CACL: Efficient Fine-Grained Protection for Objects

Joel Richardson Peter Schwarz Luis-Felipe Cabrera
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

CACL is a protection scheme for objects that offers a simple and flexible model of protection and has an efficient, software-only implementation. The model, based on Access Control Lists (ACLs) integrated with the type system, allows owners to control who may invoke which methods on which objects, permits cooperation between mutually suspicious principals, allows ownership of objects to be transferred safely, prevents unwanted propagation of authority between principals, and allows changes to the authorization information to take effect on the next method invocation. The implementation, based on the integration of Capabilities with method dispatch, avoids the overhead of access checking in the majority of invocations, at the cost of space for extra dispatch vectors. CACL offers a viable mechanism for fine-grained protection in an object-oriented database system.

1 Introduction

For a number of years, object-oriented database systems (OODBs) have been an active area of research. Most of the attention has been given to traditional database concerns, such as defining more expressive data models, inventing query languages for objects, and devising schemes for concurrency control and recovery. Substantially less attention has been paid to the problem of protection. In some systems, e.g., O_2 [8], protection is not addressed at all (at least, in the published literature). Presumably, one would protect objects by protecting the operating system files in which they reside. Many other OODBs protect objects at the segment level [1][9]. A user can grant (or deny) to other users the ability to read or write the objects in his/her segments. One disadvantage of these approaches is that the granularity of protection is coarse. The entire object-base is partitioned into relatively few operating system files or object segments. While in theory, individual objects could be assigned to individual files or segments, doing so would carry sizeable overhead, both in administrative complexity and in system resources. Another disadvantage is that the model of protection is different from the model of the objects being protected. By offering only Read/Write/Execute protection, the protection model treats all objects as if they were of type File. Not only is this counterintuitive for the user, but it is probably sufficient only for get- and put-style methods. An object usually hides much complexity behind a relatively simple interface. What may appear to be a retrieval from a given object may actually involve "reading" many different objects and "writing" others. It appears that any assignment of rights to a user either grants too little privilege to get the job done or far too much to be safe.

There have been attempts to provide more sophisticated models of protection for OODBs. For example, a recent proposal allows both explicit and implicit authorizations[10]. An explicit authorization on one part of the database implies certain rights over other

© 1992 ACM 0-89791-539-9/92/0010/0263...$1.50

OOPSLA'92, pp. 263-275

parts, unless overridden by another explicit authorization. The scope and interaction of authorizations is tied to the semantics of the various abstractions offered by the model, e.g., classes, objects, composite objects, versions, and methods. This protection model is certainly integrated with the data model to which it applies. However, it is quite intricate, and given an arbitrary set of explicit authorizations, it is not immediately clear whether a given invocation by a given user will succeed. Nor is it obvious what will be the overall effect of a given change in authorizations.

Clearly, some form of protection is needed in any system that intends to manage objects of value to its users. However, we believe that a protection mechanism must balance the sophistication of its protection model against the ability of the users to understand it. CACL offers a very simple model. Essentially, the owner of an object controls who may invoke which of the object's methods. As we shall see, however, this simple model is quite effective in solving some interesting protection problems.

The seamless nature of an integrated object-oriented system also forces one to consider the issue of when authorization checks should be carried out. In conventional operating systems, a process' right to access a file can be checked at a distinguished point in the process' execution, i.e., when the file is opened. In an integrated object-oriented system, such distinguished points do not exist; a program, beginning at some root of persistence, simply follows references and invokes methods on objects. It may be necessary to check authorization at any point in the program's execution. Furthermore, access rights may be revoked or granted at arbitrary times. If such revocations and restorations are to appear to be "immediate," access rights must be checked (conceptually) at every invocation.

Clearly, the kind of mechanism we are suggesting poses major implementation challenges. In particular, if we literally place an authorization check in the path of every method invocation, performance will be severely degraded. Furthermore, we believe the vast majority of such checks will succeed because they are, in fact, unnecessary. There are three reasons for this. First, experience shows that most objects exist only to implement a higher-level abstraction. Method invocations that occur "inside" the larger object should not be subject to access checks. Second, we expect revocation and restoration of rights to occur far less frequently than method invocation. This means that in most cases, a process' right to access an object will not change from one invocation to the next. Third, we expect that a large percentage of all invocations will occur because a public method invokes a series of other public or private methods on self. If the caller has the right to invoke the public method, that should imply the right of the method to execute without further hindrance. Thus any practical implementation must meet the rather severe requirement of doing little or no work on each method invocation, yet giving the appearance of performing a full access check.

CACL is a fine-grained protection mechanism for strongly-typed, object-oriented programming environments or database systems. In this paper, we will discuss the model's semantics and its efficient implementation. The protection model is easy to understand, yet flexible enough to solve realistic protection problems. Our implementation meets the stringent performance requirement stated above, paying the cost primarily in the space required for additional dispatch vectors. The name, CACL, is a combination of *C*apabilities and *A*ccess *C*ontrol *L*ists, the two protection mechanisms whose properties are combined in this design. While the semantics of CACL can be described in terms of access control lists integrated with the type system, the implementation can be seen as capabilities integrated with method dispatch.

The remainder of the paper is organized as follows. In the next section, we describe some basic assumptions concerning the object model and runtime environment required by CACL. Section 3 describes the CACL protection model, and Section 4 provides some examples of its use. Section 5 describes an implementation of CACL that does not require specialized support from the underlying operating system or hardware, relying instead on cooperation between the language compiler and runtime system. Section 6 reviews related work, and Section 7 summarizes the contributions of this work.

2 Assumptions

The design and implementation of CACL assumes certain properties of the object model and runtime environment in which it operates. While CACL was developed in the context of a specific object model (Melampus [2][11]), the essential ideas are widely applicable. In this paper, therefore, we shall limit the set of assumptions to a minimum.

The first assumption is that every protected object is an instance of an abstract type. An abstract type specifies an interface and an implementation for its instances. An interface consists of a set of signatures that syntactically define what operations may be performed on an instance. The implementation consists of a representation for instances and a set of procedures which implement the operations in the interface. For the purposes of this paper, the key feature of an abstract type is that instances are encapsulated; there is no way for a program to access an object except by invoking its operations. Note that the object model may or may not also support subtyping. While such an assumption is made for this paper, it is not strictly necessary.

The first assumption leads to the second, which is that all methods and application code are written in a strongly typed language. In particular, it must not be possible to forge a reference (e.g., by casting) nor to invoke any operation on an object that is not supported by the object's implementation. This implies that the compiler must be a trusted component of the object system; indeed, in Section 5 we shall see how the implementation of CACL relies on the type checking performed by the compiler.

The last assumption is that the runtime environment is safe from outside attack. CACL is designed to provide protection within the confines of a particular data model that conforms to the assumptions described above, but if the model's implementation is embedded in a hostile environment, additional mechanisms may be required. In particular, we assume that all users of the system are reliably authenticated, that communication channels are safe from message replay and insertion, and that only trusted utilities and programs compiled by a trusted compiler can access the runtime interface. Clearly, a complete system requires several protection mechanisms in addition to CACL.

3 The Protection Model

This section describes the CACL model of protection. We will describe the basic concepts around which the model is built, followed by a description of the properties of certain critical events: method invocation, object creation, and transferral of authority between users.

3.1 Principals

In CACL, the locus of authority is called a *principal*[12]. A principal may correspond to a human user, a project group, or any other abstract entity on whose behalf actions are carried out within the system. We make no assumptions as to whether principals are themselves modeled as objects in the system, subject to protection. We assume only that principals are authenticated to the system and that there is some representation for a principal's identity. Principals appear in four different roles in CACL:

1. Each object has an *owner*, the principal responsible for authorizing access to the object. For any object *o*, *Owner(o)* returns the identifier of the principal that owns *o*.

2. Each object has a *method principal* (*MP*) on whose behalf the object's methods will execute. The MP is often, but not always, the object's owner. This point will be explained shortly. *MP(o)* returns the identifier of the method principal for *o*.

3. At runtime, the MP of the currently executing method is called the *current principal* (*CP*). Since CPs are pushed and popped along with activation records, we often speak of the *principal stack* associated with an execution thread. *CP()* returns the identifier of the current principal.

4. Finally, each implementation has an *implementor*, the principal responsible for the correctness of the implementation. *Impl(o)* returns the identifier of the principal that defined the implementation of *o*.

3.2 Access Control Lists

An object's owner controls which principals may invoke which operations by means of an *access control list* (*ACL*). A default ACL is attached to an object when the object is created. Conceptually, the complete set of ACLs in the system constitutes a mapping:

$$ACL(Object \times Principal) \to \{Method\}$$

That is, p is allowed to invoke method m on object o only if $m \in ACL(o, p)$. The primary motivation for CACL is to enforce the semantics of ACL-based protection for objects, while avoiding the cost of checking an ACL on every method invocation. As with principals, we do not specify whether ACLs are modelled as objects.

3.3 Typed References

Objects are accessed by means of typed references. The *effective type* of a reference is the interface described by the intersection of the following three components: 1) the interface of the declared type of the variable containing the reference (the reference's *static type*), 2) the interface of the type of the object denoted by the reference (the object's *creation type*), and 3) the interface specified in the object's ACL for the current principal (the principal's *authorization type*). The relationship between these types is illustrated in Figure 1. Static type checking ensures that the static type of the reference is always a subset of the creation type of the object to which it refers. Likewise, the authorization type (by definition) must always be a subset of the creation type. The relationship between the static type and the authorization type, however, depends on the principal who is accessing the object and the permissions given to that principal in the object's ACL. In this example, the effective type is narrower than the static type, as indicated by the shading.

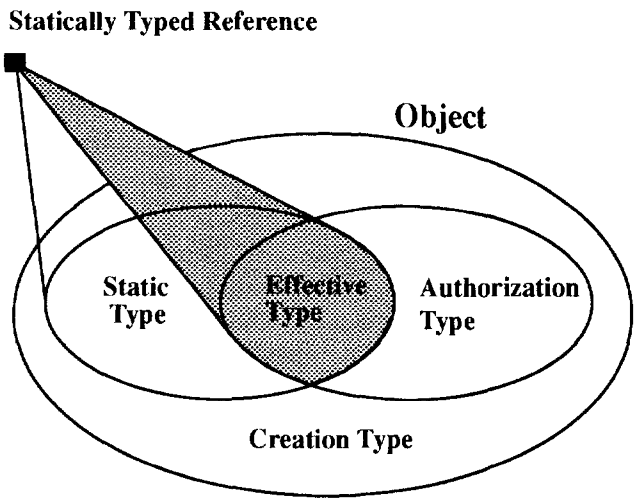

Figure 1. Types Associated with an Object

3.4 Method Invocation

From the point of view of protection, an execution thread is simply a sequence of method invocations. CACL focuses on this event and maintains the following invariant:

> *A method invocation succeeds if and only if that method is part of the effective type of the reference at the time of the invocation.*

Conceptually, the maintenance of this invariant requires an access check with every method invocation. As we will see, however, the implementation of CACL maintains the invariant in a way that stresses fast invocation for those cases (expected to be the majority) in which access checks are not needed.

Although we have described protection in terms of type checking, there is an important difference for the programmer between the assurances provided by static type checking and those provided by the protection mechanism. In a conventional statically-typed programming language, the programmer is assured that a program which compiles without type errors will be free from type errors during execution. Protection violations, however, by their very nature, cannot be detected at compile time. Access permissions vary from instance to instance within a class, and the specific objects to which a method will be applied are not known until runtime. Furthermore, access permissions may be changed at any time, and our desire to make such changes take effect on the next invocation further constrains any attempt to do protection checking in advance of method invocation.

We wish to stress, therefore, that although protection violations appear as runtime type errors in CACL, the runtime nature of such errors is inevitable and does not represent any loss of type safety. Any type errors that would be detected in a conventional approach will be detected by static checking, as usual.

3.5 Object Creation

Principals create objects with a system operation, new, which requires its caller to designate an implementation to be used for the new object. The current principal becomes the new object's owner and the implementation's implementor becomes the object's method principal. The reason that the object's initial MP is the implementor (and not the CP), will be discussed shortly. The object's initial ACL grants all rights to the owner, and none to any other principal. Object creation is summarized in Table 1.

$\mathrm{Owner}(o) \leftarrow \mathrm{CP}$

$\mathrm{MP}(o) \leftarrow \mathrm{Impl}(o)$

$\mathrm{ACL}(o, p) = \phi$, if $p \neq \mathrm{Owner}(o)$

$\mathrm{ACL}(o, \mathrm{Owner}(o)) = \mathrm{CreationType}(o)$

TABLE 1. Object Creation

3.6 Changing the Owner and Method Principal

The owner of an object can transfer ownership to any other principal by means of the operation Change-Owner(o, p). In so doing, the former owner transfers the right to manipulate the object's ACL to the new owner. However, the object's method principal does not change. The current owner of an object can set the object's method principal to him/her self, but not to any other principal, with the operation Set-MethodPrincipal(o). Together, these two operations provide for controlled transfer of an object between a donor and a recipient in a way the exposes neither party to involuntary risk. In the first phase of the transfer, an owner delegates the ability to update the object's ACL to another principal. This unilateral action does not pose a risk for the recipient, since the object's methods still execute with the authority of the old MP, i.e., it is not possible for an attacker to create an object of his/her choosing, "give" it to an unsuspecting user, then invoke the object's methods with the authority of the victim. In the second phase of the transfer, the recipient agrees to accept responsibility for the object's behavior by becoming the method principal. Here, the recipient accepts some risk, but only voluntarily. It is assumed that the recipient will first verify that the object is not a Trojan Horse, before agreeing to become MP.

4 Discussion and Examples

In this section, we discuss the features of the CACL protection model and show how they can be used to solve several important protection problems.

4.1 Trust Among Principals

CACL was designed with the assumption that different levels of trust exist between the principals who implement, administer, and use objects. In order to accomplish any cooperative work in a computer system, some mutual trust is obviously necessary. The client of a document editor entrusts the editor to make changes to a given document. Users of a mail system trust that mail will be delivered to all of (and only) the named recipients. All users trust the kernel to implement its abstractions correctly. However, "some trust" is not the same as blind faith. Allowing the document editor to update a particular document should not (necessarily) imply the granting of rights to any of the client's other objects. Furthermore, while the client may trust the editor, he/she may not trust the owner of the editor. Thus, allowing the editor to read a document should not imply the right of the editor's owner to read the document. These arguments led to the explicit support in CACL for the role of the implementor and to the separation of owner and method principal.

When a method executes, code written by the implementor executes on behalf of the method principal. Any client that invokes the method, passing it object references as parameters, must therefore grant to the MP enough rights over the parameter objects to allow the invoked method to execute properly. The set of methods required by the invoked method is likely to be advertised to the client as the static type of the parameter. If the client is unwilling to grant that many rights to the MP, then he/she cannot use the service (and is free to go elsewhere). There is a motivation here, familiar to software engineers, for an implementor to declare parameter types that are no more specific than necessary to perform the service.

Even though a client may trust a particular implementation to perform some function, the method principal may not be the implementor, but some other principal. In this case, the client still has a measure of protection in using the service, even though the MP may not be trusted in general. The reason is that passing the object to the method is not equivalent to simply handing the reference to the method principal; his ability to use the reference is constrained by the implementor's design. If the MP has no other access to the client's object, and if the implementation is sufficiently trusted, then the client can grant the required rights to the MP, even if the MP is not trusted.

4.2 Examples

4.2.1 Mutual Suspicion

Consider a scenario in which two mutually suspicious principals wish to cooperate to accomplish some task. Neither principal wishes to grant the other more than the minimal authorizations needed to accomplish the task. The fine-granularity protection supported by CACL makes this possible. Each principal need only authorize the other to perform exactly those operations on exactly those objects that are necessary to get the job done, and this authorization can be revoked immediately once the need for cooperation has ended. Furthermore, because references are unforgeable, neither principal can even attempt to invoke operations on objects that were not explicitly given to them.

For instance, suppose Joel wishes to print one of his documents using a printer owned by Peter. For the moment, let us also assume that Peter is the method principal for the printer's methods. Joel obtains a reference to the object that represents the printer, and invokes the Print method. If Peter has agreed to let Joel use his printer, the authorization type for Joel in the printer's ACL will include this method and the invocation will succeed. Once invoked, the Print method invokes various methods on Joel's document object to extract the document's text for printing. Since Peter is the method principal, these invocations will only succeed if the authorization type for Peter in the document's ACL includes these methods. Note, however, that the printer software has no need to update the document, and hence Peter need not be authorized to do so. Furthermore, by granting Peter access to this document, Joel does not grant Peter access to any of his other documents. Likewise, Peter does not need to grant Joel authority to use any other printer.

4.2.2 Trusted Implementations

In some cases, a principal may own, and wish to control access to, a resource which requires authority exceeding his own in order to operate correctly. For instance, the printer owned by Peter in the preceding example may require access to a proprietary font database in order to print documents. Only the implementor of the printer software, Acme Software, should be permitted to acccss this database. CACL provides the tools to solve this problem. As the implementor of the printer software, Acme Software is (by default) the method principal of the Print method when Peter instantiates a new printer object. The printer object will be owned by Peter, who can therefore control which other users may print on it, but Peter need not be authorized to access the font database because the CP during execution of the Print method will be Acme Software. As owner of the printer, Peter can make himself the method principal at any time, but thereafter the software will simply cease to work because he is not authorized to access the font database.

Note that in our revised example, Joel must authorize Acme Software, but not Peter, to access his document. This represents an additional reduction in the amount of trust Joel must place in Peter in order to use his resource. Joel must simply trust the (immutable) implementation of the Print method supplied by Acme Software.

Finally, we note that a considerable amount of mutual protection can be obtained even when the method principal is not a trusted third party, as in the example above. Suppose the Print method requires access to an audit file proprietary to Peter, instead of the font database, and therefore the method principal must be Peter. If Joel trusts Acme Software's implementation sufficiently, he will be willing to authorize Peter to read his document even though he doesn't trust Peter. This is because, as long as Peter cannot obtain a reference to the document through some other means, his ability to access the document is constrained by Acme's trusted implementation.

5 An Implementation Design for CACL

5.1 Introduction

A naive implementation of the CACL protection model would simply check the ACL of the target object before each method invocation. As we noted in Section 1, such an implementation would be prohibitively and unnecessarily costly. In designing a practical implementation for CACL, we sought to take advantage of the observation that the effective type of a reference can only change as a result of certain relatively infrequent events:

1. **Crossing a Protection Domain Boundary:** If a method executing with p_1 as current principal invokes a method that will execute on behalf of principal p_2, the effective type of each reference passed as a parameter must be recomputed based on p_2's authorization type for each referenced object. A similar recomputation must be performed for each reference that p_2 returns to p_1.

2. **Widening:** Object models that support subtyping often allow assignments in which the static type of the destination variable is a subtype of the static type of the source variable, provided that the creation type of the referenced object (as determined by a runtime check) is a subtype of the type of the destination variable. If a method executing with p_1 as current principal holds a reference of static type T_1 and tries to widen its view to type T_2, the effective type must be recomputed based on both the referenced object's creation type and its authorization type for p_1.

3. **Changing an Object's Method Principal:** Each instance variable in an object is a reference whose effective type is limited by the method principal's authorization type for the referenced object. If the method principal changes from p_1 to p_2, the effective type of each instance variable must be recomputed based on p_2's authorization type for the referenced object.

4. **Update to an ACL:** If the ACL for an object is modified or replaced, the effective type of every reference to the object must be recomputed.

Unless one of these events occurs, possession of a reference with a particular effective type is very much like having a capability for the referenced object. Possession of the reference, like possession of a capability, represents the authority to invoke a specified set of operations on the referenced object. In particular, there is no need to consult the ACL before allowing the invocation to proceed. CACL treats references like capabilities that are revoked when the reference's effective type changes as a result of one of the four events listed above. CACL also takes advantage of the fact that even if one of these events does occur, there is no need to recompute the effective type until just prior to the reference's next use.

Our proposed implementation for CACL is derived from a standard implementation of late binding in object-oriented systems. Such implementations use a dispatch vector (DV) to map an invocation in the user's program to the address of actual method code at runtime. CACL augments the role of the DV to include caching of authorization information. Instead of one DV per type, as in the standard implementation, our design requires one or more DVs per protected object. An entry in a DV may point to method code, as usual, or to a system procedure called the Protection Manager (PM). Depending on the contents of the DV, a method invocation will therefore be dispatched either to the appropriate method or, if an authorization check is needed, to the protection subsystem. The PM is responsible for checking the rights of the caller with respect to the target object, and will either raise an exception or continue the original invocation. The only "nonstandard" technol-

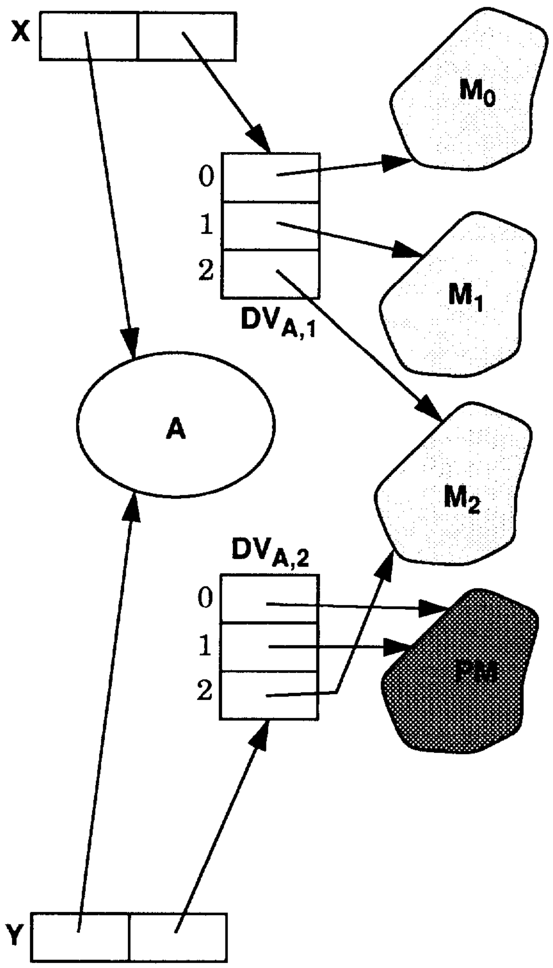

Figure 2. Overview Example

ogy required by this scheme is a small amount of assembly code that allows the PM to continue the original invocation without pushing a new stack frame.

Each of an object's DVs contains a combination of method and PM pointers that reflects a specific effective type. Object references contain two pointers: one to the referenced object and one to an appropriate DV.[1] Method invocation follows the DV pointer in the reference, and thus different references to the same object may behave differently, depending on the access rights granted to the principal holding each reference. By dynamically altering the contents of a reference and/or a DV when a reference's effective type changes, the system forces subsequent invocations to be directed to the PM to recheck the caller's access rights

An example of these data structures is given in Figure 2. References X and Y both refer to object A, which has two dispatch vectors, DV$_{A,1}$ and DV$_{A,2}$. However, the principal holding reference X, which refers to DV$_{A,1}$, is authorized to invoke any of A's methods, while the principal holding Y, which refers to DV$_{A,2}$, is only authorized to invoke method M$_2$. If method M$_0$ or M$_1$ is invoked via Y, control is transferred to the PM.

Although the design described here eliminates many unnecessary authorization checks, the performance improvements do not come for free. Using one or more DVs per object, instead of one per type, and the expansion of references to two pointers, represent time-space tradeoffs that increase performance at the cost of increased use of space.

The next two sections describe the data structures and algorithm that constitute our design in greater detail.

5.2 Data Structures

We assume that all data structures used to implement CACL are persistent, and that a separate mechanism is used to fault objects and method code into memory as needed.

5.2.1 Objects

An object consists of references to other objects (instance variables) plus some auxiliary data used for protection purposes. Every object has an owner field (OP), a method principal field (MP), and a pointer to the list of the object's dispatch vectors (DVp). The structure of an object is shown in Figure 3.

[1] One could eliminate the object pointer, and access the object indirectly through a pointer in the DV. This alternative would save space, but increase the time cost of comparing two references to determine whether they denote the same object.

OP	MP	DVp	Instance Variables

Figure 3. An Object

5.2.2 References

An object reference consists of two fields: a pointer to the object (OBJp) and a pointer to a dispatch vector (DVp). An object reference is shown in Figure 4.

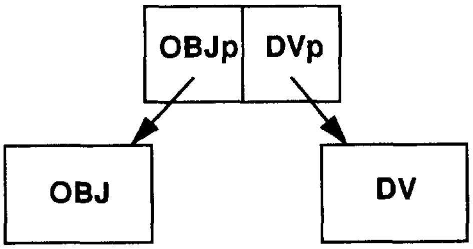

Figure 4. An Object Reference

5.2.3 Dispatch Vectors

A dispatch vector (DV) consists of an array of code pointers plus two additional pointers. If all of the code pointers in a DV point to the Protection Manager, we call it a PMDV. One of the extra pointers in the DV (PMDVp) points to a PMDV, if one has been created for the object. The other extra pointer (DVnext) is used to link together all of an object's DVs. There is no significance to the order in which the DVs appear in the list. Figure 5 shows an object with three DVs. The first contains all method pointers, the second contains one method pointer and two PM pointers, and the third contains all PM pointers (and thus is a PMDV).

While an object supporting *n* methods could theoretically have as many as 2^n DVs, we expect that most objects will be "private" objects used only within the implementation of some higher-level abstraction. Such objects will have exactly one DV. Furthermore, we expect that for most object types, only a small number of combinations of access rights will correspond to useful abstractions. Hence, even most "public" objects will only have a few DVs.

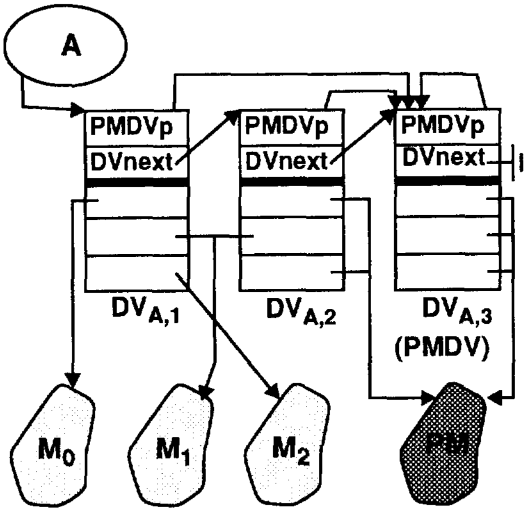

Figure 5. An Object and its DVs

5.3 Algorithms

Our algorithms stress fast invocation, based on the assumption that method invocation occurs far more frequently than any other operation of concern to the protection subsystem. Moreover, we also assume that the vast majority of all invocations are authorized, and will succeed. Thus, only when one of the four infrequent events listed in Section 5.1 has occurred will it be necessary for invocations to be directed to the PM. In the following subsections, we describe what processing steps occur for each event of interest.

5.3.1 Method Invocation

Method invocation consists of two steps. In the first step, the current principal is compared to the method principal field (MP) of the object that is the target of the invocation. If they differ, a boundary between protection domains will be crossed and special argument processing, described below, will be required. This step can be eliminated if it is possible to determine at compile time that the invocation cannot result in a change of principal. Two simple cases in which such a determination is possible are: 1) invocation of a "private" method not exported by the object's implementation, and 2) invocation of a method by another method of the same object.

In the second step, after evaluating arguments, the invocation sequence at the point of call indexes into the DV denoted by the DVp in the target object reference, and calls the procedure whose address is retrieved. This may result in an invocation of either the target method or the Protection Manager. The required index is generated by the compiler based on the method to be invoked, just as it would be in a standard implementation of late binding.

5.3.2 Crossing a Protection Domain Boundary

As we noted in Section 5.1, when a reference is passed across a protection domain boundary, its effective type may change. Rather than immediately recalculating the effective type based on the new principal's authorization type, CACL delays this determination until the reference's next use. As each reference is copied onto the argument stack, its DVp is updated to point to a PMDV. If a PMDV already exists for the object, its address can be obtained from the PMDVp in the DV associated with the old principal. If the PMDVp in this DV is null, a new PMDV must be allocated. Making the reference point to a PMDV ensures that the new principal's first method invocation using the reference will be directed to the Protection Manager for authorization checking.

This algorithm is used whenever invoking a method or returning from one causes a reference to cross a protection domain boundary. We feel this lazy approach to protection checking will be more efficient than the eager alternative, because references obtained by a method may not actually be used, but rather simply stored or passed on to other methods.

5.3.3 Widening a Reference

Widening a reference requires a runtime inspection of the referenced object to determine its creation type, regardless of any protection concern. One could recompute the effective type at this time, but to be consistent, we take the same lazy approach used when a reference is passed across a protection domain boundary. That is, the reference is copied and the new reference's DVp is set to point to a PMDV. The first method invocation through the widened reference will be intercepted by the PM.

5.3.4 Change of Ownership

Change of ownership does not change the effective type of references, but a check is required to make sure that the current principal of the process requesting the ownership change is the current owner of the object. This is done by comparing the current principal to the contents of the owner (OP) field of the object.

5.3.5 Change of Method Principal

A change of method principal is not permitted unless the object's owner (OP) field is equal to the current principal, in which case the method principal (MP) field is set equal to the OP. In addition, if the change succeeds, each reference in the object has effectively been passed to a new protection domain. Each reference's DVp field is therefore updated to point to a PMDV. The next invocation through each reference will be directed to the PM, to recompute the effective type.

5.3.6 Replacement or Modification of an Object's ACL

When an object's ACL is modified or replaced, every reference to the object is potentially affected. Locating all references and recomputing effective types accordingly is impractical. Instead, the system traverses the list of the object's DVs, and transforms each one into a PMDV by overwriting all method code pointers with the address of the entry point of the PM. Subsequent method invocations through any reference to the object will be redirected to the PM. The next section describes how references are revalidated, so that there is no permanent penalty for changing authorization information.

5.3.7 Processing in the Protection Manager

We have previously described several situations in which a method invocation is redirected to the Protection Manager. This section describes how the PM handles these invocations. The PM must perform two tasks: 1) compute the reference's effective type, check whether the invocation is allowed to proceed, and either continue the invocation in a manner that is transparent to the method's caller or raise an exception, and 2) modify the reference that was used to perform the invocation so that future invocations compatible with the reference's effective type are no longer intercepted by the PM.

In order to perform the first task, the PM must be able to ascertain which method is being invoked on which object, and on which principal's behalf. This is accomplished by suitable calling conventions implemented in the compiler, and will vary from system to system. Similarly, the mechanics of transparently intercepting and continuing a procedure call will vary from system to system. These aspects will not be discussed further in this paper.

To prevent future invocations using the reference from being redirected to the PM, the PM scans the list of the object's DVs for one that corresponds to the reference's (newly-computed) effective type. If none is found, a new DV is created and linked into the list. The reference's DVp is then updated to point to this DV. Note that in order for such modification to be useful, the method calling convention must permit the PM to locate the actual reference that was used to perform the invocation, rather than a copy of the reference.

Since references are often copied to temporary variables before being used to invoke methods, the scheme described above may not discover and convert all outstanding references that point to a PMDV. A special process that runs in conjunction with the garbage collector can be used to locate and update such references. Garbage collection must also be used to reclaim unreferenced DVs and PMDVs.

6 Related Work

Protection mechanisms have received considerable attention in both experimental and commercial systems, and we cannot possibly review all of this work here. However, we know of no prior work that is both as flexible and as efficient as CACL. Object-oriented databases, and other systems that require method invocation to be extremely efficient tend to offer relatively simple protection mechanisms. Conversely, the most flexible and sophisticated protection mechanisms are typically found in operating systems, file systems, and distributed systems, in which the operations to be performed are relatively complex, and hence the need for an extremely efficient authorization mechanism is reduced. We give some representative examples of both types of systems below.

Traditionally, protection mechanisms have fallen under the purview of operating systems. Two notable examples of systems that use Access Control Lists are Multics[12] and the Andrew File System[14]. In both systems, the ACL associated with an object (segment or file) specifies the permitted access in terms of Read/Write/Execute permissions. One can view these systems as providing type-specific protection for a small, fixed set of data types, while the CACL mechanism supports an extensible set. A major difference between Multics, AFS, and CACL is the frequency of access checking and its effect on the semantics of revocation. When Multics maps a segment to a process' address space, it performs a (long) access check and sets the appropriate permission bits in the process' descriptor word for that segment. Thereafter, the hardware performs an access check on every machine reference. If the segment's ACL is changed, Multics updates the segment descriptor word for that process (and for any other process that has the same segment mapped). Thus revocation in Multics is truly immediate. In AFS, a process requests certain access modes when it opens a file, and its permissions are checked at that point. Subsequent access to the opened file is not checked, except that the access must be one of those requested in the open call. If the file's ACL is changed in the meantime, the process will not observe the change until (and unless) it again opens the file. Thus revocation in AFS may take arbitrarily long to take effect. CACL takes an intermediate approach by checking access at every method invocation. As in AFS, revocation may take arbitrarily long to take effect, since a process may already be executing a method at the time its permission to that method is revoked. In this case, however, the implementor's code, not the client's, determines the length of the delay. Finally, by assuming a tight integration with an object-oriented data model, CACL's access checking can be done very efficiently, essentially by avoiding checks in most cases.

Many object-oriented database systems, including Gemstone[1] and ObjectStore[9], offer protection for objects that is reminiscent of file protections in Unix or AFS. In these systems, users partition their objects into segments or "databases" and control Read/Write (and in ObjectStore, Execute) access at the partition level. In fact, ObjectStore provides a model of protection and associated administration tools that closely resemble Unix; the name space is a hierarchy of directories and databases, both owners and groups are recognized, and there are ObjectStore analogues to the Unix commands chown, chgrp, chmod, etc. The SORION (Secure ORION) system[13] allows both principals and objects to be assigned security levels. SORION grants or denies access (in terms of Read and Write operations) based on the relative security levels of the requesting principal and the target of the operation. Neither of these approaches is as flexible as CACL, which allows individual objects to be protected, and grants or denies the right to invoke specific methods on a principal-by-principal basis. The Itasca object-oriented database[4] allows principals to be authorized to execute specific methods or to examine specific attributes, but, at least according to available product descriptions, such authorization applies to entire classes of objects, as opposed to individual instances. Authorization to instances is in terms of Read and Write.

The fine-grain, type-specific protection supported by CACL is more common in capability-based operating systems, such as Hydra[15] and, more recently, Amoeba[6][7] and ICAP[3]. Classic capability-based systems such as Hydra do not support changes to authorization information. Once a capability for an object has been given out, the authorization it represents cannot be revoked. In fact, the capability can be replicated or passed on to other principals without restriction. By contrast, CACL retains the ability to invalidate all outstanding references to an object and force access permissions to be recomputed. In so doing, CACL gives the owner of an object complete control over propagation of access rights, but does not allow an owner to delegate to another principal the authority to grant or revoke access. Some capability-based systems, like Amoeba and ICAP, support revocation mechanisms that are more flexible than CACL. Both systems rely on encryption schemes using random numbers to prevent forgeries. When a capability is presented for use, the server responsible for the named object validates the capability, either by decryption (Amoeba) or by re-encryption (ICAP). Both systems support revocation by allowing the owner of an object to interact with the server and change the server's internal key. One difference is that in Amoeba, revocation is universal, while in ICAP, revocation can be targeted to specific principals.

Recently, Luniewski, Stamos and Cabrera[5] described an access control mechanism for Melampus with properties similar to CACL. There are important differences, however, particularly in the implementations. The mechanism described in [5] depends on support from the underlying operating system and paging hardware, while CACL was designed to have an efficient, software-only implementation.

Finally, we should note that the manner in which CACL transparently intercepts procedure invocations is not itself new. For example, similar techniques are used, to implement dynamic linking. However, we know of no system in which this technique is used to implement an authorization mechanism.

7 Summary

Issues of data security, and hence access control mechanisms, will be of critical importance in future object-oriented programming environments and databases. This paper presented a software mechanism for fine-grained access control, called CACL, applicable to such systems. CACL combines the properties of two traditional access control mechanisms: capabilities and access control lists. The result is that CACL allows the owner of an object to control the ability to invoke individual methods on a per-principal, per-object basis. Our mechanism is based on the use of object-specific customized dispatch vectors that, once established, encode authorization information so that the system can directly invoke methods without explicit authorization checking. Nevertheless, the mechanism retains the ability of a principal to change authorization information at will, with such changes taking effect on the next method invocation. Our mechanism requires support from the language compiler and runtime system, but no support from the underlying hardware or operating system.

Finally, we should point out that there are many possible variations to our implementation. For example, when a reference is passed across a protection boundary (Section 5.3.2), we could set the DVp field in the reference to **NULL** instead of retrieving the PMDVp. Doing so would require some cooperation from the operating system (to catch the null pointer dereference) as well as additional convention in the calling sequence (to determine whether a null pointer dereference should trap to the PM or signal an error). Furthermore, the model itself can be improved. For example, CACL does not include a notion of protection *groups*, i.e., an aggregation of objects (of the same type) that share the same ACL. Such an abstraction would be quite useful and could be implemented (in part) by sharing the list of DV's among the objects.

8 Acknowledgements

We would like to thank the other members of the Melampus group, especially Alan Luniewski and Jim Stamos, for numerous discussions and helpful comments.

9 References

[1] R. Bretl, D. Maier, A.Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams and M. Williams. "The GemStone Data Management System" in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., ACM Press, 1989.

[2] L.F. Cabrera, L. Haas, J. Richardson, P. Schwarz, and J. Stamos, "The Melampus Project: Toward an Omniscient Computing System," IBM Research Report #RJ 7515, June 1990.

[3] L. Gong. "A Secure Identity-Based Capability System" in *Proc. 1989 IEEE Computer Soc. Symposium on Security and Privacy.* Computer Society Press, Oakland CA, 1989.

[4] Itasca Systems, Inc. *Itasca Technical Summary.* June 1990.

[5] A. Luniewski, J. Stamos, and L.-F. Cabrera. "A Design for Fine-Grained Access Control in Melampus" in *Proc. of 1991 International Workshop on Object-Orientation in Operating Systems*, Palo Alto CA, October 1991.

[6] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," in *IEEE Computer*, 23(5), May 1990.

[7] S.J. Mullender and A.S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," in *The Computer Journal*, 29(4), 1986.

[8] O. Deux, *et. al.*, "The Story of O_2," in *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[9] Object Design, Inc. *ObjectStore User's Guide.*

[10] F. Rabitti, E. Bertino, W. Kim, and D. Woelk, "A Model of Authorization for Next-Generation Database Systems," in *ACM Transactions of Database Systems*, 16(1), March 1991.

[11] J. Richardson and P. Schwarz, "MDM: An Object-Oriented Data Model," in *Third Int'l Workshop on Database Programming Languages*, Nafplion, Greece, August 1991. Also available as IBM Research Report #RJ 8228, July 1991.

[12] J.H. Saltzer, "Protection and the Control of Information Sharing in Multics," in CACM, 17(7), July 1974.

[13] M. B. Thuraisingham. "Mandatory Security in Object-Oriented Database Systems" in *Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications.* ACM Press, New Orleans LA, October 1989.

[14] Transarc Corp. *AFS 3.0 User's Guide.* Pittsburgh PA, 1990.

[15] W. Wulf, R. Levin, and S. Harbison. *Hydra/ C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.