1) an authentication system that controls access of users to named catalog capabilities, and 2) the general capability system that controls access of the holder of a capability to other objects stored in the system.

The authentication system associates the newly created virtual processor with the principal accountable for its future activities. Once the virtual processor is started, however, the capability mechanism identifies the user of an object or a capability. Although there is a sequence of accountability, we cannot view the whole system as an entity. The user is finally a user, with a capability enabling others to execute his programs, just as the capability mechanism is an entity whose programs do not exec its programs. The whole system, including the capability mechanism, is the virtual processor.
Fig. 7. A capability system with provision for authentication.

The scheme described so far admits any desired static arrangement of accessing authorization. It could be used in an application for which a simple, rarely changed, authorization pattern is useful. For example, a company data base management system might have a relatively static authorization pattern, which changes only when major revisions are made to the style of maintaining the data base. We have not yet provided, however, for the possibility that Doe, upon creating a new segment, might wish to authorize access to it for Smith. Such a need would probably arise if the computer system is used for the creation and editing of interoffice memoranda and letters or for constructing programs. We shall call this operation dynamic authorization. The dynamic authorization of sharing a topic that must be examined quite carefully, since it raises several subtle issues that are fundamental to sharing and protection.

2 The Dynamic Authorization of Sharing: One might propose to handle dynamic authorization very simply by arranging that Doe have a capability to write into Smith's catalog. Then Eve could store a copy of the capability for the new segment in Smith's catalog. But this approach has a defect. Allowing Doe to have a capability to write into Smith's catalog would enable Doe to overwrite and destroy all of Smith's capabilities. The inverse strategy of giving Smith a capability to read Doe's catalog would give Smith access to all of Doe's segments. A more "secure" approach to the problem is needed. To develop this approach, we will consider a clumsy strategy with square-law growth, and then refine it.

If the possibility of sharing had been anticipated, both Doe and Smith might initially have had a capability allowing read and writing a communication segment used only to pass messages and capabilities between Doe and Smith. Doe's program deposits the capability for his newly created object in the communication segment for Smith, and Smith's program can pick it up and use it or catalog it at Smith's convenience. But that description oversimplifies one step. Both Doe's and Smith's programs somehow have to locate the capability for the common communication segment. How do they know what to look for? Consider the case of the sender, Doe's program, first. Presumably it looks in some trusted catalog for the name "Smith" and finds the capability for the communication segment next to Smith's name. But how does Doe's program know to look for the name "Smith"? The character-string name may be embedded in the program by Doe or he may type it into his program as it runs, but either one way is crucial—that there be a secure path from Doe, who is authorizing for the purpose of the capability, to the program, which is carrying it out. Next, we should ask, where does Doe find Smith out of the character-string name "Smith" so that he could type it in or embed it in his program? Presumably, he learns Smith's name via some path outside the computer. Perhaps Smith shouts it down the hall to him. The method of communication is not important, but the fact of the communication is. For dynamic authorization of sharing within a computer, there must be some previous communication from the recipient to the sender, external to the computer system. Further, this reverse external communication path must be sufficiently secure that the sender is certain of the system-cataloged name of the intended recipient. That name is, by definition, the identifier of the recipient's principal within the computer system. Thus the sender can be sure that only programs run under the accountability of that principal will have access to his new object.

An analogous chain of reasoning applies to Smith's program as the recipient of the capability for the new object. Smith must learn from Doe some piece of information sufficient that he can instruct his program to look in the correct communication segment for the capability which Doe is sending. Again,
Do's principal identifier should be the same used in Smith's rating of communication segments, so Smiths can be certain that only some program run under Doe's accessibility could possibly have sent the capability. In summary, Hwy is a complete protocol for dynamically authorizing sharing of a new object.

Sender's part:
1) Sender <hs> receiver's principal identifier via a communication path outside the system.
2) Sender transfers receiver's principal identifier to some program running inside the system under the accessibility of the sender.
3) Receiver's program uses receiver's principal identifier to ensure that only virtual processors operating under the accessibility of the receiver will be able to obtain the capability being transmitted.

Receiver's part:
1) Receiver learns sender's principal identifier, via a communication path outside the system.
2) Receiver transfers sender's principal identifier to some program running inside the system under the accessibility of the receiver.
3) Receiver's program uses the sender's principal identifier to ensure that only a virtual processor operating under the accessibility of the sender could have sent the capability being received.

This protocol provides protection for the authorization mechanism (copying of a capability) by requiring an authority check (comparison of a principal identifier found inside the system with authorization information transmitted from outside). Although the analysis may seem somewhat strained, it is important because it always applies, even though parts of it may be implicit or hidden. We have described the protocol in terms of a capability system, but the same protocol also applies in access control list systems.

Our analysis of the dynamics of authorization sharing has been in terms of private communication segments between every pair of users, a strategy which would lead, with N users, to some N² communication segments. To avoid this square-law growth, one might prefer to use some scheme that dynamically constrains the communication paths also, such as having special hard ware or a protected subsystem that implements a single "mailbox" segment for each user to receive messages and capabilities sent by other users. Of course, the mechanism that implements the mailbox segments must be protected, reliable mechanism, since it must accurately determine the principal identifier of the sender of a message and label the message with that identifier, so the receiver can reliably carry out this step 3) of the protocol. Similarly, the sender's agency, it must be able to associate the recipient's principal identifier with the recipient's mailbox, so that the sender's agency in his step 3) of the protocol is carried out correctly.

3) Reorganization and Control of Propagation: The capability system has as its chief virtues its inherent efficiency, simplicity, and flexibility. Efficiency comes from the ease of testing the validity of a proposed access; if the accessor can present a capability, the request is valid. The simplicity comes from the natural correspondence between the mechanical properties of capabilities and the semantic properties of addressing variables. The semantics for dynamically changing addressability that are part of such modern languages as PL/1 and Algol 68 fit naturally into a capability-based framework by using capabilities as address (pointers) variables. Straightforward additions to the capability system allow it gracefully implement languages with dynamic-type extension [21]. Facility comes from the defining property of a capability system: the user may decide which of his address may carry certain capabilities. The user can develop a data structure with as arbitrary pattern of access authorizations to his liking.

On the other hand, there are several potential problems with the capability system as we have sketched it so far. If Doe has a change of heart—he suddenly realizes that there is confidential information in the segment he permitted Smith to read—there is no way that he can disable the copy of the copy, unless that Smith now has stored away in some unknown location. Unless we provide additional control, his only hope is to destroy the original segment, an action which may be disruptive to other users, still trusted, who also have copies of the capability. This revocation of access is a problem.

A second, related property of a capability system is that Smith may now make copies of the capability and distribute them to other users, without the permission or even the knowledge of Doe. While in some cases, the ability of a system to pass access authorization along is exactly what the original grantor intended, in others it is not. We have not provided for any control of propagation.

Finally, the only possible way in which Doe could make a list of all users who currently can reach his segment would be by searching every segment in the system for the necessary capability. That search would be only the beginning, since there may be many paths by which users could reach those capabilities copy. Every such path must be found, a task that may involve a fair amount of computation and the allocation of memory space. We have completely ignored the protection mechanisms. This review of access is a problem.

To help counter these problems, constraints on the use of capability systems have been proposed and implemented in some systems. For example, a bit added to a capability (the copy bit) may be used to indicate whether or not the capability may be stored in a segment. If one user uses another user's access to a capability with the copy bit off, then the second user would not make copies of the capability he had borrowed. Propagation would be prevented, at the price of lost flexibility.

Alternatively, some segments (perhaps one per user) may be designated as capability-holding segments, and only those seg- ments may be targets of the instructions that load and store descriptor registers. This scheme may reduce drastically the effort involved in auditing and making provision possible, and only capability-holding segments used be examined. (The CAP system [20] and the Plessey 210 [35] are organized in approximately this way, and the Burroughs B5000 family restricts descriptor storage to the virtual processor stack and a single table of outbound references [47].) In systems that make a programming-language distinction between short-term (procedural-addressable memory (addressed by LOAD and STORE instructions) and long-term storage (addressed by GET and PUT subroutines), it is possible to restrict capabilities so that...

3 A fourth problem, not directly related to protection, is the "garbage collection" or "lost object" problem. If all copies of an object are inaccessible, the object should not be "garbage" in the sense that it is inaccessible. The object should not exist. Conversely, if an object is "garbage" in the sense that it is inaccessible, the object may be reactivated by some process activity. The simplest solution is to insist that the creator of an object is responsible for destroying the object before the object is destroyed the last capability copy. Since the creation of an object provides for the destruction of the object, the process that destroys object is usually adequate. See, for example, Rosenthal et al. [45]...
they may be stored only in processor-addressable memory. This restriction not only reduces the effort required for designing, but also limits the lifetime of a capability to that of a virtual process. When the system shuts down, the only records left behind are those that are stored in the system in long-term storage and all capabilities vanish. Of course, the next time the system starts up, it can create virtual processes need some way (such as a new access control list system, described in the next subsection) to acquire the capabilities they need.

A third approach is to associate a depth counter with each process descriptor register. The depth counter initially would have the value, say, of one, placed there by the supervisor. Whenever a program loads a descriptor register from a file in memory, that descriptor register receives a depth count that is one greater than the depth count of the descriptor register that contained the capability that permitted the load attempt. This depth counter prevents a depth count from exceeding a limit in a file. When a process attempts to load a file that is too deep, the access control list would fail and the process would be terminated.

A fourth approach to access control and protection of capabilities is to use an object-specific access control mechanism. Each object has a set of privileges that can be used to control access to the object. This approach can be used to implement user-specific protection mechanisms. It is also possible to combine the object-specific access control with process-specific access control to provide a more flexible protection mechanism.

C. The Access Control List System

1) Access Controllers: The usual strategy for providing responsibility of bindings is to control when they occur—typically by delaying them until the last possible moment. The access control list system provides exactly such a delay by inserting an extra authorization check at the latest possible point. Where the capability system was basically a ticket-oriented strategy, the access control list system is a list-oriented strategy. Again, there are many possible mechanisms, and we must choose one for illustration. For ease of discussion, we will describe a mechanism implemented completely in hardware (perhaps through microprogramming), although, historically, access control list systems have been implemented partly with interpretive software. Our initial model will impose the extra authorization check on every memory reference, an approach that is unlikely in practice but simpler to describe. Later we will show how a couple of a simple access control list system can be extended to a capability system, a more realistic implementation that reduces the number of extra checks.

The system of Fig. 2 identifies protection descriptors as a processor mechanism and addressing descriptors as a memory mechanism. Suppose that the memory mechanism is further augmented as follows. Whenever a user requests that a segment be created, the memory system will actually allocate two linked storage areas. One of the storage areas will be used to store the data of the segment as usual, and the second will be treated as a special kind of object, which we will call an access controller. An access controller contains two pieces of information: an addressing descriptor for the associated segment and an access control list, as in Fig. 2. An addressing descriptor for the access controller itself is assigned a unique identifier and placed in a map used by the memory system to locate objects. The access controller is used to be used as a kind of indirect address, as in Fig. 9. In order to access a segment, the processor must supply the unique identifier of that segment's access controller. Since the access controller is protected, however, there is no longer any need for these unique identifiers to be protected. The former protection descriptor required can be replaced with unprotected pointer registers, which can be loaded from any addressable location with arbitrary bit

82For example, in the Motoset system [55], capabilities are recognized by the hardware only if they are placed in special capability-holding elements, and the supervisor never has a separate copy of capabilities for those segments to other domains. The supervisor also associates with each access control list a thread header to every copy it makes of a capability, so that revocation is possible.
2) The processor uses the unique identifier found in the \textit{principal identifier register} to address access controller AC\textsubscript{1}. The processor uses the same time present to the memory system the user's principal identifier, a request to write, and the offset \textit{k}.

3) The memory system searches the access control list of \textit{AC\textsubscript{1}} to see if this user's principal identifier is recorded there. If the principal identifier is found, the memory system examines the permission bits associated with that entry of the access control list to see if writing is permitted.

4) If writing is permitted, the addressing descriptor of segment \textit{X}, stored in \textit{AC\textsubscript{1}}, and the original offset \textit{k} are used to generate a write request inside the memory system.

We need one more mechanism to make this system work.

The set of processor registers must be augmented with a new protected register that can contain the identifier of the principal currently accessing the activity of the virtual processor, as shown in Fig. 9. (Without that change, one could not implement the second and third steps.)

For example, we may have an organization like that of Fig. 10, which implements essentially the same pattern of sharing as did the capability system of Fig. 6. The crucial difference between these two figures is that, in Fig. 10, all references to data are made indirectly via access controllers. Overall, the organization differs in several ways from the pure capability system described before.

1) The decision to allow access to segment \textit{X} has known, auditible consequences. Doe cannot copy a code in the addressing descriptor of segment \textit{X} since he does not have direct access to it, eliminating propagation of direct access. The pointer to \textit{X}'s access controller itself may be freely copied and passed to anyone, but every use of the pointer must be via the access controller, which prevents access by unauthorized principals.\textsuperscript{24}

2) The access control list directly implements the sender's third step of the dynamic sharing protocol—verifying that the requestor is authorized to use the object. In the capability system, verification was done once to decide if the local capability copy should be made; after that, further copying was unrestricted. The access control list, on the other hand, is consulted on every access.

3) Revocation of access has become manageable. A change to an access control list removing a name immediately excludes all future attempts by that user to use that segment.

4) The question of "who is the unique identifier of some segment's access controller will work. A data reference for the processor proceeds in the following steps, keyed to Fig. 9.

1) The processor encounters an instruction that would write in the segment described by pointer register 3 at offset \textit{k}.

\textsuperscript{24}We should note that nothing prevents a program running under an authorized principal from copying the data of a segment in another segment where other principals might be authorized to read it.\textsuperscript{25} In general, a program running under an authorized principal may not write into any forms of access permissions, for example, by writing into the access control list, in any way that changes the authorization associated with the object. Partly because of this possibility, the importance of direct or indirect object of each principal has been emphasized.
Fig. 10. A protection system using access controllers containing access control lists. In this system, every segment has a single corresponding access control list with its own unique identifier for addressing purposes; pointer registers always contain the unique identifiers of access controllers. Program A is in control of the processor, and it has already acquired a pointer to the library catalog. Since the access control list in the access controller for the library catalog contains Doe's name, the processor can use the catalog to find the pointer for the shared math routine. Since his name also appears in the access control list of the math runtime, the processor will then be able to use the shared math routine.

If any desired grouping for protection purposes. Thus, in Fig. 10, a library catalog has been introduced.

It is also apparent that implementation, especially direct hardware implementation, of the access control list system could be quite an undertaking. We will later consider some changes to simplify implementation with minimum compromise of functions, but first it will be helpful to introduce some more functional property—protection groups.

2) Protection Groups: Cases often arise where it would be convenient to list by name every individual who is to have access to a particular segment, either because the list would noticeably be long or because the list would change frequently.

To handle this situation, most access control list systems implement factoring into protection groups, which are principally that be used by more than one user. If the name of a protection group appears in an access control list, all users who are members of that protection group are to be permitted access to that segment.

Methods of implementation of protection groups vary widely. A simple way to add them to the model of Figs. 9 and 10 is to add the "principal holding" register of the processor so that it can hold two (or more) principal identifiers at once, one for a personal principal identifier and one for each protection group of which the user is a member. Fig. 10 shows this extension in dashed lines. In addition, we upgrade the access check list checker so that it searches for a match between any of the principal identifiers and any entries of the access control list. Finally, those who are allowed to use those principals that represent protection group identifiers must also be controlled systematically.

We might imagine that for each protection group there is a protection group list, that is, a list of the personal principal identifiers of all users authorized to use the protection group's principal identifier. (This list is an example of an access control list that is protecting an object—a principal identifier—rather than a segment.) When a user logs in, he can specify the set of principal identifiers he proposes to use. His right to use his personal principal identifier is authenticated, for example, by a password. His right to use the remaining principal identifiers can then be authenticated by looking up the now-authenticated personal identifier on each named protection group list. If everything checks, a virtual processor can safely be created and started with the specified list of principal identifiers.

3) Implementation Considerations: The model of a complete protection system as developed in Fig. 10 is one of many possible architectures, most of which have essentially identical functional properties; our choices among alternatives have been guided more by pedagogical considerations than by practical implementation issues. There are at least three key areas in which a direct implementation of Fig. 10 might encounter practical problems.

1) As proposed, every reference to an object in memory requires several steps: reference to a pointer register, indirect reference to another pointer register, and finally to the object.

In some systems (notably CAL TES [17]), principal identifiers are treated as special cases of a capability, known as an access key, that can be copied about, stored anywhere, and passed on to friends. Although this approach appears to produce the same effect as protection groups, accountability for the use of a principal identifier no longer resides in an individual, since any holder of a key can make further copies for his friends.
reference through an access controller including search of an access control list; and finally, access to the object itself via addressing descriptors. Not only are these steps serial, but several memory references are required, so fast memory access would be needed.

2) An access control list search with multiple principal identifiers is likely to require a complex mechanism, or be slow, or both. (This tradeoff between performance and complexity contrasts with the capability system, in which a single comparison is always sufficient.)

3) Allocation of space for access control lists, which can change in length, can be a formidable implementation problem. (Compared to a capability system, the mechanics of changing authorization in an access control list system are inherently more cumbersome.)

The first of these problems is attacked by recognizing that the purpose of the access control list is to establish authorization rather than to mediate every detailed access. Mediation of access would be handled more efficiently by a capability system.

Suppose we provide for each pointer register a "shadow" capability register that is invisible to the virtual processor, as in Fig. 11. Whenever a pointer register containing the unique identifier of an access controller is first used, the shadow register is loaded with a capability consisting of a copy of the addressing descriptor for the segment protected by the access controller, together with a copy of the appropriate set of permission bits for this principal. Subsequent references via that pointer register can proceed directly using the shadow register rather than indirectly through the access controller.

One implication is a minor change in the revocability properties of an access control list: changing an access control list does not affect the capabilities already loaded in shadow registers of running processors. (One could restore complete revocability by clearing all shadow registers of all processors and restarting any current access control list searches. The next attempted use of a cleared shadow register would automatically trigger its reloading and a new access control list check.) The result is a highly constrained but very fast capability system beneath the access control list system. The detailed checking of access control falls on the capability mechanism, which can in individual memory references exactly enforce the constraints specified by the access control list system.

The second and third problems, allocation and search of access control lists, appear to require more compromise of functional properties. One might, for example, use access control lists to contain, say, exactly five entries, to simplify the space allocation problem. One popular implementation allows only three entries on each access control list. The first is filled in with the personal principal identifier of the user who created the object being protected, the second with the principal identifier of the (single) protection group to which he belongs, and the third with the principal identifier of a universal protection group of which all users are members. The individual access permission bits for these three entries are specified by the program creating the segment. A completely different way to provide an access control list system is to implement it in interpretive software in the path to the secondary storage or file system. Primary memory protection can be accomplished with either base-and-bound registers, or more generally, with a capability system in which the capabilities cannot be copied into the file system. This approach takes the access control list checking mechanisms out of the heavily used primary memory access path, and reduces the pressure to compromise its functional properties. Such a mixed strategy, while more complex, typically proves to be the most practical compromise. For example, the Multics system uses software-interpreted access control lists together with hardware-interpreted tables of descriptors. Similarly, the "guard file" of the Burroughs B6700 Master Control Program is an example of an access controller implemented in an interpretable language. Authority to Change Access Control Lists: The access control list organization brings one issue into focus: control of who may modify the access control information. In the capability system, the corresponding consideration is diffuse. Any program having a capability may make a copy and put that copy in a place where other programs, running in other virtual processors, can make use (or further copies) of it. The access control list system is devised to provide more precise control of authority, so some mechanism of exercising that control is needed. The goal of any such mechanism is to provide within the computer an authority structure that models the authority structure of whatever organization uses the computer. Two different authority-controlling policies, with subtly different modeling abilities, have been implemented or proposed. We name these two self-controlling and hierarchical control.

The simplest scheme is self-control. With this scheme, we extend our earlier concept of access permission bits to include not just permission to read and write, but also permission to modify the access control list that contains the permission bits. Thus, in Fig. 12, we have a slightly more elaborate access controller, which by itself controls who may make modifications to it. Suppose that the creation of a new segment is accompanied by the creation of an access controller that contains one initial entry in its access control list, the entry giving all permissions to the principal identifier associated with the creating virtual processor. The creator receives a pointer for

Variations of this strategy are implemented in the UNIX and VINES (11) and UNIX-16 (11) systems. This also seems to have originated at the University of California at Berkeley (11) and elsewhere.

We have thus merged, for speed, the processor descriptor and the addressing descriptor.
Hierarchy. Permission to modify access at any one node of the hierarchy permits the holder to grant himself access to any-thing in the entire subtree based on that node.40

The hierarchical control scheme might be used in a time-sharing system as follows. The first access controller created is given an access control list naming one user, a system administra-
tor. The system administrator creates several access con-trollers (for example, one for each department in his company) and grants permission to modify access in each controller to the department administrator. The department administrator can create additional access controllers in a tree below the one for his department, perhaps for subdepartments or individual computer users in his department. These individual users can develop any pattern of sharing they wish, though the use of access control lists for the segments they create. In an emergency, however, the department administra-
tor can intervene and modify any access control list in his department. Similarly, the system administrator can intervene

40 The simplest way to handle the first access controller is to have it refer to itself. This approach provides self control at one point in the system, allowing for anticipated changes in authority as well and must be countered by careful planning by the system administrator.
in case a department administrator makes a mistake or is unavailable.\(^\text{41}\) The hierarchical system in our example is subject to the objection that the system administrator and department administrators are too powerful; any hierarchical arrangement inevitably leads to concentration of authority at the higher levels of the hierarchy. A hierarchical arrangement of authority actually corresponds fairly well to the way many organizations operate, but the hierarchical control method of modeling the organization has one severe drawback: the user and possible abuse of higher level authority is completely unchecked. In most societal organizations, higher level authority exists, but there are also checks on it. For example, a savings bank manager may be able to authorize a withdrawal despite a lost passbook, but only after advertising its loss in the newspaper. A creditor may remove money from a debtor's bank account, but only with a court order. A manager may open an employee's locked file cabinet, but (in some organizations) only after temporarily obtaining the key from a security office, an action which leaves a record in the form of a logbook entry. A policeman may search your house, but the search is illegal unless he first obtained a warrant. In each case, the authority to perform the operation exists, but the use of the authority is coupled with checks and balances designed to prevent abuse of the authority.Briefly, the hierarchical control scheme provides for exercise of authority but, as sketched so far, has no provision for preventing abuse of that authority.

One strategy that has been suggested in various forms \(^{58}\), \(^{59}\) is to add a field to an access control; which we may call the prescript field. Whenever an attempt is made to modify an access control list (either by a special store instruction or by a call to a supervisor entry, depending on the implementation), the access-modifying permission of the higher level access controller regulating the access control list is checked as always. If the permission exists, the prescript field of the access control list that is about to be modified is examined, and some action, depending on the value found, is automatically triggered. The following list suggests some possible actions that might be triggered by the prescript value, and some external policies that can be modeled with the prescript scheme.

1) No action.
2) Identifier of principal making change is logged (the "audit trail").
3) Change is delayed one day ("cooling-off" period).
4) Change is delayed until some other principal attempts the same change ("buddy" system).
5) Change is delayed until it is received from some specific (system-designated) principal ("court order").

The goal of all of the policies (and the prescript mechanism in general) is to ensure that some independent judgment moderates otherwise unfettered use of authority.

The notion of a prescript, while apparently essential to a protection system intended to model typical real authority structures, has not been very well developed in existing systems. The particular prescript mechanism we have used for illustration of the concept can model only a small range of policy situations. For example, it arranges that a prescript be invoked on every access to some segment, rather than just on changes in the authorizations. One could implement more complex policies by use of preprocessed subsystems, a general escaper mechanism described briefly in a later section.

5) Discretionary and Non discretionary Controls: Our discussion of authorization and authority structures has so far rested on an unarticulated assumption: the principal that creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. In the description of the selfcontrol scheme, for example, it was suggested that a newly created object begins its existence with one entry in its access control list, giving all permissions to its creator.

We may characterize this control pattern as discretionary\(^\text{42}\) implying that the individual user may, at his own discretion, determine who is authorized to access the object he creates. A variety of situations, discretionary control may not be acceptable and must be limited or prohibited. For example, the manager of a department developing a new product line may want to "compartmentalize" his department's use of the company computer system to ensure that only those employees with a need to know have access to information about the new product. The manager thus desists to apply the principle of least privilege. Similarly, the marketing manager may wish to compartmentalize use of the company computer for calculating product prices, since pricing policy may be sensitive. Either manager may consider it not acceptable that any individual employee within his department can abridge the compartmentalization decision merely by changing an access control list on an object he creates. The manager has a need to limit the use of discretionary controls by his employees. Any limits he imposes on authorization controls are that are out of the hands of his employees, and are viewed by them as nondiscretionary. Similar constraints are imposed in military security applications, in which not only isolated compartments are required but also entitled sensitivity levels (e.g., top secret, secret, confidential) that must be modeled in the authorization mechanisms of the computer system. Non discretionary controls may be needed in addition to or instead of discretionary controls. For example, the department manager may be prepared to allow his employees to adjust their access control lists any way they wish, within the constraint that no one outside the department is ever given access. In that case, both nondiscretionary and discretionary controls apply.

The key reason for interest in nondiscretionary controls is not so much the threat of malicious subordination as the need to safely use complex and sophisticated programs created by suppliers who are not under the manager's control. A contract software house may provide an AFP interpreter or a fast file sorting program. If the supplied program is to be useful, it must be given access to the data it is to manipulate or interpret. But unless the borrowed program has been completely audited, there is no way to be sure it does not access the data (for example, by making an illicit copy) or expose the data in a way that is either accidentally or intentionally. One way to prevent this kind of security violation would be to forbid the use of bor-

\(^{41}\) A term suggested by R. Schell \(^{40}\).
7/12. The dual strategy of maintaining a "low water mark" has been suggested as a way of monitoring the trustworthiness of a computer. The Maltese tiny-room-type register maintains such a low water mark on its address evaluation (63).
<table>
<thead>
<tr>
<th>TABLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TYPICAL SYSTEM-PROVIDED PROTECTED OBJECTS</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object</th>
<th>Typical Separately Permissible Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data segment</td>
<td>READ data from the segment</td>
</tr>
</tbody>
</table>
| Access controller | READ access control list | PROCESS READ
 Modify access control list | MODIFY PERMISSIONS ON ACCESS CONTROL LIST |
| FIFO message queue | PROCESS QUEUE | PROCESS MESSAGE |
| Input/Output device | READ data | WRITE data |
| Removable recording medium (e.g., magnetic tape unit) | READ data | WRITE data | WRITE data in new area |

queues and other segments can be made by introducing the concept of type in the protection system.

Consider, for example, the capability system in Fig. 6. Suppose we add a capability an extra field, which will name the type field. This field will have the value 1 if the object described by the capability is an ordinary segment, and the value 2 if the object is to be considered a queue. The protection descriptor registers are also expanded to contain a type field. We add to the processor the knowledge of which types are suitable as operands for each instruction. Thus the special instructions for manipulating queues require that the operand capability have type field 2, while all other instructions require an operand capability with type field 1. Further, the interpretation of the permission bits can be different for the queue type and the segment type. For the queue type, one might use the first permission bit to control use of the enqueue instruction and the second permission bit for the dequeue instruction. Finally, we should extend the "create" operation to permit specification of the type of object being created.

Clearly, one could extend the notion of type beyond segments and queues; any data structure could be similarly distinguished and protected from misuse. Further, input and output systems attached to interactive terminals, printers, and the like could be considered distinct types with their own repertoire of separately permitted operations. The concept of type extension is not restricted to capability systems; in an access control list system one could place the type field in the access controller and require that the processor present to the memory, along with each operand address, an indication of the type and permission bits required for the operation being performed. Table 1 lists some typical system-implemented objects and the kinds of operations one might selectively permit. This table could be extended to include other objects that are basically interpreted data structures, such as accounts or catalogs.

Finally, one may wish to extend dynamically the range of objects protected. Such a goal might be reached by making the type field large enough to contain an additional unique identifier, and allowing for software interpretation of the access to typed objects. This observation brings us to the subject of user-programmed controls on sharing and the implementation of protected objects and protected subsystems. We shall not attempt to examine this topic in depth, but rather only enough to learn what problems are encountered.

8. Protected Objects and Domains

Both the capability system and the access control list system allow controlled sharing of the objects implemented by the system. Several common patterns of use can be independently controlled, such as reading, writing, or running as a program. While it is a great improvement over "all-or-nothing" sharing, this sort of controlled sharing has two important limitations.

The first limitation is that only those access restrictions provided by the standard system facilities can be enforced. It is easy to imagine many cases where the standard controls are not sufficient. For example, an instructor who maintains his course grade records in a segment on an interactive system may wish to allow each student to read his own grades to verify correct recording of each assignment, but not the grades of other students, and to allow any student to examine the histogram of the class grades for each assignment. Implementing such controls within systems of the sort discussed in the last few sections would be awkward, requiring at least the creation of a separate segment for each student and for the distributions. If, in addition, the instructor wishes an assistant to enter new grades, but wants to guarantee that each grade entered cannot be changed later without the instructor's specific approval, we have a situation that is beyond the ability of the mechanisms so far described.

The mechanisms described so far cannot handle this situation because the manipulations we wish to perform on a grade or a set of grades are not fundamental operations of the base-level system. In essence, we wish to dynamically define a new type. The grade record, and provide a set of programs that interpretively implement the operations appropriate for this new type. The second limitation concerns users who borrow programs constructed by other users. Execution of a borrowed program in the borrower's domain can present a real danger to the borrower, for the borrowed program can exercise all the capabilities in the domain of the borrower. Thus a user must have a certain amount of faith in the provider of a program before he examines the program in his own domain.

The key to removing these limitations is the notion of a protected subsystem. A protected subsystem is a collection of programs and data segments that is "encapsulated" in such a way that no other executing programs cannot read or write the program and data segments and cannot disrupt the intended operation of the component programs, but can invoke the programs by calling designated entry points. The encapsulated data segments are the protected objects. Programs in a protected subsystem are enforcing arbitrary complex controls on access to them. Programs outside the protected subsystem are allowed to manipulate the protected objects only by invoking the entry points.
The notion of protected subsystems, then, provides mutual protection for multiple program complexes cooperating in the same computation and removes two limitations of facilities providing simple controlled sharing: 1.) unclear from a description of protected subsystems that each must operate in its own domain. Implementing protected subsystems requires mechanisms that allow the association of more than one domain with a computation and also requires means for changing from one protection domain to another as control passes from one protected subsystem to another. The design must ensure that one protected subsystem cannot interfere in any way with the correct operation of another subsystem involved in the same computation.

We note in passing that the supervisor in most computer systems is an example of a protected subsystem. If general facilities are provided for supporting user-constructed protected subsystems, then these mechanisms can be applied to protect the supervisor from user programs as well. Thus the protection mechanisms are protecting their own implementation. The resulting uniformity is consistent with the design principle of economy of mechanism. In order to implement protected subsystems, then, there must be a way of associating multiple domains with a single computation. One way would be to use a separate virtual processor, each with its own domain, for each protected subsystem, a notion proposed by Dennis and Van Hors [41] and discussed by Lampson [30]. A computation involving multiple protected subsystems would require multiple cooperating virtual processors. The invocation of one protected subsystem by another, and the communication of any response, would be done using the interprocessor communication facilities of the system [67]. An implementation using multiple virtual processors, however, is complicated and, in this case, tends to be awkward and inefficient in practice. Furthermore, it needs to obscure important features of the required mechanisms. Unless there is an inherent reason for the protected subsystems in a computation to be expressed as asynchronous activities, a single virtual processor implementation seems more natural. Such an implementation would require the association of multiple domains with a single virtual processor, a strategy proposed by LeClere [68], [69] and explored in detail by Lampson [19], Schroeder [70], Neelam [10], Stroess [17], Jones [17], and Rotenberg [59]. In this case, communication among protected subsystems could be via interprocessor call and return mechanisms.

The essence of changing domains is, in access control list terms, to change principal identifiers; in capability terms it is to acquire the set of capabilities of the new domain. In both cases, it is also essential that the virtual processor begin execution at some agreed-to starting point in the new domain.

Let us consider first an access control list implementation. Suppose we extend the possible permissions on a segment, as recorded in an access controller, to include ENTER permission, and add one more field to an access controller, the domain identifier, which is the principal identifier of the domain to be entered. The meaning of ENTER permission on a segment is that a virtual processor having only that permission may use (the first address in) that segment only as the target of a GO TO or CALL instruction. Further, upon executing a GO TO or CALL instruction, the processor will automatically pick up the domain identifier field in the access controller and use it as the principal identifier in transactions with the memory system.
We now have a controlled domain entry facility. A user wanting to provide a protected subsystem can do so by setting the access control lists of all objects that are to be internal parts of the system to contain one of his own principal identifiers. He also adds to the access control list of the initial procedure of his subsystem ENTER permission for any other principal who is allowed to use his protected subsystem.

In a capability system, a similar addition produces protected subsystems. The permission field of a capability is extended to include an entry permitting and, when capability is used as the target of a GO TO or a CALL instruction, control is passed to the procedure in the segment pointed to by the capability. Simultaneously with passing control to the procedure, the processor switches on the READ permission bit of the capability, thereby making available to the virtual processor a new domain—all those objects that can be reached starting from capabilities found in the procedure.

Two mechanisms introduced earlier can now be seen to be special cases of the general domain entry. In the initial discussion of the capability system, we noted that the authentication system starts a new user by allowing a virtual processor to enter that user's domain at a controlled starting point. We could use the domain entry mechanism to accomplish this result as follows. A system program is "listening" to all currently unused terminals or system ports. When a caller walks up to a terminal and attempts to use it, the system program creates a new virtual processor and has that processor enter the domain named by the prospective user. The entry point would be to a program, perhaps supplied by the user himself, which authenticates his identity before doing any other computation. Because a protected subsystem has been used, the program that monitors the unused terminal does not have access to the data in the protected subsystem (in contrast to the system of Fig. 7), a situation in better accord with the principle of least privilege. Instead, it has an entry capability for every domain that is intended to be entered from a terminal, but that capability leads only to a program that demands authentication.

We have sketched only the bare essentials of the mechanism required to provide domain switching. The full mechanism of a practical system that implements protected objects and subsystems is beyond the scope of this tutorial, but it is useful to sketch quickly some considerations those mechanisms must handle.

1) The principle of "separation of privilege" is basic to the idea that the same structured set of data objects is accessible to virtual processor 4, but only when the virtual processor is executing in program 7. For example, the protection system requires possession of a special capability before it allows access to the internal contents of some objects, then the program responsible for maintenance of the objects can hold the capability while the objects are being read. Morris [72] has described an elegant semantics for separation of privilege in which the first capability is known as a seal. In terms of the earlier discussion of types, the type field of a protected object contains a seal that is unique to the protected subsystem; access to the internal structure of an object can be achieved only by presenting the original seal capability as well as the capability for the object itself. This idea apparently was suggested by H. Strugis. The HY DRAM and CAL systems illustrate two different implementations of this principle.

2) The switching of protection domains by a virtual processor should be carefully coordinated with the mechanisms that provide for dynamic activation records and static (non-variable) storage, since both the activation records and the static storage of one protection domain must be distinct from that of another. (Consider multiple virtual processor implementations provide a neat automatic solution to these problems.)

The passing of arguments between domains must be carefully controlled to ensure that the called domain will be able to access its arguments without violating its own protection restrictions. Calls by value represent no special problem, but other forms of argument reference that require access to the original argument are harder. One argument that must be specially controlled is the one that indicates how to return to the calling domain. Schroeder [70] explored argument passing in depth from the access control point of view, while Jones [71] explored the same topic in the capability framework.

The easier interested in learning about the mechanisms of protected objects and subsystems in detail is referred to the literature mentioned above and in the Suggestions for Further Reading. This area is in a state of rapid development, and several ideas have been tried out experimentally, but there is not yet much agreement on which mechanisms are final.

For this reason, the subject is best explored by one study.
nal problems (c) (c) and the (f) follow: 31 implement (I) these must be profirable or use brute force techniques such as complete dedication of a system to a single task at a time [71]. The Department of Defense guide for safeguarding classified information stored in computers provides a good example of such brute force techniques [76]. In the decade between 1964 and 1974, several protection architectures were implemented as research and development projects, usually starting with a computer that provided only a privileged mode, adding minor hardware features and interacting with software the desired protection architecture. Among these were M.I.T.'s CTSS which, in 1961, implemented user authentication with all-or-nothing sharing and, in 1965, added shared files with permission lists [12]. In 1967, the ADEPT system of the System Development Corporation implemented in software on an IBM System/360 a model of the U.S. military security system complete with clearance levels, clearances, need-to-know, and centrallized authorization control [14]. At about the same time, the IBM Cambridge Scientific Center released an operating system named CP/67, user markeded under the name VM/370, that and descriptor-based hardware to implement virtual System/360 computers using a single System/360 Model 67 [11]. In 1969, the University of California at Berkeley CA system implemented a software-interpreted capability system on a Control Data 6400 computer [17]. Also in 1969, the Multics, a joint project of M.I.T. and Honeywell, implemented in software and hardware a complete descriptor-based access control list system with hierarchical control of authorization on a Honeywell 645 computer system [26], [17]. Based on the plans for Multics, the Hitachi Central Research Laboratory implemented a simplified descriptor-based system with hardware-implemented owned domains (rings of protection) on the IBTAC 5020E computer in 1968 [78]. In 1970, the Berkeley Computer Corporation also implemented rings of protection in the BCC 500 computer [19]. In 1973, a hardware version of the idea of rings of protection together with automatic argument address validation was implemented for Multics in the Honeywell 6100 [61]. At about the same time, the Plessey Corporation announced a telephone switching computer system, the Plessey 250 [53], based on a capability architecture. Current experimentation with new protection architectures is represented by the CAP system being built at Cambridge University [20] and the HYDRA system being built at Carnegie-Mellon University [21]. Recent research reports by Schroeder [70], Rottenberg [59], Spier et al [79], and Redell [54] propose new architectures that appear practical to implement.

B. Current Research Directions

Experimentation with different protection architectures has received little attention recently. Instead, the trend has been to concentrate in the following five areas: 1) certification of the correctness of protection system design and implementation 2) Inability to single faults, 3) constraints on use of information after release, 4) enforcement of information with secret keys, and 5) improved authentication mechanisms. These five areas are discussed in turn below. A research problem attracting much attention today is how to certify the correctness of the design and implementation of hardware and software protection mechanisms. There are actually several subproblems in this area.

1) One must have precise model of the protection goals of a system upon which to measure the design and implementation. When the goal is complete isolation of independent users, the model is straightforward and the mechanisms of the virtual machine are relatively easy to match with. When controlled sharing of information is desired, however, the model is much less clear and the attempts to clarify it generate many unsuspected questions of policy. Even attempts to model the well-documented military security system have led to surprisingly complex formulations and have exposed formidable implementation problems [14], [62].

2) Given a precise model of the protection goals of a system and a working implementation of that system, the next challenge is to verify somehow that the presented implementation actually does what it claims. Since protection functions are usually a kind of negative specification testing by sample cases provides almost no information. One proposed approach uses proofs of correctness to ensure formally that a system is implemented correctly. Most work in this area consists of attempts to extend methods of proving assertions about programs to cover the constructs typically encountered in operating systems [52].

3) Most current systems present the user with an intrinsically interface for specifying his protection needs. The result is that the user has trouble figuring out how to make the specification and verifying that he requested the right thing. User interfaces that more closely match the mental models people have of information protection are needed.

4) In most operating systems, at unreasonable large quantities of "system" software runs without protection constraints. The reasons are many: fancied higher efficiency, historical accidents, misunderstood design, and inadequate hardware support. The usual result is that the essential mechanisms that implement protection are thoroughly tangled with a much larger body of mechanisms, making certification impossible complex. In any case, a minimum set of protected supervisor functions a protected kernel has not yet been established for a full-scale modern operating system. Groups at M.I.T. [80] and at Mrite [81], [82] are working in this area.

Most modern operating systems are vulnerable in their reaction to hardware failures. Failures that cause the system to misbehave are usually easy to detect and, with experience, candidates for automatic recovery. Far more serious are failures that result in an undetected disabling of the protection mechanisms. Since routine use of the system may not include attempts to access things that should not be accessible, failures in checking circuitry may go unnoticed indefinitely. There is a challenging and probably solvable research problem involved in guaranteeing that protection mechanisms remain vulnerable in the face of all single hardware failures. Molcho [83] explored this topic in the IBM System 360/Model 50 computer and made several suggestions for its improvement. Fabry [84] has described an experimental "complete isolation" system in which all operating system decisions that could affect protection are duplicated by independent hard- ware and software. Another area of research concerns constraining the use with information may be put after its execution program. In Section II, we described such constraints as a fifth level of desired function. For example, one might wish to "tag" a file with a notation that any program reading that file is to be restricted forever after from putting output on remote terminals located outside the headquarters building.
For this restriction to be complete, it should propagate with all results created by the program and into other files it writes. Information concerning mechanisms such as these are common in legal agreements (as in the agreement between a taxpayer and a tax return preparing service) and the problem is to identify corresponding mechanisms for computer systems that could help enforce (or detect violations of) such agreements. Rotenberg explored this topic in depth [59] and proposed a "privacy restriction processor" to aid enforcement.

A potentially powerful technique for protecting information is to encrypt it using a key known only to authorized accessors of the information. (Thus encryption is basically a ticket-oriented system.) One research problem is how to communicate the keys to authorized users. If this communication is done inside the computer system, schemes for protecting the keys must be devised. Strategies for securing multinode computer communication networks using encryption are a topic of current research; Braustad has summarized the state of the art [40]. Another research problem is development of encryption techniques (sometimes called privacy transformations) for random access to data. Most well-understood encryption techniques operate sequentially on long bit streams (as found in point-to-point communications, for example). Techniques for encrypting and decrypting small, randomly selected groups of bits such as a simple word or byte of a file have been proposed, but finding simple and fast techniques that also require much effort to cryptanalyze (that is, with high work factors) is still a subject for research. A biviel encrypting system based on a scheme suggested by Feistel was developed at the IBM T. J. Watson Research Laboratory by Smith, Nott, and Oechsle [38]. One special difficulty in this area is that research in encryption encounters the practice of military classification. Since World War II, only three papers with significant contributions have appeared in the open literature [27], [39], [85]; other papers have only updated, reexplained, or rearranged concepts published many years earlier.

Finally, spurred by the need for better credit and check-cashing authentication, considerable research and development effort is going into better authentication mechanisms. Many of these strategies are based on attempts to measure some combination of personal attributes, such as the dynamics of a handwritten signature or the rhythm of keyboard typing. Others are directed toward developing machine-readable identification cards that we have to duplicate.

Work is in progress not well represented by published literature. The reader interested in further information on some of the current research projects mentioned may find useful the proceedings of two panel sessions at the 1974 National Computer Conference [86], [87], a recent workshop [88], and a survey paper [89].

C. Concluding Remarks

In reviewing the extent to which protection mechanisms are systematically understood (which is not a large extent) and the current state of the art, one cannot help but draw a parallel between current protection inventions and the first mass-produced computers of the 1920s. At that time, by virtue of experience and strongly developed intuition, designers had confidence that the architecture being designed were complete enough to be useful. And it turned out that they were. Even so, it was quickly established that matching a problem statement to the architecture—programme—was a major effort whose performance was quite sensitive to the exact architecture.

In a parallel way, matching a set of protection goals to a particular protection architecture by setting the bits and locations of access control lists or capabilities or by devising protected sub-systems is a matter of programming the architecture.

Following the parallel, it is not surprising that users of the current first crop of protection mechanisms have found them relatively clumsy to program and not especially well matched to the users' image of the problem to be solved, even though, the mechanisms may be sufficient. As in the case of all programming systems, it will be necessary for protection systems to be used and analyzed and for their users to propose different, better views of the necessary and sufficient semantics to support information protection.

ACKNOWLEDGMENT

R. Needham, A. Jones, J. Dennie, J. P. Anderson, B. Lindsey, L. Rotenberg, B. Lampson, D. Redell, and M. Wilkes carefully reviewed drafts of this manuscript and offered technical suggestions. In addition, the preparation of this paper was aided by discussions with many people including H. Forssdick, P. Larson, A. Huber, V. Vaidyappa, D. Mend, and R. Fabry. L. Schroeder ruthlessly edited out surplus jargon and prose

REFERENCES FOR FURTHER READING

The following short bibliography has been selected from the reference list to direct the reader to the most useful, up-to-date, and significant material currently available. Many of these readings have been collected and reprinted by L. J. Hoffman in [90]. The five bibliographies and collections (item 8 below) provide access to a vast collection of related literature.

1. Privacy and the impact of computers. [1]–[3], [91], [92].
2. Case studies of protection systems [14], [17], [20], [26], [61], [83], [84].
3. Protected objects and protected subsystems [30], [45], [54], [59], [70]–[72].
4. Protection with encryption [38]–[40], [91], [14].
5. Military security and noncooperative controls [63], [91], [95], [96].
6. Comprehensive discussions of all aspects of computer security [6]–[8].
7. Surveys of work in progress [86]–[89].
8. Bibliographies and collections on privacy and protection [91], [97]–[100].

REFERENCES

References are presented in order of first citation. The sections in which each reference is cited appear in parentheses following the reference. Methods for inclusion in additional references are presented in [67].

The Role of Rain in Satellite Communications

DAVID C. HOGG, FELLOW, IEEE, AND TA-SHING CHU, MEMBER, IEEE

Invited Paper

Abstract—The most fundamental stochastic encounter in the design of satellite communication systems at frequencies above 10 GHz is attenuation by rain. The microwave power radiated into an earth station is severely attenuated by a given amount of rainfall if the earth station and the satellite are in line of sight. This attenuation is a random variable, being limited by factors such as available primary power and size of earth station. The solution to this problem is still to be found. Techniques for dealing with the problem include diversity and spatial filtering, but these are known to reduce the diversity gain, so that the system is still subject to rain fading. The only way to avoid this problem is to design a system that is immune to rain fading. A system that is immune to rain fading is one that can operate satisfactorily in the presence of rain. This paper presents an overview of the work that has been done on the problem of rain fading in satellite communication systems. It is shown that the problem can be solved by using a diversity system, but that the diversity gain is limited by the amount of rain. The paper concludes with a discussion of the future of rain fading in satellite communication systems.

I. INTRODUCTION

In early experiments using microwave for broad-band transmission via satellites, it was quickly apparent that rain affected the performance of the system. For example, in the Telstar experiment [1], in which 4-GHz microwave signals were received, it was found that a 4-GHz frequency was affected by rain. The microwave power radiated into an earth station was severely attenuated by a given amount of rainfall if the earth station and the satellite were in line of sight. This attenuation is a random variable, being limited by factors such as available primary power and size of earth station. The solution to this problem is still to be found. Techniques for dealing with the problem include diversity and spatial filtering, but these are known to reduce the diversity gain, so that the system is still subject to rain fading. The only way to avoid this problem is to design a system that is immune to rain fading. A system that is immune to rain fading is one that can operate satisfactorily in the presence of rain. This paper presents an overview of the work that has been done on the problem of rain fading in satellite communication systems. It is shown that the problem can be solved by using a diversity system, but that the diversity gain is limited by the amount of rain. The paper concludes with a discussion of the future of rain fading in satellite communication systems.

1. A. Jonsen, "Prohibition in prehistoric times," Ph.D. disserta-


3. G. Andolf, G. Blauw, and F. Brocks, "Architecture of the

IBM System/360," IBM J. Res. Dev., vol. 3, pp. 47-70,

Apr. 1969. (IEEE-ESF)


5. R. Beatty, B. and G. Pappas, "Disconnection: An approach to

operating system security," in Proc. ACM 17th Ann. Conf.,

pp. 464-468. (IEEE-ESF)

6. G. Tschudin, "Technique and Procedures for Precipita-

tion Measurement, Decontamination, Testing, and Evaluating


1972. (IEEE-ESF)

7. H. Graham, "Protection in an information processing utility," 


(IEEE-ESF)

9. E. Spier, T. Hastings, and D. Cutler, "An experimental inte-

gration of the human/human architecture," ACM Operating


10. H.T. Prot. MAN, "Computer systems research," in Project


(IEEE-ESF)


14. B. Fahey, "Software verification of operating system designs," 


15. C. Shannon, "Communication theory of secrecy systems," Bell


17. K. Smith et al., "A panel session—Research in data security,


vol. 4, pp. 993-996. (IEEE-ESF)

18. J. Nolte, et al., "Intrusion detection and protection in computer


(IEEE-ESF)


(IEEE-ESF)

23. J. Metten, "Cryptology, computers, and common sense." in


24. J. Anderson, "Computer security technology planning study,


(IEEE-ESF)

25. W. W. Ware, "Security controls for computer systems," Air


26. R. Anderson and E. Pagalo, "Privacy and the computer: A

motivated bibliographer," ACM Comp. Rev., vol. 13, pp. 529-


27. L. Nasser, A. M. Diano, and D. Harris, "An motivated and tran-

smission-encrypted bibliography on computer security and auto-

control in computer systems," Ohio State Univ., Computer

and Information Science Report, Ohio State Univ., CSC/COM 72-15,

1972. (IEEE-ESF)


29. J. Scharf, "Computer and disk base security: A comprehensive


1974. (IEEE-ESF)