PROJECT MAC
Computer Systems Research Division

1 April 1975
Request for Comments No.72

A NEW PROTECTION SYSTEM.

by Philippe A. Janson.

Introduction.

The main goal of this paper is to describe a new protection system. However we would like to take advantage of this introduction to briefly define the original features of this design and compare it to other existing protection systems.

One of the design goals is to try to reconcile the two apparently incompatible views of an access control list mechanism and a capability mechanism so that the resulting system will not inherently have the limitations of either while at the same time providing most of the advantages of both.

A second design goal is to allow for the power of the protection system to be extended to objects of types not directly handled by the hardware and totally meaningless to the base level protection system. This feature is not new. However installing it on top of a system which handles both access control lists and capabilities together is not trivial. Because we want the set of capabilities for an object to reflect a

---

distribution of privileges as strict as that imposed by the access control list for that object, it seems that the access control lists and capabilities should be managed by the base level. However the very nature of an extended type object is that it is meaningless to the base level. Therefore one must answer the question: could the access control lists and capabilities for extended type objects be handled by the base level?

The third goal is called for by the second one. It is an approach to the problem of confinement. Andrews has defined the total confinement and message confinement concepts. We will propose a simple and straightforward solution to the problem of message confinement for protected subsystems in general and extended type managers in particular. The solution is based on very practical considerations which claim that it is easier to certify that a protected subsystem is "discreet" than to force its confinement at run time.

As will be seen further, the three goals proposed sofar allow a separation of privileges over extended type objects: the owner of the extended type object has full control over the object but does not have direct access to it; the extended type manager has direct access to the object but can exercise this right only when asked to by the owner of the object. What is achieved is not only an extendable protection system as defined

by Redell for instance but an extendable protection system about which certain properties can be certified to hold by simple inspection of the base level system. This is not to say that the base level of the system is the security kernel of the system. One may want to say more about the system than what the base level allows to say. However, at least all protection properties can be certified by inspection of only the base level and they can be further used to certify other pieces of the kernel.

The fourth design goal is to make the system as simple as possible in the broad sense. Many protection systems are hard to implement, hard to certify, hard to use and/or hard to understand. The proposed system attempts to achieve a reasonable power while remaining simple. No complex machinery is proposed, many tradeoffs between performance and sophistication are considered. Of course the success with which this fourth goal is achieved is very subjective and hard to evaluate. This is even more so since we will not attempt to evaluate it in this paper.

Access Control Lists and Capabilities.

We now turn our attention to the first goal of the system: the access control list based capability mechanism.

The advantages of an access control list system are usually described as its reviewing power, its facility to revoke

access rights and its power to control the propagation of access rights. The disadvantages of a pure access control list system are its cost and lack of flexibility: even with the help of associative memories and descriptor segments, verifying access to an object is slow and computation consuming because the access list must eventually be searched. The system lacks flexibility because an entry must be added to some sort of access list each time an access right is passed. Capability system partisans claim that the revocation and propagation control features are of minor interest because once access to an object is given out, the beneficiary B can save and pass around a copy of the object. We disagree with this claim because access to a copy is not as valuable as access to the object itself. Dynamic changes to the object cannot be observed unless there is a collusion between B and the observer; from the legal point of view, the mere fact of copying the object may one day become a violation of some copyright law for computer stored information; even if partial revocation (from one user) fails, complete revocation is possible and very often useful. To conclude we conjecture that an access control list mechanism is a good thing to have in a system.

The advantages of a capability mechanism are its flexibility and ease of use. Capabilities can be copied and passed without the need to add any name on any access list. Their use requires no searching of any list. Of course the disadvantages of the capability system are the advantages of the

access control list system. Because no access list exists in the former, control of any kind is much harder if not impossible. Redell has proposed an interesting scheme for introducing revocation in a capability system. But the scheme has several drawbacks. The scheme is based on a hierarchical dependancy of capabilities. Firstly, this makes the use of a revocable capability at the tenth level more expensive than the use of a capability for the same object at the first level. Secondly we do not like the intrinsic notion of a hierarchy of capabilities. Although this may fit the needs of many protection policies it cannot fit the needs of all of them.

Having presented our views on the virtues and drawbacks of the two systems, we can now explain what kind of features we would like to see in our system. Our goal is to have an access control list system and a capability system operating simultaneously, in parallel to exploit the advantages of both systems. The idea is to use the access control list mechanism as little as possible because of its cost but just enough to benefit of its power, and the capability mechanism as much as possible as long as it does not interfere with the control features of the access control list mechanism.

Considering the inherent conflict of purposes between an access control list system and a capability system, it is interesting to draw the line that separates them in this

protection system. Without going into any detail, one can roughly identify the areas where either aspect has taken precedence. Domain invocations is the area where the capability system is predominant. If domain X invokes domain Y and wants Y to operate on object A, X can give Y a capability for A eventhough Y may not be on the access control list for A. Inter process communication is the area where the access control list aspect is predominant. No access rights can flow from one process to another via the capability mechanism. In a last area of concern, namely shared c-lists, a compromise has been established whereby capabilities can be stored into shared segments to the extent no access control list entry is violated by the resulting sharing. This last constraint is enforced by way of comparing the "scope" (see later) of a capability with that of a c-list before the capability can be stored in the c-list. The choice of the areas where either aspect has taken precedence is motivated by the desire to maximize the use of the flexible capability mechanism while retaining most of the power of the access control list mechanism. Thus emphasis was put on the capability system for inter domain communication because inter domain communication requires frequent passage of temporary access rights. Emphasis was put on the access control list system for inter process communication because inter process communication requires less frequent communication of more static access rights.

The access control list mechanism of this system is

most similar to the system proposed by Schroeder. However the flexibility of the system is enhanced by an extendable capability mechanism allowing the user to pass access rights in a very convenient fashion. The capability mechanism of this protection system is most similar to the system proposed by Redell. However in addition to selective revocation, the system has access right reviewing and propagation control features thanks to the parallel access control list mechanism.

Extendibility and Confinement.

Having stated how access control lists and capabilities should work in the system, we quit the topic, leaving the implementation for a later paragraph. We now examine the second and third goals of our design.

Extendibility is no new concept. In some sense Multics is extendable. Directories are extended type objects as users have no direct access to them. Access to their representing segment is confined to the kernel. However this notion of extendibility is present only in the access control list mechanism and not in the (degenerate) capability mechanism. No SDW will ever contain sma access mode bits and a type field. A system like the one described by Redell is quite appealing. Capabilities for directories (in addition to capabilities for their representation) can exist in the world and may be used to

pass access rights to the directories. The advantages of the scheme is that the directory manager, when invoked to perform some task on a directory, immediately knows which directory to work on and what access the caller has to the directory. This involves no list search or other obscure hierarchy considerations as on Multics.

In trying to produce an extendable protection system, one is confronted with the choice between two possible approaches.

The "kernelists" approach will prefer to keep the management of access control lists and capabilities for extended type objects under the responsibility of the extended type manager because user created extended types are irrelevant to the kernel and because the kernel should not be concerned about the access control lists and capabilities formats of such extended type objects.

The "layerists" will claim that since the access control list and capability concepts exist for base level objects and since it only takes an extra type field to apply the same concepts to all extended type objects, it would be a waste of effort to force each extended type manager to reinvent concepts that are common to all of them. Instead those basic concepts should be supported by the base level.

Being unable to reconcile the kernelists and the layerists, we tried another approach. Is there any advantage to adopting one point of view rather than the other for the sake of the protection of the system? In response to this question we found at least one strong argument towards adopting the layerists' design.

There are two ways an extended type manager can cause damage to its customers. Firstly, it may contain a glitch or misinterpret the modes of a capability for an extended type object thereby causing damage to the representation of one object. Secondly it could violate inter user protection, for instance by matching one object with the representation of another one or by broadcasting information about objects it handles. The first kind of breach of protection involves only one object at a time while the second kind of threat involves two objects at least. Avoiding the first kind of trouble requires certifying the extended type manager to establish confidence it does the things it claims it does. Certification would certainly serve the purpose of defeating the second kind of threat. However we do not need that much. All we want to know is that the extended type manager is unable to access more than one object representation at a time (isolation property) and that it cannot leak information about objects it handles (confinement property). The former property is actually implied by the latter because one

way to leak information about one object would be to encode it into the representation of another object.

Jones designed a capability system with properties somewhat similar to what we just described. However her design would not work for an access control list system and it is based on a system wide rule that all working storage is temporary. This severely restricts the power of the system.

Unsealing.

For the time being, the only important conclusion is that access control lists and capabilities for extended type objects must be handled by the base level. If this were not the case, inter-user isolation and confinement would be hopeless as the extended type managers could fiddle directly with access control lists and capabilities for extended type objects and thus defeat any effort to isolate the objects from one another.

The real power left to the extended type managers lies in their ability to unseal a capability for an extended type object and to turn it into a capability for its representation. An extended type object is made up of a collection of component objects. For each extended type object , there exists a c-list containing the capabilities for its component objects. Thus the

c-list is called the representation of the extended type object (rep-c-list). Unsealing a capability for an extended type object yields a capability for its rep-c-list. The privilege of unsealing capabilities for a given extended type is materialized by a special capability called an extender for that type. Extenders are unique, not copiable and not passable. The owner of an extender is the manager for the extended type denoted by the extender. The protection system distinguishes between extended type managers and other protected subsystems. Extended type managers have the privilege to unseal capabilities but do not have the right to set access control list entries for any kind of object. Consequently, all objects can be classified into one of two categories: component objects which are owned and managed by extended type managers and have degenerate access control lists, and other objects with fully grown access control lists. Access control lists for component objects are degenerate in that from creation on, they automatically contain only one entry bearing the name of the extended type manager owning the component object and indicating a null access mode. This will prevent the extended type manager from ever getting a capability for a component object directly from the base level. The only way it can get a capability for a component object is by unsealing a capability for an extended type object and retrieving the desired capability from the now accessible rep-c-list of that object.

The unsealing feature in conjunction with control mode

bits (see later) which may make a capability not storable allow us to assert that an extended type manager is able to access only one object at a time.

Confinement.

As stated in the introduction, we are not concerned about the total confinement problem. We believe the only solution to this problem is a line by line auditing of the protected subsystem under concern. We hope to just solve the message confinement problem.

Solving the message confinement problem requires preventing the protected subsystem from statically storing and from passing information derived from its arguments. These conditions were stated by Andrews. However Andrews did propose no implementation, only an abstract solution to the problem.

Rotenberg proposed a system where the caller could state his desire to confine or not to confine a called protected subsystem. We do not believe that this is a correct approach. The caller does not know how the called protected subsystem works. This one may inherently not be amenable to confinement because it needs static storage. In this case auditing is necessary to guarantee trustworthiness, but no fast checking or enforced

confinement will work.

Therefore we have taken the approach that the caller should have no power to confine a protected subsystem. Instead it is the owner of the protected subsystem who should get a certificate of trustworthiness for his subsystem and the caller should rely on this certificate. Just like we distinguished between total confinement and message confinement, we now distinguish between "total discretion" and "message discretion" or simply discretion. Discretion means that the subsystem is guaranteed not to pass or store any information about its arguments via overt channels. A protected subsystem is guaranteed to be discreet if it is registered with the Discreet Protected Subsystems Administration (DPSA). This authority issues certificates of discretion on the basis of the following principles:

1. It is assumed that all discreet protected subsystems are frozen under control of the DPSA and listed in a table by uid.
2. It is assumed that all automatic storage vanishes after a protected subsystem invocation (no retention or residues).
3. It is imposed that all capabilities to be used by a confined protected subsystem be stored in non-writeable c-lists.
4. The certificate of confinement will be issued iff:
- gate capabilities in the c-lists correspond to gates into protected subsystems already registered with the DPSA;

_ segment capabilities in the c-lists show no writeable segments;
_ the c-lists contain no extended type object capabilities (this is because the DPSA does not know which extended type modes correspond to the writeability of the extended type object representation).

We believe a great majority of useful protected subsystems could be certified discreet by the simple method of the DPSA without going through the hassle of a manual auditing. Many extended type managers for instance (e.g. directory managers) do not need any internal static storage and need to call only other discreet protected subsystems (of which the protected subsystems of the kernel are examples since they are certified). For these protected subsystems at least, a certificate of discretion can be issued and the users can call them with a certain confidence (not that they are correct but that they will not leak information).

Basics about an implementation.

The fundamental principle of operation of the whole system to be presented is an associative relation between each access control list entry and the capabilities derived from this entry. This relation is based on identical uids in the entry and in the corresponding capabilities. The implementation of access control lists, capabilities, revocation and propagation control

is based on the associative relation. Hence it is very important and should be kept in mind throughout the rest of this paper.

The base level knows essentially about three kinds of objects: segments, extenders and all extended type objects in one whole (from the protection point of view only).

Segments are gate, procedure, data and capability containers. They correspond to areas of physical storage. Access modes are G (for gates), R, W, E and C (for c-lists).

Extenders have no physical storage realization. The owner of an extender is unique and unchangeable and has the power to unseal a capability for an extended type object to turn it into a capability for the rep-c-list of the object. No mode is defined for extenders.

Extended type objects of any type are handled uniformly by the base level. All the base level knows about them is it creates them, destroys them, updates access control lists for them and fabricates capabilities for them. Needless to say it does so in total ignorance of the semantics of the extended types. Thus the bit patterns encoding the modes must be given by the caller when setting an access control list. This does not put too much of a constraint on the caller provided each owner of an

extended type manager publishes the semantics of each mode bit that is relevant to his extended type.

Before we can describe by an example how the base level operates on an extended type object, we will briefly give the (logical) format of access control lists and capabilities.

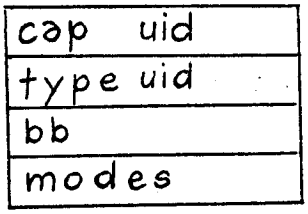
Consistent with the three classes of objects distinguished by the base level, we have three kinds of access control list entries which all fit into one common access control list entry template:

cap uid
user uid
group uid
pss uid
modes

The capability uid is the uid on which the associative relation between an access control list entry and its corresponding capabilities is based. The capability uid will be reproduced as the uid of all capabilities corresponding to this access control list entry. The user, group and protected subsystem uids are the "names" on the access control list entry. A uid of all zeroes is the Multics equivalent of a "*". The mode field contains the authorized access mode for this entry.

Access control list entries for segments map directly into the above template. Access control list entries for extenders have an empty mode field and the capability uid is the same as the extended type uid. Access control list entries for extended type objects map directly into the template.

Now we take a brief look at capabilities before we describe how they are derived from access control list entries and how they are used. There are also four kinds of capabilities fitting into the same template:



Given the format of access control list entries, the interpretation of capability fields should be clear. Notice that if the n-1 first bits of the type field are off the hardware will recognize a directly enforcable (base level) capability for a segment (0) or for a extender (1). The bb field is used only to specify the offset of a gate in a gate segment or to pass a base and a bound for subsegments as arguments. Capabilities issued by the base level always have a base of all zeroes and a bound of all ones.

With the above sketches of access control list entries and capabilities, we can validly discuss an example of their operation. We will describe the creation of an extended type object (a directory). This will give us a chance to describe operation on an extended type objects as well as on its base level representation.

The creation of a directory D is a two step operation. Firstly a null object of type directory is created by the base level. Secondly the null directory is expanded into the desired directory by the directory manager. The first step creates an empty rep-c-list C for the directory and sets up the protection attributes of D. The second step creates and initializes the segment D is composed of and which will later contain the information conceptually embedded in D.

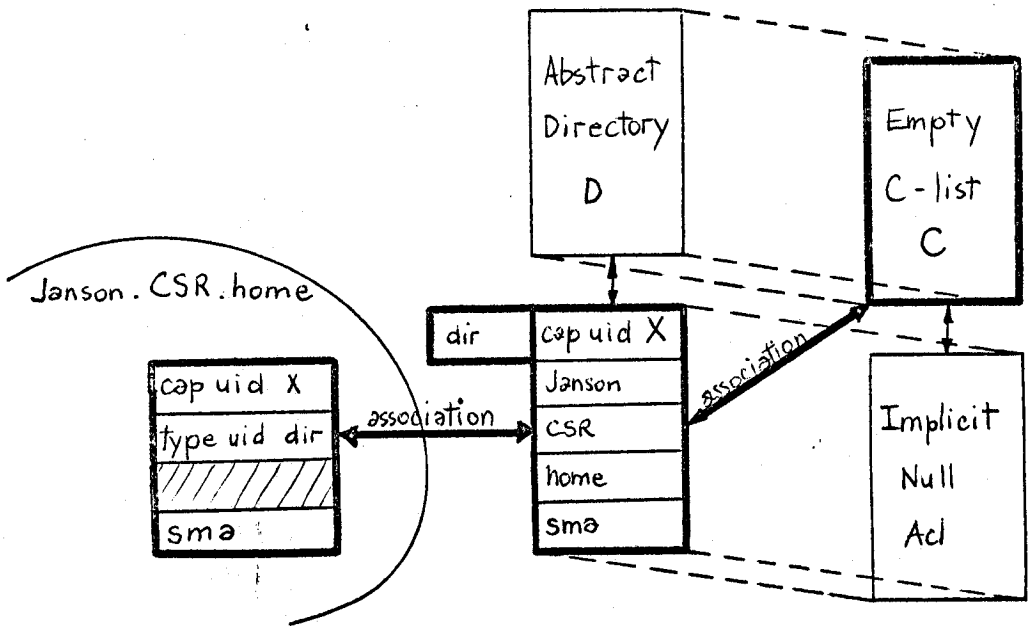
Step 1: Unlike a random c-list created by a user, C is a rep-c-list created solely for the purpose of holding capabilities for the component objects of D. The only protected subsystem which will ever use C is the directory manager. And even the directory manager may only reach C by unsealing a capability for D. Thus nobody in the world may ever be allowed to recieve a capability for C directly from the base level. Hence the access control list for C must be null. Actually, for reasons to be explained soon, let us assume that the access control list of C is physically non null but that it is made logically null

(inoperative) by adding to it a "directory" tag indicating that this is a rep-c-list for a directory.

Now consider D as an extended type object. As such it must have an access control list. The initial access control list will simply show that the creator of D has the requested access to D (e.g. Janson.CSR.home sma). Notice that while C was a base level object but did not require an explicit access control list, D is meaningless to the base level but needs an access control list. Also notice that the base level must establish a connection between C and D. Consequently, it seems very appealing and it will indeed prove very efficient to establish the missing connection by using the tagged access control list of C to implement the access control list of D. This feature will also be used later to implement revocation of access to D.

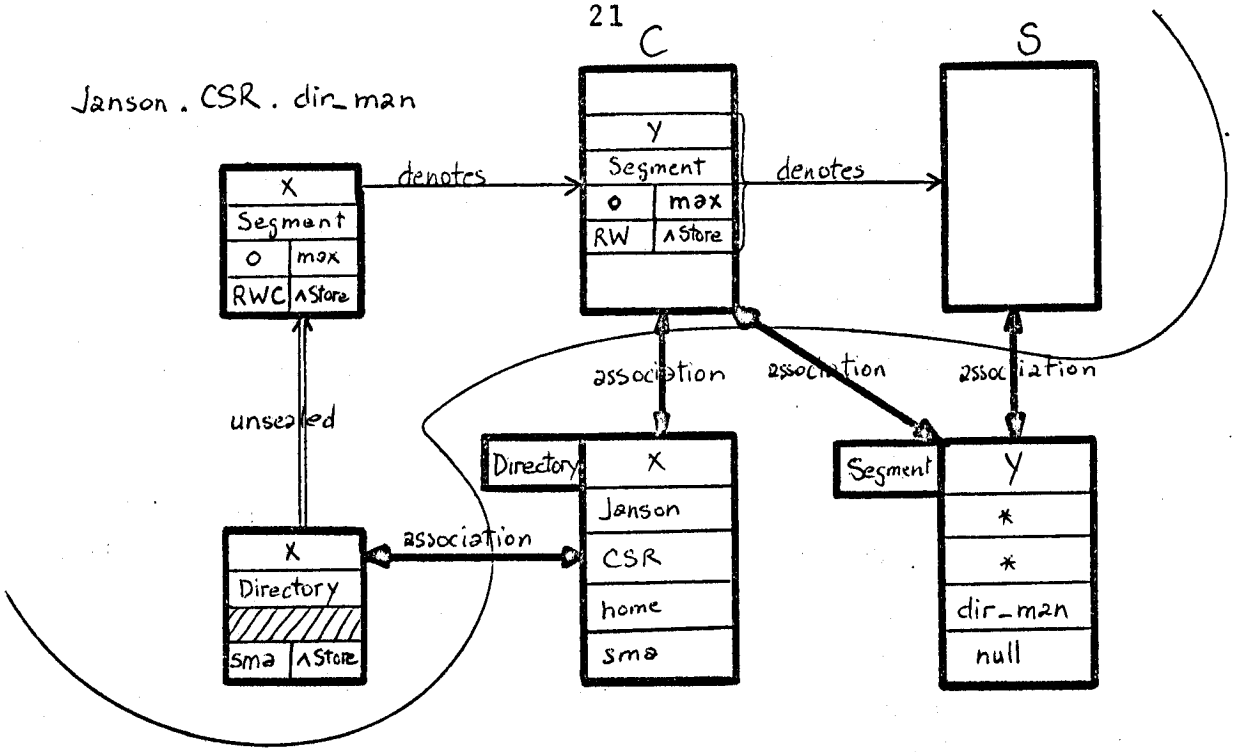
The first step will only be complete when the base level returns a capability for D to its creator. The state of D is now pictured as follows:

Step 2: Now, Janson.CSR.home must call the directory manager to initialize D. Having received a non storable (see later) capability for D, the directory manager unseals it and gets a non storable capability for C. Unsealing preserves the capability uid X since this is the only way to later go from the unsealed capability to C. The type field is set to zero (segment). The



mode field is set to RWC. The interpretation of the mode field sma is left to the discretion of the directory manager for the moment. This will be the topic of a later paper. Now the directory manager, which knows how to implement directories, requests the base level to create a segment S to implement D. As the request comes from an extended type manager, the base level sets a degenerate access control list on S (*.*.dir_man null). It then stores a capability for S into C. This capability can be used by the directory manager to access S. However the directory manager cannot save it (isolation property) because appropriate control mode bits are set in it. The state of D is now as follows:

Upon return from the call to the directory manager, capability Y remains in C but capabilities X vanish.



Revocation and Propagation Control.

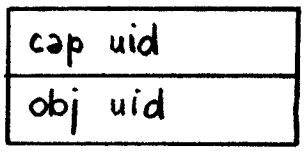
This is the most important part of the paper as far as the implementation is concerned. We will be talking first about the working of capabilities and then about their use. Everything to be said is of crucial importance to the correct implementation of capabilities, access control lists, extendibility, revocation and propagation control, and confinement.

First we consider the case of capabilities for extended type objects. A capability for an extended type object cannot be interpreted directly by the hardware. It can only be passed to an extended type manager which has the power to unseal it (i.e. set the type field to "segment" and the mode bits to RWC) to obtain a capability for the extended type object rep-c-list. As access control list entries for extended type objects are implemented by

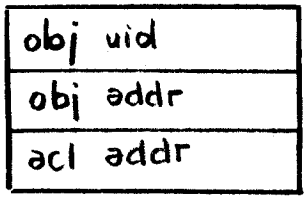
tagged access control list entries for the rep-c-lists of the extended type objects, revocation of access to extended type objects works by way of revocation of access to the object representations and thus boils down to what is stated in the following paragraphs.

Secondly, we consider the case of capabilities for segments and gates. These are directly interpretable by the hardware as stated earlier. Two system wide maps are of interest for them.

The Cmap contains one entry per capability uid in the world. The entries read like:



The Omap contains one entry for each object uid in the world. The entries read like:



The maps are implemented as segments and reside mainly on secondary storage. The Cmap is similar in function to Redell's system map.

Corresponding to the two maps are two resident mini-maps, the cmap and the omap which are condensed versions of the most recently used main map entries.

We do not want to go here into the details of how each map or mini-map must be known, wired, paged and maintained for proper operation and consistency.

Below the mini-maps are two distinct fast associative memories, the cam and the oam, to contain the most recently used of the mini-maps entries.

When a capability is presented to the hardware, its uid is used to get the segment uid out of the cam/cmap/Cmap. This uid is in turn used to get the segment address out of the oam/omap/Omap. The address is finally used to get at the segment itself.

What we have achieved by adding one level of indirection to the Multics way of doing business is what Redell achieved by his system map: a total separation between addressing

and protection to ease revocation.

We want revocation to be as powerful as on a pure access control list system. For the time being, we assume the desirable functionality of an access control list system to be that exhibited by Multics. We propose to develop a better access control list system in a later paper. Thus revocation of access to "foo" for (*.CSR.home rw) can take the following forms:

delete acle	set-acl(foo,null,*.CSR.home)
delete mode	set-acl(foo,r ,*.CSR.home)
delete a set	set-acl(foo,null,Janson.CSR.home)

Case 1: entry deletion. This is the simplest case. All it takes is to delete the access control list entry and any entry in the Cmap/cmap/cam which bears the same capability uid as the access control list entry. Thus any capability which may still have this uid will no more work as the connection between the capability and the object it denotes has been ruptured in the Cmap.

Case 2: mode restriction. In this case we want capabilities owned by *.CSR.home to stop working for W but they should still work for R. If we simply proceeded as in case 1 and replaced the deleted entry by a brand new one, the functionality would be correct, but users would find themselves with capabilities disabled even for the R operation. In addition there would be no way to "fix" the disabled capabilities as the connection between

them and the object they used to denote would be lost.

To solve this problem, the idea is to disable the capabilities without breaking the connection to the object they denote so that a standard recovery routine may be invoked to fix the disabled capabilities without the user even being aware of it. This can be done by splitting the uid of an access control list entry (and of its associated capabilities) into two parts: the true-uid which is indeed unique and the instance-uid which is unique only for any single access control list entry. The inst-uid is initially zero. When an access mode is revoked, the access control list entry is first modified, the inst-uid is then incremented by one, the entries in the Cmap/cmap/cam corresponding to the true-uid||inst-uid are set to true-uid||inst-uid+1.

Consequently, the old capabilities bearing a uid of true-uid||inst-uid will no more work as such a global uid does no more exist in the Cmap. However, by using the true-uid which has not changed, a recovery routine can retrieve the now modified access control list entry and recompute the correct access mode field. It then can increment the inst-uid in the capability thereby enabling it again but only for R. A true access violation noticed by the user would now occur only if he tries to do a W.

Case 3: set substraction. This case is just a generalisation of

case 2. On each set-acl operation, the set-acl primitive must sort the new access control list entry into the existing access control list. In doing so it should go down the list past the new entry, to see if any set substraction has occured as a result of the modification of the list. If so, the base level must proceed as in case 2 for each entry affected by a set substraction. The inst-uid counter feature will work for each such entry as it worked for the single entry of case 2.

We finally turn our attention to the control of the propagation of capabilities which will obviously lead us to the discretion problem. Access control lists say exactly who can access what in what way. Our system claims that the distribution of an access right via capabilities must at any time (except during a domain invocation) be constrained by what the controlling access list says. To enforce this we introduce in the mode field of <u>any</u> capability four fixed-semantics control bits: S, U, G and E.

U reflects the fact that the capability corresponds to an access control list entry where a user name was specified. G means that the access control list entry specified a group or project. E reflects the fact that the access control list entry specified a protection environment.

U, G and E control the scope of a capability in the

user.project.environment space. If all are on, the capability was derived from an access control list entry of the form "user.group.envt".

The S control bit specifies whether a capability may be stored in a c-list.

We have identified exactly three ways a capability could go from one domain to another, potentially violating what the original access control list entry says. The capability could be passed via ipc, could be stored in a shared c-list or could be passed as an argument.

We rule out the operation of passing a capability via ipc as being illegal in an access control list based system. This is where the access control list aspect takes precedence.

Capabilities can be stored into shared c-lists if S is on, but there are certain restrictions. A capability with a given scope (as defined by U, G and E) can be stored only into a c-list with equal or smaller scope (set theoretically). That is if capability X is of scope UGE, it cannot be stored into a c-list denoted by a capability of scope E because the c-list is accessible in other instances of the protected subsystem in other processes and capability X should never go there given the meaning of U, G and E.

Notice that three bits are used to actually mean access control list entry names like user.group.envt. At any point in time, the meaning of U and E is unambiguous. U means the currently active user and E means the current domain of execution (because capabilities for other users in other domains could not be in use now!). However the meaning of G is not clear as a user may want to be considered part of several groups at a time. A solution to this ambiguity is already worked out but will be presented in a later paper.

Finally, capabilities may be passed as arguments. However one problem remains. When a capability is passed, it should not be stored in any kind of c-list of the callee as it does not statically belong to the callee according to the access control list. For that purpose when such a capability is passed, S is turned off thereby making U, G and E meaningless and useless and confining the capability to one invocation of the current protected subsystem in the current process. It can be loaded only in registers or in argument lists which by definition go away after the invocation is completed. This is where the capability aspect takes precedence over the access control list aspect.

One more comment is in order about the use of control mode bits with respect to unsealing. The unseal operation not

only sets the type field to "segment" and the normal mode bits to RWC, but also, regardless to its current value, turns off S to prevent the extended type manager from saving the unsealed capability which would enable it to later access more than one object at a time. Notice that if this extended type manager is registered as discreet with the DPSA, then however passable this capability is, it cannot be passed to indiscreet protected subsystems by definition.

Conclusion.

This paper has attempted to outline the design of a new protection system. Features of the system are: a parallel implementation of access control lists and capabilities with the resulting tight access control and flexibility, the extendibility applying to both the access control lists and the capabilities, the message discretion of protected subsystems.

References.

Andrews G.R.
COPS - A Protection Mechanism for Computer Systems.
University of Washington - TR-74-07-12 - 1974.

Fabry R.S.
The Case for Capability-Based Computers.
CACM 17, 7 - P403-412 - 1974.

Jones A.K.
Protection in Programmed Systems.
CMU, Dept. of Comp. Sc. - 1973.

Lampson B.W.
A Note on the Confinement Problem.
CACM 16, 10 - P613-615 - 1973.

Lampson B.W.
Redundancy and Robustness in Memory Protection.
Proc. IFIP Cong. - 1974.

Redell D.D.
Naming and Protection in Extendible Operating Systems.
UC Berkeley - 1974.

Rotenberg L.J.
Making Computers Keep Secrets.
MIT Projet MAC - TR-115 - 1974.

Schroeder M.D.
Cooperation of Mutually Suspicious Subsystems
in a Computer Utility.
MIT Project MAC - TR-104 - 1972.