Object-Oriented Hierarchies Across Protection Boundaries

David W. Dykstra
dykstra@cs.uiuc.edu

Roy H. Campbell
roy@cs.uiuc.edu

July 31, 1991

1 Introduction

In the context of programming systems, *protection* is the mechanism used to control access of programs and users to resources, including data [PS85]. Protection is often introduced to provide integrity, to prevent inappropriate use that would contravene some resource access policy that is beneficial to the users of a system. Protection may also be used to ensure security, but this is a broader topic than the issues we wish to discuss. Our work is concerned with protection and the design of an operating system that supports an object-oriented interface to the application. Specifically, this paper examines the relationship between protection and object-oriented systems.

Protection can be enforced by *convention* or agreement amongst the users of the system to program in a specific manner, by a *compiler* that statically checks protection rules are not violated or inserts run time checks to verify the access rules are not violated at execution time, and by *hardware* that prevents violation of access rules physically at execution time. *Encapsulation* is a major object-oriented programming protection mechanism that restricts manipulation of objects defined in the representation of an abstract data type from being accessed except by a set of predefined methods or operations. Encapsulation is beneficial because it reduces the amount of coupling between code fragments and data, encouraging modification, reuse, porting, and understanding. Encapsulation can be implemented by convention as in languages that do not have features that are object-oriented, by compiler as in Smalltalk, or by hardware, as in various capability-based computer architectures. Where capabilities are not built into a hardware architecture, hardware protection may also be provided by a combination of hardware and operating system primitives. For example, the supervisor/user state distinction or virtual memory mechanism may be used to provide hardware-enforced encapsulation. Protection by convention can easily be violated. Similarly, protection by compilers depends on the effectiveness of the compiler, the nature of the language, and whether multiple languages are used to build applications. Hardware-enforced protection offers more security for access policies and can be applied to more general access problems.

Encapsulation is, however, only one aspect of an object-oriented system. As defined by [Weg87], objects in object-oriented systems belong to classes and class hierarchies that can be incrementally defined by an inheritance mechanism. In addition, [Lis87] demonstrated that polymorphism through the use of type hierarchies is more significant than inheritance hierarchies, and that both kinds of hierarchies are useful in an object-oriented system.

In our efforts to build an object-oriented operating system that supports an object-oriented application interface, we have discovered a need to split both inheritance hierarchies and type hierarchies across hardware-enforced protection boundaries. The subject of this paper is an analysis of the problems caused by splitting these object-oriented hierarchies. Section 2 presents some motivating examples as to why we want to be able to split hierarchies across protection boundaries. Section 3 gives background for the detailed analysis of splitting hierarchies that is in Section 4. Section 5 describes our solution for the *Choices* operating system and C++.

2 Motivating Examples

Before we begin the analysis, we would like to propose a few motivating examples for our desire to split object-oriented hierarchies across protection boundaries. While developing the *Choices* object-oriented operating system, which is written in C++, we have encountered several categories of situations in which it would be helpful to extend the object-oriented hierarchies out of a protected kernel.

2.1 Providing System Services

The most fundamental category involves simply providing system services to users. For example, if the operating system provides a Semaphore object to synchronize processes, user programs should be allowed to invoke methods on the Semaphore as easily as the kernel itself can. This example does not show an extension of an object-oriented hierarchy, but it does show an extension of the use of a hierarchy.

2.2 Customizing Services at User Level

The next category involves customizing operating system services at the user level. For example, there is an OutputStream type in *Choices* that has a **write** method to send data to the stream and a **flush** method that is called when data is to be written out. We would like to be able to provide a BufferedOutputStream in user space that overloads the **write** and **flush** methods to save data in a local buffer and reduces the number of times that system calls need to be made. The BufferedOutputStream should be a subtype of OutputStream so the user can treat it as an OutputStream through the polymorphism of the type hierarchy. This example also shows an extension of the implementation hierarchy in that the overloaded methods in BufferedOutputStream invoke corresponding methods on the OutputStream type.

2.3 Customizing Services at an Intermediate Level

We would like to have a minimal kernel and to move many shared system services into an intermediate protection level. The kernel will be protected from the intermediate system services, and the intermediate system services will be protected from user programs. There are many advantages to a minimal kernel as demonstrated by other modern operating systems [CZ83, R$^+$89, BA$^+$89].

When there are more than two levels of protection, there will be cases where a type is defined on one level, a subtype is defined on another level, and the type is used on a third level. For example, there is a Timer type in *Choices* that defines **start** and **await** methods. We want to define the Timer type at the minimal kernel level, define a subtype called TimeoutTimer at an intermediate level, and allow the user level to use the TimeoutTimer type and treat it as a Timer through polymorphism.

The Timer example also shows another kind of use of object-oriented hierarchies across protection boundaries: the minimal kernel may have cause to use a TimeoutTimer that is defined at the intermediate level. The kernel only knows about Timers, but if it treats a TimeoutTimer as a Timer, method calls will call out to the TimeoutTimer at the intermediate level.

2.4 Separating Policy from Mechanism

Another category is that of providing a mechanism at a protected lower level and a policy at a less-protected higher level. For example, we want to allow higher levels to control the policy of handling page faults. The operating system will define an interface type called PageFaultPolicy and methods implementing the default policy for handling a page fault. User programs will be able to make a subtype of PageFaultPolicy and tell the operating system to invoke methods on the subtype instead through polymorphism.

This same kind of feature could be provided without using subtypes, but it is a convenient way to provide the feature in an object-oriented system.

3 Background

This section discusses several background topics that will be useful for our analysis.

3.1 Operating System Protection

Protection in operating systems is defined in terms of objects, subjects, protection domains, and processes.

object A unit of data. In an object-oriented system, an object is more than just a unit of data: an object is encapsulated such that only the classes that "own" them can manipulate them.

subject A unit of code. In an object-oriented system, each class is a subject.

protection domain A list of access rights to objects. Each subject is associated with one protection domain that specifies the access rights that the subject has to objects. There can be more than one subject for each protection domain. In our discussion, the access right that we are concerned with is the ability to modify objects; that is, both **read** and **write** access rights together.

process A thread of control. Processes wind their way through different subjects, changing protection domains as they change from one subject to another. Changing from one protection domain to another is called crossing a protection *boundary*. This occurs at the time a process calls a method on an object that belongs to a subject in a different protection domain than the one it was executing in.

3.2 Trust

In operating systems, different kinds of *trust* relationships exist between subjects. For example, subjects in a kernel are more trusted than subjects in applications, and application subjects do not normally trust each other.

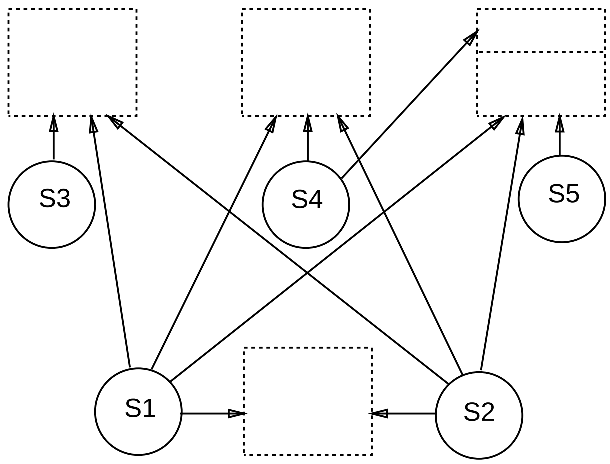

Figure 1: Trust Relationships

Figure 1 illustrates the trust relationships that can exist between subjects. Circles represent subjects, boxes represent collections of objects, and arrows indicate an access right in the protection domain from a subject to an object. These are the trust relationships between two subjects:

more trusted The more trusted subject is able to modify all objects belonging to the less trusted subject. The protection domain of the more trusted subject is a superset of the protection domain of the other subject. S1 and S2 are more trusted than the other subjects in the figure. They could be two parts of the kernel of an operating system.

less trusted The converse of more trusted: the protection domain of the less trusted subject is a subset of the protection domain of the other subject.

partially trusted The partially trusted subject is able to modify some of the objects that belong to the other subject but not all. S5 partially trusts S4. They could be two applications that are working together and have some shared virtual memory.

mutually distrusted Mutually distrusted subjects do not trust each other with any of the other's objects. S3 and S4 mutually distrust each other. They could be two independent applications.

mutually trusted Mutually trusted subjects trust each other with all of each other's objects. They both have the same protection domain. S1 and S2 mutually trust each other.

When a subject is more trusted than another subject, a protection boundary is placed between them so the less trusted subject cannot modify the objects belonging to the more trusted subject. With this kind of a trust relationship, we say that the two subjects are at different *protection levels*.

Even though trusted subjects *can* modify the objects that belong to subjects that trust them, that does not mean they *will* modify the objects. Trust simply implies that one subject is willing to believe that the other subject will not modify more than it is supposed to, including that the trusted subject will abide by any compile-time or run-time enforcement of object encapsulation. The existence of trust relationships, however, is important to our analysis of splitting object-oriented hierarchies across protection boundaries.

3.3 Inheritance, Delegation, and Forwarding

Inheritance and delegation are two ways of looking at object-oriented implementation hierarchies. The *Treaty of Orlando* [LSU87] identifies several kinds of distinctions between inheritance and delegation; when we use the two terms in this paper, the distinguishing property that we imply is that of whether the sharing in the implementation hierarchy is *static* or *dynamic*. That is, inheritance specifies that the sharing pattern is staticly determined by the time an object is created, and delegation allows the sharing to be determined dynamically during run-time. C++ [Str85] and Smalltalk [GR83] are inheritance-based languages and Self [US87, CUL89] is a delegation-based language.

The way that delegation-based languages provide dynamic determination of the sharing pattern is by giving each level of the implementation hierarchy a separate object, and forwarding unrecognized methods to the parent object. The parent object can be changed at any time by choosing to forward methods to a different object.

True delegation is more than just forwarding a method call to the parent object [Lie86]. In true delegation (and inheritance), if a child overloads a method that its parent invokes from a different method, the child's version of the method will be used. That is, the method gets invoked on the original object. On the other hand, if method calls are simply forwarded to the parent object, the parent would only invoke its own version of the method. For example, suppose a parent defines a method M1 that invokes another of its methods M2, and its child redefines method M2. When a call to M1 comes to the child, it will forward the call to its parent. With simple forwarding, the parent will invoke its own M2, but with delegation, the parent will invoke the child's M2. Delegation-based languages implement this by sending a reference to the original child object along with method calls, and always beginning with the child object when invoking methods.

4 Analysis

We now turn to our analysis of splitting object-oriented hierarchies across protection boundaries. We will address the issues related to implementation hierarchies and to type hierarchies separately.

These are the questions that we need to address:

Implementation hierarchies

1.1. What is the protection afforded to the portions of an object that correspond to different levels of an implementation hierarchy that spans protection boundaries? That is, should the portions of the object that belong to levels of the implementation hierarchy on different sides of protection boundaries be protected at one protection level or at different protection levels?

1.1.1. If at one level, should the object be placed at the protection level of the more trusted portion of the hierarchy or the less trusted level? What are the problems with placing the object at one protectoin level? Are there situations in which placing the object at one level is acceptable and others in which it is unacceptable? If so, what are the distinguishing factors?

1.1.2. If at different protection levels, what additional problems are caused? Should the forwarding of method calls between the objects at different levels be true delegation or simple forwarding?

1.2. Does it make a difference if a child in the implementation hierarchy does not trust its parent?

Type hierarchies

2.1. What are the concerns when polymorphism in a type hierarchy, with types and subtypes on different sides of protection boundaries, results in a method call to a subject that is not trusted? Should a subject be able to control whether an untrusted subtype is able to take the place of a trusted type?

2.2. What are the concerns when a type that is available across protection boundaries results in a method call from an untrusted subject? What precautions should be taken by the called subject?

2.2.1. Should an untrusted subject be allowed to invoke methods on all of the objects belonging to the called subject?

2.2.2. Should an untrusted subject be allowed to invoke all of the methods on the objects it has the right to invoke methods on, or should some of the methods be restricted from use by untrusted subjects?

2.2.3. What precautions should be taken when an untrusted subject attempts to pass in parameters that refer to other objects?

2.3. Should a subject be able to create or delete objects that belong to a subject that does not trust it?

2.4. Are the concerns different if the method calls are between mutually distrustful or partially distrustful subjects?

These questions form an outline of our analysis.

4.1 Implementation Hierarchy Split Across Protection Boundaries

What happens when an implementation hierarchy is split across protection boundaries? Not only code is shared in an object-oriented implementation hierarchy. Each level of the hierarchy may also add data members to the definition of objects. The data members added for a particular level are primarily intended to be accessed and modified by the code for that level of the hierarchy, although some languages make exceptions and allow code further down the hierarchy to modify the data. (C**++** even goes so far as to allow parts of some objects, the **public** parts, to be modified by any subject.)

4.1.1 Unified vs. Split Objects

Should the portions of an object belonging to different levels in an implementation hierarchy be protected at one level or should the object be split so it can be protected at different levels? Languages such as C++ and Smalltalk place objects in contiguous memory. If protection hardware is not fine-grained enough to distinguish between portions of an object, it is difficult to protect portions of an object at different levels.

Objects at a Single Protection Level Assume that an entire object must be in contiguous memory and that an object can only be protected by hardware at a single protection level. Consider the case of a two-level hierarchy where the parent subject is more trusted than the child subject.

At what protection level should the object then be placed? Clearly, placing the entire object at the parent's protection level is not appropriate because the child would be prevented from manipulating its own portion of the object. Therefore the object should be placed at the child's protection level.

What are the implications of placing the object at the child's lower protection level? The parent subject cannot trust that the child has left the parent's portion of the object intact because the child has complete access to the object. This is acceptable if one of the following special cases is true:

1. The parent has not defined any data in the object. If there is no data, there is nothing for the child to modify.

2. The parent causes no side effects based on the data in the object that the child does not have the right to cause itself. For example, the parent might only modify the object itself, not other objects or other protected data that the parent has the ability to modify, such as "class" data. If a parent did cause side effects on protected data, the less trusted child could circumvent the protection by manipulating the parent's portion of the object which could cause the parent to modify the protected data to the child's advantage.

3. The parent validates the data that it has defined at the beginning of every method to ensure that any side effects that it will cause will not violate trust. For example, perhaps some data items will cause correct operation if they have any of a range of values. If that is so, the parent must check to make sure the data items are still within the correct range; it cannot trust that the data items have been left intact because the less trusted child may have modified them.

Few situations are covered by the first case because classes usually define data; however, it is easy for the compiler or run-time system to verify whether the parent has defined data. There may be many situations covered by the second case where the parent would cause no harmful side effects based on object data, but that is difficult to verify automatically. More situations will be covered by the third case, but it requires careful analysis by the programmer for each situation and is prone to error. Even the third case does not cover all situations: it may be that the data values were influenced by more trusted subjects and any modification to their value by a less trusted subject would cause incorrect operation. Also, if the child has concurrent threads of control, a different thread could modify the data during the execution of a parent method after the parent had verified the data.

What if the object is placed at the child's lower protection level, and the parent lowers its protection level to the child's level while operating on the object? That would prevent the parent from causing any side effects that the child could not cause itself. This is much like the second case above, and it verifies correct operation. It can be easily implemented by essentially copying the code for the parent subject to the level of the less trusted child subject. It is appealing for the cases in which it may be applied because no protection boundaries need to be crossed, and this improves performance. However, it is too restrictive for the general case because the parent might need to cause side effects at the elevated protection level, especially if the parent is an operating system service.

The same analysis applies to a multi-level protection hierarchy, not just a two-level hierarchy, at the point at which the trust level changes. The analysis can simply be applied iteratively.

Thus, placing the entire object at a single protection level is not appropriate for the general case in an implementation hierarchy where children subjects are less trusted than their parent subjects.

Objects Split Across Multiple Protection Levels Since placing an object at a single protection level for an implementation hierarchy split across protection levels is not appropriate for all cases, objects should be split across the protection levels as well. The portions of the object that belong to each level in the implementation are placed at the protection level of the subject they belong to. The child object then delegates methods it does not recognize to its parent object.

Note that the ability to dynamically determine the parent class is not the motivation for using delegation here. The motivation is simply to be able to split an object into separate portions. If the system does not allow dynamic determination of the parent, it may be technically considered an inheritance system even though the object is split. That does not make a difference for protection purposes.

What additional problems are caused by splitting an object across protection levels? Some inheritance-based languages allow child subjects to directly access the data members of parent objects. Clearly this cannot be allowed across protection boundaries. It is not usually allowed in delegation-based languages, and [Sny86] advocates that all access to parent instance variables should be through methods even when protection boundaries do not exist [Sny86].

Some languages also distinguish between methods that are callable only by child subjects and methods that are callable by all subjects, for example C++ with its **protected** and **public** methods. Unfortunately, this cannot be strictly enforced across protection boundaries because an extra child can be easily created that makes the protected methods available to other subjects. Language restrictions can still be used, but programmers of the more trusted subjects should not assume that enforcement of the restriction is strict.

Should true delegation be used between the child object and the parent object, or should simple forwarding be used? If true delegation were used, a method call by the parent object could result in a call back to the child object if the child overloaded a parent method. Since the child in this case is less trusted than the parent, it may be that forwarding is better so the parent does not invoke an untrusted method when it is expecting a trusted method. Method calls to untrusted subjects have to be handled carefully, as we will discuss further in Section 4.2.1.

4.1.2 Untrusted Parents

Up to this point we have only been analyzing the case when a parent is more trusted than a child. Does it make a difference if the child in the implementation hierarchy does not trust its parent? This could be the case if either the child and parent mutually distrust each other or if the child is more trusted than the parent. Essentially, it makes very little difference to the analysis; the result of needing to split the object across the protection boundaries is the same. If objects could not be split and the parent is less trusted than the child, the object would have to be placed with the child, and the roles reversed in the analysis. If the objects could be split and mutual distrust existed between the parent and the child, the object would have to be made accessible to both the parent and child, and they would both have to protect against unauthorized modification by the other by verifying the data on every method call.

Delegating a method to an untrusted parent would result in a method call to an untrusted subject. Again, those calls have to be handled carefully as we will discuss further.

4.2 Type Hierarchy Split Across Protection Boundaries

What happens when a type hierarchy is split across protection boundaries? A process crosses a protection boundary when it calls a method on an object that is at a different protection level. The discussion is divided into three parts: method calls to untrusted subjects, method calls from untrusted subjects, and method calls between mutually distrustful or partially distrustful subjects.

4.2.1 Calls to Untrusted Subjects

What are the concerns when a subtype defined by a less trusted subject can take the place of a trusted type? Object-oriented systems use type hierarchies to provide polymorphism. Subtypes in a hierarchy can transparently take the place of supertypes. Making a method call on an object of an untrusted type will cross the protection boundary to the level of the untrusted subject before the untrusted method is executed.

Should a subject be able to control whether an untrusted subtype can take the place of a trusted type? There are reasons why a subject may not want to allow particular object types to be replaced by less trusted subtype objects. For one thing, the method can not be trusted to do what it is supposed to do, and return values may need to be validated. The method may never return at all. Another potential problem is that a subject may want to pass references to data items that are not accessible to the untrusted subjects. Also, crossing protection boundaries take time, and that may not be acceptable in some cases.

In general, a system with a type hierarchy that can cross protection boundaries should allow subjects to control whether less trusted subtype objects can take the place of trusted type objects.

4.2.2 Calls From Untrusted Subjects

What are the concerns when a type that is available across protection boundaries results in method calls from untrusted subjects? The concerns that we will address in this section are valid whether or not a type-subtype hierarchy crosses the protection boundaries. The concerns are simply caused by the use of a type across a boundary.

What precautions need to be taken when an untrusted subject calls a method on an object? Precautions are necessary to prevent the untrusted subject from circumventing protection by taking advantage of the increased privileges of the subject that owns the object. Either the subject or a subject it trusts should verify that the following are true:

Method Calls on All Objects? The untrusted subject must first of all have the *capability* to invoke methods on that object. Since calls to methods at different protection levels first cross over to the object's protection level, untrusted subjects should not be allowed to call methods on all objects in a system. A subject will want to selectively give away the capability for untrusted subjects to access its objects. For example, referring back to Figure 1, suppose there are two untrusted subjects S3 and S4 that mutually distrust each other and a trusted subject S1 that is more trusted than both. In that case, S1 might give away the capability to access one object to S3 and give away the capability to access a different object to S4, but not give away the capability to access a single object to both untrusted subjects. Also, there may be objects that should not be accessible to any untrusted subject.

All Method Calls on Objects? The untrusted subject must also be invoking a method that is accessible across protection boundaries. There may be individual methods on the objects accessible across protection boundaries that should not be available to untrusted subjects. Programmers often want to have methods that are only used for internal purposes. Also, methods that are callable across protection boundaries must be written with the awareness that callers are not to be trusted: for example, parameters may need to be checked to ensure that they are within a valid range.

A system that supports method calls across protection boundaries should provide a mechanism so a programmer of a more trusted subject can control which methods can be called across boundaries. If a special mechanism does not exist, the programmer could introduce an intermediate object and subject that only exposes the methods to be callable across boundaries and forwards method calls to the real object, but that is inconvenient.

Parameters Referring to Other Objects If there are any parameters passed to the method that refer to other objects, the untrusted subject must have capabilities to invoke methods on those objects as well. If they did not but the called subject did, the called subject might use its elevated capabilities to invoke methods on the objects, violating trust.

In addition to having the capability to invoke methods on objects that it passes in, the untrusted subject must pass in objects of correct types. A correct type is either the type that the called subject expects or a subtype in the type hierarchy. If an object of an incorrect type were allowed to be passed through to the called subject, the methods that the called subject expects to be available on the object would not be available. In a language like C++ that has no run-time type checking, that could cause severe consequences; it could even cause the system to crash.

4.2.3 Creation and Deletion of Objects by Untrusted Subjects

Should a trusted subject be able to create or delete objects belonging to a subject that does not trust it? Clearly the creation and deletion of objects must to be under the control of the subjects that own them, but subjects should be able to selectively give away the capability for untrusted subjects to create or delete its objects. Often objects exist to provide services for the untrusted subjects, and the untrusted subjects need to be able to request more services and to relinquish them.

4.2.4 Calls Between Mutually or Partially Distrustful Subjects

Are the concerns different if the method calls are different between mutually or partially distrustful subjects? No, calls across protection boundaries between two subjects that mutually distrust each other simply have the problems of both the previous categories. The invoker of a method views the invocation as a call *to* an untrusted subject, and the owner of the method views the call as a call *from* an untrusted subject. Each subject protects itself against the other subject both when it calls and when it is called. Validations can either be done by the two subjects individually or by a mutually trusted subject.

Partially trusted subjects can be decomposed into their trusted and untrusted components. A call from a partially trusted subject only needs to be protected against the components of the calling subject that are not trusted, and likewise a call to a partially trusted subject only needs to protect against the components of the called subject that are not trusted.

5 The *Choices* Solution

This section describes a system that is object-oriented across protection boundaries. The system is a part of the *Choices* operating system, and has been designed using the principles discussed in the analysis Section 4. A prototype of this part of *Choices* has been developed, and a full implementation is in progress.

The two major constraints on this system are the use of the C++ language and the use of traditional processor hardware. *Choices* is written in C++ and a transparent C++ interface across protection boundaries is desired. *Choices* currently runs on National NS32332, Motorola 68030, and Intel 80386 processors, so the only hardware protection available is the memory management units and in the case of the Intel 80386, segmentation.

C++ does not make a distinction between the implementation hierarchy and the type hierarchy, which simplifies the design somewhat. The C++ language does type checking at compile time and it does not have any notion of the type of objects at run time; to get around that deficiency, *Choices* keeps track of the type hierarchy by use of first-class classes [IL90, MCK91].

5.1 Protection Model

The protection model that we have chosen is a partitioned rings of protection model. Each successive outer ring is less trusted than the further-in rings. Furthermore, each ring can be partitioned into mutually distrustful regions. In order to make the interface as transparent to C++ as possible, address spaces are shared across rings although rings cannot access the addresses of further-in rings. Partitions within a single ring share the same range of virtual addresses, but represent different memory in different address spaces.

Figure 2 shows an example with four rings of protection. A partition of one ring is called a *ring-partition*. The number in each ring-partition is the ring number, and the letters represent the partition identifier. The last letter in each identifier distinguishes a ring-partition from other ring-partitions that belong to the same partition of the next inner ring. The other letters identify the ring-partition in the next inner ring. Ring-partitions 3aba and 3abb both share the same range of virtual addresses and mutually distrust each other, and ring-partition 2ab is mapped into the address space of both of them and is more trusted than both of them.

These are the reasons why this model of protection was chosen:

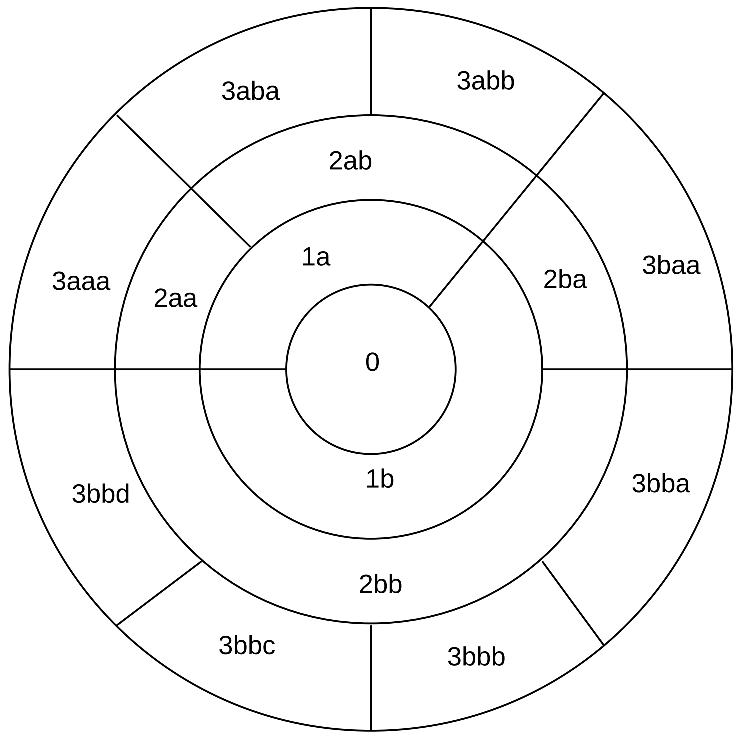

Figure 2: Four Partitioned Protection Rings Example

1. Due to the nature of the C++ language and our protection hardware, fine-grained protection is not practical. Instead, objects and subjects are grouped into relatively large portions that trust each other. Within a group of objects, no protection check overhead is incurred to invoke methods.

2. The traditional operating system approach of just two levels of protection, kernel and application, is inadequate. We want to be able to move shared services out of the kernel and into an intermediate level so we can have a minimal kernel. In addition, we want to support application developers who would like to have their own protected services shared across several higher-level applications.

   Layering the trust levels allow inner shared levels to use shared memory semantics rather than message passing semantics. That is, if the shared services were forced to be placed at the outer level in their own address spaces, messages have to be sent to them to gain access to their services. If they are at an inner trust level, they have direct access to the memory of the users they serve.

3. The partitioning of the outermost protection ring into separate address spaces that are mutually distrustful is common in operating systems. We chose to allow the intermediate rings to also be partitioned to enable a protected operating system research environment on a running system, and to allow higher-level shared application libraries to coexist on the same system without having to trust each other.

The implementation of the partioned rings of protection is processor dependent. Unfortunately, all of our processors only support two protection levels in the memory management units. However, multiple rings can still be achieved by using supervisor mode to implement the innermost ring and the user mode for the outer rings. On the National 32332 and Motorola 68030 processors, we use a separate first-level page table for each outer ring partition, and on the Intel 80386, we take advantage of the segmentation hardware to avoid the page table switch on ring crossings. For more details see [Dyk90].

5.2 Proxy Objects

Objects are represented outside of their ring-partitions with *proxy* objects [Rus90]. A proxy object gives its user the capability to invoke methods on the object it represents. The user of a proxy object is unaware of its existence and treats it as if it is the real object. The methods of a proxy object are actually stubs that trap to the kernel and translate the method calls into calls on the real object at the appropriate protection level.

A proxy object may exist between any two ring-partitions, but only inward calls from an outer ring to an inner ring (the most common case) can take advantage of the shared address spaces. That is, code in an outer ring can pass references to data in its own ring-partition and expect the inner ring to be able to access it. Outward calls from an inner ring to an outer ring, or cross calls across different address spaces cannot pass references to data in their own spaces and expect the called ring-partition to be able to access the data.

5.3 Proxify++

To assist with addressing the concerns discussed in Section 4, a tool called *Proxify++* has been created. This tool examines the definitions of C++ classes and generates information about each class. Proxify++ recognizes a new keyword called **proxiable** that identifies each method that can be called through a proxy. This keyword is stripped out by the C-preprocessor when the C++ compiler examines the class definitions, so no modifications are needed to the C++ compiler itself.

5.3.1 Calls from Untrusted Subjects

In order to do the checks on calls from untrusted subjects discussed in Section 4.2.2, Proxify++ generates a table of information about each proxiable method. The table lists the number of parameters and the types of parameters that refer to other proxiable objects, and the type of the return value of the method. The proxy trap handler uses this table to do the checks.

If a parameter refers to an object that is a proxy to another object, the proxy trap handler verifies that the caller has the right to use that proxy. If the proxy is to an object that is does not trust the caller, the trap handler checks the type of the real object to verify that it is also of the correct type or a subtype in the class hierarchy. (Types are determined by asking the object what its type is, so the type of an object that belongs to an untrusted caller cannot be believed). In addition, the trap handler strips off proxies to objects and, if the real object is not in the callee's ring-partition, a new proxy is allocated so the callee can use the object (recall that a proxy can only be used by a single ring-partition). When the method returns to the trap handler, and if the method returns a reference to a proxiable object, the trap handler allocates a proxy to that object for the caller before returning to the caller.

5.3.2 Untrusted Subtypes Replacing Trusted Types

Section 4.2.1 states that a mechanism should be provided to allow the programmer to control when an untrusted subtype can take the place of a trusted parent type. Proxify++ supports this by recognizing when the **proxiable** keyword is used to identify an entire class. If the class definition is marked as **proxiable**, then untrusted subtypes may take the place of a more trusted type. The proxy trap handler enforces this by only allowing an untrusted caller to pass in an object from its own space when the type of a parameter to the called method is of a class that is marked as **proxiable**.

5.3.3 Constructors and Destructors

If a constructor is marked as **proxiable**, then an object of that type may be created by a untrusted subject. This is the control mechanism on creation of objects discussed in Section 4.2.3. Proxify++ recognizes **proxiable** constructors specially and prepares a separate list of these constructors. This list is used to create constructor stubs to be linked with the untrusted subject code, and to create a table that the more trusted subject code uses to translate the stub calls into a call on the real constructor. This enables the programmer of the untrusted subjects to do a **new** on trusted objects as if they were local objects, and to get back a proxy to the real object.

Destructors on proxied objects are not called directly by users of the proxy objects. When the user of a proxy object calls the destructor, the proxy object itself is deleted instead. When the proxy object is deleted, the reference count on the real object is decremented by one, causing the real object to get deleted only if it is not being used by any other subjects.

5.3.4 Alternate Class Descriptions

Marking methods as **proxiable** is the mechanism used to control which methods are callable by untrusted subjects as discussed in Section 4.2.2. As mentioned above, Proxify++ enforces that by only listing information about those methods that are proxiable in the table that the proxy trap handler uses. Proxify++ also generates alternate class descriptions header files that the subjects at different trust levels include. These alternate class descriptions only list the methods that are callable, so compile-time checks can notify programmers of untrusted subjects if they accidentally try to call methods that are not proxiable. The alternate class descriptions also eliminate all implementation details, so they have the added advantage of preventing recompilation of code that uses the proxied objects as long as the interface stays the same.

5.3.5 Implementation Hierarchy Across Boundaries

As discussed in Section 4.1.1, the only general way to split an implementation hierarchy across protection boundaries is to split the object so the parts of the object that belong to each level in the hierarchy are protected at different levels. C++ allocates objects out of contiguous memory, so we make separate C++ objects. Child objects forward method calls that they do not handle to their parents objects.

As mentioned above, Proxify++ creates an alternate class description for the users of a class callable across protection boundaries. This class is of the same name as the original but only contains space for a pointer to a proxy to the real object. Subclasses inherit the data defined in the parent class, so space for the pointer to the proxy object is reserved in the subclasses.

Also as mentioned above, stubs are generated for class constructors using a list made by Proxify**++**. Each constructor stub creates a proxy to a new real object of that type and also detects whether it is invoked directly through a **new** or indirectly through the constructor of a subclass. If the constructor stub is called through a subclass constructor, the pointer space in the subclass object is filled in with the pointer to the proxy object.

In addition, the method table that Proxify**++** generates is used to make a stub for each proxiable method in the class. The stub simply invokes the corresponding method on the proxy object using the pointer saved in the object. The method stubs are there so they can be inherited by subclasses, making the forwarding transparent to the programmers of the subclasses. A destructor stub is also made to delete the proxy object; the subclass destructor calls that automatically.

Note that this scheme only forwards methods to the parent objects, it is not true delegation. That is, once the method has been forwarded to the parent object, and if the parent calls another method in the same class that the child has overloaded, the child's method will not be invoked. In order to simulate true delegation, a reference to the original object has to be passed as a parameter to the parent method. This is a recognized way to simulate delegation in C**++** [JZ91].

6 Conclusions

1. There are good reasons to split object-oriented hierarchies across protection boundaries, especially for an object-oriented operating system with an object-oriented application interface.

2. If an implementation hierarchy is split across protection boundaries, with a parent and a child on different sides of a boundary, the portion of a defined object belonging to the parent should be on the parent's side of the protection boundary and the portion of the defined object belonging to the child should be on the child's side of the protection boundary. The child object should then delegate or forward unrecognized method calls to its parent object.

3. If a type hierarchy is split across protection boundaries, subjects need to protect themselves both when calling untrusted subjects and when being called by untrusted subjects. Calling untrusted subjects occurs when an untrusted subtype in a type hierarchy takes the place of a trusted type due to polymorphism. Calls from an untrusted subject can come any time a type is made available to untrusted subjects. Called subjects should protect themselves to ensure that an untrusted subject only calls *methods* that it is allowed to call on *objects* it is allowed to call methods on, and to ensure that the untrusted subject does not try to overstep its privileges by passing parameters that refer to objects that it does not have access to or that are of an incorrect type.

4. Splitting of object-oriented hierarchies across protection boundaries can be accomplished practically, as we have demonstrated using C**++** and the *Choices* operating system.

Acknowledgements

Thanks to Ralph Johnson, Peter Madany, See-mong Tan, Aamod Sane, Panos Kougiouris, Bjorn Helgaas, and Ruth Dykstra for providing many helpful suggestions for revising this paper.

References

[BA$^+$89] J. M. Bernabeu-Auban et al. The architecture of Ra: a kernel for Clouds. In *Proceedings of the Twenty-Second Annual Hawaii International Converence on System Sciences*, pages 936–945 vol. 2. IEEE, January 1989.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF. In *Proceedings of OOPSLA '89*, pages 49–70. ACM SIGPLAN, October 1989.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128–140, October 1983.

[Dyk90] David W. Dykstra. Hardware Enforced Protection for Object-Oriented Operating Systems. Technical Report UIUCDCS-R-91-1666, University of Illinois at Urbana-Champaign, December 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[IL90] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[JZ91] Ralph E. Johnson and Jonathon M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 1991. Submitted for publication.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior. In *Proceedings of OOPSLA '86*, pages 214–223. ACM SIGPLAN, September 1986.

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of OOPSLA '87*, pages 17–34. ACM SIGPLAN, October 1987.

[LSU87] Henry Lieberman, Lynn Andrea Stein, and David Ungar. Of Types and Prototypes: The Treaty of Orlando. In *Addendum to the Proceedings of OOPSLA '87*, pages 43–44. ACM SIGPLAN, October 1987.

[MCK91] Peter W. Madany, Roy H. Campbell, and Panos Kougiouris. Experiences Building an Object-Oriented System in C++. In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[PS85] Jim Peterson and Avi Silberschatz. *Operating System Concepts*, chapter 11. Addison-Wesley, Reading, MA, 1985.

[R$^+$89] R. Rashid et al. Mach: a system software kernel. In *Digest of Papers: COMPCON Spring 89*, pages 176–178, San Francisco, California, 1989.

[Rus90] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM SIGPLAN, September 1986.

[Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1985.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*, pages 227–242. ACM SIGPLAN, October 1987.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pages 168–182. ACM SIGPLAN, October 1987.