A FRAMEWORK FOR A THEORY OF PROTECTION

by: C. V. Srinivasan

DCS Technical Report #16

Department of Computer Science
Hill Center for Mathematical Sciences
Rutgers, The state University of New Jersey
New Brunswick, New Jersey 08903

May, 1972

A Framework For a Theory of

Protection

C. V. Srinivasan

ABSTRACT

What may one expect from a theory of protection, and what framework does one have to build a satisfactory theory? These are the principal issues discussed in this paper. A generalized formal model of protection systems is proposed as a framework for a theory of protection. The paper discusses the relevance of the proposed model to the problems encountered in the design of protection systems. Also, the manner in which the model extends other existing models is pointed out.

In the context of the model, some of the consistency, implementation and reliability problems that require further study are pointed out.

1. Introduction

There is now a growing body of empirical knowledge on the design of "protection" mechanisms in an operating system.[1] Work in this area has so far concentrated on clarifying functional capabilities (kinds of protection and conditions for their regulation) necessary for protection systems in various operating systems environments.[2] The technology of design in this area is still in its early innovative stage, where solutions are often ad hoc, and practical considerations of time and efficiency dominate considerations of flexibility, robustness and generality. There is now a need for the development of a unified theoretical framework for protection systems. Such a framework should (i) provide formal means of specifying protection systems; (ii) help in the identification of the fundamental technical and legal considerations that affect the design of such systems; (iii) provide means for describing implementations, identifying the required levels of security and reliability of components of such implementations, and verifying proposed implementations. It should also provide guidelines for the general organization of such systems, and means for testing their robustness (performance in unreliable or malicious environments). Finally, the theoretical framework should also help in clarifying the protection properties of existing systems. The general goal of our work in this area is to develop such a theoretical framework and to test it in the context of specific designs. This paper introduces a general formal model of protection systems, which provides a convenient framework for such a theory. In the context of the proposed model, some of the consistency, implementation and reliability problems are discussed.

2. Formal Models of Protection Systems

The work of Lampson [Lampson 1971] represents a first attempt to present the functional capabilities of several of the existing systems within a unified context. It is probably appropriate to take Lampson's work as a point of departure for our discussions here.

---

1. See Daley, 1965; Graham, 1968, 1971; Lampson, 1969 a and b, Wilkes, 1968; Schroeder, 1972.
2. Daley, 1968; Schroeder, 1972; Evans, 1967; Lampson, 1971; Wilkes, 1968; Graham, 1971.

Lampson's model consists of the following:
A system is a set of objects, say $B = \{b_1, b_2, \ldots, b_n\}$. A domain (subjects) D is a subset of B, $D = \{d_1, d_2, \ldots d_m\}$. Each subject may have certain kinds of access privileges to objects in B. The specific kinds of access privileges between pairs (d, b) are given by an access matrix, M, whose rows are the domains (subjects) and columns are the objects. The entries in $M[d,b]$, specify the kinds of access privileges that d may have for b; d is said to have an α access privilege to b if α is a member of $M[d,b]$. Let A denote the set of all access priv leges α. The access priveleges in $M[d,b]$ are the only legal ones for d. All other access attempts should be prevented. This regulation is enforced by a monitor, which checks, for each request (d,α,b) whether α is in $M[d,b]$, and allows α access of b to d if α is in $M[d,b]$, and not otherwise. A set of rules R govern the establishment, deletion and modification of the elements of M. Different existing systems seem to roughly correspond to different ways of representing M, and to different choices of the sets R and A.

The causes of complications that arise in implementations do not appear in this model. The complications arise due to a variety of factors. Some of these, discussed below, suggest some of the structural properties that a comprehensive formal model should have.

(a) Typing of objects

In the above model all objects are treated alike by the monitor; whether they be diamonds or bits of glass, they get equal security protection. There is no hierarchy of security requirements that might be specified. This usually is not the case in practice. Not only is it true that monitors protecting different types of objects are different, also the extent of protection given to various types may vary. The protection given to a software of an operating system is certainly different from that given to a private user program. A general formal model should provide for typing objects into classes according to their protection requirements, security risks, frequency of use, etc. It should also admit of rules governing access between pairs of objects, that depend only on the object types involved.

(b) Multiple regulatory mechanisms

The existence of different object types leads naturally to multiple regulatory mechanisms (monitors) in the system. The reasons are economy and operational feasibility. Thus the protection mechanisms for memory protection, are different from those for file protection. So also, the protection mechanisms

for dynamic sharing of active files are different from those for fixed "execute only" files consisting of pure procedures. The kinds of protection monitors used depend in general upon the addressing structure of objects, their frequency of use, access time, and the kinds of conflicts that should be avoided. A general model should provide facilities for indicating these attributes.

(c) Non-uniformity of access checks

In the above model all access checks are of the type "is α in M[d,b]?". In practice this is not the case. The reasons for variation are twofold: One is that the mechanisms of implementation often transform the above question to more suitable equivalent forms. The other is that the access conditions themselves are differently formulated. Thus, in certain cases, checking for compatibility of object types may be the only necessary condition for allowing an access. In others, there may be elaborate probing between mutually suspicious objects before they agree to share something between them, as may happen when a system attempts to verify the integrity of a user. A general model should provide facilities to specify access conditions in terms of rules, more general than checking for preset attributes in a table.

(d) More structure in the object set B

In addition to type classification the set B may have other structures imposed on it. Two typical ones are the "ownership" and "control" relation structures. Also, the type of an object may change dynamically in a system. (A program may lose its security classification as a result of illegal access attempts initiated by it). A general formal model should have facilities for specifying such structures and their consequences on protection conditions.

(e) More structure in access privileges

The set of access privileges, A, itself has a structure. Some individual privileges imply others, and some combination of privileges imply other privileges. Also, the set of all subsets of access privileges form a lattice (the "ring" is a special case of this) which might be used to characterize the context of use of a given object. It should be possible to specify these properties in a general model.

(f) Laws of secure operation

In a complex system the protection state is a dynamically changing entity. To assure secure performance there should be general laws (constitution for the system) that govern admissible changes. These laws should be independent of individual objects. The design of an entire system should be consistent with its constitution.

Many details of implementation depend on the specific nature of the structures (a) through (f). The abstract model used to specify a protection system should be useful in deciding on implementation choices. It should also act as the basis for checking the validity of an implementation. It is, of course, necessary to develop schemes for describing implementations themselves at different levels of abstraction. The design of the implementation is crucially dependent on the structure of the operating system in which it is to function. If the operating system is itself unreliable then special precautions should be taken to limit possible damage that a breakdown may cause. In principle, if one knew in advance the nature of errors and their propogation properties through a system then a variety of error correction and error limiting strategies may be employed to minimize the effects of malfunctions. Critically important parts may be specially protected. A systematic approach to the specification and design is essential to understand these problems realistically.

In the following subsections we shall introduce a possible generalized model and identify in the context of the model some of the consistency, implementation and reliability problems.

3. Towards a generalized formal model: Specification of security and protection requirements

For the purpose of our discussion here we shall postulate three kinds of protection specifications:

1. Rules of Protection,
2. Secondary Access Rights, and
3. Exceptions.

3.1 Rules of Protection

These may be thought of as specifying the constitution of a system.

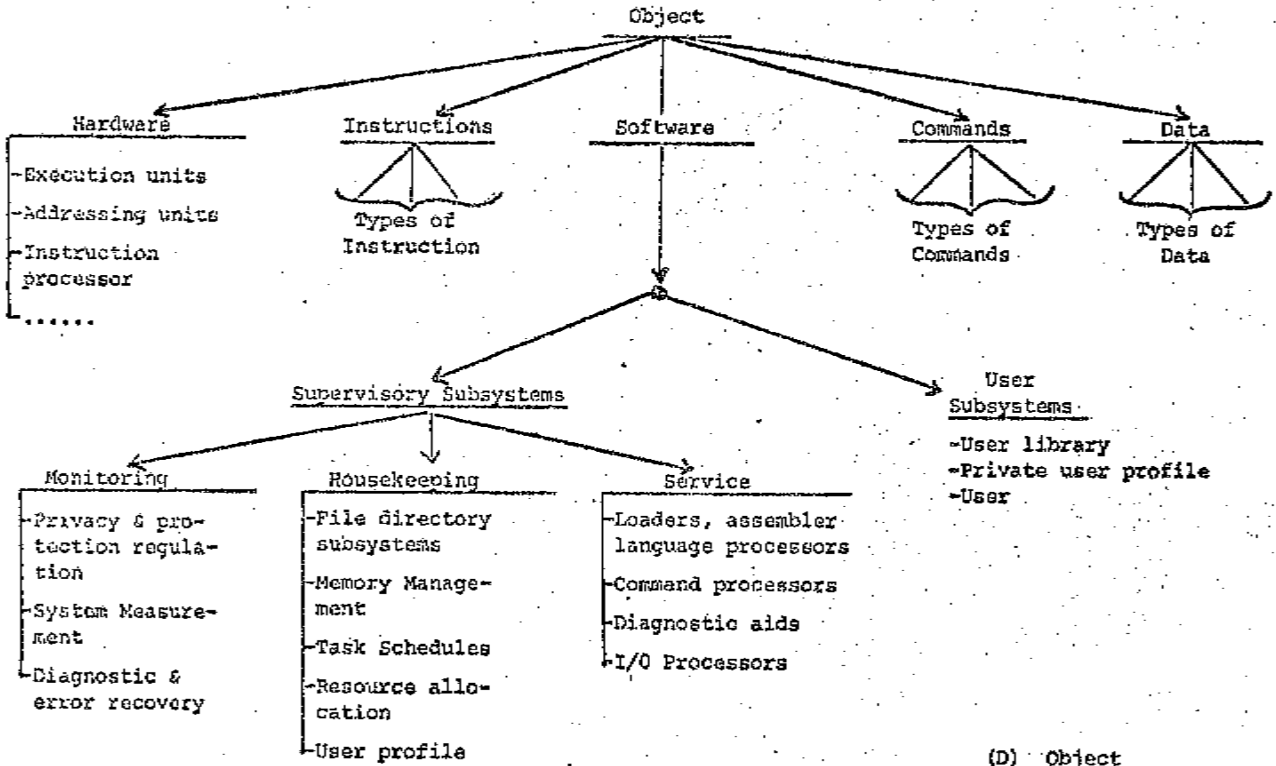
The rules are to define five kinds of access privileges:

(i) Admissible Accesses: The access rights that a given type of object may have, by virtue of its type, to other types of objects. An example of the kinds of considerations that enter into typing of objects is shown in Figure 1. For example a task scheduler should have mandatory privilege to read and update a "ready" list; a debugging routine should have the privilege to examine the core image of a user program that is being debugged; a user program should have the right to call on any system service routine; etc.

(ii) Forbidden Accesses: Access rights that a type of object could not have by virtue of its type. For example a user program cannot possibly have access to a task scheduler, page tables, or a protection monitor; or an uncertified program cannot be shared as a system resource.

(iii) Rules of Transfers: These specify the conditions under which one object may transfer or borrow or withdraw some of its access rights from others. These conditions will usually depend on the object types, certain relationships existing between groups of objects, and certain contexts of operation. Typical of the relationships between objects are the following: Ownership, Control, Subordinate (a subordinate object will have to obtain permission from its superior to acquire an access), Client (a client gets access via an agent), etc. Thus, the owner of an object may have the right to share the object owned with others; one who controls d may withdraw some of d's access rights; etc. In general, the "Client" and "Subordinate" relationships will depend on the context of operation. The context of use (or services) of an object b with respect to d is the set of rules, Secondary Access Rights (see 3.2 below) and Exceptions (see 3.3 below) which enable d to enjoy the use (or services) of the object b.

Transfers may be subject to a variety of limiting conditions, in addition to those imposed by context. Thus, an authority to pass on an access right may be subject to an upper bound on the number of objects involved, or passing on an authority to an object might require the initial owner to relinquish it; or it might be subject to confirmation of "good behavior" of the recipient. Special conditions of this type may be introduced as part of the Exceptions (Sec. 3.3).



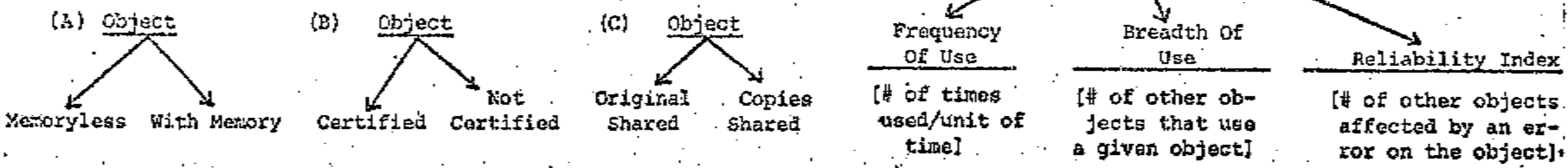


Figure 1 - Considerations that affect typing of an object.

(iv) Rules of Creation and Destruction: These pertain to which type of object may create or destroy which other types of object. The creation rules will establish the ownership and control relationships among objects and the access privileges of objects created. The destruction rules will depend on the existence of certain relationships. A user cannot usually create objects having access rights not possessed by the user himself. Usually only an owner of b may have the right to destroy b.

(v) Rules of Exceptions: These specify the conditions (in terms of object types, contexts of use, object relationships, and other attributes) under which exceptions to (i) through (iv) may occur. Thus, an object attempting illegal accesses may be forced to relinquish some or all of its rights; a subject may be given the right to read a file, normally inaccessible to it, under the supervision of a third object; two objects which require mutual isolation may be allowed to communicate with each other through an agent; etc.

The secondary access rights and exceptions in the sections below specify the consequences of the rules as they might appear in a typical system governed by them.

3.2 Secondary Access Rights

These are the rights acquired or lost by an object as a result of application of "rules of transfer" (Sec. 3.1, part (iii)). These rights are classified into two classes: Those that are independent of context and those that are context dependent.

3.3 Exceptions

These identify the objects, and access privileges they enjoy as a result of the rules of exception in Sec. 3.1, part (v).

In the above model the rules are meant to be relatively permanent and commonly used ones that govern communication between objects in a system. They presuppose a careful classification of objects according to types. Some of the considerations that enter into typing of objects are shown in Figure 1. Selection of appropriate rules is a prerequisite to assure secure and reliable

operation of a system. The model also provides a framework for the kinds of consistency checks that should be implemented in a system. The system should provide means to check that the secondary access rights and exceptions do not violate the rules. Also, access privileges granted during the operation of the system should be consistent with the model. The necessary consistency checks should be simple and efficient; they should not unduly impose "red-tape" on the users and they should contribute to the enhancement of system reliability. Part of our research will be devoted to the formulation of the rules, and identification of their effect on design (in terms of reliability, security and implementation of consistency checks). In the next section we shall review some of the considerations that affect implementations.

4. Models of Implementation:

The abstract model discussed in the previous section suggests the implementation schema shown in Figure 2. Boxes 1 through 4 in Figure 2 are data items and boxes A through F are processors. The significant differences between this model of implementation and the implementations in many of the currently existing systems are the following: There is provision for consistency checking (Box A in Figure 2), error processing (Box F), and typing of objects (Box E).

The objects grouped together as units in Figure 2 will not, of course, appear similarly grouped together in an implementation. They are usually widely dispersed among different parts of the system. There are several reasons for this. A primary reason is the variations in the enforcing mechanisms used. There are basically two kinds of these: The first kind (denoted by ENFORCER 1) checks for access privilege before a requested object is accessed. This kind of enforcer is usually part of the addressing mechanism (address decoders, or directory search routines and directory entries). The second kind (denoted by ENFORCER 2) checks for distribution privilege after a requested object has been accessed. In this case the object itself carries with it some of the information necessary to check the requisite privilege. This usually implies a software checking routine. This kind of checking is essential to regulate accesses through intermediaries (agent or certified supervisor).

The existence of a memory hierarchy and a variety of addressing mechanisms makes it desirable to transform the relevent rules of the abstract model to

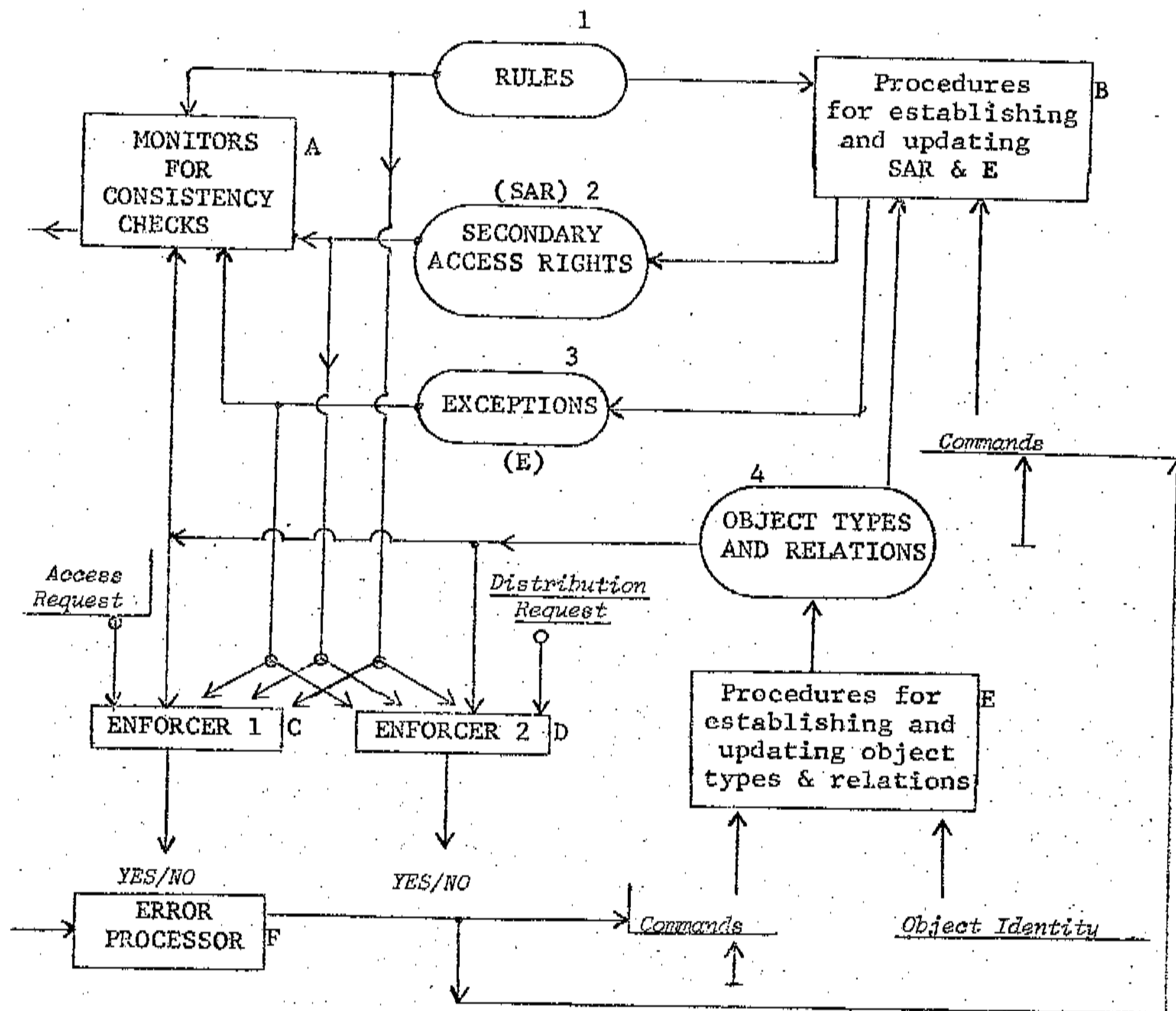


Figure 2: Basic Schema of Implementation

representations appropriate to the respective enforcers.

The second reason for dispersion of units in Figure 2 arises as a result of requirements of efficiency. In general, it is desirable to have an implementation in which it is necessary to have full access checks only when a subject d first initiates an access request for an object b. Access checks for subsequent accesses within the same context should be minimized as far as possible. This would usually require an abbreviated representation of verified access conditions (as core memory bounds within which object b might exist, or as flags in the associative memories used for address translation) for easy checking while addressing the core [Schroeder, 1972].

The third reason pertains to the degree of protection and security necessary for an object. Objects used often or widely should not require repeated protection checks. (Freely accessible objects of certain types may be for example stored in specially allotted areas of memory.) So also, objects with high reliability index (number of other objects affected by an error in a given object) should have protection mechanisms more secure than those of others (this again may be implemented by special memory allocation for such objects).

The fourth reason for the dispersal of elements in a given implementation pertains to the set of available communication paths between various subsystems in a given operating system. Many of the protection conditions for regulating communication between subsystems of an operating system may be embedded as part of the programs of the subsystems themselves.

To proceed with a systematic design it is first necessary to carefully specify the implementation requirements imposed by the considerations discussed above, and the way they relate to memory management within the system.

It is necessary to explore ways of specifying these, and develop tools to analyze a given specification to identify implementation requirements. Of particular interest here are tools to classify objects according to their frequency of use, reliability index, breadth of use, and addressing mechanisms. The degree of protection necessary for and reliability expected of an object, usually depend on the above parameters.

We are here advocating the approach that the specification of an operating system should begin with its protection and reliability model.

4.1 Implementation of Secure Systems

Our discussion so far has concentrated on the specification of a protection system, its regulatory mechanisms, and implementation requirements. The transition from this to a secure implementation requires much study. There are two approaches possible: One is to systematize implementation conventions with respect to the language used to write subsystems, and with respect to an overall schema for structuring a system. This approach is best exemplified by the THE system [Dijkstra 1968], and the SUE system of the University of Toronto. Approaches here are still empirical. We are still far from developing any closely reasoned criteria. The other approach depends on the possibility of developing automatic program verification programs. The state of the art in this area is far from being ready for use in complex systems like operating systems. Both approaches require further study. Of particular interest here would be tools for program documentation, for enforcing system conventions, and for verifying assertions about well defined program segments

4.2 Reliability and Diagnosis

Formal models at different levels of abstraction of protection systems and implementations may be used to identify failure modes and to diagnose system malfunctions. We have discussed in earlier sections some of the considerations pertaining to the initial specification of protection systems and their implementations. It is necessary also to develop description (and representation) schemes for implementations at different levels of abstraction. Such descriptions may be used to analyze failure modes, and develop strategies for error control. There are a variety of means available to control errors in a complex system: redundant encodings of data; programmed checks; flagging data types to detect addressing errors; or execution of periodic diagnostic checks. The problem usually is one of finding what techniques to apply where, and how specifically. This requires careful system planning and system analysis prior to design. Formal models, whose beginnings we have outlined here will provide the necessary context for the development of systematic techniques for error control.

REFERENCES

Daley, R. C., and Neumann, P. G., 1965, "A General Purpose File System for Secondary Storage", Proc. AFIPS 28, (FJCC), pp. 213-229.

Daley, R. C., and Dennis, J. B., 1968, "Virtual Memory, Processes and Sharing in MULTICS", CACM 11, May, pp. 306-312.

Dijkstra, E. W., 1968, "The Structure of the 'THE'-Multiprogramming System", CACM 11, May, pp. 341-346.

Evans, D. C., and LeClerc, J. Y., 1967, "Address Mapping and the Control of Access in an Interactive Computer", Proc. AFIPS 30, (SJCC), pp. 23-30.

Graham, R. M., 1968, "Protection in an Information Processing Utility", CACM 11, May, pp.85-104.

Graham, R. M., and Denning, P. J., 1971, "Protection: Principles and Practice", Dept. of Electrical Engineering Technical Report #101, Princeton University, November.

Lampson, B. W., 1969a, "On Reliable and Extendable Operating Systems", Techniques in Software Engineering, Nato Science Committee, Working Material, Vol. 2, September.

Lampson, B. W., 1969b, "Dynamic Protection Structures", Proc. AFIPS 35, (FJCC) pp. 27-38.

Lampson, B. W., 1971, "Protection", Fifth Annual Princeton Conference on Information Sciences and Systems, Princeton University, March, pp. 437-443.

Schroeder, M. D., and Saltzer, J. H., 1972, "A Hardware Architecture for Implementing Protection Rings", CACM 15, March, pp. 157-170.

Wilkes, M. V., 1968, Time-Sharing Computer Systems, American Elsevier, New York.