II. DESCRIPTOR-BASED PROTECTION SYSTEMS

A. Separation of Addressing and Protection[24]

As mentioned earlier, descriptors have been introduced here for the purpose of protecting information, although they are also used in some systems to organize addressing and storage allocation. For the present, it is useful to separate such organizational uses of descriptors from their protective use by requiring that all memory accesses go through two levels of descriptors. In many implementations, the two levels are actually merged into one, and the same descriptors serve both organizational and protection purposes.

Conceptually, we may achieve this separation by enlarging the function of the memory system to provide uniquely identified (and thus distinctly addressed) storage areas, commonly known as *segments*. For each segment there must be a distinct addressing descriptor, and we will consider the set of addressing descriptors to be part of the memory system, as in Fig. 5. Every collection of data items worthy of a distinct name, distinct scope of existence, or distinct protection would be placed in a different segment, and the memory system itself would be addressed with two-component addresses: a unique segment identifier (to be used as a key by the memory system to look up the appropriate descriptor) and an offset address that indicates which part of the segment is to be read or written. All users of the memory system would use the same addressing descriptors, and these descriptors would have no permission bits--only a base and a bound value. This scheme is functionally similar to that used in the Burroughs B5700/ 6700 or Honeywell Multics systems in that it provides a structured addressing space with an opportunity for systematic and automatic storage allocation.

The unique identifiers used to label segments are an essential cornerstone of this organization. They will be used by the protection system to identify segments, so they must never be reused. One way of implementing unique identifiers is to provide a hardware counter register that operates as a clock (counting, say, microseconds) and is large enough never to overflow in the lifetime of the memory system. The value of the clock register at the time a segment is created can be used as that segment's unique identifier.[25] As long as the memory system remembers anything, the time base of the clock register must not be changed.

The processor of Fig. 5 contains, as part of its state, protection descriptors similar to those of Figs. 1 and 2, with the addition of permissions, as in Fig. 4. All references by the processor are constrained to be to segments described by these protection descriptors. The protection descriptor itself no longer contains a base and bound; instead the descriptor contains the unique segment identifier that the memory system requires as the first part

of its two-part address for accessing that segment. Thus, from the point of view of a program stored in one of the segments of memory, this system is indistinguishable from that of Fig. 2. Note in Fig. 5 that although *addressing* descriptors exist for the segments containing program B and program S (the supervisor), they are not accessible to the processor since it has no *protection* descriptors for those two segments. It is useful to distinguish between the system address space, consisting of all the segments in the memory system, and the processor address space, consisting of those segments for which protection descriptors exist. Since the addressing descriptors are part of the memory system, which is shared by all processors, the system address space is 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 supervisor 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 system 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 Capabilities: 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].[26] 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 previously 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 OFF, the word in that location is an ordinary data or instruction word. If the bit is ON, 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 it 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 ON, and we must arrange that all other store instructions set the tag bit OFF in any memory location they write into. This gives us two kinds of objects stored in the 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.[27]

This particular tagged architecture is known as a *capability system,* one that lets the user place protection descriptor values in memory addresses that are convenient to him. A memory word that contains a protection descriptor value (in our simple tagged system, one that has its tag bit ON) is known as a *capability*.

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, Cl and C2. Program A may direct the processor to load the capability at location C2 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 Cl into a protection descriptor register, after which the processor may address the segment labeled "Private Data Base X." By a similar chain of reasoning, another processor starting with a capability for the segment labeled "Catalog for Smith" can address both the shared math routine and the segment "Private Data Base Y."

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 others receive capabilities permitting only reading from that same segment. The catalog segment actually might contain 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 calling the supervisor. The supervisor by convention might set some protection descriptor to contain a capability for the new segment.[28] The user could then file his new segment by storing this new capability in his catalog along with a name for the segment. Thus we have an example of a primitive but usable filing system to go with the basic protection structure.[29]

To complete the picture, we should provide a tie to some authentication mechanism. Suppose that the system responds to an authentication request by creating 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 a user identifies himself as "Doe" and supplies a password, the supervisor program can look up his identification in the user identification table. It can verify the password and load into a protection descriptor register the capability for the catalog associated with Doe's entry in the user identification table. Next, it would clear the remaining capability registers, destroying the capability for the user identification table, and start running some program in Doe's directory, say program A. Program A can extend its addressability to any segment for 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 A.

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 catalog 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 character-string identifier "Doe" is no longer used; the associated catalog capability is sufficient. The replacement of the character-string form of the principal identifier is possible because the full range of accessible objects for this user has already been opened up to him by virtue of his acquisition of his catalog capability. The catalog capability becomes, in effect, the principal identifier. On the other hand, some loss of accountability has occurred. It is no longer quite so easy, by examining the registers of a running virtual processor, to establish who is accountable for its activity. This lack of accountability will have to be repaired in order to allow the virtual processor to negotiate the acquisition of new capabilities.

With this example of a capability system, a catalog is not a special object. It is merely any segment in which any program chooses to store capabilities that are, by virtue of their tags, protected unforgeable objects. If in Fig. 7, program A, running under Doe's control, creates a new object, it may choose to place the new capability in segment X in a position where it can easily be found later. In such a case, segment X has become, in effect, another catalog. To establish the full range of objects that Doe may address, it is necessary to examine not only the initial catalog segment, whose capability is contained in the user identification table, but also all segments it contains capabilities for, and all segments they contain capabilities for, etc.

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 is a topic that must be examined quite carefully, since it exposes 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 Doe 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 reading 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 way one thing is crucial--that there be a secure path from Doe, who is authorizing the passing of the capability, to the program, which is carrying it out. Next, we should ask, where does Doe find out 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.[30] 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, Doe's principal identifier should be the name used in Smith's catalog of communication segments, so Smith can be certain that only some program run under Doe's accountability could possibly have sent the capability. In summary, here is a complete protocol for dynamically authorizing sharing of a new object.

Sender's part:

1. Sender learns receiver's principal identifier via a communication path outside the system.
2. Sender transmits receiver's principal identifier to some program running inside the system under the accountability of the sender.
3. Sender's program uses receiver's principal identifier to ensure that only virtual processors operating under the accountability 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 transmits sender's principal identifier to some program running inside the system under the accountability of the receiver.
3. Receiver's program uses the sender's principal identifier to ensure that only a virtual processor operating under the accountability of the sender could have sent the capability being received.

This protocol provides protection for the authorization changing 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 authorizing sharing has been in terms of private communication segments between every pair of users, a strategy which would lead, with N users, to some N^2 communication segments. To avoid this square-law growth, one might prefer to use some scheme that dynamically constructs the communication paths also, such as having special hardware or a protected subsystem that implements a single "mailbox segment" for each user to receive messages and capabilities sent by all other users. Of course, the mechanism that implements the mailbox segments must be a protected, reliable mechanism, since it must

infallibly determine the principal identifier of the sender of a message and label the message with that identifier, so the receiver can reliably carry out his step 3) of the protocol. Similarly, as 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 intent in his step 3) of the protocol is carried out correctly.

3) Revocation 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/I and Algol 68 fit naturally into a capability-based framework by using capabilities as address (pointer) variables. Straightforward additions to the capability system allow it gracefully to implement languages with dynamic-type extension [21] . Flexibility comes from the defining property of a capability system: the user may decide which of his addresses are to contain capabilities. The user can develop a data structure with an 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 capability that Smith now has stored away in some unknown location. Unless we provide additional control, his only recourse is to destroy the original segment, an action which may be disruptive to other users, still trusted, who also have copies of the capability. Thus *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 recipient 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 copies of the necessary capability. That search would be only the beginning, since there may be many paths by which users could reach those capability copies. Every such path must be found, a task that may involve a fair amount of computation and that also completely bypasses the protection mechanisms. Thus *review* of access is a problem.[31]

To help counter these problems, constraints on the use of capabilities have been proposed or 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 gives another user access to a capability with the copy bit OFF, then the second user could not make copies of the capability he has 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 segments may be targets of the instructions that load and store descriptor registers. This scheme may reduce drastically the effort involved in auditing and make revocation possible, since only capability-holding segments need be examined. (The CAP system [20] and the Plessey 250 [53] 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 programmer-visible distinction between short-term processor-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 they may be stored only in processor-addressable memory. This restriction not only reduces the effort required for auditing, but also limits the lifetime of a capability to that of a virtual processor. When the system shuts down, the only memory of the system is in long-term storage and all capabilities vanish. Of course, the next time the system starts up, newly created virtual processors need some way (such as appeal to an 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 protection 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 place 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 loading. Any attempt to increase a depth count beyond, say, three, would constitute an error, and the processor would fault. In this way, the depth counters limit the length of the chain by which a capability may propagate. Again, this form of constraint reduces the effort of auditing, since one must trace chains back only a fixed number of steps to get a list of all potential accessors. (The M.I.T. CTSS used a software version of this scheme, with a depth limit of two.)

To gain more precise control of *revocation*, Redell [54] has proposed that the basic capability mechanism be extended to include the possibility of forcing a capability to specify its target indirectly through a second location before reaching the actual object of interest. This second location would be an independently addressable recognizable object, and anyone with an appropriate capability for it could destroy the indirect object, revoking access to anyone else who had been given a capability for that indirect object. By constructing a separate indirect object for each different principal he shared an object with, the owner of the object could

maintain the ability to revoke access independently for each principal. The indirect objects would be implemented within the memory-mapping hardware (e.g., the addressing descriptors of Fig. 5) both to allow high-speed bypassing if frequent multiple indirections occur and also to allow the user of a capability to be ignorant of the existence of the indirection.[32] Redell's indirect objects are closely related to the *access controllers* of the access control list system, described in the next subsection. While providing a systematic revocation strategy (if their user develops a protocol for systematically using them), the indirect objects provide only slight help for the problems of propagation and auditing.

The basic trouble being encountered is that an authorization--a kind of binding--takes place any time a capability is copied. Unless an indirect object is created for the copy, there is no provision for reversing this binding. The ability to make a further copy (and potentially a new authorization) is coupled to possession of a capability and is not independently controllable. Restrictions on the ability to copy, while helping to limit the number or kind of authorizations, also hamper the simplicity, flexibility, and uniformity of capabilities as addresses. In particular, capabilities are especially useful as a way of communicating exactly the necessary arguments from one procedure to another. In this way, they encourage wide use of procedures, a cornerstone of good programming practice. Restrictions on copyability, then, inhibit their usefulness in the context of procedure calls, and that runs counter to the goal of providing base-level facilities that encourage good programming practice. This dilemma seems to present an opportunity for research. At the present level of understanding, the most effective way of preserving some of the useful properties of capabilities is to limit their free copyability to the bottom most implementation layer of a computer system, where the lifetime and scope of the bindings can be controlled. The authorizations implemented by the capability system are then systematically maintained as an image of some higher level authorization description, usually some kind of an access control list system, which provides for direct and continuous control of all permission bindings.[33]

C. The Access Control List System

1) Access Controllers: The usual strategy for providing reversibility 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 mechanizations, and we must choose one for illustration. For ease of discussion, we will describe a mechanism implemented completely in hardware (perhaps by 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 to couple an access control list system to a capability system, a more typical realization that reduces the number of extra checks.

The system of Fig. 5 identified 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. 8. An addressing descriptor for the access controller itself is assigned a unique identifier and placed in the map used by the memory system to locate objects. The access controller is 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 registers can be replaced with unprotected *pointer registers*, which can be loaded from any addressable location with arbitrary bit patterns. (In terms of IBM System 370 and Honeywell Multics, the pointer registers contain *segment numbers* from a universal address space. The segment numbers lead to the segment addressing descriptors stored in the access controller.) Of course, only bit patterns corresponding to the unique identifier of some segment's access controller will work. A data reference by the processor proceeds in the following steps, keyed to Fig. 9.

1. The program encounters an instruction that would write in the segment described by pointer register 3 at offset k.
2. The processor uses the unique identifier found in pointer register 3 to address access controller AC_1. The processor at the same time presents to the memory system the user's principal identifier, a request to write, and the offset k.
3. The memory system searches the access control list in AC_1 to see if this user's principal identifier is recorded there.
4. 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.
5. If writing is permitted, the addressing descriptor of segment X, stored in AC_1, and the original offset 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 accountable for 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 X has known, auditable consequences. Doe cannot make a copy of the addressing descriptor of segment X since he does not have direct access to it, eliminating propagation of direct access. The pointer to 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.[34]
2. The access control list directly implements the sender's third step of the dynamic sharing protocol--verifying that the requester is authorized to use the object. In the capability system, verification was done once to decide if the first 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 preludes all future attempts by that user to use that segment.
4. The question of "who may access this segment?" apparently is answered directly by examining the access control list in the access controller for the segment. The qualifier "apparently" applies because we have not yet postulated any mechanism for controlling who may modify access control lists.
5. All unnecessary association between data organization and authorization has been broken. For example, although a catalog may be considered to "belong" to a particular user, the segments appearing in that catalog can have different access control lists. It follows that the grouping of segments for naming, searching, and archiving purposes can be independent of 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 strategies to simplify implementation with minimum compromise of functions, but first it will be helpful to introduce one more functional property-protection groups.

2) Protection Groups: Cases often arise where it would be inconvenient to list by name every individual who is

to have access to a particular segment, either because the list would be awkwardly long or because the list would change frequently. To handle this situation, most access control list systems implement factoring into *protection groups*, which are principals that may 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 extend 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 control list checker so that it searches for a match between any of the principal identifiers and any entries of the access control list.[35] Finally, who is 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 other 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.[36]

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 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.[37] 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 contra list check.) The result is a highly constrained but very fast capabitity system beneath the access control list system. The detailed checking of access control falls on the capability mechanism, which on individual memory references exactly enforces 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, constrain all 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 permissions for these three entries are specified by the program creating the segment.[38]

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 [55] 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 interpretively [57].

4) 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 was devised to provide more precise control of authority, so some mechanism of exerting 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 control* 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-an entry giving all permissions to the principal identifier associated with the creating virtual processor. The creator receives a pointer for the access controller he has just created, and then can adjust its access control list to contain any desired list of principal identifiers and permissions.[39]

Probably the chief objection is to the self-control approach is that it is so absolute: there is no provision for graceful changes of authority not anticipated by the creator of an access control list. For example, in a commercial time-sharing system, if a key member of a company's financial department is taken ill, there may be no way for his manager to authorize temporary access to a stored budget file for a co-worker unless the absent user had the foresight to set his access control lists just right. (Worse yet would be the possibility of accidentally producing an object for which its access controller permits access to no one--another version of the garbage collection problem.) To answer these objections, the *hierarchical control* scheme is sometimes used.

To obtain a hierarchical control scheme, whenever a new object is created the creator must specify some previously existing access controller to regulate future changes to the access control list in the access controller for the new object. The representation of an access controller must also be expanded to contain some kind of pointer to the access controller that regulates it (for example, a unique identifier). In addition, the interpretation of the permission bit named "ACLmod" is changed to apply to those access controllers that hierarchically are immediately below the access controller containing the permission bit. Then, as in Fig. 13, all of the access controllers of the system will be arranged in a hierarchy, or tree structure, branching from the first access controller in the system, whose creation must be handled as a special case, since there is no previously existing access controller to regulate it. The hierarchical arrangement is now the pattern of access control, since a user with permission to modify access control lists may add his own principal identifier, with permission to modify access, to lower level access controllers, giving himself ability to change access control lists still further down the hierarchy. Permission to modify access at any one node of the hierarchy permits the holder to grant himself access to anything in the entire subtree based on that node.[40]

The hierarchical control scheme might be used in a timesharing system as follows. The first access controller created is given an access control list naming one user, a system administrator. The system administrator creates several access controllers (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, through the use of access control lists in access controllers, for the segments they create. In an emergency, however, the department administrator can intevene and modify any access control list in his department. Similarly, the system administrator can intervene in case a department administrator makes a mistake or is unavailable.[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 use 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. In brief, 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 controller, 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 signal 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 or proposed computer systems. The particular prescript mechanism we have used for illustration of the concept can model easily only a small range of policies. One could, for example, arrange that a prescript be invoked on every access to some segment, rather than just on changes in the authority structure. One could implement more complex policies by use of protected subsystems, a general escape mechanism described briefly in a later section.

5) Discretionary and Nondiscretionary Controls: Our discussion of authorization and authority structures has so far rested on an unstated 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 self-control 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*[42] implying that the individual user may, at his own discretion, determine who is authorized to access the objects he creates. In 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 desires to apply the principle of least privilege. Similarly, the marketing manager may wish to compartmentalize all 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 are controls 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 nested *sensitivity levels* (e.g., top secret, secret, and confidential) that must be modeled in the authorization mechanics of the computer system. Nondiscretionary controls may need to be imposed 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 insubordination 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 APL 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 that it does not misuse the data (for example, by making an illicit copy) or expose the data either accidentally or intentionally. One way to prevent this kind of security violation would be to forbid the use of borrowed programs, but for most organizations the requirement that all programs be locally written (or even thoroughly audited) would be an unbearable economic burden. The alternative is *confinement* of the borrowed program, a term introduced by Lampson [61]. That is, the borrowed program should run in a domain containing the necessary data, but should be constrained so that it cannot authorize sharing of anything found or created in that domain with other domains.

Complete elimination of discretionary controls is easy to accomplish. For example, if self-controlling access controllers are being used, one could arrange that the initial value for the access control list of all newly created objects not give "ACL-mod" permission to the creating principal (under which the borrowed program is running). Then the borrowed program could not release information by copying it into an object that it creates and then adjusting the access control list on that object. If, in addition, all previously existing objects in the domain of the borrowed program do not permit that principal to modify the access control list, the borrowed program would have no discretionary control at all and the borrower would have complete control. A similar modification to the hierarchical control system can also be designed.

It is harder to arrange for the coexistence of discretionary and nondiscretionary controls. Nondiscretionary controls may be implemented, for example, with a second access control list system operating in parallel with the first discretionary control system, but using a different authority control pattern. Access to an object would be permitted only if both access control list systems agreed. Such an approach, using a fully general access control list for nondiscretionary controls, may be more elaborate than necessary. The few designs that have appeared so far have taken advantage of a perceived property of some applications of nondiscretionary controls: the desired patterns usually are relatively simple, such as "divide the activities of this system into six totally isolated compartments." It is then practical to provide a simplified access control list system to operate in parallel with the discretionary control machinery.

An interesting requirement for a nondiscretionary control system that implements isolated compartments arises whenever a principal is authorized to access two or more compartments simultaneously, and some data objects may be labeled as being simultaneously in two or more compartments (e.g., pricing data for a new product may be labeled as requiring access to the "pricing policy" compartment as well as the "new product line" compartment). In such a case it would seem reasonable that, before permitting reading of data from an object, the control mechanics should require that the set of compartments of the object being referenced be a subset of the compartments to which the accessor is authorized. However, a more stringent interpretation is required for permission to write, if borrowed programs are to be confined. Confinement requires that the virtual processor be constrained to write only into objects that have a compartment set identical to that of the virtual processor itself. If such a restriction were not enforced, a malicious borrowed program could, upon reading data labeled for both the "pricing policy" and the "new product line" compartments, make a copy of part of it in a segment labeled only "pricing policy," thereby compromising the "new product line'' compartment boundary. A similar set of restrictions on writing can be expressed for sensitivity levels; a complete and systematic analysis in the military security context was developed by Weissman [14]. He suggested that the problem be solved by automatically labeling any written object with the compartment labels needed to permit writing, a strategy he

named the "high water mark." As an alternative, the strategy suggested by Bell and LaPadula [62] declared that attempts to write into objects with too few compartment labels are errors that cause the program to stop.[43] Both cases recognize that writing into objects that do not have the necessary compartment labels represents potential "declassification" of sensitive information. Declassification should occur only after human judgment has been interposed to establish that the particular information to be written is not sensitive. Developing a systematic way to interpose such human judgments is a research topic.

Complete confinement of a program in a shared system is very difficult, or perhaps impossible, to accomplish, since the program may be able to signal to other users by strategies more subtle than writing into shared segments. For example, the program may intentionally vary its paging rate in a way users outside the compartment can observe, or it may simply stop, causing its user to go back to the original author for help, thereby revealing the fact that it stopped. D. Edwards characterized this problem with the phrase "banging on the walls." Lampson [61], Rotenberg [59], and Fenton [64] have explored this problem in some depth.

D. Protecting Objects Other Than Segments

So far, it has been useful to frame our discussion of protection in terms of protecting segments, which basically are arbitrary-sized units of memory with no internal structure. Capabilities and access control lists can protect other kinds of objects also. In Fig. 9, access controllers themselves were treated as system-implemented objects, and in Fig. 13 they were protected by other access controllers. It is appropriate to protect many other kinds of objects provided by the hardware and software of computer systems. To protect an object other than a segment, one must first establish what kinds of operations can be performed on the object, and then work out an appropriate set of permissions for those operations. For a data segment, the separately controllable operations we have used in our examples are those of reading and writing the contents.

For an example of a different kind of system-implemented object, suppose that the processor is augmented with instructions that manipulate the contents of a segment as a first-in, first-out queue. These instructions might interpret the first few words of the segment as pointers or counters, and the remainder as a storage area for items placed in the queue. One might provide two special instructions, "enqueue" and "dequeue," which add to and remove from the queue. Typically, both of these operations would need to both read and write various parts of the segment being used as a queue.

As described so far, the enqueue and dequeue instructions would indiscriminately treat any segment as a queue, given only that the program issuing the instruction had loaded a capability permitting reading and

writing the segment. One could not set up a segment so that some users could only enqueue messages, and not be able to dequeue--or even directly read--messages left by others. Such a distinction between 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 to a capability an extra field, which we 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 streams 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 I 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.

TABLE I

Typical System-Provided Protected Objects

Object	Typical Separately Permittable Operations
Data segment	READ data from the segment WRITE data into the segment Use any capability found in the segment Write a capability into the segment

Access controller	Read access control list Modify names appearing on an access control list Modify permissions in access control list entries Destroy objects protected by this access controller
FIFO message queue	Enqueue a message Dequeue a message Examine queue contents without dequeueing
Input/Output	READ data WRITE data Issue device-control commands
Remove recording medium (e.g. magnetic tape reel)	READ data WRITE over data WRITE data in new area

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.

E. 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.[44]

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 executes 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 program and data segments that is "encapsulated" so that 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 can act as caretakers for the protected objects and interpretively enforce arbitrarily complex controls on access to them. Programs outside the protected subsystem are allowed to manipulate the protected objects only by invoking the care taker programs. Algorithms in these caretaker programs may perform any appropriate operation, possibly depending on the circumstances of invocation, and may even record each access request in some way in some protected objects. For example, the protected subsystem shown in Fig. 14 implements the grade keeping system discussed above. Clearly, any access constraints that can be specified in an algorithm can be implemented in this fashion. Giving users the ability to construct protected subsystems out of their own program and data segments allows users to provide arbitrary controls on sharing.

If programs inside a protected subsystem can invoke programs in another protected subsystem without compromising the security of the first subsystem, then we can plug together multiple protected subsystems to perform a computation. We also find a way around the borrowed program problem. The normal domain of a user is one example of a protected subsystem. The user arranges for programs borrowed from other users to

execute outside of this "home" protected subsystem. In this way, the borrowed programs can be invoked without giving them access to all the programs and data of the borrower. If the borrowed program is malicious or malfunctions, the damage it can do is limited. The lending user could also encapsulate the lent program complex in a protected subsystem of its own and thus insulate it from the programs of the borrower.[45]

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. It is clear from the 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 Horn [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, though conceptually straightforward, tends to be awkward and inefficient in practice. Furthermore, it tends 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 LeClerc [68], [69] and explored in detail by Lampson [19], Schroeder [70],Needham [20],Sturgis [17], Jones [71], and Rotenberg [59] . In this case, communication among protected subsystems could be via interprocedure call and return operations.

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 wishing 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 principals who are 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 ENTER permission, and when a 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. Simultaneous 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 user 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 terminals does not have access to the data in the protected subsystem (in contrast with the system of Fig. 7), a situation in better accord with the principle of least privilege. Instead, it has an enter 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 mechanics of a practical system that implements protected objects and subsystems are beyond the scope of this tutorial, but it is useful to sketch quickly the considerations those mechanisms must handle.

1. The principle of "separation of privilege" is basic to the idea that the internal structure of some data objects is accessible to virtual processor A, but only when the virtual processor is executing in program B. If, for example, the protection system requires possession of two capabilities before it allows access to the internal contents of some objects, then the program responsible for maintenance of the objects can hold one of the capabilities while the user of the program can hold the other. 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. Sturgis. The HYDRA 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 (own) variable storage, since both the activation records and the static storage of one protection domain must be distinct from that of another. (Using a multiple virtual processor implementation provides a neat automatic solution to these problems.)
3. 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 intentions. 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 especially controlled is the one that indicates how to return to the calling domain. Schroeder [70] explored argument passing in depth from the access control list point of view, while Jones [71] explored the same topic in the capability framework.

The reader interested in learning about the mechanics 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 fundamental. For this reason, the subject is best explored by case study.