Computer Systems Research Division Request for Comments No. 73

IMPLEMENTING EXTENDED TYPE OBJECTS IN AN ACL-BASED PROTECTION SYSTEM

by Douglas H. Hunt

To the author's knowledge, existent general-purpose computer systems which support the implementation of extended type objects are capability-based systems. This RFC introduces a design for supporting extended type objects in an access control list system. These results, which are a part of my doctoral thesis research, should contribute both to basic research in protection and to the certification of operating systems.

Introduction

This report describes some research results which are a part of my doctoral thesis. The report is an experiment in "top down" document writing; as such, it tends to mention most of the subject matter which I consider to be part of my thesis. At the same time, it is my intention that many of the topics included in this report be described only briefly.

This research has received inspiration from two directions. First, it is motivated by related basic research in the area of protection. The results presented here provide further insight into the similarities and differences of capability and access control list protection mechanisms. This report presents a method for implementing extended type objects in a computer system which has an access control list protection mechanism. Previously described systems which support extended type objects make use of capability-based protection. The advantages to be gained by providing an extendible access control list system are in turn the advantages of an access control list system relative to a capability system. These relative merits have been described in the literature [ref. Saltzer and Schroeder IEEE paper]. Secondly, this research is motivated by a number of related projects in the area of operating system certification. Certification researchers take the point of view that if a system is expected to conform to a specification, then it is not necessary to prove or even observe all the modules which comprise the system. Rather, there ought to exist a relatively small

collection of modules, often called the kernel, which must be shown to operate correctly. Given that the kernel operates correctly, the certifiers can deduce that the entire system adheres to the stated policy because the kernel, with the aid of the system hardware environment, constrains all other modules in such a way that they cannot violate the policy. The certification approach seems to be a viable one for establishing confidence that the protection policy of an operating system cannot be circumvented. The results described here are a direct contribution to certification research, since they indicate how a common mechanism (which would be part of the kernel) can be used to protect a large class of objects in an operating system.

The base level system

To provide a context within which the research results may be described, this section is devoted to a description of a portion of the supervisor of a hypothetical system, which shall be called the "base level." The purpose of the base level is to provide a virtual environment which will support the mechanisms to be described in later sections. In this section, only slight attention is paid to the base level implementation details.

The abstract machine provided by the base level maintains a set of objects, each of which is denoted by a unique identifier (UID). Each object can be characterized by a set of operations which is applicable to it. The set of all objects can be divided into equivalence classes, such that all objects in an equivalence class are subject to the same set of operations. Each

equivalence class is known as a type. The base level maintains three types, which are segment, protected subsystem, and process. An object of type segment may be subjected to operations such as "read" and "write." An object of type protected subsystem may be subjected to the "call" operation, and an object of type process may be subject to operations such as "block" and "wakeup."

The base level object of type "protected subsystem" (PS) is used to implement a tamperproof collection of procedures and data. The procedures and data, in turn, (presumably) provide some abstraction or service to users of the system. A PS can only be invoked at one of its defined entry points. Although a PS can be viewed as a set of base level segment objects, the set itself is defined to be a distinct base level object, of type protected subsystem. A distinguished member of the set, usually called a gate, is a special transfer vector segment which directs calls to the defined entry points within the PS. The UID of the gate into a PS also serves as the UID of the entire PS.

An agent which potentially has the ability to perform an operation on an object is known as a principal. The base level implements a principal in terms of two of its own object types: process and PS. To offer an intuitive justification of why a principal is represented in terms of the base level types "process" and "protected subsystem," consider the following example: John Doe desires that the telephone company repair his phone. John Doe is the only legitimate user of the phone; if Fred Smith requests that the telephone company modify Doe's

phone, the request should be refused. On the other hand, the telephone company is the only legitimate maintainer of the phone. Doe should not be able to authorize Fly-by-Night Phone Hackers to make the modifications to the phone. In the system being described, the process corresponds to the user of an object, and the PS corresponds to the protected subsystem which is the supplier of an object. Thus, within the system, a process executing in a particular PS represents a source of authority.

A principal is able to perform an operation on an object only if it has been granted the privilege to do so. There is a privilege flag associated with each operation defined on an object. Thus, it makes sense to refer to "read" or "write" privileges for a segment, but not to "wakeup" privileges. The collection of all privileges on objects which have been granted to a principal is called a domain. For example, the following domain may correspond to "principal 241:"

segment 22	read, write
PS 53	call
process 61	block, wakeup
segment 157	read
process 34	wakeup

There is a one-to-one correspondence between domains and principals. Thus it makes sense to refer to the domain corresponding to "process X, PS Y." If "process X, PS Y" calls "process X, PS Z" then the domain will change at the time that PS Z is called. In general, there is not a one-to-one

correspondence between protected subsystems and domains, since the domain corresponding to "process X, PS Y" is potentially different than the domain corresponding to "process W, PS Y."

The relationship between "protected subsystem," "process,"

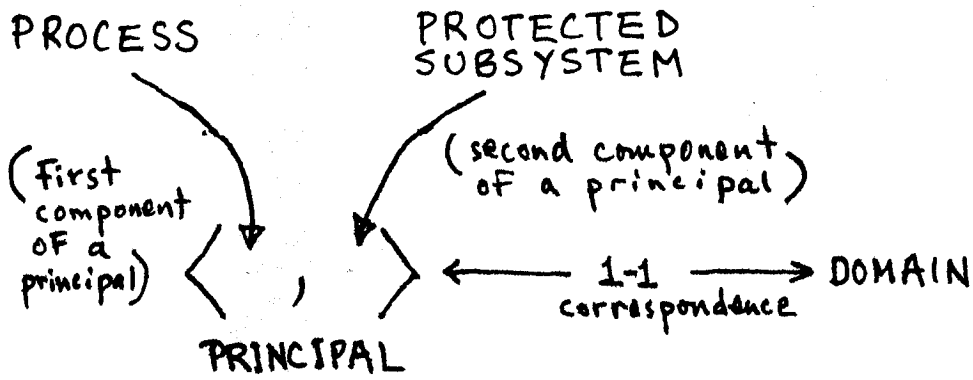


Figure I

"principal," and "domain" is illustrated in Figure I. A principal is the agent in the system which may exercise privileges to access objects. There is a one-to-one correspondence between principals and ordered pairs with "process" as the first component and "protected subsystem" as the second component. Therefore, a process executing in a protected subsystem is the exerciser of access privileges in the system. There is also a one-to-one correspondence between principals and domains. A domain is the set of privleges granted to a principal.

Associated with each object is an access control list (ACL). The ACL contains a set of entries, each of which in turn contains a principal and a set of privileges, or modes. Since a principal is a process-PS pair, it can be described by the UID of a process and the UID of a PS. The modes could be described by character

strings such as "read" and "write," but would actually be represented more compactly as a bit-encoding. The ACL for an object is a single repository displaying all the principals which have privileges for the object, as well as the particular privileges granted to each principal. It is worth pointing out the relationship between domains and access control lists. To relate these two notions, consider the access matrix as defined by Lampson [ref. Lampson]. An entry in the matrix displays the set of privileges for a given object which are available to a given principal. Of course, an entry may contain the empty set. The entry in row i and column j can be defined as shown:

entry(i,j) = the privileges of principal i for object j.

Thus, each row in the access matrix represents the domain for some principal, and each column represents the access control list for some object. The base level system provides an ACL for each object; however it does not provide, in a single repository, the domain for every principal. To determine the domain for a given principal would require searching every access control list.

The description of ACLs which has been provided so far is incomplete, since it has not covered dynamic changes of access to an object. It must be possible, for example, to grant additional privileges for an object to a principal, or to revoke privileges. An access control list is a declaration of a policy, indicating the privileges which some "guardian" of an object sees fit to grant to "consumers" of the object. The discussion up to this

point has not established the identity of the guardian. It would be possible to add new modes to every ACL, such as "add_modes" or "revoke_modes." These new modes would refer to the acl itself, rather than to the associated object. Then the set of principals which has either "add_modes" or "revoke_modes" access might be called the guardian of the object. There is an issue, though, as to whether these two new modes apply to themselves. Regardless of which choice is taken, there seem to be undesirable implications. An escape mechanism proposed by Rotenberg [ref. Rotenberg] is the "office." The office embodies a policy specification. For instance, there may be a policy that the ACL for an object should be changed only if the majority of some managing board agrees. This and other policies would be cumbersome to implement in an access control list; however they can be implemented by a procedure. Thus, an office may be implemented as a protected subsystem. Each ACL can then contain the UID of a PS, which corresponds to the office for the object. In practice, a large number of objects will be subordinate to the same office. With the addition of an office, there is a division of labor: "routine" access control decisions, such as checking to see if a principal has "read" mode on a segment, involve only the ACL; "higher-level" access control policy actions are carried out by offices. Since the base-level objects which implement an office have ACLs themselves, there is the potential of a recursive disaster. To avoid such a disaster, the ACLs which apply to offices can be changed only via a software

reconfiguration operation, initiated from the system operations console. Thus, as is the case with any system, the effectiveness of the logical protection mechanism ultimately depends upon physical security.

Each reference by a principal to an object necessitates (1) a check to find out if the principal has the proper access privilege, and (2) a reference to the representation (rep) of the object. Whereas the access control information for each object is expressed in the same ACL format, the representations for objects of distinct types may be quite different. The rep of segment and PS objects, for example, consists of a page map, while the rep of a process object consists of a set of tables (segments) which the base level can interpret. Corresponding to each existent object is an entry in a system-wide table called

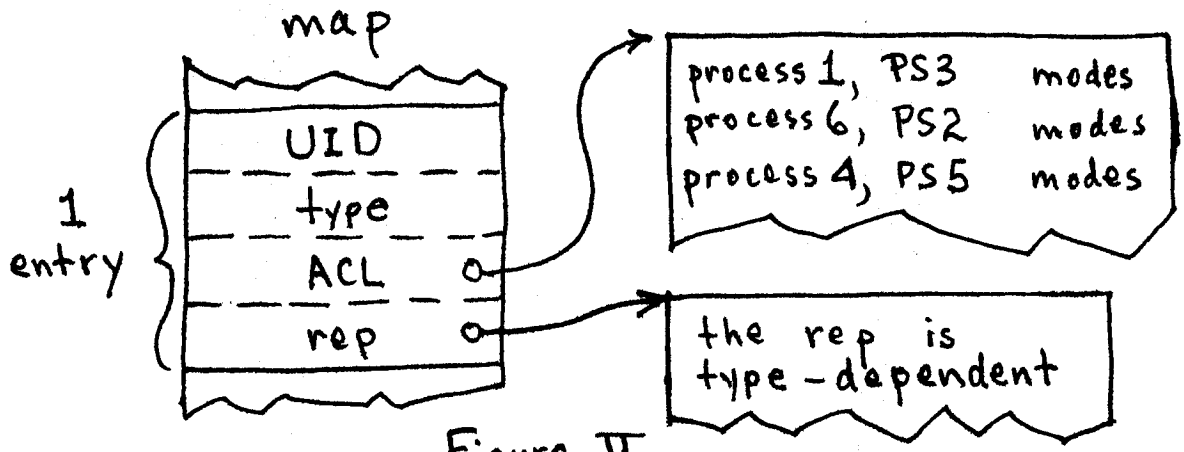


Figure II

the map. Each map entry has four fields, as shown in Figure II. The first field is the UID of the object, generated at the time the object is created. The second field is the type. More will be specified about this field later; for now it is sufficient to provide three different values, corresponding to objects of type

segment, PS, and process. The third field is a descriptor for the ACL. Since an ACL will be implemented using segments, the third field will contain a segment UID. The fourth field is a descriptor for the rep. In the case of the base-level objects, the descriptor indicates either a page map or an entry in a system-wide process table.

Conceptually, each operation on an object requires a reference to the map. For the sake of efficiency, however, each principal actually references its own local map, which is a particular subset of the (global) map. Entries in a local map contain the same information as the corresponding global map entries, except that the ACL descriptor is replaced by a set of modes. (1) A local map is created whenever a PS object is called for the first time in a process; a local map is destroyed when the process itself is destroyed. Whenever a principal references an object for which there does not correspond a local map entry, the global map is consulted. If the UID supplied by the principal is found in the global map, and if the ACL corresponding to that UID indicates that the principal has some privilege to reference the object, then a new entry is added to the principal's local map. (2) It should be emphasized that the

---

(1) The decision to put the modes directly in a local map entry, rather than to indirect through the local map entry to a central set of modes, is based on a trade-off involving efficiency of referencing an object and efficiency of revocation.

(2) The processor supporting this system contains a "principal register" which has two fields. The first field holds the UID of the current process, and the second field holds the UID of the

local map is an optimization; it does not add anything conceptually to the base level model since the services provided by the base level can be described in terms of the system-wide global map.

The global map and local maps are implemented using segment objects. There is a subsystem, called the map subsystem, which knows how to create and delete map entries, build local map entries, and how to manipulate ACLs. The map subsystem is protected, and executes in its own domain called the "map domain." It is possible to associate the map subsystem with a particular domain only because the map subsystem will be given the same privileges in all processes. The only domain which has the privileges to manipulate ACLS and maps is the map domain. Since the base level objects depend on the map subsystem, it is appropriate to classify the map subsystem as part of the base level. However, throughout this paper, the map subsystem is often referred to directly. The map subsystem has no special knowledge of the representation of any object; it merely provides a field in all map entries which contains a descriptor for the representation. The fact that the map subsystem knows about ACLs but not about representations is based upon the observation that although the semantics -- and the representations -- of distinct types may have nothing in common, there is a common mechanism for providing access control.

---

current PS. ACL searches compare principals in the ACL with the contents of the principal register.

The Implications of Protecting Extended Type Objects

The base level system, as described thus far, supports exactly three types: segment, PS, and process. For each of these types a principal may perform (subject to access controls) a "create object" operation. For example, a principal may create a new object of type segment. If a principal may also create new types -- via a "create type" operation -- then the base level is said to support dynamic type creation [ref. Jones]. Any type created by a "create type" operation will be called an extended type. As will become clear, most of the mechanism necessary to support dynamic type creation already exists in the map subsystem.

The Hydra system [ref. Wulf et.al.] provides a uniform mechanism for dynamic type creation, which stems from the view that types are objects. There is a root object, called the "type type" object. To create a new type, a principal performs a "create object" operation on the type type object. To create a new object (which is not a type) a principal performs a "create object" operation on a type object. Thus the same "create object" operation can be used to create both new types, and also new instances of an already-defined type. For the purposes of this paper, however, the two terms introduced in the preceeding paragraph will be used: "create type" operations create a new type; whereas "create object" operations create a new object which is not a type.

As mentioned earlier, a type is characterized by a set of

operations. The type "message queue," for example, may be characterized by operations such as "enqueue," "dequeue," and "list_entries." Since the operations on an extended type object (ETO) can be arbitrary, they are implemented by procedures. Thus an extended type object actually consists of a set of procedures which operate on all objects of a given type, together with some data area which serves as the representation of the particular object. Two distinct ETOs would make use of the common procedures, but would require different representations. The representation of an ETO is constrained to be some other object or objects, but not necessarily base level objects. Therefore, the representation of an ETO may be some other ETO (or ETOs); but the tree of representations is rooted in base-level objects.

In order that the operations on an ETO be well-defined, the associated procedures should be callable only at certain points. These procedures and related data segments will form a protected subsystem. Consequently, for each extended type in the system, there corresponds a PS object. To access any object of a given extended type, the same PS is called. The identifier of the particular object is passed as a parameter to the extended type manager (ETM) subsystem. (1) The ETM then interrogates the map subsystem, to see if the principal of the caller is on the ACL of the ETO which has been passed as a parameter. If the principal

---

(1) Another possible way to support ETOs is to associate a PS with every object, rather than with every type. Such an approach is similar to the "domain capabilities" approach for supporting ETOs described by Redell [ref. Redell].

appears on the ACL, and has the proper privileges, the ETM carries out the requested operation.

Although there is one PS object corresponding to each extended type, we do not propose at this time that there be one extended type corresponding to each PS object. If such a stipulation were made, it would certainly tend to enforce the view that an operating system is a manager of a collection of abstract resources; i.e. all requests of an operating system are really operations on ETOs. Although there may be merit in pursuing such a view, such an analysis is outside the scope of this paper. In this paper, we assume that the set of types maps "1-to-1 and into" the set of PS objects.

Every ETM relies on the map subsystem to check access to and to provide the representation for ETOs. For any ETO, the type field in its map entry designates the PS which is the corresponding ETM. The rep field designates one or more other objects, which may be extended or base level. We have sketched how the base level may be able to support an ETO while not having any knowledge of its semantics. It is now appropriate to ask what it means to protect ETOs (some of which may be part of the representation of other ETOs) using access control lists. For the purposes of the following analysis, we define a rep-object to be an object (either base level or extended) which is part of the rep of some other object. By introducing extended types, we are forced to consider the meaning of ACLs on rep-objects. These considerations will show that there is a trade-off between

certain protection goals and programming generality.

As an aid in describing the application of ACLs to ETOs, we introduce some new terminology. These terms correspond to relationships between principals and objects. First, a principal may be the guardian of an object. As mentioned previously, the guardian of an object is responsible for specifying the policy for control of access to the object. Second, a principal may be a consumer of an object. A consumer uses the object; that is it performs operations, subject to privileges appearing on the ACL, on the object. The consumer treats the object as a black box and relies on either an ETM or the base level to carry out operations defined on the object. Third, a principal may be the supplier of an object. The supplier of an object is the subsystem which "knows about the inside of the black box." For base level objects, the supplier is some protected subsystem in the base level. For other objects, the supplier is an ETM. Fourth, a principal may be the sponsor of an object. A sponsor is the authority in the system that pays for a given object. Like a guardian, a sponsor would be implemented as an office. The notion of a sponsor has been implemented in at least one system [ref. Project SUE]. The roles of sponsor and guardian have been described as distinct here, even though there may be some interaction. For example, a sponsor may require some form of "delete" access on an object in order to discard unused objects. Using the terminology introduced here, we can describe an ETM as the supplier of some set of ETOs, and the consumer of some other

set of (possibly extended) objects.

To explore protection goals which may pertain to a system that supports dynamic type creation, we choose the following modus operandi:

1) Propose a "reasonable" protection policy;
2) Determine if there is a way, in the hypothetical system, to implement the proposed policy;
3) Indicate whether or not there is a capability-based system which can support the proposed policy;
4) Look for deficiencies in the proposed policy, and improve on it;
5) go to step 2 (1)

Following this procedure will generate a sequence of protection policies. A particluar policy in the sequence may be "stronger" than the preceeding one, in the sense that (1) the "degrees of freedom" a principal may have (i.e. the states of the system which are considered admissible under the policy) are fewer; and consequently (2) the set of assertions which must be satisfied for a principal to access an object is larger.

The first policy to be explored, which we shall call the "black box" policy, requires that the procedures and data bases of an ETM should be "private" -- that is, they should not be accessible from other protected subsystems. This policy includes rep-objects belonging to the ETM as part the "data bases" of the ETM. This policy, if applied to the phone company example, would

---

(1) My apologies to the structured programmers!

require that

1) Only the telephone company has access to the "rep" of a telephone, and

2) No "outsiders" can affect the method the telephone company uses for repairing telephones.

The black box policy can be re-stated in the context of the proposed system: The only principal which should be able to access rep-objects for objects of type X is the ETM for objects of type X, executing in any process. There is the additional requirement that the ETM be tamperproof, but this requirement is met since the ETM is implemented as a PS. There is a straightforward way to express this policy in the ACLs of the rep-objects. Each ACL should have one entry of the following form:

*.ETM mode1, mode2, mode3, . . .

The first field in the principal, "*", means "match all processes." (1) The second field in the principal is actually the UID of the PS which is the extended type manager. The modes (which would actually be encoded as bit strings) indicate the access privileges needed by the ETM. The ACLs on components of the ETM such as procedures and own storage would also contain the principal shown above.

There is also a capability-based system which supports ETOs and satisfies this first protection policy: the system proposed

---

(1) This "*-convention" is currently used in the Multics system. [ref. Saltzer]

by Redell [ref. Redell]. In Redell's system, the ETM domain (or layer) is able -- by virtue of posessing an "extender" -- to seal capabilities for the rep of an extended object. These sealed capabilities can then be passed outside of the ETM layer without fear of compromising the integrity of the extended object. Other domains cannot unseal these sealed capabilities, since they do not posess the proper extender. As a result, only the ETM layer may access its rep-objects, after unsealing the corresponding capabilities. The function of sealing, in this example, is to prevent non-ETM domains from accessing rep-objects. The ETM, which is assumed trustworthy, has full control over sealing and unsealing capabilities for its rep-objects.

According to the methodology stated above, we now consider possible improvements to the black box policy. In terms of the phone company example, only the phone company may access the representation of the telephone black box. In particular, the representation of a telephone may be accessed by the supplier of telephones, whether or not any consumer of the particular telephone has authorized the access. One may wish to insure that the supplier accesses an object only at the consumer's request. Such a policy will be called "exclusion", since the consumers of an object have exclusive rights to authorize accesses to the object. The exclusion policy is useful in that it provides a greater degree of accountability than the black box policy. In particular, if the state of an object has changed, then (if the exclusion policy is enforced) only those principals which appear

on the ACL of the object could have effected the change. The exclusion policy does not prohibit leakage of information about any object, since an ETM may leak information about an object even while performing an authorized access.

One approach for implementing the exclusion policy would be to verify certain properties of an ETM. The goal would be to show that whether or not the ETM "does the right thing" to an ETO, at least the ETO would be accessed only upon the request of a consumer. That is, the ETM would never attempt to access any of its rep-objects, except for those which are part of the ETO (or ETOs) specified in its input argument list. However, to prove such a property (or any property of a program) is, as mentioned previously, outside the scope of this research. Rather, the approach taken here is to constrain programs, so that regardless of their behavior, a particular protection policy can be maintained.

Another approach for constraining the ETM to obey the exclusion policy would be to add the principals of the consumers of an ETO to the ACLs of all rep-objects in the rep tree, as illustrated in Figure III. The semantics of ACLs, as described up to this point, would have to be extended: on each reference to an object, two principals would be matched against the ACL. First, the ACL would be searched to locate an occurrence of the principal corresponding to the current domain (i.e. the principal of the supplier). Second, the ACL would be searched to locate an occurrence of the principal on whose behalf the operation is

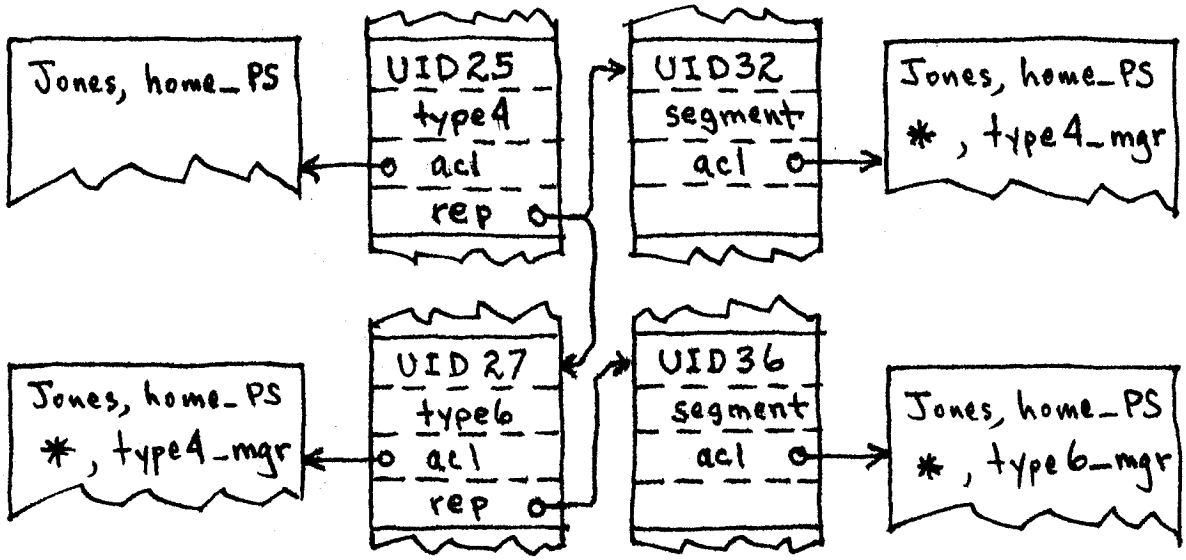


Figure III

being performed (i.e. the principal of the consumer). In order for this second search to occur, the principal of the consumer of the root of the rep tree must be made available to the ETMs which manage rep-objects in the tree. A reference to a rep-object would be allowed to take place only if both principals are found in the ACL. This approach exhibits a number of disadvantages, however. First, it is not clear what the consumer's modes of access to an arbitrary rep-object in the rep tree should be. In general, knowledge of the modes of access to a rep-object which are necessary to support a given mode of access to the parent ETO resides only in the corresponding ETM. Second, this approach does not interact well with the dynamic aspects of access control. A change in the ACL of an ETO would necessarily propagate down the entire rep tree. Consequently, this approach will not be explored further.

The chosen implementation of the exclusion policy depends on

a form of sealing. Every rep-object is defined to be either sealed or unsealed. If a rep-object is sealed, then it is not possible for a principal to access it, even if the ACL indicates that the access is allowed. An ETM may attempt to unseal a rep-object by invoking the UNSEAL primitive:

UNSEAL(parent_ETO, rep-object).

The UNSEAL operation will succeed only if

1) The parent ETO is passed to the ETM as a parameter;

2) the principal of the consumer of the ETO appears on the ACL with non-null access, and

3) the rep-object which the ETM wishes to unseal is actually part of the rep of the ETO specified in the UNSEAL operation.

The scope of the unsealing is for the current activation of the ETM. Thus, when the ETM returns to its caller, all rep-objects which it may have unsealed revert to the (default) sealed state.

The notion of sealing is not sufficient to support the exclusion policy. It must be supplemented by the following restrictions:

1) No implicit reps are allowed.

The mere fact that there exists a rep field in the map entry for each object does not constrain an ETM from using other data bases to store state information for an ETO. We make the following distinction: the "official" rep of an ETO (which is always designated by the rep field in the map entry) is called the explicit rep. Any other data base in which the ETM maintains state information for an ETO is

called the implicit rep. If the ETM is constrained to adhere to the exclusion policy, then it will not be allowed to make use of implicit reps. The danger in using an implicit rep -- i.e. an object which is not a rep-object and consequently not able to be sealed -- is twofold. First, other principals may have access to it and may therefore effectively access an ETO even if the ETM is not invoked. Second, the ETM itself would be able, during any activation, to access the implicit rep. As a result, the ETM may effectively access an ETO without authorization from a consumer. Sufficient constraints can be placed on an ETM to insure that it cannot be using implicit reps. For example, the ETM may not use a writeable data base which is readable by other principals, nor may it write into a data base which is readable by other activations of itself in a different process. These restrictions can be enforced by appropriate ACL entries on data segments used by the ETM. The "no implicit reps" restriction also rules out storing information in another ETO, as well as calling another protected subsystem which might make use of an implicit rep.

2) No non-private reps are allowed.

This restriction is related to the previous one. Even in the case of rep-objects, it must be true that no other principals have access. Therefore it must not be possible either (1) to change ACLs on rep-objects, or (2) to specify that an already-existing object become a rep-object. Consequently, only newly-created objects may be designated as part of the

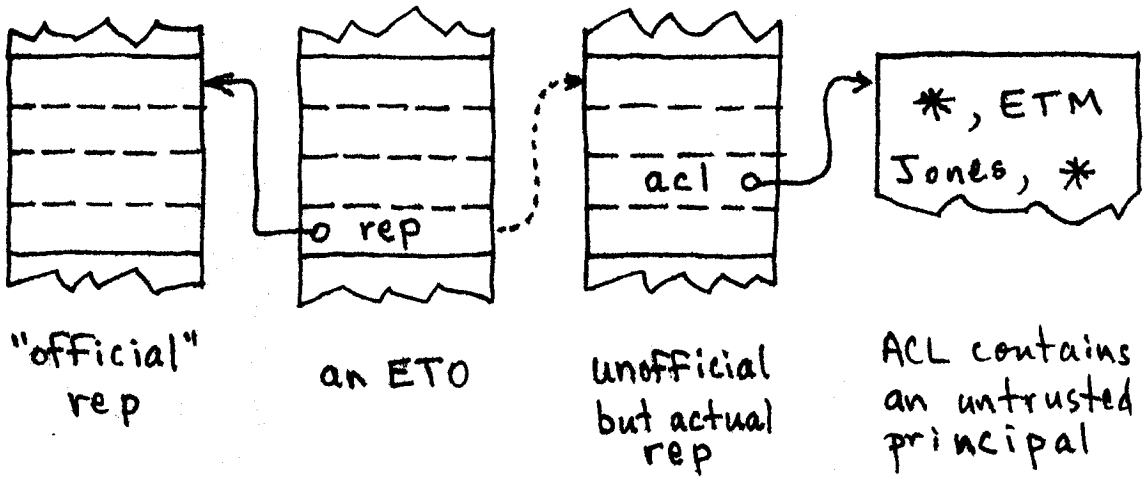
rep of an ETO. This suggests that the creation of objects destined to be rep-objects, and the assignment to the rep of some ETO should be an atomic operation. It is not necessary, however, that the entire rep of an ETO be created at the time that the ETO is created; the ETM can continually add or delete members of the rep, subject to the constraint that the added members are newly-created.

3) No non-unique reps are allowed.

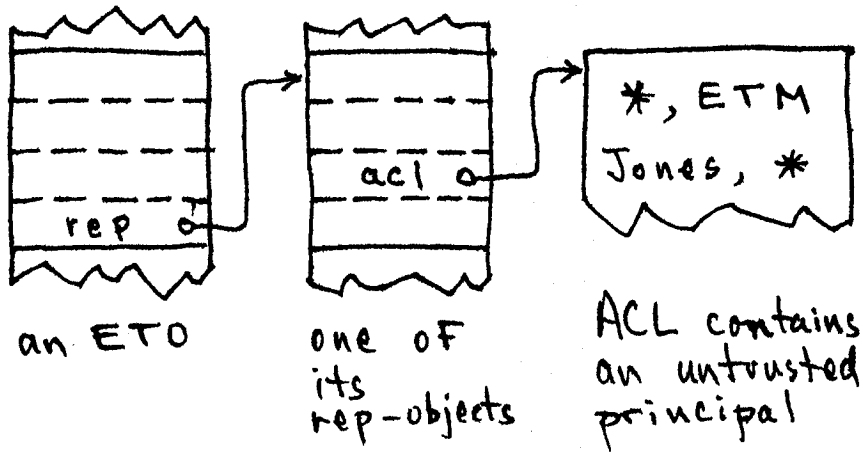
Even if the set of objects accessible to the ETM is disjoint from the set of objects accessible to other principals, the exclusion policy could be violated. If the ETM uses the same object as part of the rep of two distinct ETOs, then it may effectively access an ETO without authorization from a consumer. Therefore, a rep-object must belong to only one ETO. The restriction already mentioned, that only newly-created objects may become rep-objects, also suffices to guarantee that a rep-object has one parent ETO.

The three restrictions described here correspond to the three parts of Figure IV. If an ETM is able to operate subject to the above restrictions, then it can be given the registered attribute. Registered ETMs are certified, by system administrators, not to use implicit reps. The system administrators need not read the programs comprising an ETM to register it; rather by observing the objects for which the ETM is a consumer -- and by observing the corresponding ACLs -- they can deduce either the potential existence or else the non-existence

i) IMPLICIT REPS



ii) NON-PRIVATE REPS



iii) NON-UNIQUE REPS

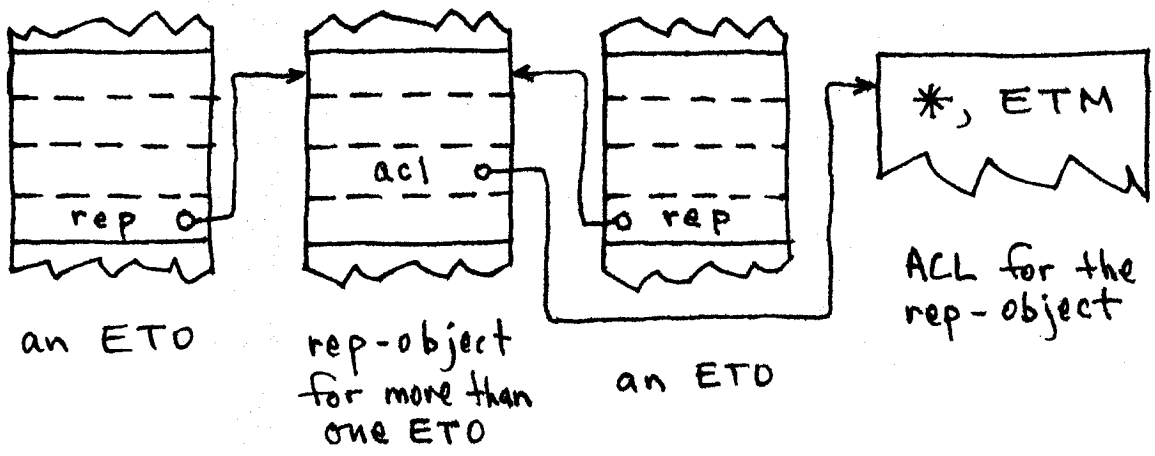


Figure IV

of implicit reps. Adherence to the "no non-private" and "no non-unique" restrictions need not be guaranteed by the system administrators; once an ETM has been given the "registered" attribute, the base level can enforce these latter two restrictions.

Although the details of the sealing mechanism are not described in this paper, a few comments about sealing are in order. The mechanism for sealing can be built on top of a mechanism for inter-domain argument passing similar to Schroeder's dynamic access stack (DAS) [ref. Schroeder]. The DAS is a per-process structure which contains, at any instant in time, a number of frames equal to the number of outstanding unreturned cross-domain calls. Each DAS frame contains pointers to arguments which are passed as part of a cross-domain call. Associated with each argument pointer is a source field, which indicates the first domain in the sequence of cross-domain calls which used that pointer (or a pointer denoting a containing subsegment) as a cross-domain argument. Schroeder's motivation for including a source field with each argument descriptor in the DAS frame was to simplify revocation of access to objects passed as cross-domain arguments. Concetptually, his scheme for passing arguments between domains does not require a source field. To implement the exclusion policy, however, the information contained in the source field is necessary to provide the identity of the consumer of an object. If we use the source field as defined by Schroeder to represent the consumer of an

object, then we must be sure that the source information propagates correctly down the DAS. In Schroeder's design, each time a new DAS frame is being constructed, a check is made for each argument pointer in the frame being constructed to see if it denotes a subsegment of a segment pointed to by an argument pointer in the previous DAS frame. Meeting this condition is necessary to allow propagation of the source field from an argument pointer in the previous frame to an argument pointer in the current frame. Since each argument pointer in the DAS has an associated bounds field, it is easy to determine whether or not any two pointers describe nested subsegments. The analogy in the case of our hypothetical system is to determine, given an arbitrary UID, whether or not it is part of the rep of some ETO. The way that this problem is resolved in our system is that the UNSEAL operation specifies not only a rep-object, but also a parent ETO.

The technique of using the source field as the identity of the consumer is a particular solution to a general problem which we call the <u>authorization</u> problem. The authorization problem for a PS is to determine on whose behalf a requested operation is about to be done. The source field technique has some drawbacks. In particular, it does not supply the invoked PS with the identity of the process of the invoker. In addition, it appears to be intimately tied to the procedure - call - and - return control structure. A more general discussion of the authorization problem is outside the scope of this paper.

The exclusion policy, if applied to a capability system, would require that an ETM be unable to reference the rep of an ETO unless the ETM has received a capability for that ETO as a parameter. The ETMs described by Redell [ref. Redell] could, as a result of error or malice, violate the exclusion policy. The Hydra system, [ref. Wulf et. al.] on the other hand, can apparently support the exclusion policy. In Hydra, each call to a procedure object generates a new local name space (LNS). The LNS associated with an incarnation of a procedure object contains two classes of capabilities, which are (1) capabilities in the template of the procedure object and are therefore part of the LNS of each incarnation, and (2) capabilities derived from parameter capabilities. By suitably restricting the capabilities contained in the template of a procedure object, it appears possible to constrain object-managing procedures in Hydra so that they obey the exclusion policy. Hydra procedure objects must be constrained in the following ways:

1) Hydra procedures must not be allowed to store capabilities into a data base with a lifetime longer than one procedure invocation. If a procedure could save a capability to the rep of some ETO, it could access the rep at some later time, without requiring a capability for the ETO. In our system, the sealing mechanism provides this constraint.
2) Hydra procedure templates must not contain capabilities for writeable segments with a lifetime longer than one procedure invocation. Otherwise, a procedure could, in effect, access

the rep of one of its ETOs during any of its invocations. This restriction corresponds to the "no implicit rep" rule.

3) Finally, the Hydra procedure object which is responsible for the creation of an ETO must be a base level (i.e. trusted) procedure. Procedures which are less trustworthy may be responsible for initializing and performing other operations on an ETO which has already been created. However, the operations of (1) generating a new UID, (2) encasing that UID in a capability, and (3) adding an entry in the system map corresponding to the new UID must be an atomic base level operation. The base level must carry out these operations for all objects, including objects in the rep-subtree of an ETO. This restriction should guarantee that all reps are private and unique.

As mentioned previously, adherence to the exclusion policy does not prevent an ETM from leaking information about its arguments. Some researchers in the field of protection are concerned with the problem of constraining subsystems so that they cannot leak information about the data which they manipulate. This problem has been called the confinement problem by Lampson [ref. Lampson]. As described in a thesis by Andrews [ref. Andrews], a general solution to the confinement problem does not appear to be possible. Most research related to the confinement problem is directed towards solving some weaker version of the problem. Andrews has characterized one weaker version as the problem of message confinement.

Andrews defines message passing as follows: an actor (i.e. an active object in the system) can send a message to another actor if either (1) the first actor has write privileges to an object and the second actor has read privileges to the same object, or (2) the first actor calls the second actor and passes parameters. A service is said to be message confined if it is necessary that a customer of the service must supply privileges enabling an actor in the service to send a message to another actor not in the service. We now consider whether or not an ETM in our system can be message confined.

In the context of our hypothetical system, let an ETM executing in a process (together with all registered ETMs which are dynamic descendents) correspond to Andrews' actor. Then the message confinement policy requires that an ETM be prevented from storing information into anything other than rep-subtrees of the consumer's ETOs. In particular, the requirements of (1) no implicit reps, (2) no non-private reps, and (3) no non-unique reps are necessary for an ETM to be message confined. Furthermore, the sealing mechanism is necessary for message confinement; otherwise an ETM could effectively send messages to another incarnation of itself executing in another process. These observations suggest, at least in the context of our system, that the same mechanisms are required to support the exclusion and message confinement policies. The policies appear equivalent, in the sense that the minimum mechanism necessary to support either policy is the same.

On closer inspection, though, there are differences between the policies. In one respect, the exclusion policy is stronger. Consider in the following scenario. Let principal A be on the ACL of ETOs X and Y, which are both of type T. Principal A calls the ETM for objects of type T, passing X, but not Y, as a parameter. Assume that the ETM modifies the rep of Y during this invocation. Such an occurrence is allowable under the message confinement policy, but not under the exclusion policy. The exclusion policy states that the ETM may access the reps of only those ETOs which (1) contain the principal of the consumer in their ACLs and (2) have been passed as parameters. The message confinement policy does not require that the second of the above conditions be satisfied.

Based on our observations about the exclusion policy and the message confinement policy, we claim that our hypothetical system can provide message-confined ETMs. This claim is based on the arguments that (1) our system can support the exclusion policy, and that (2) a system which supports the exclusion policy also supports the message confinement policy. Therefore one mechanism should be able to implement two policies which, at first glance, may not appear similar. Since the Hydra system is apparently able to support the exclusion policy, it should also be able to provide message-confined ETMs.

Summary

The significance of the results described here is twofold. First, we have shown how a system with an ACL-based protection

mechanism is able to support dynamic type creation. Previous designs to support dynamic type creation have been built on capability-based systems. Thus, we have gained further insight into the relative power of ACL and capability systems. Secondly, these results benefit ongoing research in the certification of operating systems. As mentioned in the introduction, a certification effort presupposes the existence of some relatively small kernel of hardware and software algorithms which must be well-understood. Given that the kernel is believed to be operating correctly, the certifiers can then deduce that the entire operating system must obey a policy implemented by the kernel. A primary motivation behind certification efforts is that relatively large portions of operating system software need not be scrutinized carefully. Based on our results, it appears that ETMs in an ACL-based system can be certified to adhere to the exclusion and message-confinement policies, regardless of the behavior of their component programs.

An objective of this research is to make it possible for certain operating system facilities which are usually part of the base level, such as message queues and directories, to be provided by ETMs. Hopefully, additional facilities which may be introduced into an extendible system can be implemented as extended type objects. The base level can then provide a common mechanism for controlling access to each object created within the system. As shown here, that common mechanism can be built using access control lists.