# 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 applications. Many of these new applications involve both storing information and simultaneous use by several individuals. The key concern in this paper is multiple use. For those applications in which all users should not have identical authority, some scheme is needed to ensure that the computer system implements the desired authority structure.

For example, in an airline seat reservation system, a reservation 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 passenger list from a law enforcement agency is reviewed by the correct level of management.

The airline example is one of protection of corporate information for corporate self-protection (or public interest, depending on one's view). A different kind of example is an online warehouse inventory management system that generates reports about the current status of the inventory. These reports 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 banks; law enforcement information systems; time-sharing service bureaus; 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 implement 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 administered code of ethics or a lack of knowledge about computers adequately protects the stored information. Although there are situations in which the computer

need provide no aids 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-storing systems. Not all authors use these terms in the same way. This paper uses definitions commonly encountered in computer science 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.[1]

*The term "security" describes techniques that control who may use or modify the computer or the information contained in it.*[2]

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 occurs contrary to the desire 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-accessed 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 subsequent interpretation of electromagnetic radiation,
4. enciphering 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 programs to stored information.[3] An example of a protection technique is labeling of computer-stored files with lists of authorized users. Similarly, the term *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 paper concentrates 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 all 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 appear in the strategy. In contrast, a narrow concentration on protection mechanisms, especially those logically impossible to defeat, may lead to false confidence in the system as a whole.[4]

*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 the following.

a) Unprotected systems: Some systems have no provision for preventing a determined user from having access to every 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, commercially available batch data processing systems fall into this 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 since it is common for unprotected systems to contain a variety of mistake-prevention features. These may provide just enough control that any breach of control is likely to be the result of a deliberate act rather than an accident. Nevertheless, it would be a mistake to claim that such systems provide any security.[5]

b) All-or-nothing systems: These are systems that provide isolation of users, sometimes moderated by total sharing of some pieces of information. If only isolation is provided, the user of such a system might just as well be using his own private computer, as far as protection and sharing of information are concerned. More commonly, such systems also have public libraries 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. Most of the first generation of commercial timesharing systems provide a protection scheme with this level of function. Examples include the Dartmouth Time-Sharing System (DTSS) [10] and IBM's VM/370 system [11]. There are innumerable 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][6]

d) User-programmed sharing controls: A user may want to restrict access to a file in a way not provided in the standard facilities for controlling sharing. For example, he may wish to permit access only on weekdays

between 9:00 A.M. and 4:00 P.M. Possibly, he may wish to permit access to only the average value of the data in a file. Maybe he wishes to require that a file be modified only if two users agree. For such cases, and a myriad of others, a general escape is to provide for user-defined *protected objects* and *subsystems*. A *protected subsystem* 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 specified entry points. Thus the programs of the subsystem completely control the operations performed on the data. By constructing a protected subsystem, a user can develop any programmable form of access control to the objects he creates. Only a few of the most advanced system designs have tried to permit user-specified protected subsystems. These include Honeywell's Multics [16], the University of California's CAL system [17], Bell Laboratories' UNIX system [18], the Berkeley Computer Corporation BCC-500 [19], and two systems currently under construction: the CAP system of Cambridge University [20], and the HYDRA system of Carnegie-Mellon University [21]. Exploring 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 ever having direct access to the data itself. Protection based on content is the subject of a variety of recent or current research projects [22]-[25] 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 income information to a tax advisor; constraints should prevent him from passing the information on 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 constraint on 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 consideration that cuts across all levels of functional capability: the *dynamics of use*. This term refers to how one establishes and changes the specification of who may access what. At any of the levels it is

relatively easy to envision (and design) systems that statically express a particular protection intent. But the need to change access authorization dynamically and the need for such changes to be requested 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 if O'Hara has access to file X, but also 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 has 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 contracts, ignorance, or barbed wire fences may provide some of the required functional capability. This discussion, however, is focused on the internal mechanisms.

*3) Design Principles:* Whatever the level of functionality provided, the usefulness of a set of protection mechanisms depends upon the ability 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 such unauthorized acts has proved to be extremely difficult. Sophisticated users of 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 all have shown that users can construct programs that can obtain unauthorized access to information stored within. Even in systems designed and implemented with security as an important objective, design and implementation flaws provide paths that circumvent the intended access constraints. Design and construction 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 *all* 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 apply particularly to protection mechanisms.[7]

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

b) Fail-safe defaults: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965,[8] 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 system 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 considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

c) Complete mediation: Every access to every object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control, which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.

d) Open design: The design should not be secret [27]. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose.[9] Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

e) Separation of privilege: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extendible protected data types usually depend on separation of privilege for their implementation.

f) Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.

g) Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users [28]. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it.

h) Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.

Analysts of traditional physical security systems have suggested two further design principles which, unfortunately, apply only imperfectly to computer systems.

a) Work factor: Compare the cost of circumventing the mechanism with the resources of a potential attacker. The cost of circumventing, commonly known as the "work factor," in some cases can be easily calculated. For example, the number of experiments needed to try all possible four letter alphabetic passwords is $26^4 = 456\ 976$. If the potential attacker must enter each experimental password at a terminal, one might consider a four-letter password to be adequate. On the other hand, if the attacker could use a large computer capable of trying a million passwords per second, as might be the case where industrial espionage or military security is being considered, a four-letter password would be a minor barrier for a potential intruder. The trouble with the work factor principle is that many computer protection mechanisms are *not* susceptible to direct work factor calculation, since defeating them by systematic attack may be logically impossible. Defeat can be accomplished only by indirect strategies, such as waiting for an accidental hardware failure or searching for an error in implementation. Reliable estimates of the length of such a wait or search are very difficult to make.

b) Compromise recording: It is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss. For example, if a tactical plan is known to have been compromised, it may be possible to construct a different one, rendering the compromised version worthless. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism. Although the information stored inside may be easy to obtain, the cabinet will inevitably be damaged in the process and the next legitimate user will detect the loss. For another example, many computer systems record the date and time of the most recent use of each file. If this record is tamperproof and reported to the owner, it may help discover unauthorized use. In computer systems, this approach is used rarely, since it is difficult to guarantee discovery once security is broken. Physical damage usually is not involved, and logical damage (and internally stored records of tampering) can be undone by a clever attacker.[10]

As is apparent, these principles do not represent absolute rules--they serve best as warnings. If some part of a design violates a principle, the violation is a symptom of potential trouble, and the design should be carefully reviewed to be sure that the trouble has been accounted for or is unimportant.

*4) Summary of Considerations Surrounding 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 knows how to build a system without flaws, the 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, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability.

Finally, some protection designs can be evaluated by comparing the resources of a potential attacker with the work factor required to defeat the system, and compromise recording 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 *ad 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 that 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 using 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 dynamics. The final model (only superficially explored) is of protected objects and protected subsystems, which allow arbitrary modes of sharing that are unanticipated by the system designer. These models are not intended so much to explain the particular systems as they are to explain the underlying concepts of information protection.

Our emphasis throughout the development 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 protected 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 mechanics are typically quite different. The beginnings 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 computer system is not a single object. When one is considering direct access, the information is divided into mutually exclusive partitions, as specified by its various creators. Each partition contains a collection of information, all of which is intended to be protected uniformly. The uniformity of protection is the same kind of uniformity that applies to all of the diamonds stored in the same 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 fundamental objects to be protected.

Conceptually, then, it is necessary to build an impenetrable wall around each distinct object that warrants separate protection, construct a door in the wall through which access can be obtained, and post a guard at the door to control its use. Control of use, however, requires that the guard 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 general model.

Before extending this model, we pause to consider two concrete 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 consists 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, since during the intervals 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 progress 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 be 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 checked by an extra 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 *base* value and a *bound* value. The base is the lowest numbered address the program may use, and the bound is the number 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 memory. 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 concept of a descriptor: it is central to most implementations of protection and of sharing of information.[12]

So far, we have not provided for the dynamics of a complete protection scheme: we have not discussed who loads the descriptor register. If any running program could load it with any arbitrary value, there would be no protection. The instruction that loads the descriptor register with a new descriptor must have some special controls--either on the values it will load or on who may use it. It is easier to control who may use the descriptor, and a common scheme is to introduce an additional bit in the processor state. This bit is called the *privileged state* bit.[13] All attempts to load the descriptor register are checked against the value of the privileged state bit; the privileged state bit must be ON for the register to be changed. One program (named the *supervisor*--program S in Fig. 1) runs with the privileged state bit ON, and controls the simulation of the virtual processors for the other programs. All that is needed to make the scheme complete is to ensure that the privileged state bit cannot be changed by the user programs except, perhaps, by an instruction that simultaneously transfers control to the supervisor program at a planned entry location. (In most implementations, the descriptor register is not used in the privileged state.)

One might expect the supervisor program to maintain a table of values of descriptors, one for each virtual processor. When the privileged state bit is OFF, the index in this table of the program currently in control identifies exactly which program--and thus which virtual processor--is accountable for the activity of the real processor. For protection to be complete, a virtual processor must not be able to change arbitrarily the values in the table of descriptors. If we suppose the table to be stored inside the supervisor program, it will be inaccessible to the virtual processors. We have here an example of a common strategy and sometime cause of confusion: the protection mechanisms not only protect one user from another, *they may also protect their own*

*implementation*. We shall encounter this strategy again.

So far, this virtual processor implementation contains three protection mechanisms that we can associate with our abstractions. For the first, the information being protected is the distinct programs of Fig. 1. The guard is represented by the extra piece of hardware that enforces the descriptor restriction. The impenetrable wall with a door is the hardware that forces all references to memory through the descriptor mechanism. The authority check on a request to access memory is very simple. The requesting virtual processor is identified by the base and bound values in the descriptor register, and the guard checks that the memory location to which access is requested lies within the indicated area of memory.

The second mechanism protects the contents of the descriptor register. The wall, door, and guard are implemented in hardware, as with the first mechanism. An executing program requesting to load the descriptor register is identified by the privileged state bit. If this bit is OFF, indicating that the requester is a user program, then the guard does not allow the register to be loaded. If this bit is ON, indicating that the requester is the supervisor program, then the guard does allow it.

The third mechanism protects the privileged state bit. It allows an executing program identified by the privileged state bit being OFF (a user program) to perform the single operation "turn privileged state bit ON and transfer to the supervisor program." An executing program identified by the privileged state bit being ON is allowed to turn the bit OFF. This third mechanism is an embryonic form of the sophisticated protection mechanisms required to implement protected subsystems. The supervisor program is an example of a protected subsystem, of which more will be said later.

The supervisor program is part of all three protection mechanisms, for it is responsible for maintaining the integrity of the identifications manifest in the descriptor register and the privileged state bit. If 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 bit 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 permanent storage can be added to such a system by attaching some longterm storage device (e.g., magnetic disk) 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 programs 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.[14]

Long-term storage does, however, force us to face one further issue. Suppose that the virtual machine communicates 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, padlocks can control access to 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. Conversely, 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 secrecy or the unforgeability of the user's identification.

In time-sharing systems, the most common scheme depends on secrecy. The user begins by typing the name of the person he claims to be, and then the system demands that the user type a password, presumably known only to that person.

There are, of course, many possible elaborations and embellishments of this basic strategy. In cases where the typing of the password may be observed, passwords may be good for only one use, and the user carries a list of passwords, crossing each one off the list as he uses it. Passwords may have an expiration date, or usage count, to limit the length of usefulness of a compromised one.

The list of acceptable passwords is a piece of information that must be carefully guarded by the system. In some systems, all passwords are passed through a hard-to-invert transformation[15] before being stored, an idea suggested by R. Needham [37, p. 129]. When the user types his password, the system transforms it also and compares the transformed versions. Since the transform is supposed to be hard to invert (even if the transform itself is well known), if the stored version of a password is compromised, it may be very difficult to determine what original password is involved. It should be noted, however, that "hardness of inversion" is difficult to measure. The attacker of such a system does not need to discern the general inversion, only the particular one applying to some transformed password he has available.

Passwords as a general technique have some notorious defects. The most often mentioned defect lies in choice of password--if a person chooses his own password, he may choose something easily guessed by someone else who knows his habits. In one recent study of some 300 self-chosen passwords on a typical time-sharing system, more than 50 percent were found to be short enough to guess by exhaustion, derived from the owner's name, or something closely associated with the owner, such as his telephone number or birth date. For this reason, some systems have programs that generate random sequences of letters for use as passwords. They may even require that all passwords be system-generated and changed frequently. On the other hand, frequently changed random sequences of letters are hard to memorize, so such systems tend to cause users to make written copies of their passwords, inviting compromise. One solution to this problem is to provide a generator of "pronounceable" random passwords based on digraph or higher order frequency statistics [26] to make memorization easier.

A second significant defect is that the password must be exposed to be used. In systems where the terminal is distant from the computer, the password must be sent through some communication system, during which passage a wiretapper may be able to intercept it.

An alternative approach to secrecy is unforgeability. The user is given a key, or magnetically striped plastic card, or some other unique and relatively difficult-to-fabricate object. The terminal has an input device that examines the object and transmits its unique identifying code to the computer system, which treats the code as a password that need not be kept secret. Proposals have been made for fingerprint readers and dynamic signature readers in order to increase the effort required for forgery.

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 a secret after all. On the other hand, the scheme is convenient, 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 interceptor'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 communication with some unsurprising 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 that 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 may consist of a sequence of 1000 binary digits read from a magnetically striped plastic card. In order that a recipient of such an enciphered signal may comprehend it, he must have a deciphering circuit primed with an exact copy of the transformation key, or else he must cryptanalyze the scrambled stream to try to discover the key. The strategy of encipherment/decipherment is usually invoked 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 Feistel [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 this 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 unintelligible streams of bits. If the exchange succeeds, the computer 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 transmitted through the communication system. If

communication fails (because the user is unauthorized, the system has been replaced by a masquerader, or an error occurred), each party to the transaction has immediate warning of a problem.[16]

Relatively complex elaborations of these various strategies have been implemented, differing both in economics and in assumptions about the psychology of the prospective user. For example, Branstad [40] explored in detail strategies of authentication in multinode computer networks. Such elaborations, though fascinating to study and analyze, are diversionary to our main topic of protection mechanisms.

*5) Shared Information:* The virtual machines of the earlier section were totally independent, as far as information accessibility was concerned. Each user might just as well have his own private computer system. With the steadily declining costs 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 useful, or even essential. For example, there may be a library of commonly used, reliable programs. Some users may create new programs that 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 implement a biomedical statistics system. In all these cases, virtual machines are inadequate, because of the total isolation of their users from one another. Before extending the virtual machine example any further, let us return to our abstract discussion of guards and walls.

Implementations of protection mechanisms that permit sharing fall into the two general categories described by Wilkes [37]

a) "List-oriented" implementations, in which the guard holds a list of identifiers of authorized users, and the user carries a unique unforgeable identifier that must appear on the guard's list for access to be permitted. A store clerk checking a list of credit customers is an example of a list-oriented implementation in practice. 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] corresponding to the objects to which he has been authorized access. A locked door that opens with a key is probably the most common example of a ticket-oriented mechanism; the guard is implemented as the hardware of the lock, and the matching key is the (presumably) unforgeable authorizing identifier.

*Authorization*, defined as giving a user access to some object, is different in these two schemes. In a list-oriented system, a user 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 mechanism 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 places 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 need 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 comparison, list-oriented mechanisms are not often used in applications where traffic is high. On the other hand, ticket-oriented mechanisms typically require considerable technology to control 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 arrangement is accomplished by providing, at the higher level, a list-oriented guard[18] 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 authorizations, as represented in the two systems, synchronized with each other. Computer protection systems differ mostly in the extent to which the architecture of the underlying 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 again have a "complete isolation" kind of protection system. Thus the "complete 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 interacting protection mechanisms, some of which are list-oriented, some of which are ticket-oriented, and some of which provide complete isolation and therefore may happen to be implemented as degenerate examples of either of the other two, depending on local circumstances.

We should understand the relationship of a user to these transactions. We are concerned with protection of information from programs that are executing. The user is the individual who assumes accountability for the actions of an executing program. Inside the computer system, a program is executed by a virtual processor, so one or more virtual processors can be identified with the activities directed by the user.[19]

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 two 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 abstraction for that accountability--the *principal*. A principal is, by definition, the entity accountable for the activities of a virtual processor.[20] In the situations discussed so far, the principal corresponds to the user outside the system. However, there are situations in which a 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 occasion he may wish to use the computer for some purpose unrelated to the valuable information. To prevent accidents, he may wish to identify himself with a different principal, one that does not have access to the valuable information--following the principle of least privilege. In this case there is a need for two different principals corresponding to the same user.

Similarly, one can envision a data base that is to be modified only if a committee agrees. Thus 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 see later (in the section on dynamic authorization of sharing) that authorizing access is done in terms of principals. That is, if one wishes a friend to have access to some 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 a 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 authentication 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 that principal, which is considered accountable for all further actions of these virtual processors. The authentication mechanism has allowed the virtual processor to enter the domain of that principal. That description makes apparent the importance of the authentication mechanism. Clearly, one must carefully control the conditions under which a virtual processor enters a domain.

Finally, we should note that in a ticket-oriented system there is no mechanical need to associate an unforgeable identifier with a virtual processor, since the tickets themselves are presumed 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 identifier for that principal. Thus accountability in ticket-oriented systems can be difficult to pinpoint.

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 subroutine. 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 subtly, 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 *master copy* of the program. The storage space will be smaller and the communication of updated versions will be easier.

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, placing the math routine somewhere in memory by itself and placing a descriptor for it in the second descriptor register. Following the previous strategy, we assume that the privileged state bit assures that the supervisor program is the only one permitted to load either descriptor register. In addition, some scheme must be provided in the architecture of the processor to permit a choice of which descriptor register is 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 addresses in the lower half of the address range would be interpreted relative to descriptor register 1, and addresses in the upper half of the address range would be relative to descriptor register 2. An alternate scheme, suggested by Dennis [42], is to add explicitly to the format of instruction words a field that selects the descriptor register intended to be used with the address in that instruction. The use of descriptors for sharing information is intimately related to the addressing architecture of the processor, a relation that can cause considerable confusion. The reason why descriptors are of interest for sharing becomes apparent by comparing

parts a and b of Fig. 2. When program A is in control, it can have access only to itself and the math routine; similarly, when program B is in control, it can have access only to itself and the math routine. Since neither program has the power to change the descriptor register, sharing of the math routine has been accomplished while maintaining isolation of program A from program B.

The effect of sharing is shown even more graphically in Fig. 3, which is Fig. 2 redrawn with two virtual processors, one executing program A and the other executing program B.

Whether or not there are actually two processors is less important than the existence of the conceptually parallel access paths implied by Fig. 3. Every virtual processor of the system may be viewed as having its own real processor, capable of access to the memory in parallel with that of every other virtual processor. There may be an underlying processor multiplexing facility that distributes a few real processors among the many virtual processors, but such a multiplexing facility is essentially unrelated to protection. Recall that a virtual processor is not permitted to load its own protection descriptor registers. Instead, it must call or trap to the supervisor program S which call or trap causes the privileged state bit to go ON and thereby permits the supervisor program to control the extent of sharing among virtual processors. The processor multiplexing facility must be prepared to switch the entire state of the real processor from one virtual processor to another, including the values of the protection descriptor registers.

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 $P_1$ can overwrite the shared math routine, then it could disrupt the work of virtual processor $P_2$.

2) The shared math routine must be careful about making modifications to itself and about where 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 supervisor 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 shared area of memory is a procedure, then to avoid the possibility that virtual processor $P_1$ will maliciously overwrite it, we can restrict the methods of access. Virtual processor $P_1$ needs to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants embedded in the program, but it has no need to write into any part of the shared procedure. We may accomplish this restriction by extending the descriptor registers and the descriptors themselves to include *accessing permission*, an idea introduced for different reasons in the original Burroughs B5000 design [32] . For example, we may add two bits, one controlling permission to read and the other permission to write in the storage area defined by each descriptor, as in Fig. 4. In virtual processor $P_1$ of Fig. 3, descriptor 1 would have both permissions granted, while descriptor 2 would permit only reading of data and execution of instructions.[21] 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 less satisfactory because, unlike the descriptors so far outlined, permission bits attached to the data would provide identical access to all processors that had a descriptor. Although identical access for all users of the shared math routine of Figs. 1-2-3 might be acceptable, a data base could not be set up with several users having permission to read but a few also having permission to write.

The second implication of a shared procedure, mentioned before, is that the shared procedure must be careful about where it stores temporary results, since it may be used simultaneously by several virtual processors. In particular, it should avoid modifying itself. The enforcement of access permission by descriptor bits further constrains the situation. To prevent program A from writing into the shared math routine, we have also prohibited the shared math routine from writing into itself, since the descriptors do not change when, for example, program A transfers control to the math routine.[22] The math routine will find that it can read but not write into itself, but that it can both read and write into the area of program A. Thus program A might allocate an area of its own address range for the math routine to use as temporary storage.[23]

As for the third implication, 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 supervisor 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. Most real systems use a combination of these two forms, the capability system for speed and an access control list system for the human interface. Before we can pursue these generalizations, and 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 copyable descriptors as tickets to provide a flexible authorization scheme. In this context we establish the general rule that communication external to the computer must precede dynamic authorization of sharing. The limitations of copyable descriptors--primarily lack of accountability for their use--lead to analysis of revocation and the observation that revocation requires indirection. That observation in turn leads to the model of access control lists embedded in indirect objects so as to provide detailed control of authorization.

The use of access control lists leads to a discussion of controlling changes to authorizations, there being at least two models of control methods which differ in their susceptibility to abuse. Additional control of authorization changes is needed when releasing sensitive data to a borrowed program, and this additional control implies a nonintuitive constraint on where data may be written by the borrowed program. Finally, Section II explores the concept of implementing arbitrary abstractions, such as extended types of objects, as programs in separate domains.