Capability-Based Protection for Integral Object-Oriented Systems

María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, Juan Manuel Cueva Lovelle

Department of Computer Science, University of Oviedo
Calvo Sotelo, s/n, 33007 OVIEDO - SPAIN
{fondon, darioa, lourdes, falvarez, cueva}@pinon.ccu.uniovi.es

Abstract
Protection is an essential issue for the kind of heterogeneous distributed interoperable object environments, which Java and CORBA are a glimpse of. A uniform protection mechanism such as capabilities, integrated with the object model of the system is a good and flexible solution for these environments. Oviedo3 is an example of an Integral Object-Oriented System (IOOS) based solely on the OO paradigm, developed over an OO Abstract Machine (OOAM) and OO Operating System (OOOS). When integrated with the object model of the abstract machine, capabilities gain new advantages such as automatic protection. Capabilities also allow more flexible security policies. Existing platforms such as Java, without an uniform protection mechanism, could be made more suitable for these new environments by applying these benefits.

1. Introduction

The adoption of the object-oriented paradigm is growing steadily. With the Internet boom and the distributed systems, a heterogeneous distributed interoperable object environment is appearing. Platforms such as Java and architectures such as CORBA are responsible for the evolution towards this environment, which is increasingly seen as an environment for the future.

With this kind of environment, a comprehensive protection mechanism is needed. This mechanism should be able to establish access-level restrictions to objects, checking the access rights to the objects of the environment.

With the wide use of object-oriented languages and object-oriented technology in general research laboratories start to develop systems aimed to support object-orientation. These are commonly in the form of operating systems, which facilitate object interoperability, offer direct support for objects or are built in an object-oriented way. For example, Amoeba [1] allows to use OS resources by using objects managed by server processes. Munsg [2] or Monads [3] provide the concept of objects as contiguous memory spaces and manage them as that. Others such as Choices [4], or Guide [5] support object interoperability between user objects created by an OO programming language, even when located in different address spaces or different machines in a network.

Every one of these systems provides a protection mechanism, which can restrict the communication between objects when necessary. Nevertheless, protection is commonly bound to traditional concepts such as process and address space and is not applied down to the level of fine-grained objects in user applications.

An integral object-oriented system (IOOS) is an example of the kind of “world of objects” environments which are hinted as the evolution of current environments. In this paper, a protection mechanism based on capabilities is deemed as the best match for an IOOS. Protection extends to the level of individual fine-grained objects in the system, protecting object methods separately. Moreover, the protection is introducing in the system in a homogeneous way, without disrupting the unique object model of the system.

2. The Integral Object-Oriented System Oviedo3

The Integral Object-Oriented System Oviedo3 is an IOOS under development at the University of Oviedo. Oviedo3 [6] is structured upon an Object-Oriented Abstract Machine (OOAM) and an Object-Oriented Operating System (OOOS). They define a system in which the concept of object (of any granularity) is the only abstraction. The machine supports a uniform object model for the rest of the system, intentionally standard to

---

*This work has been partially supported by the second regional plan (FICYT) of research of the Principality of Asturias, Spain, under project PBP-TIC97-01 / Object-Oriented Integral System: Oviedo3*. 
support popular OO Methodologies. It has the usual features of encapsulation, inheritance and polymorphism, with global object identity for references, exceptions, and aggregation and generic relationships. Objects are homogeneous, active entities encapsulating all semantics. The OS is composed of a set of normal user objects transparently providing the rest of the object with OS functionality: persistence, concurrence, distribution and security. With a reflective architecture, even the OOAM is a set of objects visible to user objects that can be used like any other object in the system. Thus, an IOOS (fig. 1) is achieved: a single object space where objects exist indefinitely, virtually infinite, and where objects placed in different machines cooperate transparently by means of method invocation through unique object references.

3. Design of the protection mechanism for the IOOS: Capabilities

For this kind of distributed object system, objects are supposed to be autonomous entities, which can migrate from node to node. Keeping this in mind, a protection mechanism based on Access Control Lists (ACLs) is not very adequate. ACLs require a per-object list in which objects that can access the object are registered, as well as the access rights held by every one of these objects. So every object must record and be aware of all the objects that could use it. To know every object in the system would not be possible always. In an IOOS, with many distributed objects, ACLs would rapidly grow big, and this (among other things) limits mobility. Besides, this kind of centralized protection management introduces additional complexity in the system by forcing the use of an object-identification and registering system for every object in order to construct the ACL.

3.1. Capabilities

Capabilities [7] are a better match for an IOOS with a uniform object model. A capability mechanism does allow complete autonomous objects. To make a method invocation on an object, the invoking object must show a capability for that object. A capability is like a ticket entrance; its possession grants access to a specific object. However, a capability also has access permissions on the methods of the object, specifying which methods can be invoked by holding the capability.

Like object references, a capability can be obtained only by creating a new object or by parameter passing in a method invocation. Once obtained, the capability is sent with the rest of the parameters of the method. The protection system checks the rights before granting access to the method.

Many operating systems using objects in one form or another have researched capabilities. One differentiating aspect is how to implement capabilities in a way that forbids forgery. A user should not be able to arbitrarily alter a capability, modifying the rights to have access to otherwise inaccessible methods.

Some systems like Monads [3] use specialized hardware, which tags capabilities in hardware so that user manipulation is deemed impossible. But this can not be used on conventional hardware, restricting the use of these systems, even more for a heterogeneous distributed system.

Other systems like Grasshopper [8] use segregated capabilities. Capabilities are stored in memory regions inaccessible by users, managed only by the operating system. The drawback is that capabilities can not be embedded in user structures like any other user data. Therefore, they can not be passed freely between applications.

Encryption or sparse capability spaces [1] ensure that it is almost impossible by blind-generation to guess a valid capability. This avoids tampering while using capabilities in normal user space. Amoeba [1], for example, uses these techniques for its “password capabilities”.

When capabilities can be used in user structures, there is a natural confluence of the protection mechanism with the object model of the system. Just by adding protection information (access rights) to object references gives the system a uniform protection mechanism without changing the object model, keeping its conceptual simplicity.

In an IOOS with an OOAM as the substrate, capabilities can be used combining the advantages of hardware and segregated capabilities while using...
capabilities in user objects without the need to use encrypted or sparse capabilities.

3.2. Architecture of the abstract machine

The architecture of the object-oriented abstract machine for the Oviedo3 is described below. This is a generic OOAM architecture. Its organization may be considered as a reference of the features that an abstract machine should have to support this kind of IOOS.

The OOAM supports the object model for the whole system. Objects are homogeneous and the only supported entity. Every action upon an object is done through a reference to it. References, like objects, have an associated type. Machine instructions (which are object invocations) have references as parameters, as well as methods in classes. Return values are also references.

Fig. 2 shows the architecture of the Carbayonia abstract machine \[8\], consisting of four areas: class area, references area, instance area, and system references area. Memory addresses are not used directly, object references holding object identifiers are used instead. Each area can be considered as an object in charge of the management of its data.

![Figure 2. Architecture of the Carbayonia abstract machine](image)

**Class area.** Maintains the description of classes. There is a set of primitive classes defined permanently in this area.

**References area.** Stores references. Every reference has a type (relates to the class area) and points to an object of this type (relates to the instance area).

**Instance area.** Stores objects created. At run time, the information of its class can be accessed in the class area.

**System references area.** Contains references with specific functions in the machine.

The machine language is a pure OO low level language. It allows class declaration, method definition and exception handling. It will be the intermediate language used by compilers of programming languages like Eiffel, C++, etc. Once compiled into this intermediate language, the source language of object creation is irrelevant.

3.3. Capabilities as the Protection Mechanism

Object references are merged with protection information to form capabilities. Thus, the object references used in the machine (now capabilities) have a dual function: naming and object protection. A reference in the abstract machine is an object that contains an object identifier (points to an object). Adding attributes (fig. 3) that specify which methods are allowed to be invoked on the object by using the reference gives a pure capability. Consequently, the protection mechanism is introduced at the level of the abstract machine.

![Figure 3. Addition of protection information to a reference](image)

The machine should check the protection information now present in the references. When an object A wants to invoke method M of an object B, it must get somehow a reference to object B. This is usual practice and it is not related to the existence of a protection mechanism. Then, the method M will be invoked using the reference (which will carry protection information encoding the access rights for the object). When the machine executes the method invocation instruction (it is basically the only instruction available), it should previously check if there is a right to invoke method M in the protection information in the reference. Normal execution continues if that is the case, with an exception thrown when not.

To augment the machine with the protection mechanism as shown above, a number of modifications are needed in the machine, in the references, the management of references and in the invocation mechanism:

- The class for references in the machine has to add attributes to store the protection information. These attributes must be able to specify individually, for every method, if it is accessible or not. The number of methods of an object is not a fixed number.
- Operations on references must take this protection information into account. For example, when creating a new object, initial access rights should be established. Cloning of references can be affected, too.
The method invocation of the machine must be modified. An additional step of checking the protection information is needed. The invocation should be stopped by throwing an exception when not having permission.

In addition to these basic modifications, the use of capabilities can introduce other changes, be it just for the use of capabilities or for the use of certain variations of the concept:

- New operations to perform protection management. Typical operations include restriction of access rights in a reference to get a reference with fewer permissions than the original, etc.
- Special rights can be used, such as an “owner” right. These special rights will make the machine behave in different ways when present. For example, the “owner” right could be used to bypass the protection check, speeding execution. The rationale is that the owner of an object can access all methods without restriction.

### 3.4. Advantages of this approach

These aspects should be investigated, such as which special rights should be included, which operations should be defined to manage protection, etc. Nevertheless, the basic properties of capabilities used as described in an abstract machine for an IOOS have some clear advantages:

- **Automatic protection of capabilities.** Advantages of both segregated and sparse capabilities are combined: The machine guarantees capabilities can not be tampered with or altered without permission. Only the operations provided by the machine can be used to handle capabilities. The additional cost of sparse techniques is avoided.
- **Simplicity and easy to use.** Neither the object model nor the way of using objects changes. The protection mechanism is transparent for the rest of the system. On the one hand, by using capabilities, the objects are not aware of the management of its protection. On the other, using capabilities as normal references in user structures facilitates system use, because there is no special way to access the segregated area where capabilities are stored.
- **Flexible implementation with application independence.** Having capabilities at the level of the abstract machine, the internal implementation in the machine can be tuned and modified without breaking user applications. For example, special ways of encoding access rights can be used to accelerate protection-checking, etc. The structure to store protection information is not visible to user objects. So user objects are not dependent on a special structure chosen to represent capabilities and this structure can be changed without concern.

- **Arbitrarily long access rights for every method of an object.** In other systems, access rights in capabilities are restricted to a fixed size. For example in Amoeba they are eight bits protecting common operations for coarse-grained server objects. Mungi has just Read, Write, and Execute rights for its objects (just contiguous memory regions). But for an object with full semantics as in an IOOS, the number of methods is not fixed. The abstract machine presents references as primitive concepts for the applications, so its implementation is not visible. As said before, the flexible implementation allows implementing arbitrarily long, even variable-size capabilities if wanted without breaking applications.

### 3.5. Efficiency

A possible drawback of this kind of capabilities is efficiency. First, abstract machines emulated in software are not as fast as native code. This is not related to protection. However, the success of Java shows that users are willing to trade speed for the convenience of having portable, mobile objects in distributed networks. The second point is the additional step required for protection checks. While this must be measured and additional, check should not add that much overhead to an emulation machine. It is expected that users will again trade some more speed for the flexibility provided by this protection mechanism. The same can be argued to the additional storage needed by protection information. Besides, the internal implementation of capabilities can be tuned to reduce the performance penalty.

### 4. More flexible security policies.

This describes the basic protection mechanism for the IOOS. Capabilities are used to restrict access at the level of individual methods. The protection mechanism involves managing the protection information in the capabilities as well as checking permissions in method invocations.

However, there are many aspects that are not the task of the protection mechanism. These aspects are policies of the system, such as who decides which access rights are passed to a given object, etc.

Policies can be defined that establish this kind of parameters. They can be provided as a set of classes and
objects of the OOOS. Capabilities are general enough to allow the implementation of a wide range of policies. Many policies are strongly linked with an object naming service, which gives objects a symbolic name to help users find them. It is easy to implement these policies. When asked for an object, a reference is returned to the user. Nevertheless, the reference will have the access rights of the capability set in different ways according to the policy.

There can be different policies coexisting in the system. For example, a policy for untrusted objects arrived from the net can be defined, akin to the Java sandbox policy. The object is simply given only capabilities to access objects, which will not harm the system. But capabilities are flexible enough to permit less restrictive policies than this: complete access to not important objects versus no access at all to important objects. Usually access to any file at all is restricted. Instead, the level of access can be configured to the level of individual methods, tailored for a particular object. For example, a spreadsheet-report object could be allowed to have write access to a window in the screen, and read access to a specific spreadsheet object. More flexible security policies can then be used.

5. Application to other systems like Java

The use of capabilities as described above is being implemented in the abstract machine of the Oviedo3 IOOS. A number of design alternatives for operations on capabilities, special rights information is being considered to find which are best suited for an IOOS. Prototype implementations will be compared to versions of the abstract machine without protection and to similar systems like Java to measure the real cost of protection. But as said before, based on our experience with the software implementation of the abstract machine, it is expected not to add that much overhead to the machine.

The benefits of this approach can be applied, however, to similar systems. On the one hand, capabilities are a perfect match for systems based uniformly on the OO paradigm. Systems based on abstract machines get more advantages. The obvious example of this kind of systems is the Java platform. Current mechanisms used for security with Java as defined by browsers are not very flexible [9]. The adoption of this kind of strategy to add capabilities to the Java Virtual Machine will make protection an integral part of the system and not an add-on. Thus, more flexible security policies can be developed, tailored for a variety of user needs.

Applications would not have to be rewritten, for an approach like hidden capabilities [10] can be used. In this approach, user objects are separated from protection. Capabilities are managed by external objects, which give user objects capabilities (normal references) according to policies. A security policy for untrusted objects is such an example. Existing applications just use the same object references as before, but now the machine extends references transparently with protection information. However, new applications could manage protection directly if wanted, by using operations on capabilities. Obviously, security policies are one of this kind of applications that must be aware of capabilities and protection.

6. Related Work

There are other research projects with this goal of an integral OO system. Tunes [11] and Merlin [12] are systems which share this approach of using reflective architectures and OO to obtain such environments, organized around the SELF language in the case of Merlin, and with a new language developed for Tunes. Protection, however, is not melted with the object model of these systems.

The CORBA object security services [13] specify a complete security architecture for the CORBA platform with no specific implementation or mechanisms, while this work deals only with a concrete protection mechanism. However, it could be used by CORBA servers for access control within the CORBA framework.

Other projects have a specific focus on bringing capability-based protection to existing environments, sharing the idea of using capabilities as the basic protection mechanism in which to build future flexible distributed object environments. The particular capability model varies. There are proposals to incorporate the model of hidden capabilities into the CORBA and Java platforms. The SLK project [14] intends to introduce type-capabilities into the Java platform. Type-capabilities offer a combination of static link-time protection checking based on common access rights for all objects of the same type with usual dynamic, individual capability protection. However, the implementation of these different kinds of capabilities, however, must be done by user-libraries and object-managers added to the existing platforms. So, the use of most of the functionality of capabilities is done by self-limitation in programming, and is not an integral part of the object model and infrastructure of the system.

7. Conclusion

The evolution towards heterogeneous distributed interoperable objects multiplies the need to have robust
object protection mechanisms in order to allow only authorized method invocations. An Integral Object-Oriented System (IOOS) is an example of this new “world of objects” environments based solely on the OO paradigm.

The conception of objects as autonomous entities and the unit of structuring of systems leads to a uniform protection mechanism at the level of individual methods of objects. Traditional protection mechanisms linked to other abstractions such as process and address space are not suited to these environments.

Oviedo3 is an IOOS with a reflective OO Abstract Machine as a substrate, coupled with an OOOS composed of a set of user objects providing OS functionality.

Capabilities are a best match for an IOOS because they fit very well in the uniform object model of an IOOS without breaking homogeneity. This is achieved by extending object references with protection information to form capabilities.

In the case of an OOAM, the use of capabilities at this level brings a number of additional benefits. These include automatic protection of capabilities, a flexible implementation with application independence, and a pure capability model with arbitrarily long access rights for every method of an object.

This basic mechanism can be used to implement security policies according to the different needs of the system. It specially allows to define more flexible security policies for distributed object systems. For example, security policies for untrusted mobile objects more tailored and less restrictive than other systems like Java. Security, nevertheless, is not compromised because the machine guarantees no object can bypass the protection checks.

This model could be easily applied current similar systems; especially systems based on abstract machines, like Java, to empower them with the flexibility and benefits of uniform fine-grained protection.

Although this model imposes a performance loss for the checking of protection, it is expected that it would be again accepted for users as was the case with software-emulated abstract machines because: i) the performance hit of the additional step on an already emulated machine should not be that high and ii) the flexibility offered by the model.

References


