# WM Protection: The Base Mechanism

Wm. A. Wulf & Anita K. Jones

Computer Science Report No. TR-91-03
March 4, 1991

Department of Computer Science
University of Virginia
Charlottesville, Va. 22903

# WM Protection: The Base Mechanism

Wm. A. Wulf & Anita K. Jones

Computer Science Department
University of Virginia
Charlottesville, Va.

March 1991

## 1. Introduction

The security of our information processing systems is appallingly poor. That *should* be unacceptable. It is accepted, however -- but only because we do not know how to do better cost-effectively.

There are many reasons for the present lack of security, and many of them are not technical. However, even within the technical domain, there is a great deal that needs to be learned before we can provide adequate, efficient, and convenient security for today's applications, much less those of tomorrow. This report addresses one of the technical problems in secure systems; a broader view of the context for it may be found in [Pfl89].

Specifically, we describe a hardware protection mechanism, derived from the notion of a capability [Den66], that we believe is significantly better than that currently used in most computers. The mechanism permits decomposition of systems into units whose size and complexity is amenable to rigorous analysis -- and specifically to contemporary formal verification technology[1]. The mechanism is also simple enough to be efficient. As a pleasant side benefit, it seems to allow (and encourage?) the implementation of a richer class of security policies than one currently finds.

The present report is not a complete description of the mechanism; it is our "stake in the ground" -- a description of the basic idea, intuitive arguments about its functionality and implementability, and some remarks on what we don't know. Our purpose in writing this is to inform and involve our colleagues and students in the research effort.

## 2. Description of the Mechanism

Most current computers use a variant of a single protection scheme; it has two parts: (1) access control to memory implemented as part of the virtual memory system (typically read/write/execute rights associated with a page), and (2) a set of hierarchically ordered "modes" such that certain machine instructions can only be executed in more privileged modes.

---

[1] See [Che81] for a survey of verification of security properties, and [Den77] for a more detailed presentation of verification under a particular model.

In principle this scheme is adequate for any implementable security policy. It also seems intuitively plausible that the more sophisticated variants of the scheme, such as that used in MULTICS [Sch72, Sal74], reduce the amount of the system that needs to be implemented in the *kernel* , i.e. that portion of the system executing in the least protected mode. Reducing the amount of code at this level is very important if one intends to verify the system because the state of verification technology severely limits the number of lines of code that it is feasible to verify today.

In practice, unfortunately, the size of the kernel of real systems  has been enormous, and routine verification has been unrealistic. Mechanisms such as capabilities (c.f. [Wul80]) provide a better base for implementing security policies -- but unfortunately so far these have proven to be too inefficient to be useful in practice (c.f. [Col88]).

The mechanism we propose eliminates the notion of *modes*, and relies entirely on the virtual memory system to enforce protection. There are no "privileged instructions"; instead there are a handful of hardware-defined segment *types*. Certain instructions operate only on certain types, and to execute those instructions requires type-specific access rights to the specific segment in question. We define the base mechanism as follows:

1. The domain of discourse is a finite collection of *segments*, **S**.

2. There is a finite set of *types*, $\mathbf{T} = \{t_1, t_2, \ldots t_m\}$; each of the segments in **S** is associated with one type in **T**.

3. Associated with each type, $t_j$, is a finite set of *access rights*, $\mathbf{R}_j = \{r_{j,1}, \ldots, r_{j,k_j}\}$ applicable to instances of type $t_j$.

4. A *capability* is an unforgeable pair, $<\sigma,\rho>$, consisting of an reference to a segment $\sigma$, and a set of access rights, $\rho$, for that segment;  $\rho \subseteq R_{type(\sigma)}$. A capability with no rights is called a *null* capability; all nulls are equivalent.

5. There is a distinguished segment type called *domain*; a domain contains capabilities, and has three access rights -- *take*, *grant*, and *delete*; a domain may be considered be a vector of capabilities -- positions within the vector are called *slots*. A slot

containing a null capability is called *empty*. Only domains contain capabilities, and domains contain only capabilities.

6. There is another distinguished type called *task*, that can execute *operations*. A task is always associated with one domain which defines the objects to which the task has access. Note that not all domains need to be associated with a task.

7. An *operation*, $op(o_1,...,o_q)$, consists of an *operator, op,* and a set of associated *operand specifiers*: $o_1,...,o_q$. The operand specifiers are interpreted relative to the domain associated with the task executing the operation; that is, they name (elements of) segments referenced by capabilities in the domain of the task.

   For each operation, $op(o_1,...,o_q)$, the mechanism defines the type and rights required of each operand, $o_k$.

The basic rule of the mechanism is:

> *An operation is valid iff each of its operands, interpreted relative to the domain associated with the current task, has the type and rights required for that operation.*[2].

The trivial example of the mechanism is a type "memory" with read and write rights; the operations are load and store. "load $\alpha$" requires that $\alpha$ name an element of a segment of type memory with read writes, for example. Things only get interesting when there are more than one type -- such as type "domain". One cannot perform load from a domain; it is not the right type. Nor can one perform domain operations on memory segments.

This mechanism is at least as powerful as a mode-based one. To see this, consider a mode-based system with a hierarchy of N modes. The instruction set, I, is partitioned into disjoint subsets:

$$I = \{I_1, I_2, ..., I_N\}$$

such that in mode i, $1 \leq i < N$, instructions in subsets $I_1$ - $I_i$ are legal, but those in $I_{i+1}$ - $I_N$ are not.

[2] In addition, of course, the instruction must come from a segment of the proper type and with the proper rights.

One can construct a system using the proposed mechanism which exactly mimics this behavior. Partition the instruction set in exactly the same way, and associate at least one distinct object type with each member of the partition. Now construct a sequence of domains; the first such, $D_1$,will contain unrestricted capabilities for all (and only) objects of the type associated with $I_1$. Subsequent domains, $D_i$, will contain all the capabilities of their predecessors plus those of the type associated with $I_i$. Obviously, a task associated with $D_i$ can execute instructions in $I_1 \cup I_2 \cup ... \cup I_i$, but none in $I_k$ $(k>i)$.

Of course, considerably finer discriminations can be expressed in a natural way with the mechanism; it was not our intent to suggest that this construction is a preferred one. Quite the opposite is true.

We have tried to be careful in the above description; we intend that it be sufficiently precise to form a semantic foundation on which verification of various system properties can be based. It should be understood, however, that our intent is a highly efficient, practical implementation. To illustrate that this is so, in the next section we'll sketch a possible implementation -- showing that it has the same cost as a conventional virtual memory system. Later we'll return to a formal model of the mechanism.

## 3. A Sketch of a Possible Implementation

Consider a RISC-like, load-store architecture with a conventional 32-bit address space. In addition to the two required types of the model (task and domain), we'll add type *memory* -- the familiar notion of primary storage. Let the rights associated with these three types be:

| type | rights |
|---|---|
| memory | read (**r**), write(**w**), execute(**x**) |
| task | swap(**s**) |
| domain | take(**t**), grant(**g**), delete(**d**) |

Since this is a load/store architecture, most instructions do not specify addresses (have *operand specifiers*); for those that do we require the following:

| instruction class | required type and rights |
|---|---|
| load $\alpha$ | type($\alpha$) = memory, $\mathbf{r} \in$ rights($\alpha$) |

| | |
|---|---|
| store $\alpha$ | type($\alpha$) = memory, **w** $\in$ rights($\alpha$) |
| jump/call $\alpha$ | type($\alpha$) = memory, **x** $\in$ rights($\alpha$) |
| cntxswap $\alpha$ | type($\alpha$) = task, **s** $\in$ rights($\alpha$) |
| mvcapa $\alpha,\beta,\delta$ | type($\alpha$) = domain, **t** $\in$ rights($\alpha$), |
| | type($\beta$) = domain, **g** $\in$ rights($\beta$), |
| | empty($\beta$) or **d** $\in$ rights($\beta$), |
| | $\delta$ is a literal |

There are two unusual instructions are in this list, 'cntxswap' and 'mvcapa'.

'cntxswap' saves the current state of the processor in the "current" task object, restores the state saved in the named task object, $\alpha$, and begins to execute that task.

'mvcapa' performs an indivisible movement of a capability from a slot, $\alpha$, in one domain to a slot, $\beta$, in another. The rights of the capability from slot $\alpha$ are anded with $\delta$, potentially restricting the rights stored into $\beta$. Note that the target domain slot must be empty, or the caller must have delete rights to $\beta$.

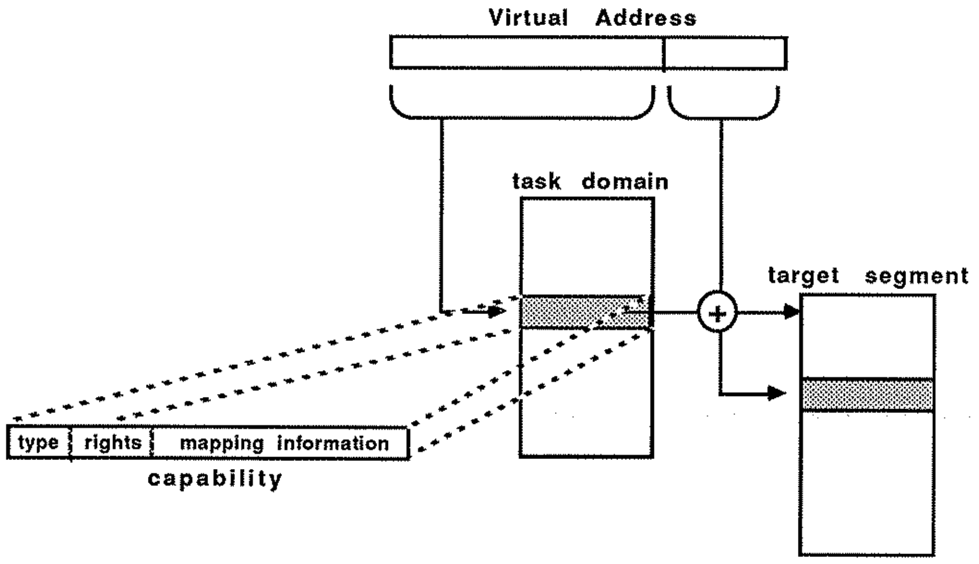
We'll return to discussing these objects and operations a bit later; first let's consider a conceptual implementation of the mechanism.

When associated with a task, a domain defines the segments that can be named directly by that task; that is, it plays something like the role of a segment/page table in a traditional architecture, and the information in it defines the mapping from virtual/typed segments to physical resources (physical memory, for example). Thus, a plausible implementation of domains is very like that of segment/page tables, and consists of a vector of capabilities for segments. These capabilities contain the type, rights, mapping, and such other information as may be required.

Virtual to physical address translation also proceeds in a conventional fashion. As sketched in the figure below[3], the high-order bits of a virtual address are used to index into the task's domain and select a capability. The low order bits of the virtual address are then

[3] We have intentionally avoided the question of 1- vs. 2-level translation tables, as well as field sizes and a number of other crucial issues for the final implementation; they are not relevant to the logical design discussed here.

combined with mapping information contained in the capability to identify the element of the target segment. In parallel, the type and rights contained in the capability are checked to verify that the operation is valid.



In a conventional architecture the rights test, eg 'r ∈ rights(α)' would be performed for memory load/store instructions. We have merely added the type test , e.g. 'type(α) = memory'. It should be clear that this can be performed in parallel with the rights check. Hence, it seems plausible that there is no performance penalty for any of these checks relative to conventional virtual memory checks.

The similarity of the above mapping scheme to conventional virtual memory schemes might mislead one to believe that nothing interesting is going on; not so! The next section discusses some of the ways in which the mechanism can be used to express, naturally and efficiently, a variety of system structures that would be awkward in a mode-based system.

## 4. Example System Structures Using the Mechanism

As noted above, a variety of system structures can be naturally modeled with the mechanism. One major reason is the existence of the 'mvcapa' instruction. There are many implications to the ability to move capabilities from one domain to another. For example:

- it enables one domain to act as a resource manager for others. All resources will be modeled as segments in this mechanism, so the manager (de)allocates them to and among a set of client tasks by moving the representing capabilities. See the discussion of scheduling, below, for an example of this.

- it allows fine grain control of resource policies. A task can act as a manager for only those resources it can name -- simultaneously there may be other tasks managing other instances of the same type of resource, but with a different policy more appropriate to their use. The granularity can be adjusted as needed by specific applications.

- by granting a task access to its *own* domain, it allows more flexible use of the task's own address space. Because we assume a fixed size virtual address, the number of segments accessible at any one time is fixed. However, because one or more of the segments can be domains, each of which can name more domains, the total set of (indirectly) addressable segments is unbounded. (To access one of the indirectly named segments requires moving its capability into the task's root domain, of course, but this can be done with 'mvcapa' instruction rather than an operating system call.)

An unusual operation introduced earlier was 'cntxswap', whose operand must be a segment of type *task*, and which required *swap* rights to that task. Consider a domain that contains tasks with swap rights. Such a domain can act safely as a scheduler for those tasks it can name[4]. While it can select and swap to one of its tasks

- it cannot modify the state information of a task since load and store are defined only on type memory , and

- it cannot 'forge' a task by swapping to a memory segment since 'cntxswap' is not defined for type memory.

For the moment assume that this domain is the only one that has access to these tasks. Then to demonstrate that there is no denial of service, for example, we need only verify the code

[4] To be a full scheduler, obviously, the task must be able to resume control when the selected process blocks and/or after a specified interval; we ignore these complications here.

for this one task. We have managed to create a protected domain that is concerned only with scheduling and whose correctness cannot be affected by other tasks; hence we are able to verify the property of concern locally. Typically scheduling algorithms are not very large, so this should be well within the state of the art.

The "for the moment assume" that begins the previous paragraph is a big assumption. The creation and distribution of segments, insertion of them into domains, and setting of access rights must be done with great care -- especially since we also expect this to be done by tasks whose only special status is derived from the segments in their domain. This leads us back to the formal model of the mechanism. It is introduced in the next section.

## 5. The Take-Grant Model

In order to enhance our ability to verify properties of systems built on the mechanism, it is useful to have a formal model of it. For reasons that will quickly become clear, we have chosen the "Take Grant" model -- a model with a long history [Jon78, Lip77, Sny81, Bis81, Bis88]. It is especially interesting to us because

- it does not presume a "monotone" security model [Rab88], and hence is suitable for more complex security policies than the military policy -- such as those needed by industry.

- it is known that *safety* can be determined in linear time[5]; determining safety is NP-complete in most models [Den82, Sny81].

This last point speaks directly to the "for the moment we assume" caveat that ended the last section. To satisfy that assumption, we must be able to determine that no other task can acquire access to the tasks controlled by the scheduler. The take-grant model allows us to examine the state of the system at some time, $T_0$, and efficiently determine whether at any future time another task could acquire access to these segments!

[5] Safety has a formal definition, but intuitively corresponds to the notion that specific subjects cannot acquire specific rights to specific objects through *any* sequence of operations.

We won't give a complete definition of the take-grant model here (see [Sny81], for example) but rather give an informal definition and build the correspondence to the protection mechanism defined in section 2. The model is defined in terms of a graph with labeled directed arcs. Nodes of the graph represent *objects*. Some objects are also called *subjects* and represent tasks with their associated domain. Arcs are references from one object to another, and the arc labels are (sets of) access rights. Operations are characterized as transformations on the graph. Four such operations are defined:

*take*: if **x** is a subject with **t** rights to **y**, and **y** has $\beta$ rights to **z**, then *take* defines a new graph in which there is an additional edge from **x** to **z** labeled $\alpha$, $\alpha \subseteq \beta$.

$$\begin{tikzcd} x \arrow[r, "t"] & y \arrow[r, "\beta"] & z \arrow[r, phantom, "\vdash"] & x \arrow[r, "t"] \arrow[rr, bend right, "\alpha"'] & y \arrow[r, "\beta"] & z \end{tikzcd}$$

*grant*: if **x** is a subject with **g** rights to **y** and $\beta$ rights to **z**, then *grant* defines a new graph in which there is an additional edge from **y** to **z** labeled $\alpha$, $\alpha \subseteq \beta$.

$$\begin{tikzcd} y & x \arrow[l, "g"'] \arrow[r, "\beta"] & z \arrow[r, phantom, "\vdash"] & y \arrow[rr, bend right, "\alpha"'] & x \arrow[l, "g"'] \arrow[r, "\beta"] & z \end{tikzcd}$$

*create*: if **x** is a subject, *create* defines a new graph with an edge from **x** to a new node **z**, the edge is labeled $\beta$. $\beta$ = all rights appropriate to type(z).

$$\begin{tikzcd} x \arrow[r, phantom, "\vdash"] & x \arrow[r, "\beta"] & z \end{tikzcd}$$

*remove*: if **x** is a subject with $\beta$ rights to **z**, then *remove* defines a new graph by deleting $\alpha$ labels; if the set of labels becomes empty as a result, the edge itself is removed.

$$\begin{tikzcd} x \arrow[r, "\beta"] & z \arrow[r, phantom, "\vdash"] & x \arrow[r, "\beta - \alpha"] & z \end{tikzcd}$$

It should be clear that the definition of the protection mechanism in Section 2 is closely related to this model.

- Segments in the mechanism are objects in the model. It is convenient to think of a task and its associated domain as a single subject node.

- Capabilities in the mechanism are arcs in the model. Both the capability and the labeled arc identify a unique object and specify access rights to it. Because only domains contain capabilities, only they will have out-arcs in the model.

- The 'mvcapa' operation introduced in Section 3 is a combination of the *take*, *grant*, and *remove*. In fact, if there are no delete rights for any domain, 'mvcapa' is isomorphic to an indivisible sequence of *take*, *grant*, and *remove* operations.

Thus, except for *create*, the mechanism can be viewed as a direct and efficient implementation of the model. The best way to handle create is unclear, as discussed under the 'agenda' portion of Section 7. There is one difference between the mechanism and the model, however:

- In the mechanism, a segment is named by specifying the offset of a slot in a domain that holds a capability that names the object. Indeed, in general a name will consist of some path through a set of domains. Hence, one can remove the ability to name a segment by overwriting a capability anywhere along this path.

There is no counterpart to this in the model; in the model an object is named with a unique node label. Thus, while the various operations of the model can affect one's ability to access an object, they cannot affect one's ability to name it. It should be clear, however, that the mechanism's ability to eliminate wholesale access to objects by eliminating the ability to name them cannot make a safe system into an unsafe one; specifically, the mechanism's effect can be obtained by a sequence of remove operations in the model. Thus we have preserved the ability to test for safety in linear time.

## 7. Representing Physical Resources

Our objective is to allow/encourage traditional operating system functions to be migrated to non-privileged, application-specific programs whenever sensible to do so. That implies the need for safe control of hardware resources by these programs -- resources such as the processor, memory and io devices.

We do not yet know the best way to do that in all cases, but we have an existence proof that it is feasible. The type *task* and associated cntxswap operation from Section 3 are an instance of processor control, for example. Memory can be represented as a set of the segments of type *memory* also discussed in Section 3. In a less developed version of the proposed mechanism [Wul90] we propose type *device*, which is similar to the conventional notion of memory-mapped io registers.

In each of these cases, the real issue is not representing the resource -- but wresting control at the proper times and transfering it to the proper places. The scheduler must be awakened when a scheduling decision needs to be made. The paging system needs to be awakened when there is a page fault. The device handler needs to be awakened when the device reports completion. Remember, there may be several of each of these -- schedulers, pagers and device handlers; we have to ensure that the right one is notified. A companion technical report [Wul91] explores a possible design for a mechanism to support this.

## 7. Relation to Other Capability Systems

Capability mechanisms, both proposed and implemented, have a long history, e.g. [Ili68, Den66, Fab74, Nee77, Wul80, Int82]. Our objective in the design of this mechanism was to preserve the basic strengths of these mechanisms while simultaneously achieving acceptable performance. Thus, for example,

- We constrain the object types to a fixed and finite set of hardware-relevant entities: physical memory, domains, devices and processes; we avoid more abstract operating system entities such as messages [Int82] or application-specific types [Wul80].

- We limit capabilities to non-persistent objects, thus avoiding the need for (large) unique names as in [Fab74]. In effect, our capabilities are little more than the entities in segment/paging tables of most contemporary machines.

- Access control is performed on "memory references", just as in traditional systems. Complex checking at domain crossing, as in Multics [Sch72, Sal74] or Hydra [Wul80] is not required. And, as noted earlier, the checking adds only modestly to the complexity and not-at-all to the execution time of conventional schemes.

- We limit our capabilities to exist in one type of object, domains. This avoids the need for tagging [Ili68], but also allows the type checking mechanism to be used to control manipulation of capabilities.

The last point is, perhaps worth repeating for emphasis. In conventional systems there is always the need for a mode in which protection is disabled so that the protection state (e.g., segment/page tables) can be manipulated. By making domains "just another type" with it's own set of operations, we hope to avoid this. Certainly for moving resources among domains we have avoided it -- the only open question relates to creation of new objects, which is still a subject of research.

## 8. Summary and Agenda

The basic idea behind the proposed mechanism is very simple:

- Map all protected objects, including the protection/mapping mechanism itself, into the address space of tasks.
- Distinguish among classes of objects by giving them a type.
- Define the appropriate hardware operations on these object types, and
- Use a minor variant of traditional virtual memory protection to keep the bad guys from doing what they shouldn't.

The implications of the mechanism all seem very pleasant indeed.

- It is demonstratably at least as powerful as a mode-based scheme.
- It is demonstratably no more expensive than a mode-based scheme.
- There is a formal model of it that has been extensively studied, and in which safety is decidable in linear time.
- It is "more flexible" than a mode based scheme in the sense that a wide variety of useful/interesting system structures are expressable in a natural way.
- It has the basic, positive features of capability mechanisms without performance penalties incurred by more elaborate mechanisms.
- The ability to isolate functionality in relatively small domains may actually make verification feasible.

- The ability to give resource allocation decisions to tasks that actually know something about the application may materially improve those decisions.
- The ability to avoid operating system overheads at critical times may improve the performance of important applications -- especially real-time embedded ones.
- The ability to define a rich, possibly non-monotone security policy may open the doors to commercial application.

However, there is a great deal to do before those potential benefits are realized; the following is a partial list of issues worth at least some consideration.

- It is not clear that we want 'mvcapa' to be defined as in Section 3, for example one might have "capability registers" in the CPU and load/store to/from them. This seems more in keeping with the load/store character of the rest of the architecture.

- What is the best to handle the functionality of *create*; in particular, are there hardware primitives that could assist? (It seems to us that the full functionality of create in hardware would be inappropriate.)

- It is not yet clear what the full set of types should be, or what operations should be defined on them. Specifically, we want to use typed segments to model *all* of the resources of the computer -- including i/o devices, processors, and physical memory. Precisely how to do that in all cases is not yet clear.

- It is not clear what, besides take and grant, should be the generic rights associated with all capabilities, or how many of the 'classic' protection problems one ought to try to solve directly (if any).

- What additional features, if any, are desirable for programming convenience and/or to permit a broader range of security policies to be implemented efficiently?

- There have been a number of extensions of the basic take-grant model, e.g. [Bis81, Bis88], to cope with issues such as ownership, information flow, etc. Which, if any, of these should we consider adding to the base mechanism?

- There are a number of implementation options that one might also introduce back into the formal model -- eg, the use of indirection to implement revocation. Which of these, if any, are  good ideas?

- Earlier capability-based systems suffered from unacceptable domain switching overhead. What "light weight" mechanism best fits the current mechanism? Ideally a single mechanism should replace conventional interrupt, trap, and 'svc' (system call) facilities.

That's why this report is just a "stake in the ground"; we're still researching these and related questions.

## Bibliography

[Bel73] Bell, D. and Lapadula, L., "Secure Computer Systems: Mathematical Foundations and a Model", Mitre Report MTR 2547, v2 Nov 73.

[Ben84] Benzel, A., et. al., "Analysis of Kernel Verification", Proceeding of 1984 IEEE Symposium of Security and Privacy, pp 125-31

[Bib77] Biba, K., "Integrity Considerations for Secure Computer Systems", US Air Force Electronic Systems Division, 1977.

[Bis81] Bishop, M. "Heirarchical Take-Grant Protection Systems", Proceedings of the 8th Synmposium on Operating Systems Principles, Dec 1981, pp107-123

[Bis88] Bishop, M., "Theft of Information in the Take-Grant Protecton Model", Proceedings of the Computer Security Foundations Workshop, June 1988, pp 45-54.

[Che81] Cheheyl, M., et. al., "Verifying Security", Computer Surveys, v13 n3, Sept 1981, pp 279-339.

[Col88] Colwell, R.P., Gehringer, E.F., and Jensen, E.D., "Performance Effects of Architectural Complexity in the Intel 432", ACM Transactions on computer Systems, v6 n3, August 1988, pp 296-339.

[Dav80] Davida, G., "A System Architecture to Support a Verifiably Secure Multilevel Security System", Proceedings 1980 IEEE Symposium of Security and Privacy, pp 137-44.

[Den66] Dennis, J., and van Horn, E., "Programming Semantics for Multiprogrammed Computations", CACM v9 n3, March 1966, pp 143-55.

[Den76] Denning, D., "A Lattice Model of Secure Information Flow", CACM, v19 n5, May 1976, pp 236-43.

[Den77] Denning, D. and Denning, P., "Certification of Programs for Secure Information Flow", CACM, v20 n7, July 1977, pp 504-13.

[Den82] Denning, D. "Crytography and Data Security", Addison-Wesley, 1982.

[DoD85] "Trusted Computing System Evaluation Criteria", DOD5200.28- STD Dec 1985 [The "orange book"]

[Fab74] Farby, R., "Capability Based Addressing", Communications of the ACM, v17 n7, July 1974

[Gra72] Graham, G. and Denning, P., "Protection -- Principles and Practice", Proceedings 1972 Joint Computer Conference, pp 417-29.

[Ili68] Illiffe, J. K., "Basic Machine Principles", American Elsivier, 1968

[Har85] Harrison, M., "Theoretical Issues Concerning Protection in Operating Systems", Advances in Computers, v24, 1985, pp 61-100.

[IBM90] IBM Corp., "IBM RISC System/6000 POWERstation and POWERserver Hardware Technical Reference - General Information", SA 23-2643-00 First Edition, 1990.

[Int82] Intel Corporation, "iAPX432 General Data Processor Architecture Reference Manual, Rev 3", Manual 171860-003, Intel Corporation Santa Clara, Ca. 1982.

[Jon75] Jones, A. and Wulf, W., "Towards the Design of Secure Systems", Software-Practice and Experience, v5 n44, Oct-Dec 1975, pp 321-36.

[Jon78] Jones, A., "Protection Mechanism Models: Their Usefullness", Foundations of Secure Computation, Academic Press, 1978 pp 237-52.

[Jon78] Jones, A. and Lipton, R., "The Enforcement of Security Policies for Computation", Journal of Computer and Systems Science, v17 n1, August 1978, pp 35-55.

[Kan89] Kane, Gerry, MIPS RISC Architecture, Prentice Hall, 1989.

[Lam73] Lampson, B., "A Note on the Confinement Problem", CACM v16 n 10, October 1973, pp 613-15.

[Lam74] Lampson, B., "Protection", Proceedings of the 5th Symposium on Operating Systems, v8n1, Jan 1974, pp 18-24

[Lan81] Landwher, C., "Formal Models for Computer Security", Computing Surveys, v13 n3, September 1981, pp 247-78.

[Lev75] Levin, R., et al., "Policy/Mechanism Separation in Hydra," Fifth Symposium on Operating Systems Principles, November 1975.

[Lip77] Lipton, R and Snyder, L., "A Linear Time Algorithm for Deciding Subject Security", JACM, v24 n3, July 1977, pp 409-45.

[Nee77] Needham, R., and Walker, R., "The Canbridge CAP Computer and Its Protection System", Poceedings of thew Sixth Symposium on Operating System Principles, Nov. 1977, pp 1-10.

[Pfl89] Pfleeger, C., "Security in Computing", Prentice Hall, 1989.

[Rab88] Rabin, M., "An Integrated Toolkit for Operating System Security", Harvard Univ TR-05-87, Aug 1988 (revised).

[Sal74] Saltzer, J., "Protection and the Control of Information Sharing in Multics", CACM, v17 n7, July 1974, pp 388-402.

[Sch72] Schroeder, M., and Saltzer, J., "A Hardware Architecture for Implementing Protection Rings", CACM, v15 n3, March 1972, pp 157-70.

[SUN87] Sun Microsystems, Inc., "The SPARC Architecture Manual, Part No: 800-1399-08, 1987.

[Syn81] Snyder, L., "Formal Models of Capability-Based Protection Systems", IEEE Transactions on Computers, vC-30 n3, Mar 1981, pp 172-81.

Wul74] Wulf, W. , et al, "Hydra - The Kernel of a Multiprocessor Operating System," CACM, June 1974.

[Wul80] Wulf, W. , et. al., "Hydra/C.mmp: An Experimental Computer System". McGraw-Hill Publishing Co., 1980.

[Wul88] Wulf, W. , "The WM Computer Architecture", Computer Architecture News, March 1988.

[Wul90] Wulf, W., "The WM Computer Architecture -- Principles of Operation", U. Virginia Computer Science TR-90-2, Jan 1990.

[Wul91] Wulf, W. "A Proposal for an Interprocess Communication in WM", U. Virginia Computer Science Technical Report, March 1991