An Augmented Capability Architecture to Support Lattice Security and Traceability of Access

Paul A. Karger
Digital Equipment Corporation
295 Foster Street
Littleton, MA 01460

Andrew J. Herbert
Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge, England CB2 3QG

Abstract

This paper describes a protection system that supports the confinement of access as required by non-discretionary access control models such as the Bell and LaPadula lattice model. The approach is to use capability-based protection at the lowest level for implementing confined domains, in support of access control lists for expressing security policies outside the security kernel. The implementation of such a system in the context of hardware support for capabilities is discussed.

Introduction

Capability-based addressing schemes historically have been developed as an efficient means of structuring a computer system into a collection of non-hierarchical protection domains with separately defined access rights specified for relatively small memory objects. A large number of non-hierarchical protection domains are useful in supporting software structuring and reliability, because the damage that any software module can do is limited to that protection domain.

Capabilities were first proposed by Dennis and Van Horn [8]. A number of software capability systems have been implemented, including the MIT PDP-1 [1], CAL-TSS [16], and HYDRA [33]. Hardware-based capability implementations have had greatly reduced overhead compared to the software systems and a consequent increase in performance. Such hardware-based systems have included the Plessey System 250 [6], the Cambridge CAP Computer [31], the IBM System/38 [12], and the Intel 432 [13]. Levy [18] provides a survey and comparison of a number of these systems.

Despite the protection advantages of small domains, capability systems have not been well matched to the security policies that most users desire for real applications. In particular, traditional capability systems do not support traceability of access, because they cannot easily answer the very common user question, "Who has access to this object?" Furthermore, traditional capability systems have had difficulty supporting lattice security policies. In particular, the philosophies of passing capabilities from domain to domain as part of procedure calls and of storing capabilities in arbitrary shared objects are at odds with the confinement goals of lattice security policies.

There have been various attempts to augment capability architectures with complex rules for the confinement of propagated capabilities. It is our opinion that these augmentations have seemed clumsy when compared to the access control list approach, and in several cases, not so effective.

We believe that the access control list model is ideal for describing the relationships between users and stored data. It is less well suited to describing the use of protected subsystems where there are three issues to consider: the access class of the user, that of the data, and those rights belonging intrinsically to the subsystem independent of its user. We feel that by a combination of access control lists for describing the user/stored data relationship and capabilities to implement subsystem protection, we achieve a simpler mechanism to understand and implement than is possible by using either model in isolation. Thus, our approach is to consider those modifications necessary to a capability-based protection kernel that will support access control lists at the outer level.

This paper presents the opinion of its authors, which is not necessarily that of Digital Equipment Corporation. Opinions expressed in this paper must not be construed to imply any product commitment on the part of Digital Equipment Corporation.

	OBJECT 1	OBJECT 2	OBJECT 3	• • •
SUBJECT 1	R	RW	RW	
SUBJECT 2	RW	NULL	R	
SUBJECT 3	NULL	NULL	R	
• • •				

FIGURE 1. LAMPSON'S ACCESS MATRIX

Traceability of Access

The most general model of security is Lampson's access matrix [17] in which the rows of the matrix represent the active entities or subjects of the system and the columns represent the information containing objects of the system. * Access rights are determined by the values specified at the intersection of the appropriate row and column. (See figure 1.)

In a capability system, each subject has a list of objects with specified access rights, corresponding to the non-empty elements of a row of the access matrix. When a subject passes a capability to another subject, an entry is copied into the capability list of the receiving subject.

By contrast, in an access control list system, each object has a list of the subjects who have access to that object. The list corresponds to the non-empty elements of a column of the access matrix. (Access control list implementations frequently use compression techniques to reduce the total number of entries required.) A suitably privileged subject may grant other subjects access to an object by modifying the access control list.

Viewed abstractly, capabilities and access control lists are equally powerful. Both systems can express the full generality of the Lampson access matrix. However, in practice the two systems are not necessarily identical because of the differences in representing privilege. There are two questions that a user frequently will ask of an access control system:

1. To which objects may a given subject access, and

2. Which subjects have access to a given object?

The questions themselves are symmetric. In a capability system, the first is easily answered by inspecting a single capability list, but the second requires a search of all capability lists. Alternatively, for access control list systems, the second question is easily answered by inspecting a single access control list, but the first question requires a search of all such lists.

In our view, pure capability systems are less obvious ways of expressing security policies than those based on access control lists, because the second question is more frequently of interest than the first. This is because in the world of security, one is concerned with who is to be granted access to particular data. A security officer investigating an incident needs to know who has access to a compromised object. It is less common for a user to want a list of all the objects to which he or she has access.

Thus, although capability and access control list systems are functionally equivalent, the asymmetry of access traceability has pushed many of the implementors of secure systems toward

* Lampson actually used the term domain rather than subject. The term subject was first used by Graham and Denning [10].

preferring access control lists. Such user preference has been expressed by Woolf [32].

Lattice Security Model

The lattice security model was developed by the MITRE Corporation [3] and Case Western Reserve University [29] as a security policy to meet national defense requirements and to confine the potential damaging actions of programs that contain malfeasant software modifications such as Trojan Horses. *

The lattice security model defines a set of access classes that form a mathematical lattice. Two access classes may be compared and one may be greater than, equal to, or less than the other; or the two access classes may be disjoint. There exist a minimum access class called "system low" and a maximum access class called "system high." System low is less than all other access classes, and system high is greater than all other access classes. Each information containing object and each active subject in the system are assigned an access class. For a subject to read information from an object, the access class of the subject must be greater than or equal to the access class of the object. Put more simply, the subject must be cleared for the information contained in the object. For a subject to write information into an object, the access class of the subject must be less than or equal to the access class of the object. This write permission rule has been called the *-property [3] or confinement property [14]. The confinement property is chosen to ensure that a Trojan Horse cannot leak information to a lower access class. Note that active entities here are processes executing programs that may be untrustworthy. By contrast, people are considered trustworthy up to their maximum access classes (their security clearances). Thus, a human being may have processes (subjects) of different access classes at different times. The confinement problem is discussed at greater length in [15] and [20]. While the lattice model was first developed for military use, it has potential applications for commercial use [21]. A product version of the lattice security model is available as the Access Isolation Mechanism of the Multics operating system [30].

* The term "Trojan Horse" was first used in the context of computer security systems by Dan Edwards [2].

Protected Subsystems

One of the more important ideas to emerge from capability systems is that of a protected subsystem, the motivation for which comes from the doctrine of minimum privilege or "need to know". An example might be that of a package for computing missile flight trajectories that relies on internal tables giving the performance parameters of various missiles. While a particular user may be authorized to run the package to analyze trajectories, it is quite probable that he should not be allowed access to the parameters from which the trajectory is computed. It is possible to think of many other examples of subsystems that rely on data or program code that is not to be accessible to users of the program and in the most general case, this leads to the requirement for non-hierachical domains of protection. That is to say the domain of protection for a subsystem will include only those capabilities that describe the intimate apparatus of the subsystem and argument capabilities passed to the domain by the caller. This should be contrasted with ring-based protection systems [27] where moving to an inner ring (i.e. a protected subsystem), causes all of the access rights of the outer rings to be available as well. A protection domain is itself represented as a capability and control over the use of a subsystem can be exercised by limiting propagation of this capability.

Attempts to implement protected subsystems using access control lists for protection have not been common. In part, the difficulty arrives from the duality of a protected subsystem as a passive object which a user may be authorized to call, and as an active subject when it is invoked. It would be necessary for security managers to declare not only which users are allowed access to data, but which subsystems can process those data. This task becomes especially hard when a new program is introduced, since many access control lists in the filing system will require alteration to permit use of the program. Furthermore, such an access control list approach does not have the property of minimum privilege, because a subsystem can access any object to which it has been granted access in an access control list. By contrast, a capability-based domain can only access objects for which it has capabilities built into its structure, or are explicitly passed as parameters when the sub-system is called.

Capability systems have also provided better support for typed objects than have access control list systems. A subsystem may well wish to give the caller some token or data to represent a privilege or resource associated with the subsystem. An example might be that of a filing system that gives its caller capabilities representing the directories or catalogs they may use. It is an essential part of minimum privilege that the caller should not be able to interfere with the representation of such things, and a number of capability systems [32, 10] provide support for sealing capabilities and marking their type so that they can be recognized and unsealed by the subsystem that made them. A user is prevented from direct access to the object by virtue of the capability being sealed and must ask the subsystem, or type manager to perform operations on the user's behalf.

In a capability-based system, a filing system directory or catalogue contains capabilities for the objects in the directory. Subsystems are filed as capabilities for their domain, although of course the objects that go to make up the domain may be filed under other names as individual objects, but only accessible to the maintainer of the subsystem, and not to its users. (Figure 2 shows a filing system directory containing a subsystem that itself contains embedded capabilities.) At this outer filing system level, it more rational to move to an access control list view of protection, since the purpose is that of indicating which users are entitled to access the various objects for which capabilities have been preserved in the filing system.

Pure access control list systems cannot be discounted, however, in the support of protected subsystems and typed objects. Reportedly, Data General has built a machine [7] that supports protected subsystems using a pure access control list approach that may have overcome some of the difficulties described above. The Data General scheme is based in part on Schroeder's work on mutually suspicious subsystems [28]. However, little information is available in the public domain on their system.

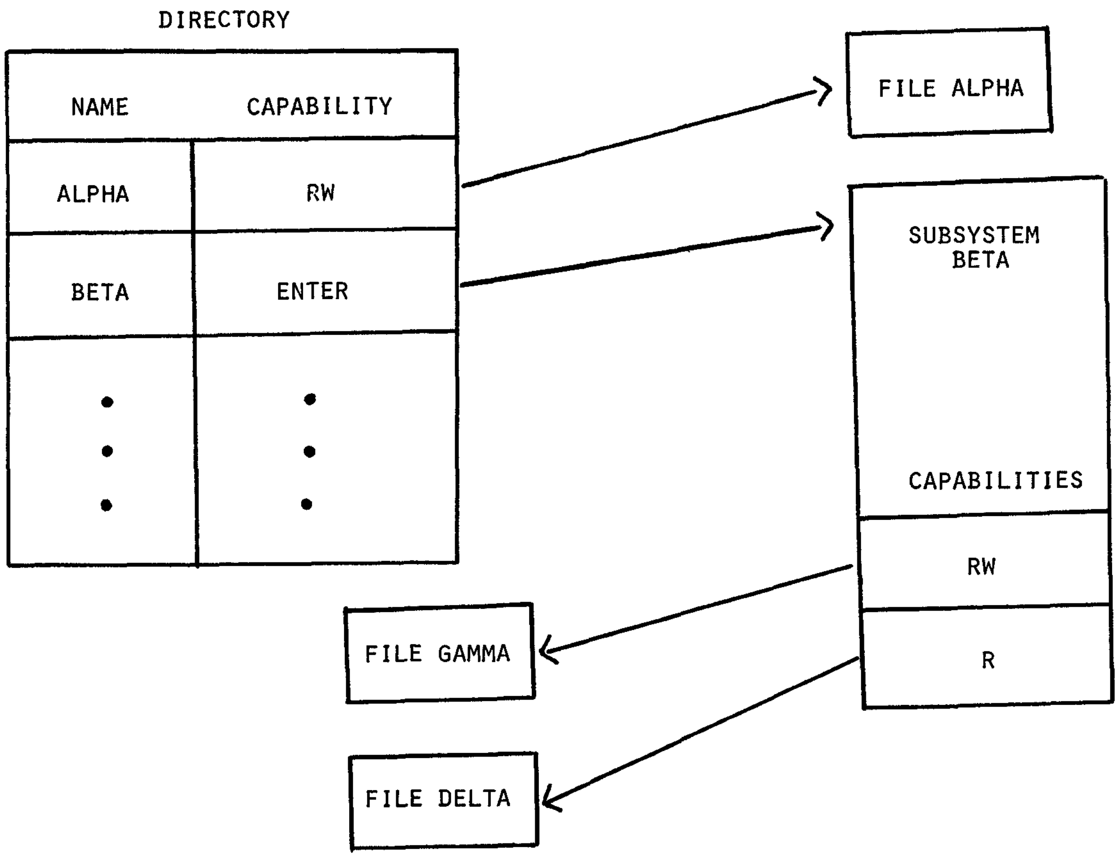

FIGURE 2. FILING SYSTEM DIRECTORY

Supporting the Lattice Model with Traditional Capabilities

Incorporating the lattice security model into an access control list system, such as Multics [30], is relatively straightforward, because all protection rights are associated with the object, and those rights cannot pass from subject to subject without operating system intervention. When the operating system intervenes, the lattice security rules can be enforced. By contrast, capability systems normally allow unrestricted flow of access rights from subject to subject as part of procedure calls. Two traditional capability systems, HYDRA [33] and PSOS [24], have proposed support for the lattice model. Each of these approaches have drawbacks (discussed below) that we will overcome in our scheme.

HYDRA was a capability-based system developed at Carnegie-Mellon University to study both capability systems and tightly coupled multiprocessors. HYDRA attempted to solve the confinement problem by introducing a special right called UncfRts or "unconfined rights" to limit the propagation of information. A domain in HYDRA is confined if it is called without UncfRts. Such a confined domain loses the right to store capabilities or information into any inherited (and potentially shared) objects. The confined domain may create new objects, but cannot share those new objects with any other domains. The confined domain is limited to modifying the arguments with which it was called.

Unfortunately, the HYDRA solution to the confinement problem is both overly restrictive and is not well-suited to protected subsystems outside the security kernel. In HYDRA a confined domain loses the right to store into any inherited objects, even objects that the lattice security model would have permitted. As a result, software that was written for an unconfined environment will likely not work when run in a confined environment, even when all operations are at a single access class and no security violations in fact take place. Furthermore, these restrictions on storing mean that system programs such as the filing system, command language interpreters, compilers, and editors, etc. cannot be confined. Since formal verification of software is very difficult in practice, it is necessary that most software a user runs (including these utilities) must be untrusted and run in a confined environment, so that only a small security kernel remains to be verified. Under these assumptions, the HYDRA solution to the confinement problem becomes unworkable.

PSOS, the Provably Secure Operating System, is a design for a capability-based system about which strong security properties could be proven. The basic PSOS design does not support the lattice model, but instead, contains a secure object manager layered above the basic elements of PSOS. The secure object manager solves the confinement problem by ensuring that capabilities with write permission are never stored into secure documents. (These are the basic unit of information storage of the secure object manager.) Thus, improper propagation of write permissions in violation of the *-property are barred. PSOS prevents improper propagation by a complex set of rights that separately determine whether a write capability can be stored.

PSOS has never been implemented and remains only a design concept. However, the design as presented in [24] appears to be overly restrictive about capability propagation and could have difficulties when a user wished to transport software from a PSOS that did not support the lattice model to a PSOS that did support the lattice model. In particular, such software might make explicit calls on modules that were concealed by the secure object manager. In addition, such software could not function with the secure object manager if it stored write capabilities into objects. Even if the software attempted no security violations, the secure object manager would prevent the storing of write capabilities and thus cause the software to malfunction.

We shall see below that our augmented capability architecture will support the lattice security model without the restrictions of the HYDRA and PSOS solutions. Further, our augmented architecture will also support traceability of access which neither HYDRA nor PSOS support.

The Augmented Capability Architecture

To illustrate the mechanics of our proposed architecture we will use the example (shown in figure 3) of a user U running the missile trajectory sub-system M. U will have logged in to the system at some authorized access class A (e.g. one of unclassified, confidential, secret, or top secret) and has a file called D that contains information at this level about trajectories to be analyzed. Thus the access control list for D will list U as an authorized user.

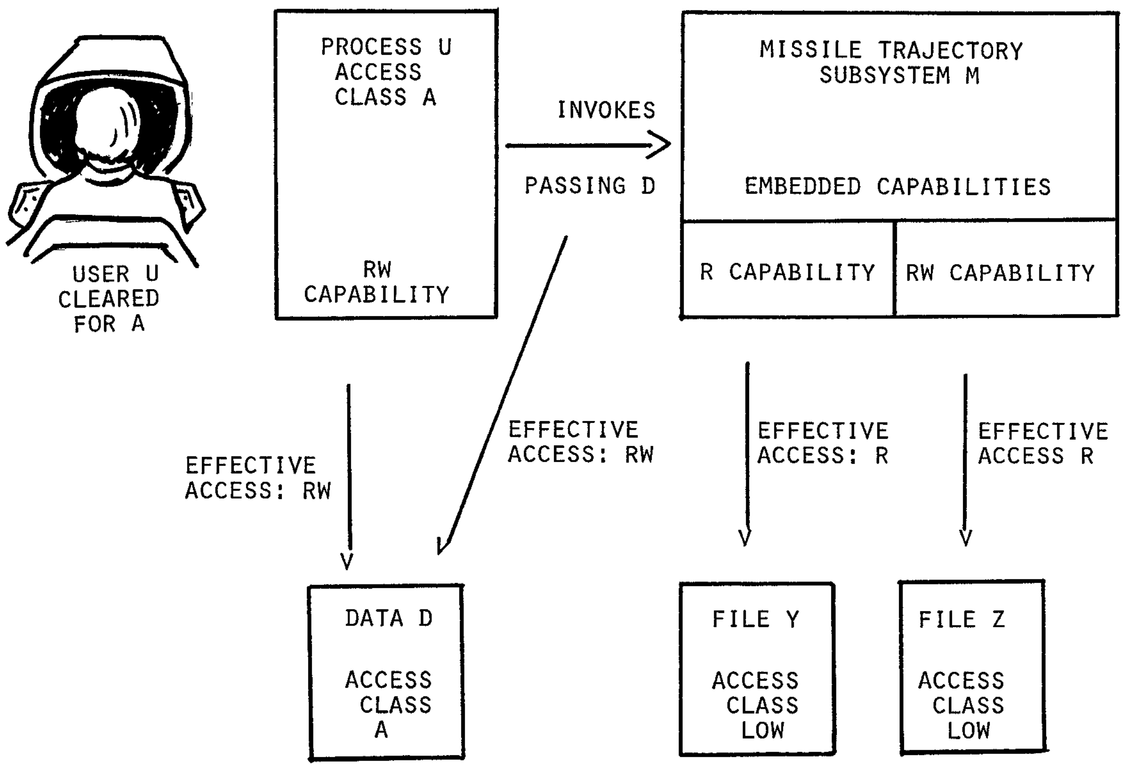

FIGURE 3. NON-DISCRETIONARY EXAMPLE

In our capability system, in common with many others, there will be a table of descriptions of objects, and capabilities will indirect to entries in this table. The exact implementation of this table is not important here, save to say that all capabilities for a given object will point at the same entry. To be practical, a capability system must cache the results of evaluating capabilities in a hardware lookup table, so that efficiency shall not suffer greatly. We assume that the table is so structured that it is possible to flush all entries in the cache derived from a given entry in the central object table. (Flushing of this sort is available in the capability unit, i.e. capability cache, of the Cambridge CAP computer [31].) Further we assume some sort of associative search or direct look up is provided in the cache so that the overheads of capability evaluation will not be unduly expensive. To avoid the obvious storage channels, we will assume that the central object table can only be inspected by trusted code (presumably the security kernel), and that the indexes into the central object table (that must be stored in capabilities) are not visible to the user.

Non-Discretionary Security

The first problem to address is that of non-discretionary security: that is to prevent the flow of material in one access class to a lower access class. The Bell and LaPadula requirements state that a process may read information from an object of lower or equal access class and can write information to an object of equal or greater access class. In our example, the process acting for U will be labelled as running in access class A. Therefore a capability granting read access to an object must only be successfully evaluated if the object is marked as belonging to access class A or lower. Capabilities granting write access must only be successfully evaluated if the object is marked as belonging to access class A or above. Initially, let us assume that the access class of the object and the access class of the process are directly available to the capability unit of the CPU. Enforcement of the Bell and LaPadula rules then becomes trivial. However, making such information directly available to the capability unit has two major drawbacks. First, it needlessly adds complexity to the capability unit;

and second, it restricts the CPU to supporting only the Bell and LaPadula lattice model, and not some other incompatible security model.

However, we can take advantage of the caching property of the system to implement the lattice model more effectively. If the processor does not automatically load capabilities into the cache, but instead, causes a fault or trap to the most privileged software domain (which we shall call the security kernel domain), then the security kernel domain can evaluate whether the lattice model permits the capability to be used. The CPU hardware and microcode need not understand the lattice model. They need only ensure that capabilities are only loaded into the cache upon command of the security kernel domain.

How would a protected subsystem operate under the lattice model? Suppose that all writeable objects in M belonged to the highest access class and all readable objects to the lowest class. In that case, M could be run by any process at any access class, although no objects in M would be simultaneously readable and writable. Now in practice, the objects of M will not be so simply classified, and any reduction in classification of writeable objects, or increase in classification of readable objects reduces the access classes at which M can be run. For this reason a sub-system will have some a priori set of access classes in which it will be able to run without hindrance. To this purpose it should be possible for a program to inspect the access class in which it is run so that it can elect to retire gracefully, rather than being forcibly prevented from accessing objects outside its domain of protection.

This leaves some question about data preserved between runs of the sub-system, such as accounting records; at what access class should they be classified? The answer is either in the highest access class to enable 'writing up', or else the domain should have a separate account file classified at each access class and the sub-system must chose on the basis of the access class of the caller. In this respect, it should be noted that the restriction of use of objects by a domain should be applied dynamically, so that a domain may contain capabilities for objects inaccessible in the present access class and will be allowed to continue running, provided that these capabilities are not exercised. Figure 3 shows that subsystem M contains two embedded capabilities, one for a read-only file Y at access class Low (Low < A) and one for a read-write file Z at access class Low. However, the augmented capability architecture will ensure that if M is operating on behalf of U at access class A, M's effective access to Z will be restricted to read-only. Thus, even if M contains a Trojan Horse, M will be unable to violate the Bell and LaPadula model and compromise the data D.

To implement non-discretionary security then, it is necessary to label every process with the access class of the user logged in to it and to label each object in the map with the access class to which it belongs. It is then possible to implement the Bell and LaPadula 'read down' and 'write up' rules to prevent information flow between access classes. These rules will be applied uniformly to all objects in which data can be stored; namely, files, data segments, capability-lists filing system directories, inter-process communication objects, etc. Uniform enforcement is possible, because a capability must be present in the cache before a process can perform any action, and because loading capabilities into the cache is controlled by the security kernel domain.

The security kernel domain could grant certain other domains the right to not be subject to the lattice security model. Such domains might be used to implement portions of the security kernel itself, to implement a sanitization or downgrading facility, to enforce the lattice model on finer grained objects, such as database records, or to structure a monolithic kernel domain into a collection of smaller domains. Such privileged domains are the analog of the trusted processes that can be found in many existing security kernel implementations [21, 8]. Such trusted domains must be verified, just as the security kernel domain must be. However, we hope that the use of a capability architecture to limit the access of even privileged domains may make the security verification of the code of the domain easier. (It is beyond the scope of this paper to discuss security verification. The interested reader may find a useful treatment of the subject in [5].)

Discretionary Security

The second problem to address is that of discretionary security - the restriction of rights of access to objects by particular named users. Since we have explained above how to achieve non-discretionary control, the implementation of discretionary security will be considered independently. Let us suppose in the example in figure 4 that the sub-system M was written by a user V who wishes to steal the data D passed to M by U and that V can work in the same access class A as U. V's strategy might be to cause M to store a copy of the capability for D in file F and then subsequently to run M himself and cause it to give him a copy of the capability. If there is a revocation scheme for disabling argument capabilities after completion of a domain call, V might try to subvert the revocation by running another, parallel instance of M in his own process and passing the capability via a shared capability-list common to both instances. A brute force solution to these copying problems is to insist that all capability-lists in a domain that are shared between instances cannot be writeable, but this is a rather restrictive view and limits the utility of protection domains.

Our full solution is to include access control list inspection as part of the basic capability evaluation mechanisms. Suppose each object description includes its access control list (or some representation of it). Further, let the user name of the user logged in to a process be available to the capability machinery. Discretionary controls can then be applied if the rule is imposed that the access control list should be checked as part of capability evaluation and that capabilities from a shared capability-list in different instances of a domain are held at different places in the capability cache. (The capability unit of the Cambridge CAP computer has exactly this property.) Then the first time a domain comes to use a capability, the access control list will be checked and in the example above, the instance of M run by V will not be able to use a capability for D stored in file F, because V is not in the access control list for D.

Using the hardware and microcode of the system to handle access control lists would make the capability unit needlessly complex and restrict the system to a particular style of security policy. However, we can take advantage of the same caching strategy that we used for non-discretionary controls to support access control lists. We note that the access control list checking need only be done the <u>first</u> time a capability is used by an instance of a domain; thereafter ordinary capability re-evaluation is sufficient. Therefore, the security kernel domain can evaluate the access control list at the time that it loads the capability into the cache, and can

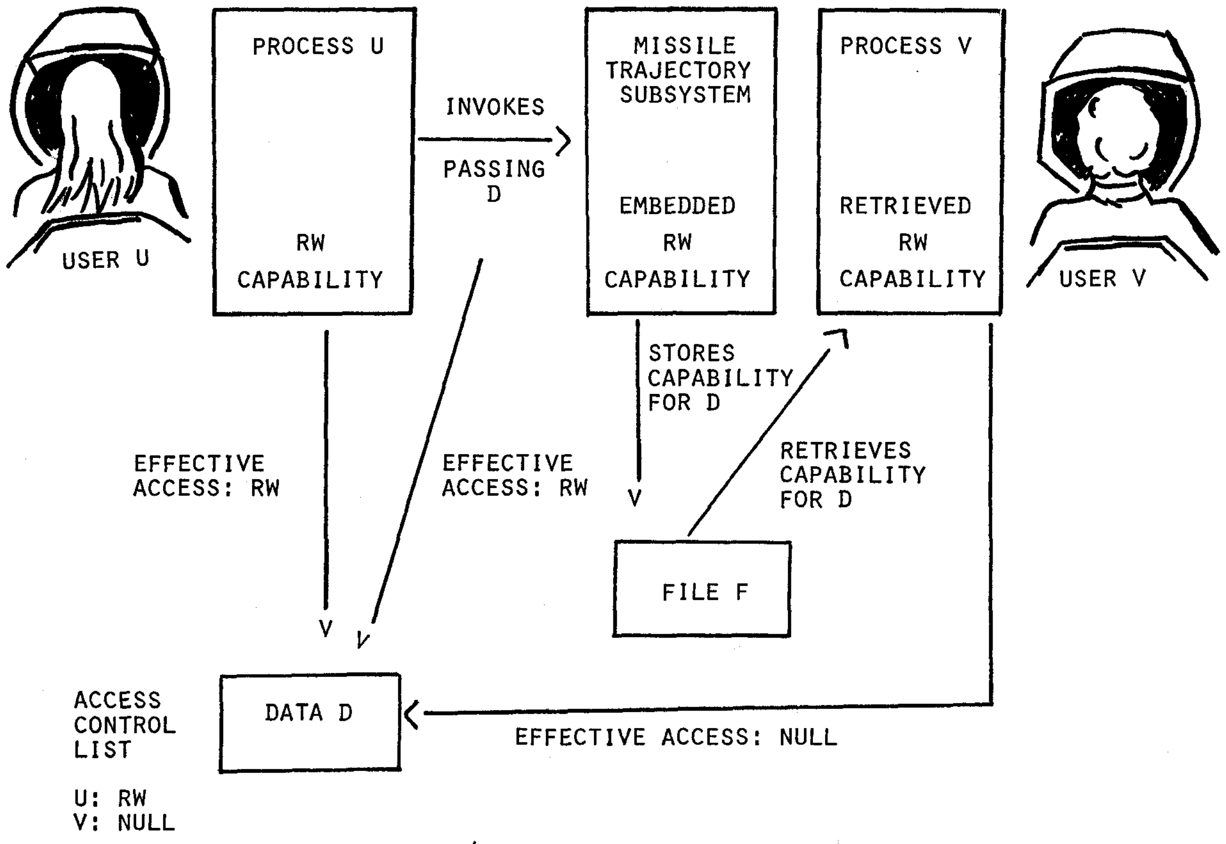

FIGURE 4. DISCRETIONARY EXAMPLE

refuse to load the capability, if the access control list so directs. So by a simple change to the capability mechanisms that support intervention during the evaluation of capabilities, we can permit the system to support arbitrary non-discretionary and discretionary security policies. This type of intervention is analogous to missing segment fault processing in Multics [25] and similar operating systems. Therefore, given a suitably large enough cache to prevent excessive re-evaluation of capabilities, we do not expect the performance cost to be excessive.

In the context of discretionary security, there is another line of attack by which V can acquire a copy of the data D. The technique is simply to make a copy to another object within M. (It should be noted that D cannot be overwritten by this route, but only stolen.) The access control list on the second object must indicate that the object is writable by U so that M can make the copy and readable by V so that the data may be read out subsequently. It is precisely this problem, inherent to any discretionary policy, that motivates practical security systems to employ non-discretionary and discretionary policies in conjunction. By varying the access classes of U and V appropriately, the *-property can be relied upon to restrict information flow from D to the second object.

The security manager in a protected system needs the ability to engage in operations such as modifying the security class of information and setting up group access control lists. These privileges can themselves be represented as capabilities that are recognized by the security kernel and the mechanisms outlined allow them to be kept in the filing system and treated as <u>bona fide</u> capabilities or privileges.

It is also possible to cope with immediate revocation in our scheme without resort to the complication of indirection through special revoker mappings fround in other capability systems [26]. If an access control list is modified for a particular object, all entries in the cache for that object need to be flushed. This will cause all capabilities for the object to be evaluated afresh and any changes in discretionary policy will be noted directly.

Our proposed capability architecture supports both discretionary and non-discretionary access controls by augmenting a basic capability mechanism. It is important to note that the scheme relies on 1) carrying information about access classes and access control lists down to the lowest level of the kernel and 2) taking some care in the rules for leaving evaluated capabilities in a capability cache. The basic rule is that when a capability is first used in an instance of a domain it must be subject to both access control list checking and non-discretionary checking. It is our belief that such a scheme could be implemented on the Cambridge CAP Computer which has most of the necessary mechanisms built into its capability unit. Furthermore it is significant to note that we can support the lattice security model transparently. That is, we can take a software module and wrap it up as a protection domain, and as long as it operates at a single access class and makes no security violations, all of its capability operations will work correctly. It should be noted that with most capability machines the creation of domains and organization of capabilities is done by language compilers, and the user is blissfully unaware of the underlying machinery until his program commits some transgression. This combined transparency is especially important, because commercial software developers and users may not run the lattice security model or a capability system and may not design software to be constrained in such ways. However, users of the lattice security model will want to run such software, albeit in guarded circumstances.

<u>Comparison With Other Systems</u>

After developing our strategy for supporting access control lists and non-discretionary controls, Levy [19] pointed out to the authors that the IBM System/38 supports a concept of authorized and unauthorized pointers that is similar to our proposal. In the System/38, unauthorized pointers cannot be used without first checking an access control list (called a user profile in the System/38 documentation). For efficiency, the owner of an object or anyone with "create authorized pointer" rights to an object may create an authorized pointer that carries access rights to the object within itself and does not require checking of the access control list.

If the System/38 had only unauthorized pointers, it would support the architecture proposed in this paper. However, because the System/38 permits a user to create, store, and pass an authorized pointer, an implmentation of the lattice security model could be bypassed. The System/38 manuals recommend that the user avoid the use of authorized pointers, because "a user can pass them to other users who have not been explicitly authorized to use an object." [11, p. 2-42] However, the creator of an object can always create an authorized pointer to that object, resulting in the potential for a lattice security model violation.

The Multics [25] strategy of combining segment descriptor words with access control lists is also similar to the architecture proposed in this paper (and in fact was the inspiration for our work.) However, Multics segment descriptors are not true capabilities in that they cannot be passed from user to user or protection domain to protection domain.

The IBM SWARD system [23] also has some similarity to our proposed architecture. The SWARD operating system [4] defines access sets to control inter-user sharing. Access sets are similar to access control lists as used in this paper. SWARD also contains a primitive control over the propagation of capabilities, in that a SWARD capability cannot be copied unless the capability itself grants copy permission. This type of propagation control seems similar to that in PSOS, but SWARD has not attempted to support the lattice model, and it is unclear from the available documentation whether confinement is possible.

Conclusions

We have proposed an augmented capability architecture that can support the lattice security model and supply the traceability of access that access control lists offer. However, the augmented architecture preserves the benefits of a capability system by offering support for small non-hierarchical protection domains, sealed objects, protected abstract data types, etc. In addition, the architecture's support of the lattice security model is transparent to software operating at a single access class. Thus, software that was written without considering any security implications can continue to run under the lattice model, as long as no security violations occur.

We feel it is important, on grounds of performance, that any implementation of a security system must have hardware support. It is our belief that an a architecture of the sort described could be readily supported by a machine such as CAP [31] which has the necessary hardware and microprogram facilities to cache capabilities and implement operations upon them.

Acknowledgments

Maurice Wilkes, Steven Lipner, Roger Needham, and Dorothy Denning provided comments and suggestions that greatly influenced this paper. We must also thank Dolores Miller for preparing the figures.

References

[1] Ackerman, W. B. and Plummer, W. W., "An Implementation of a Multiprocessing Computer System," Proceedings of the ACM Symposium on Operating System Principles, October 1967.

[2] Anderson, J. P., Computer Security Technology Planning Study, James P. Anderson and Co., ESD-TR-73-51, Vols. I and II, HQ Electronic Systems Division, Hanscom AFB, MA, October 1972.

[3] Bell, D. E. and LaPadula, L. J., Computer Security Model: Unified Exposition and Multics Interpretation, The MITRE Corp., ESD-TR-75-306, HQ Electronic Systems Division, Hanscom AFB, MA, June 1975.

[4] Buckingham, B. R. S., The SWARD Command Language (CL/SWARD), IBM Systems Research Institute, TR-73-011, New York, NY, February 1981.

[5] Cheheyl, M. H., et. al., "Verifying Security," Computing Surveys, XIII, 3, September 1981, pp. 279-339.

[6] Cosserat, D. C., "A Capability Oriented Multi-Processor System for Real-Time Applications," Proceedings of the International Conference on Computer Communications, October 1972, pp. 282-289.

[7] DeLashmutt, L., "Trusted Computing Research at Data General Corporation," Proceedings of the Fourth Seminar on the DoD Computer Security Initiative, National Bureau of Standards, Gaithersburg, MD, 10-12 August 1981, pp. J-1 - J-21.

[8] Dennis, J. B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations," CACM, IX, 3, March 1966, pp. 143-155.

[9] Gold, B. D., et al, "A Security Retrofit of VM/370," Proceedings of the 1979 National Computer Conference, AFIPS Press, Montvale, NJ, 1979, pp. 335-344.

[10] Graham, G. S. and Denning, P. J., "Protection - Principles and Practice," Proceedings of the 1972 Spring Joint Computer Conference, AFIPS Press, Montvale, NJ, 1972, pp. 417-329.

[11] Herbert, A. J., "A Hardware Supported Protection Architecture", in Lanciaux, D., editor, Operating Systems Theory and Practice, North Holland Publishing Co., Amsterdam, pp. 293-3Ø6, 1979.

[12] IBM Corporation, IBM System/38 Functional Concepts Manual, GA21-933Ø-1, Rochester, MN, June 1982.

[13] Intel Corporation, iAPX 432 General Data Processor Architecture Reference Manual, Preliminary Edition, Aloha, OR, 1981.

[14] Karger, P. A., Non-Discretionary Access Control for Decentralized Computing Systems, SM and EE thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-179, Cambridge, MA, 1977.

[15] Lampson, B. W., "A Note on the Confinement Problem," CACM, XVI, 1Ø, October 1973, pp. 613-615.

[16] Lampson, B. W. and Sturgis, H. E., "Reflections on an Operating System Design," CACM, IXX, 5, May 1976, pp. 251-265.

[17] Lampson, B. W., "Protection," Proceedings Fifth Princeton Conference on Information Sciences and Systems, Princeton University, March 1971, pp. 437-443, reprinted in Operating Systems Review, VIII, 1, January 1974, pp. 18-24.

[18] Levy, H. M., Capability-Based Computer Systems, Digital Press, Bedford, MA, 1983.

[19] Levy, H. M., Digital Equipment Corporation, private communication, 1982.

[2Ø] Lipner, S. B., "A Comment on the Confinement Problem," Proceedings of the Fifth Symposium on Operating System Principles, Operating Systems Review, IX, 5, November 1975, pp. 192-196.

[21] Lipner, S. B., "Non-Discretionary Controls for Commercial Applications," Proceedings of the 1982 Symposium on Security and Privacy, IEEE, Oakland, CA, 26-28 April 1982, pp. 2-1Ø.

[22] McCauley, S. M. and Drongowski, P. J., "KSOS - The Design of a Secure Operating System," Proceedings of the 1979 National Computer Conference, AFIPS Press, Montvale, NJ, 1979, pp. 345-351.

[23] Myers, G. and Buckingham, B. R. S., "A Hardware Implementation of Capability-Based Addressing," Operating Systems Review, XIV, 4, October 198Ø, pp. 13-25.

[24] Neumann, P. G., et al, A Provably Secure Operating System: The System, Its Applications, and Proofs, Computer Science Laboratory Report CSL-116, SRI International, Menlo Park, CA, 7 May 198Ø.

[25] Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, Cambridge, MA, 1972.

[26] Redell, D. D., Naming and Protection in Extendible Operating Systems, PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Project MAC TR-14Ø, Massachusetts Institute of Technology, Cambridge, MA, November 1974.

[27] Schroeder, M. D. and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings," CACM, XV, 3, March 1972, pp. 157-17Ø.

[28] Schroeder, M. D., Cooperation of Mutually Suspicious Subsystems in a Computer Utility, PhD thesis, Department of Electrical Engineering, Project MAC TR-1Ø4, Massachusetts Institute of Technology, Cambridge, MA September 1972.

[29] Walter, K. G., et al, Primitive Models for Computer Security, Case Western Reserve University, ESD-TR-74-117, HQ Electronic Systems Division, Hanscom AFB, MA, 23 January 1974.

[3Ø] Whitmore, J., et al, Design for Multics Security Enhancements, Honeywell Information Systems, Inc., ESD-TR-74-176, HQ Electronic Systems Division, Hanscom AFB, MA, December 1973.

[31] Wilkes, M. V. and Needham, R. M., The Cambridge CAP Computer and Its Operating System, Elsevier North Holland, New York, 1979.

[32] Woolf, A., General Motors, private communication, 1981.

[33] Wulf, W. A., Levin, R., and Harbison, S. P., HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, New York, 1981.