A SURVEY OF PROTECTION SYSTEMS

Charles B. Weinstock

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

July, 1973

This work supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

One of the more important tasks that an operating system performs is the protection of programs and data from interference (intentional or not) by other programs. Recently it has been realized that it is also desirable to protect a program from itself. A survey of techniques used to achieve this protection is presented. Works of Dennis and Van Horn, Graham, Lampson, Wulf, and Morris are discussed. Others are addressed more briefly.

INTRODUCTION

In days of old, the story goes, there was an undergraduate student who came up with a brilliant solution to the problem of getting more computer time for himself. He merely took control of the machine away from the monitor. To avoid detection, he made the machine look as if it were processing a normal job stream. In fact it was devoting almost all of its time to processing his job. When he was through executing, he would give control back to the monitor and it would continue as if nothing had happened. This trick went undetected for quite a while. The student was only discovered because he had the misfortune to be running at the same time that a systems programmer was debugging a new feature in the monitor.

There are two ways that you can take care of this kind of problem. You can do what the Computation Center did (hire the user), or you can design protection mechanisms into the monitor, making it impossible to take control away in the first place.

With single user batch processing it was only necessary to protect the system from the user to prevent occurances such as theft of resources. With the advent of multi-programming it became desirable to protect the users from each other as well. Now, with sophisticated timesharing and multi-processing systems, it would be nice to allow selective sharing of resources among users.

Ideally, it should be possible for the Democrats and the Republicans to store their confidential records on an ITT owned computer located in the Watergate building and operated by a member of the SDS with complete assurance that no unauthorized person could access them. Furthermore each should be able to use the same information retrieval system for accessing their data, again without fear of theft of information.

Additionally, individual users should have the ability to selectively protect parts of their programs from other parts. For example, a debugging system should be totally protected from the program being debugged, but the reverse is not true. The debugger needs access to the program in order to set breakpoints or insert patches.

This paper is a survey of the various methods which are currently in use to implement protection of the nature described in the preceding paragraphs.

HISTORY

Early computers had no special hardware for protection purposes. This meant that for any protection to exist, a software machine had to be defined with the proper facilities. Then all user programs had to run interpretively. Obviously this was very inefficient and therefore very little protection was ever available on these machines.

It soon became apparent that something more was needed, so computers were designed to operate in two modes, a problem state and a supervisor state. The idea was that the supervisor would have unrestricted access to the entire computer, but the user program would only have restricted access. There were many implementations of this idea, some of which will be described here.

A number of schemes involved base limit registers, a register with fields to accommodate the base address and the length of a segment. Machines having one limit register would use it to define the upper and lower bounds of the problem program area. Machines with two limit registers (the PDP-10 is an example) would use one to define the limits of a data area, and the other to define the limits of the program area. This achieved the ability to protect and share pure code. There were other schemes involving more limit registers to further segment the users core.

The IBM 360 series of computers solved the problem without limit registers. Core

was divided into equal size chunks each with a 4 bit register called the storage key. A user program could read or execute any address in the machine. However, the supervisor assigned the program a protection key while running, and the only addresses it could modify were in chunks with a storage key that matched the protection key. The supervisor, of course, could modify anything, anywhere.

Wilkes discusses these primitive protection techniques in some detail [Wil68]. The major problem with all of these schemes (from a protection point of view) is that any user that has access to an area has the same kind of access as any other user. Hence to share anything with a feeling of safety implies that the sharing users be infallible and completely trustworthy. The fact is that, with few notable exceptions, most commercially available systems go no further than this. Only recently has hardware been designed with advanced protection ideas in mind. The remainder of this paper is concerned with a description of hardware and software of a more sophisticated nature.

PROTECTION SOFTWARE FOR OPERATING SYSTEMS

Dennis and Van Horn

The protection systems currently in use can be divided into two catagories. In one catagory a process has access to an object (whatever that may be) if it has a ticket allowing it. In the other catagory, each object has associated with it a list of those processes that may access it. Wilkes [Wil68] likens these catagories to methods for running a public library. A person who wants to borrow books can be issued a library card, in which case possession of this ticket is proof of borrowing privileges. On the other hand, a master directory of all persons allowed to borrow books can be

kept at the front door and only those whose name appears on the list admitted. In this case presence inside the building is proof enough.

The first references to the "ticket" form of protection appeared in an article by Dennis and Van Horn [Den66]. They define the sphere of protection of a computation as a list of the objects that the computation is able to touch in some manner. That is, a list of objects for which the computation possesses tickets. Each entry in this list is called a capability and the list is called the C-list. The C-list may be thought of as an extended segment table. With appropriate hardware, implementation is simple. A capability tells the computation where the object is and specifies operations that the computation is allowed to perform on it. Thus it is possible to make a given object execute only, read only, execute and read, read-write or execute read and write. In addition, the capability specifies whether or not the computation is the owner of the object. If it is, it has special privileges, such as the right to grant capabilities for access to the object to other computations.

It should be noted here that an object need not be in core. It might be a file on a storage device. One of the things necessary for capabilities to be meaningful on this type of object, it must be possible to restrict a computation from doing its own I/O. Otherwise, there would be nothing to prevent a program from accessing a file by other than normal channels. Thus a capability is necessary to issue I/O commands.

As mentioned earlier, it is often useful to have different levels of protection within the same computation. An example is the debugger discussed above. It should be in a different sphere of protection than the program being debugged. Dennis and Van Horn allow a computation to define inferior spheres of protection by creating a new C-list. It is then possible to grant capabilities to the inferior sphere. It should be obvious that for all practical purposes this defines a separate process.

Means are provided for processing exceptions caused by inferior spheres. For instance if the process halts or encounters a debugger breakpoint, the superior sphere is notified. The superior sphere then has the ability to completely examine and modify the inferior spheres environment. In particular it may revoke capabilities.

Another feature that Dennis and Van Horn claim is necessary for complete protection is protected entry points. If a computation could jump to an arbitrary point in a protected computation, undesirable results would ensue. Thus the jump is only allowed to designated spots called entry points. For one computation to jump to a specific location in another computation, the location must be an entry point and the calling computation must possess a capability for that entry point

To provide all of these features Dennis and Van Horn define a number of meta-instructions to be implemented in the protected supervisor.

&caham

In 1968, Robert Graham used the ideas of Dennis and Van Horn to develop a protection scheme for Multics[Gra68]. Multics is designed as a large information processing utility (IPU) serving many users with diverse interests on possibly more than one processor. Efficiency is important in such a utility. Just as users of a telephone system get upset when they have to wait for a dial tone, users of the IPU would not tolerate the delays imposed by inefficiency. For this reason interpretation was ruled out,and the need for, special hardware was recognized.

In addition to the principles mentioned before, a principle found in Grahams design is that access is controlled by a "need to know" policy similar to that of the military. That is, each process should have the minimum amount of access that it takes to get the job done. In particular, when sharing is not necessary, a process should be completely isolated.

In line with these ideas and several others, a modified capability scheme seemed resonable. Thus Graham describes descriptors and descriptor segments which are essentially capabilities and C-lists with some modification. The sphere of computation (or domain) of a process is then defined by the descriptor segment, the address of which is in the descriptor base register while the process is actually running.

To help accomplish the "need to know" policy, layers of protection called rings are defined. The rings are assigned numbers from 0 to some maximum value. The lower the number, the more protected the ring is. Procedures executing in a ring have absolutely no access to segments assigned to lower rings. They do, however, have controlled access to segments in their own or higher rings. The implication of this is that it is possible for a process to define parts of itself that are to be protected from other parts. To accomplish this, a field is added to the descriptor for a segment. This is the ring number that the segment is in. The field is also added to the location counter so that at any moment he knows exactly in which ring a process is executing.

To enforce these restrictions, it is important to be able to detect a "change of ring" condition in the process. To do this requires that the hardware produce a fault whenever a change is attempted. Then software can make the approprate checks.

This, Graham claims, is enough to implement a solution to the protection problem. However, keeping in mind the efficiency requirements, he proposes a modification to take care of the following problem. There is a class of shared utility routines which need only as much access as the calling program. In this case the routine will run correctly in whatever ring the calling procedure is running in. Thus, he can avoid needless faults by allowing a procedure segment to be assigned to a consecutive set of rings. To do this he modifies the ring number field to contain an upper and a lower ring bound. This is called the access bracket, and any procedure running in one ring

can transfer to a segment whose access bracket includes that ring without causing a fault. For data segments there is another interpretation for the access bracket. If a procedure is running in the lower bound ring (for the data segment) or below, read/write access is allowed. If it is running within the access bracket, but above the lower bound, read only access is allowed. If it is running in a ring higher than the upper bound, it may have no access at all. All of this is also subject to further restrictions in the access control information contained in the descriptor.

To achieve further control over transfers between rings, a tflll bracket is implemented in the software. The call bracket is merely a consecutive set of rings immediately above the access bracket of the segment, possibly empty, from which a restricted jump into the segment is to be allowed. Recall that any attempt to transfer to a segment whose access bracket upper bound is lower than the number of the ring that the calling process is executing in, results in a fault. A system support process called the gatekeeper monitors these faults. When it sees that the calling process is within the call bracket, it allows the transfer only if it is to a location mentioned in the gate list. This is merely a list of those locations to which the called segment is prepared to take transfers from processes executing in its call bracket. Of course a transfer to any location is valid if the process is executing within the access bracket.

In summary then, the major difference between the Dennis and Van Horn and the Graham models is that rings are introduced as a means of achieving layers of protection. Of course the ability to define unlimited inferior spheres of protection is more desirable, but Multics was designed to be practical, whereas the system proposed by Dennis and Van Horn is mostly theoretical. In practice it seems that nested hierarchies are enough for the kinds of things that are normally done on timesharing systems.

Lampson

Butler Lampson worked more recently in the field of protection [Lam69a,69b,71]. The first two papers deal with proposed implementations of actual systems; the last deals with protection in the abstract. Since most of the same concepts appear in each of the papers (with possibly different terminology) they will be discussed together. Most of what follows is derived from [Lam71]. The reader is invited to refer to [Lam69a,69b] to see how the ideas have progressed with time.

Lampson recognizes that for protection purposes, the words "program" and "user" are equivalent. That is, statements about why protection is needed will still be meaningful when "program" is interchanged with "user" in the text. To this end he points out that enforcement of the rules of modular programming and application of a protection system are beneficial. One consequence of this is that debugging is made much easier.

Lampson describes an idealized message system similar to that of the RC4000 discussed by Hansen[Han70]. It consists of processes that share nothing but can communicate with each other by means of messages. A message is composed of an unforgeable senders ID, and the message part.

This is adequate to simulate protected suboutine calls. To call another process, the caller sends it a message. The called process then has the option of deciding if it wishes to be called by the caller. If not, it throws away the message unread. The problem of a process that was never called returning is also avoided. In his earlier papers, this required a protected call stack. With the message system it becomes easier because the calling routine knows when it expects a return and can ignore all others in its mailbox. Finally, by invoking a "trusted" process to send back a message after some elapsed time, the calling program can protect itself from a subroutine that

never returns. Implicit in all of this, of course, is the access list approach to protection.

Lampson points out two major problems with this system. The first is esoteric. It is impossible to regain control of runaway processes. The worst that this can do is waste resources and perhaps make debugging harder. The other problem is much more important; it is painful to get processes to cooperate. A process that wishes to be shared must maintain and search a list of who is allowed to do so.

Therefore, it is necessary to have a systematic way of controlling access to a process. It is also necessary to have a systematic way of describing what is to be shared by whom. Lampson proposes as a solution another idealized process he calls the object system. It has three major components: a set of objects with unique names that must be protected, a set of domains, and an access matrix A. The rows of A are indexed by domain names, the columns by object names. Thus the elements of A describe the types of access, if any, that domain d has to object x. Note that domains are also considered objects.

Now he adds a new behavior to the message system. He requires that if a domain owns an object x, it will do certain things to x upon demand by a domain that has access to x. This allows the sharing of objects. The rules will be enforced by the underlying system.

Each element of the access matrix has a set of indicators that tell what the domain is allowed to do to the object. Associated with each indicator in the element is a copy flag. This flag says whether or not the domain is allowed to transfer the right to another domain. Other rules are:

1) A domain d_1 can remove access attributes from $A[d_2,x]$ if it has control access to d_2.

2) A domain d_1 can copy to $A[d_2,x]$ any access attributes that it has for x that

have the copy flag set, and can decide whether or not the new attribute will have the copy flag set.

3) If d_1 has owner access to x, it can add any access attributes to $A[d_2,x]$.

4) If desired, we can allow d_1 to remove any access attribute from $A[d_2,x]$ if it has owner access to x, and provided that d_2 does not have protected access to x. This latter is to allow one owner to defend his rights from the other owners.

Note that A is probably sparse. The simplest alternative to keeping the whole matrix is to keep triples <d,x,A[d,x]>. That this is also inefficient should be obvious. Lampson proposes our old standby, the capability, as an alternative. This, of course, attaches the protection information to the domain. Thus he keeps pairs <x,A[d,x]> with the domain.

The other approach is attaching the protection information to the object being protected. Thus he would keep pairs <d,A[d,x]> with the object instead. Since it would be expensive and inconvenient to keep a list of each domain allowed access, he modifies this approach somewhat. He allows capabilities to be used as identification instead of the unique name of the domain. This will not violate protection standards because it is impossible to forge a capability. When used this way he calls the capability an access key. Note that the access key is merely a generalization of the domain name. Then all the access control procedure need know is what keys to recognize.

When a group of domains want access to x, the owner merely gives them the same access key (generated by the system). This method is quite different from the capability method. It is also likely to be more expensive. Thus some systems, including those described in Lampson's earlier papers, have a hybrid implementation in which an access key is used once to obtain a capability. When this process is applied to files, it is usually called "opening" the file.

Wulf et. al.

Hydra, the operating system that runs on C.mmp has been recently described by Wulf[Wul73a]. Actually, Hydra is the kernel of a group of operating systems: a set of primitives considered common to any of a number of possible operating system designs. One of the major facilities in any operating system is its protection mechanism. A main part of the Hydra design philosophy was to provide a complete set of strong, reliable, and flexible protection primitives. The realization of this design provides a powerful protection structure in which arbitrary types of protection can be implemented.

The Hydra primitives, provide, as special cases, all of the normal types of protection (e.g. Read Only, Read/Write, Execute Only). It is also easy to provide new and useful types of protection. For example, it might be desirable to give some users customer-name-only access to a mailing list.

In Hydra, protection is applied to resources. The user may define his own types of resources. Examples of resources are procedures, files, disks, pages, bibliographies, "gorps" and "thuds". A specific incarnation of an resource is called an object.

An object in Hydra is divided into several parts. The major ones are; a unique name, a type, and a representation. The unique name is different for every object ever created by Hydra.

Each type of resource has a distinguished object that is the representative of the resource's equivalence class. This object is of type "type". All objects in the same equivalence class have a type field that is merely the unique name of the class's representative. Thus, to define a new type all a user need do is create an object to represent that type.

The representation of an object is also divided into two parts. The first of these

is the data part. This part is uninterpreted by the system and may be manipulated by the user with appropriate access to the object. The second part is the item part which is merely a list of (protected) references to other objects. The program has no control over this part, no matter what kind of access it has. Either of these two parts may be empty for some particular object.

The item part of the representation of an object consists of a list of items which are similar to capabilities in that they are references to objects. These items play an important part in Hydra's protection mechanism. They determine what kind of access a program may have to an object. In addition to the name of the object being referenced, they contain a set of bits indicating access rights called the rights list. One of the things that makes the Hydra system unique is that it assigns no meaning to the rights list. It is able to determine if a protection failure has occurred without any interpretation.

One of the object types defined by the kernel is the Local Name Space or LNS. There is a different LNS for each executing procedure in the system. An instance of a LNS defines the execution environment for the associated process at a given instant; it is essentially the C-list of the program.

Objects are manipulated in Hydra by operations. The procedure is the abstraction of an operation. Thus, the user can also define his own operations (for example the customer-name-only access mailing list operation) by writing procedures.

Each bit in the rights list of an item is associated with an operation. If the bit is on, the operation may be requested. If it is off, requesting the operation results in a protection failure. Note that it is only necessary to keep a list of operation/bit pairs, for each user, to determine if a failure has occurred. It is not necessary to know what the operation does.

A procedure, of course, is an object like everything else. Its data part consists of the pure code of the procedure. Its item part serves as a prototype for the LNS that the procedure will have when executing. This prototype contains two kinds of items. The first group is the items for referencing all objects that the procedure needs regardless of who calls it. These are called the caller independent capabilities. The other kind of item is the templates for items which will be passed to the procedure as parameters when it is called. The template contains the type of the expected parameter, and a list of the minimal rights that a caller must have for the object passed. If either of these conditions is not satisfied at call time, a protection failure results. The template also contains a list of rights to the object that the called procedure will require. At call time, if the call is allowed, these rights will be merged with the callers rights to form the rights list of the actual item. Thus, the called procedure may have greater freedom to operate on an object than the calling procedure.

Two primitives are defined in the Kernel to effect procedure calls, CALL and RETURN. To call a procedure the executing program notifies the system that it wishes to do so by executing a CALL. The system checks the parameters for protection failures. If all is well with the parameters, it creates a new LNS for the process using the called procedures item part as a prototype, and pushing the old LNS onto a stack. This defines the new environment. The called procedure then receives control and executes. When it is done, it notifies the system by issuing a RETURN. The system then pops the LNS stack and returns to the caller. Note that Lampson's LNS stack is much like Lampson's protected call stack, except that it keeps information in addition to the return address.

HARDWARE

Although it really is outside the scope of this paper, it seems appropriate to include a very brief summary of the types of special protection hardware that have been designed. The major purpose of this section is to provide pointers into the literature for those readers who desire more information on hardware. It will become clear that not very much has been done in this area.

Besides the base limit register hardware already described, the first real effort to provide special protection hardware was that of the Burroughs Corporation which implemented a hardware descriptor to delimit segments and provide basic address mapping[Bur61]. This was done in the early sixties for their 5000 series of computers.

Then, in 1967, Evans and LeClerc described address mapping hardware that made use of protection information to limit access[Eva67]. In particular they describe a hierarchy of domains and the hardware to implement them. This scheme allows the efficient sharing of segments.

Later, R. S. Fabry proposed a machine design to support capabilities efficiently[Fab68]. It is essentially a hardware implementation of the schemes proposed by Dennis and Van Horn. Fabry also describes an operating system to run on the machine.

More recently, R. M. Needham has been working on another hardware protection design based on capabilities[Nee72]. He approach relies on indirect references to protected objects so that they can be moved around with freedom.

Finally, Michael D. Shroeder and Jerome H. Saltzer describe hardware for the implementation of protection rings as used in Multics[Shr72a,72b]. In particular the

cross domain call (change of ring) validation for linearly nested domains is handled automatically by their hardware. This design is currently being implemented by Honeywell for Multics. The second paper is Shroeder's thesis. It describes hardware for handling the unconstrained cross domain call. It also describes software to make use of the improved hardware.

PROTECTION IN PROGRAMMING LANGUAGES

Morris

Until quite recently, very little thought had been given to the problems of protection within programming languages. In fact it was barely considered to be a problem. Lately, it has been apparent that something more than the scope rules of ALGOL is desirable. This section will give an indication of what has been done in the area.

As mentioned in the section on operating systems, Lampson has noted the desirability of modularizing programs. These modules can then be protected from each other. His purpose for doing this is primarily to make debugging easier. However, the subject was not considered in much detail until the recent paper by Morris[Mor73]. Morris considers how a programmer might allow his program to communicate with other friendly programs while minimizing the confusion that can arise if one of these programs malfunctions. Note that if the word "process" is substituted for "program" in the above sentence, the paper would be talking about protection as discussed earlier. Protection in programming languages, in Morris's view, is just a more microscopic case of protection in general.

One of the goals of a programming system is that a programmer be able to prove

properties about his program. It would be particularly nice if he could prove that his program will not malfunction. He must be able to do this without regard to other programs in the system. Morris includes in the definition of "program" a single textual region of a larger program (for example a subroutine).

To implement this goal, he proposes a modularization similar to that described by Parnas[Par72]. Essentially he requires that a user of a procedure have no knowledge of the details of the operation of that procedure, only its effect. This provides the basis for a flexible protection mechanism.

He considers the procedure or subroutine as an object capable of being protected. He can protect another object (perhaps a variable) by defining it as local to a procedure. Then, when the object is to be referenced, he passes the procedure instead. By doing this he is in a position to be able to prove things about the object because the only place it is referenced is inside of the procedure.

Suppose that a variable x is defined inside a procedure R. Then any statement inside R can access x, but no statement outside of R can do so. However, any kind of restricted access imaginable can be allowed programs outside of R, by defining within R procedures that operate on x and providing their names to the outside program. This assumes a language with different properties than ALGOL. The language that Morris uses for examples in the paper is GEDANKEN[Rey70].

One of the things that programmers do is provide representations of new kinds of objects for other programs (for example, the stack). They typically provide routines manipulate the new object types (for instance Push and Pop). It is possible that some user could misuse an object (as by treating a stack as a vector just because he happened to know the current implementation). This could, of course, lead to future errors when the implementation is changed.

There are three ways for a program to misuse an object. It can alter the object without using the manipulating procedures (alteration), it can look at the object without the use of the primitives (discovery), and it can present a fake object to the manipulating procedures (impersonation). If either the first or the last of these misuses occurs, the primitive procedures may fail and cause trouble for the rest of the system. Since checking of input parameters is in some cases difficult, some way of avoiding these types of misuse would be useful.

To take care of these problems, Morris defines some universally available functions. The first of these is CREATESEAL When it is called, it returns a pair of unique procedures SEALj and UNSEALj, where i changes for every call of CREATESEAL. SEALj(x) yields a new object x' which can only be accessed after UNSEALj(x') is executed. Then x' can be passed to a program with complete safety if the program has no access to the proper UNSEAL. This mechanism provides two types of protection, authentication and access limitation.

In some cases he is only concerned with authentication. In this case a simpler procedure can be used. It amounts to having the UNSEAL operation publicly available, while keeping SEAL private. Morris defines a procedure CREATETRADEMARK which returns the pair MARK|,i. Again i changes every time the procedure is called. Calling MARKj(x) produces an object x' that is exactly like x except that it bears the trademark i.

By defining a procedure that returns a list of the trademarks borne by x', a routine can authenticate any object presented to it. Note that this is essentially the capability model of protection. An object must possess the capability of being passed to a procedure that desires protection from impersonation.

A model of protection by access keys is also possible. This is done by making the

UNSEAL operation private and the SEAL operation public. CREATESEAL then returns the pair $UNSEAL_i$,i, and SEAL is called as SEAL(i,x) to put seal i on object x. An object so sealed can only be accessed by that group of programs possessing the appropriate UNSEAL operation.

Morris points out that implementation of all or any of this is expensive. However, he presents no way of overcoming this problem.

Wulf and Shaw

In a recent paper, Wulf and Shaw indirectly point out the need for protection in programming languages[Wul73b]. Their major premise is that the use of the global variable is, in some cases, a bad idea. Since the writing of this paper they have done further research in this area. One of the solutions currently under investigation is the application of Hydra style protection to variables and procedures in a programming language. They point out that most protection can be checked at compile time. Thus, run time need not be sacrificed[Sha73].

OTHER WORK ON PROTECTION

The previous sections of this paper have attempted to give a reasonably thorough overview of protection. Obviously it is impossible to cover every single paper on the subject in a work of reasonable length. Also there is a degree of duplication in the literature. Some of the available papers that have not been covered are mentioned here so that the interested reader may pursue the subject further.

The protection system for the CAL-TSS system described by Lampson in one of his papers is further discussed in a paper by J. Gray et. al.[Gra--].

An overview of protection, by Graham and Denning, was recently published at a

Spring Joint Computer Conference[Gra72]. It covers most of the ideas discussed in this paper, but adds a few additional features.

Wilkes has updated his book referenced earlier into a new edition[Wil72]. In it he discusses the development of the software and hardware proposed by Fabry.

Finally, A. Jones, in a recent Phd. Thesis, considers formal models of protection in some detail[Jon73].

BIBLIOGRAPHY

[Bur61] Burroughs Corporation, The Descriptor - A Definition of the B5000 Information Processing System, Detroit, Mich., Feb. 1961

[Den66] Dennis, J. B., and Van Horn, E. C., Programming Semantics for Multiprogrammed Computations, CACM 9, 3 (Mar. 1966), 143-155

[Eva67] Evans, D. C., and LeClerc, J. Y., Address Mapping and the Control of Access in an Interactive Computer, Proc. AFIPS 1967 SJCC, Vol. 30, AFIPS Press, Montvale, N.J., 23-30

[Fab68] Fabry, R. S., Preliminary Description of a Supervisor for a Computer Organized Around Capabilities, Quarterly progress report 18, Institute of Computer Research, University of Chicago, 1968, 1-97

[Gra68] Graham, R. M., Protection in an Information Processing Utility, CACM 11, 5 (May 1968), 365-369

[Gra--] Gray, J., et. al., The Control Structure of an Operating System, to be published

[Gra72] Graham, G. S., and Denning, P. J., Protection - Principles and Practice, Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., 417-429

[Han70] Hansen, P. B., The Nucleus of a Multiprogramming System, CACM 13, 4 (Apr. 1970), 238-250

[Jon73] Jones, A. K., Protection in Programmed Systems, Phd. Thesis, Carnegie-Mellon University, 1973

[Lam69a] Lampson, B. W., On Reliable and Extendable Operating Systems, Techniques in Software Engineering, NATO Science Committee Workshop Material, Vol. II, Sep. 1969

[Lam69b] Lampson, B. W., Dynamic Protection Structures, Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., 27-38

[Lam71] Lampson, B. W., Protection, Proc. Fifth Annual Princeton Conference on Information Sciences and Systems, Dept. of Electrical Engineering, Princeton University, Princeton, N.J., Mar. 1971, 437-443

[Mor73] Morris, J. H. Jr., Protection in Programming Languages, CACM 16, 1 (Jan. 1973) 15-21

[Nee72] Needham, R. M., Protection Systems and Protection Implementations, Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., 571-578

[Par72] Parnas, D. L., On the Criteria To Be Used in Decomposing Systems Into Modules, CACM 15, 12 (Dec. 1972), 1053-1058

[Rey70] Reynolds, J. C., GEDANKEN: a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept, CACM 13, 5 (May 1970), 308-319

[Sha73] Shaw, M., Internal Correspondance, Dept. of Computer Science, Carnegie-Mellon University

[Shr72a] Shroeder, M. D., and Saltzer, J. H., A Hardware Architecture for Implementing Protection Rings, CACM 15, 3 (Mar. 1972) 157-170

[Shr72b] Shroeder, M. D., Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Phd. Thesis, Mac Tr-104, Mass. Institute of Technology, 1972

[Wil68] Wilkes, M. V., Timesharing Computer Systems, first edition, American Elsevier, 1968

[Wil72] Wilkes, M. V., Timesharing Computer Systems, second edition, American Elsevier, 1972

[Wul73a] Wulf, W. A. et. al., HYDRA: The Kernel of a Multiprocessor Operating System, Carnegie-Mellon University, 1973

[Wul73b] Wulf, W. A., and Shaw, M., Global Variable Considered Harmful, SIGPLAN Notices 8, 2 (Feb. 1973), 28-34