An Approach to Safe Object Sharing

Ciarán Bryce & Chrislain Razafimahefa
Object Systems Group,
University of Geneva, Switzerland
bryce,razafima@cui.unige.ch

Abstract

It is essential for security to be able to isolate mistrusting programs from one another, and to protect the host platform from programs. Isolation is difficult in object-oriented systems because objects can easily become aliased. Aliases that cross program boundaries can allow programs to exchange information without using a system provided interface that could control information exchange. In Java, mistrusting programs are placed in distinct loader spaces but uncontrolled sharing of system classes can still lead to aliases between programs. This paper presents the *object spaces* protection model for an object-oriented system. The model decomposes an application into a set of spaces, and each object is assigned to one space. All method calls between objects in different spaces are mediated by a security policy. An implementation of the model in Java is presented.

KEYWORDS

Protection domains, access control, aliasing, sharing, Java.

1 Introduction

In the age of Internet programming, the importance of sound security mechanisms for systems was never greater. A host can execute programs from unknown network sources, so it needs to be able to run each program in a distinct *protection domain*. A program running in a protection domain is prevented from accessing code and data in another domain, or can only do so under the control of a *security policy*. Domains protect mistrusting programs from each other, and also protect the host environment. There are several aspects to protection domains: access control, resource allocation and control, and safe termination. This paper concentrates on access control.

There are several ways to implement protection domains for programs. Operating systems traditionally implement domains using hardware-enforced address spaces. The trend towards portable programs and mobile code has led nonetheless to virtual machines that enforce protection in software. One example in the object-oriented context is a guarded object [10]. In this approach, a guard object maintains a reference to a guarded object; a request to gain access to the guarded object is mediated upon by the guard. Another example of software enforced protection is found in Java [2], where each protection domain possesses its own name space (set of classes and objects) [17]; domains only share basic system classes. Isolation between Java protection domains is enforced by run-time controls: each assignment of an object reference to a variable in another domain is signaled as a type error (`ClassCastException`).

The difficulty in implementing protection domains in an object-oriented context is the ease with which object aliases can be created [12]. An object is aliased if there is at least two other objects that hold a reference to it. Aliasing is difficult to detect, and unexpected aliasing across domains can constitute a *storage channel* since information that was not intended for external access can be leaked or modified [15]. Aliasing between domains can be avoided by making their object sets disjoint. Any data that needs to be shared between domains is exchanged by value instead of by reference.

Partitioning protection domains into disjoint object graphs is cumbersome when an object needs to be accessible to several domains simultaneously. This is especially true if the object is mutable e.g., application environment objects, as this necessitates continuous copies of the object being made and transmitted. A more serious problem is that some system objects must be directly shared by domains, e.g., system-provided communication objects, and even this limited sharing can be enough to create aliases that lead to storage channels. For instance, there is nothing to prevent an error in the code of a guard object from leaking a reference to the guarded object to the outside world.

Techniques that control object aliasing are often cited as a means to enforce security by controlling the spread of object references in programs [12]. These techniques are fundamen-

OOPSLA '00, 10/00 Minneapolis, MN, USA
© 2000 ACM ISBN 1-58113-200-x/00/0010...$5.00

tally software engineering techniques; their goal is to enforce stronger object encapsulation. Security requires more than this for two main reasons. First, aliasing control techniques are often class-based: they aim to prevent all objects of a class from being referenced, though cannot protect selected instances of that class. Second, security constraints are dynamic in nature and aliasing constraints are not. One example is *server containment* [7], where the goal is to allow a server to process a client request, but after this request, the server must "forget" the references that it holds for objects transmitted by the client. The goal of server containment is to reduce the server's ability to act as a covert channel [15].

The crux of the problem is that once a reference is obtained, it can be used to name an object and to invoke methods of that object. We believe that naming and invocation must be separated, thus introducing *access control* into the language. Least privilege [23] is one example of a system security property that requires access control for its implementation. Least privilege means that a program should be assigned only the minimum rights needed to accomplish its task. Using an example of file system security, least privilege insists that a directory object not be able to gain access to the files it stores [7] in order to minimize the effects of erroneous directory objects. Thus, a directory can name file objects, but can neither modify nor extract their contents.

This paper introduces the *Object Spaces* model for an object-oriented system; Java [2] is chosen as the implementation platform. A space is a lightweight protection domain that houses a set of objects. All method calls between objects of different spaces are mediated by a security policy, though no attempt is made to control the propagation of object references between spaces. The model allows for safe and efficient object sharing. Its efficiency stems from the fact that copy-by-value of object parameters between domains is no longer needed. The model is safe in the sense that if ever an object reference is leaked to a program in another space, invocation of a method using that reference is always mediated by a security policy. In addition, access between objects of different spaces is prohibited by default; a space must be explicitly granted an access right for a space to invoke an object in it, and the owner of this space may at any time revoke that right.

The implementation of the model is made over Java and thus no modifications to the Java VM or language are needed. Each application object is implicitly assigned a space object. An object can create an object in another space, though receives an indirect reference to this object: A bridge object is returned that references the new object. Our implementation assures the basic property that access to an object in another space always goes through a bridge object. Bridges contain a security policy that mediate each cross-space method call.

A space is a lightweight protection domain as it only models the access control element of a domain; thread management and resource control issues are not treated since these require modifications to the Java virtual machine.

The remainder of this paper is organized as follows. Section 2 outlines the object space model and explains its design choices. Section 3 presents a Java API for the model and examples of its use. Section 4 describes the implementation of the model over Java 2 and gives performance results. Section 5 reviews related work and Section 6 concludes.

2 The Object Space Model

The basis of the object space model is to separate the ability to name an object from the ability to invoke methods of that object. This is done by partitioning an application's set of objects into several object *spaces*. A space contains a set of objects, and possibly some children spaces. Every object of an application is inside of exactly one space. An object may invoke any method of any object that resides in the same space. Method invocation between objects of different spaces is mediated by an application-provided security policy.

We start in Section 2.1 with an overview of the object space model. A formal definition of the model is given in Section 2.2, and we present some examples of how the model addresses well-known protection problems in Section 2.3.

2.1 Model Overview

The set of spaces of an application is created dynamically. On application startup, the initial objects occupy a `RootSpace`. Objects of this space may then create further spaces; these new spaces are *owned* by the `RootSpace`. These children spaces may in turn create further spaces. For each space created by an object, the enclosing space of the creator object becomes the owner space of the new space. The space graph is thus a tree under the ownership relation.

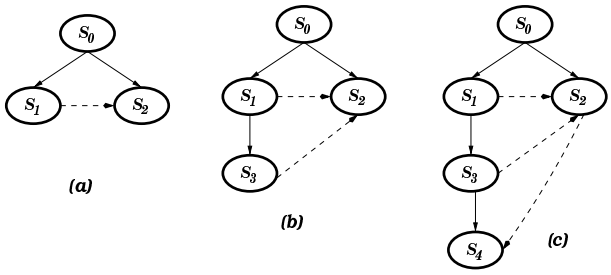

Figure 1 Ownership and authorization relations on spaces.

An object in a space may invoke methods of an object in another space if the second space is owned by the calling object's space. When the calling object is not in the owner space of the called object, then the calling space must have been explicitly granted the right to invoke methods of objects in the second space by the owner of the second space.

The set of spaces is organized hierarchically because this models well the control structure of many applications [22]. Typically, a system separates programs into a set of protection domains since they must be protected from each other. A program's components may also need to be isolated from one another, since for example it may use code from different libraries. In the object space model, a program in a space can map its components to distinct spaces.

We decided not to include space destruction within the space model, because of the difficultly of implementing this safely and efficiently. A space, like an object, can be removed by the system when other spaces no longer possess a reference to it (or to the objects inside of it).

An object may create other objects in its space without any prohibition. A space may also create objects in its children spaces – this is how a space's initial objects are created.

A space s_1 may not create objects in another space s_2 if s_1 is not the parent of space s_2. The goal of this restriction is to prevent s_1 from inserting a Trojan Horse object into space s_2 that tricks s_2 into granting s_1 a right for s_2.

Figure 1 gives an example of how the space graph of an application develops, and how access rights are introduced. Ownership is represented by arrows and access rights are represented by dashed line arrows. When the system starts, `RootSpace` (represented by space s_0 in the figure) is created. In this example, space s_0 creates two children spaces, s_1 and s_2, and permits objects of space s_1 to invoke objects of space s_2. Objects of space s_0 can invoke objects of spaces s_1 and s_2 by default since space s_0 is the owning space of s_1 and s_2.

In Figure 1b, space s_1 creates a child space s_3, and grants it a copy of its access right for s_2. Only space s_1 possesses a right on S_3 though. In Figure 1c, s_3 has created a child space s_4, and granted space s_2 an access right for s_4. This means that objects of spaces s_2 and s_3 can call objects in space s_4.

The access control model could be seen as introducing programming complexity because an object that possesses a reference is no longer assured that a method call on the referenced object will succeed. This is also the case for applets in Java where calls issued by applets to system objects are mediated by `SecurityManager` objects that can reject the call.

2.2 Formal Definition

The state of the object space protection system is defined by the triple

$$[\mathcal{S}, \mathcal{O}, \mathcal{R}]$$

where $\mathcal{S}$ is the set of spaces, $\mathcal{O}$ is the ownership relation and $\mathcal{R}$ represents the space access rights. Let $\mathcal{N}$ denote the set from which space names are generated. $\mathcal{S}$ is a subset of $\mathcal{N}$ ($\mathcal{S} \subset \mathcal{N}$); names of newly created spaces are taken from $\mathcal{N} \backslash \mathcal{S}$. $\mathcal{O}$ is a relation on spaces ($\mathcal{N} \times (\mathcal{N} \cup \{\texttt{null}\})$); we use $s_1\ \mathcal{O}\ s_2$ to mean that space s_1 "is owned by" space s_2. The expression $s_1\ \mathcal{O}\ s_2$ evaluates to true if $(s_1, s_2) \in \mathcal{O}$. The `null` value in the definition of $\mathcal{O}$ denotes the owner of `RootSpace`. Finally, $\mathcal{R}$ is a relation of type ($\mathcal{N} \times \mathcal{N}$); $s_1\ \mathcal{R}\ s_2$ means that space s_1 possesses the right to invoke methods in space s_2.

A space always has the right to invoke (objects in) owned spaces: $s_1\ \mathcal{O}\ s_2 \Rightarrow s_2\ \mathcal{R}\ s_1$. Further, an object can always invoke methods on objects in the same space as itself: $\forall s{:}\mathcal{S}.\ s\ \mathcal{R}\ s$.

We now define the semantics of the object space operations. The `grant`(s_0, s_1, s_2) operation allows (an object of) space s_0 to give (objects of) space s_1 the right to invoke objects of space s_2. For this operation to succeed, s_0 must be the owning space of s_2 or, s_0 must already have an access right for s_2 and be the owner of s_1. The logic behind this is that a parent space decides who can have access to its children, and a space may always copy a right that it possesses to its children spaces.

`grant`$(s_0, s_1, s_2)[\mathcal{S}, \mathcal{O}, \mathcal{R}] \triangleq$
  if $(s_2\ \mathcal{O}\ s_0) \quad \vee \quad (s_0\ \mathcal{R}\ s_2 \wedge s_1\ \mathcal{O}\ s_0)$
    then $[\mathcal{S}, \mathcal{O}, (\mathcal{R} \cup \{(s_1, s_2)\})]$
    else $[\mathcal{S}, \mathcal{O}, \mathcal{R}]$
  fi

Access rights between spaces can also be revoked. A space s can revoke the right from any space that possesses a right for a space owned by s. When a space loses a right, then all of its descendant spaces in the space hierarchy implicitly lose the right also. This is because a space might have acquired a right, granted a copy of that right to a child space, and have the child execute code on its behalf that exploits that access right. The operation `revoke`(s_0, s_1, s_2) is used by (an object of) space s_0 to remove the right of (objects of) space s_1 to access objects of space s_2. The operation is the reverse of grant. $\mathcal{D}(s)$ denotes the set of descendant spaces of s in the space tree; as said, these spaces also have their right revoked. $\mathcal{D}(s) \triangleq \{\ s' \mid (s'\ \mathcal{O}\ s) \vee (\exists s''.\ s''\ \mathcal{O}\ s \wedge s' \in \mathcal{D}(s''))\}$. This operation does not allow an owner to lose its right to access a child space.

`revoke`$(s_0, s_1, s_2)[\mathcal{S}, \mathcal{O}, \mathcal{R}] \triangleq$
  if $(s_2\ \mathcal{O}\ s_0) \quad \vee \quad (s_1\ \mathcal{O}\ s_0 \wedge s_1\ \mathcal{R}\ s_2))$
    then $[\mathcal{S}, \mathcal{O}, (\mathcal{R} \setminus \{(s_1, s_2), (s_j, s_2) \mid s_j \in \mathcal{D}(s_1)\})]$
    else $[\mathcal{S}, \mathcal{O}, \mathcal{R}]$
  fi

A space s may create a new space for which it becomes the owner. The new space is given a fresh name s' ($s' \notin \mathcal{S}$).

`create`$(s)[\mathcal{S}, \mathcal{O}, \mathcal{R}] \triangleq$
  $[(\mathcal{S} \cup \{s'\}), (\mathcal{O} \cup \{s', s\}), (\mathcal{R} \cup \{(s, s'), (s', s')\})]$

On system startup, the `RootSpace` s_0 is created. The initial system state is thus $[\{s_0\}, \{(s_0, \texttt{null})\}, \{(s_0, s_0)\}]$.

Finally, each time a method call is effected in the system, an access control decision is made using the `checkAccess` operation to determine if space s_0 may invoke methods on objects in space s_1:

`checkAccess`$(s_0, s_1)[\mathcal{S}, \mathcal{O}, \mathcal{R}] \triangleq s_0\ \mathcal{R}\ s_1$

In Figure 1a, $\mathcal{S}$ is $\{s_0, s_1, s_2\}$, and $\mathcal{O}$ is $\{(s_1, s_0), (s_2, s_0), (s_0, \texttt{null})\}$. $\mathcal{R}$ contains $\{(s_0, s_1), (s_0, s_2), (s_1, s_2)\}$ as well as the pairs (s_i, s_i) for each s_i in $\mathcal{S}$.

In Figure 1b, the sets $\{s_3\}$, $\{(s_3, s_1)\}$ and $\{(s_3, s_2), (s_1, s_3\}$ are included in the three elements of the system state. In Figure 1c, s_3 has created a child space s_4, and granted space s_2 an access right for s_4. Thus, $s_3\ \mathcal{R}\ s_4$, $s_2\ \mathcal{R}\ s_4$, and $s_4\ \mathcal{O}\ s_3$.

2.3 Examples

We give some brief examples of how the model can be exploited. More detail is added in Section 3 after a Java API for the model has been presented.

2.3.1 Program Isolation

Today's computer users cannot realistically trust that the programs they run are bug or virus free. It is crucial then that the host be able to run a non-trusted program in isolation from its services. This means that client programs not be able to communicate with the services, or that they can

only do so under the control of a security policy that decides whether each method call from a program to the servers is permitted.

The basis to achieving isolation using the object space model is shown in Figure 2a. The `Root` space creates a space (`Server`) for the host service objects, and a `client` space for each of the user programs. The host's security policy is placed in the `Root` space, and controls whether the user programs may access the services using the `grant` and `revoke` operations. The code of this example is given in Section 3.2.

In comparison, the ability to isolate programs in this fashion is awkward in Java using loader spaces. In Java, each program is allocated its own class loader [17], which is responsible for loading versions of the classes for the program. An object instantiated from a class loaded by one loader is considered as possessing a distinct type to objects of the same class loaded by another loader. This means that the assignment of an object reference in one domain to a variable in another domain constitutes a type error (`ClassCastException`). This model is inconvenient for client-server communication, since parameter objects must be serialized (transferred by value).

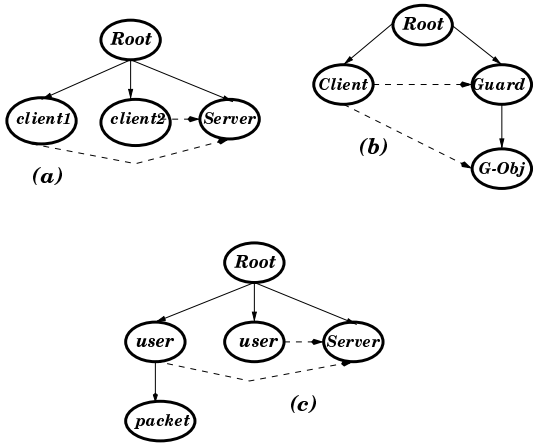

Figure 2 Examples with spaces.

2.3.2 Guarded Objects

A common example of a mechanism for controlled sharing is guarded objects. We consider two different versions of the guarded object notion here: a Java 2 version [10] and a second more traditional version [23].

In Java, a *guard* object contains a *guarded object*. On application startup, only the guard object possesses a reference to the guarded object. This object contains a method `getObject()` which executes a `checkAccess` method that encapsulates a security policy, and returns a reference to the guarded object if `checkAccess` permits. This mechanism is useful in contexts where a client must authenticate itself to a server before gaining access to server objects, e.g., a file server that authenticates an access (using `checkAccess`) before returning a reference to a file.

An implementation of the more traditional guarded object notion would never return a reference to the guarded object. Rather, each method call to the guarded object would be mediated by the guard, which would then transfer the call if `checkAccess` permits, and then transfer back the result object.

Both of these approaches have weaknesses however. In the Java version, there is no way to revoke a reference to the guarded object once this reference has been copied outside of the guard object. Revocation is needed in practice to confine the spread of access rights in a system. The problem with the traditional notion of guarded object is that a method of the guarded object may return an object that itself contains a reference to the guarded object. This clearly undermines the role of the guard.

Figure 2b illustrates how guarded objects are implemented in the object space model. A guard object is placed in its own space (`Guard`), and the guard creates a child space (`G-Obj`) in which it instantiates the guarded object. Space `Guard` controls what other spaces can access `G-Obj`. To implement the traditional version of the guarded object paradigm, the guard would never grant access to `G-Obj` to other spaces. To implement the Java version, `getObject()` of the guard grants the `Client` space access to `G-Obj` in the event of `checkAccess` succeeding. The guard can at any time revoke access, which is something that cannot be done in traditional implementations, so even if a reference is leaked, a `grant` operation must also be effected for access to the guarded object to be possible. We give the code of this example in Section 3.2.

Guards are required in all systems for stronger encapsulation. The goal of encapsulation is be able to make an object public - accessible to other programs - without making its component objects directly accessible. This is often a requirement for kernel interface objects, since a serious error could occur if a user program gained hold of a reference to an internal object. An example of this is the security bug that allowed an applet to gain a reference to its list of code signers in JDK1.1.1, which the applet could then modify [4]. By adding signer `Identity` objects to this list, the applet could inherit the privileges associated with that signer.

```
private Vector /* of Identity object */ signers;
...
public Vector getSigners(){
  return signers;
}
```

The JDK actually used an array to represent the signers [4]; arrays require special treatment in the object space model, as will be seen in Section 4. This example also shows that declaring a variable as `private` is not enough to control access to the object bound to that variable. In the object space model, stronger encapsulation of internal objects (e.g., `signers`) is achieved by having these objects instantiated in a space (`G-Obj`) owned by the kernel interface objects' space (`Guard`).

2.3.3 Server Containment

Servers are shared by several client programs. In an environment where mistrusting programs execute, a server should not be allowed to act as a covert channel by holding onto references to objects passed as parameters in a service request and then subvertly passing these references to a third party.

Security requires that a server be *contained* [7] - the server can no longer gain access to any object after the request has been serviced. A schema for this using spaces is shown in Figure 2c. The `packet` space is for objects that are being passed as parameter. The server is granted access to these objects for the duration of the service call. This access is revoked following the call. Server containment requires the ability to isolate programs from one another, and the ability to revoke rights on spaces. As such, it uses features also present in the preceding two examples.

3 The Object Space API

This section describes the classes of the object space system API, and then presents an example of its use.

3.1 Basic Java Classes

There are only three classes that an application requires to use the object space model: `IOSObject`, `Space` and `RemoteSpace`. We briefly describe the role of each, before presenting an implementation over Java 2 in Section 4.

The class `IOSObject`[1] describes an object that possesses an attribute `Space`. This attribute denotes the space that houses the object. Not all objects of an application need inherit from `IOSObject`; the only requirement is that the first object instantiated within each space be a subclass of `IOSObject` since in this way, there is at least one object from which others may obtain a pointer to their enclosing `Space` object. The API of `IOSObject` is the following:

```
public class IOSObject
{
  protected Space mySpace;
  public IOSObject();
  public final Space getSpace();
}
```

The `getSpace` method enables an object to get a handle on its enclosing space from an `IOSObject`.

The `Space` class represents an object's handle on its enclosing space. Handles on other spaces are instances of the `RemoteSpace` class. `SpaceObject` is an empty interface implemented by both `Space` and `RemoteSpace`.

```
public final class Space implements SpaceObject
{
  public static RemoteSpace
                    createRootSpace(IOSObject iosObj);
  protected Space();
  public RemoteSpace createChildSpace();
  public void grant( SpaceObject sourceSpace,
                     SpaceObject targetSpace );
  public void revoke( SpaceObject sourceSpace,
                      SpaceObject targetSpace );
  public Object newInstance( String className,
                         RemoteSpace target );
  public RemoteSpace getParent();
  protected void setParent(Space parent);
  static boolean checkAccess( Space protectedObjSpace,
                              Space callerSpace )
}
```

[1] "IOS" comes from "Internet Operating System", which is the name of the project in which the space model was developed.

Recall that spaces are organized in a hierarchy: the root of the hierarchy is created with the static method `createRootSpace`. This method returns a `RemoteSpace` object, and the system ensures that this method is called only once. The `createChildSpace` method creates a child space of the invoking object's space. The `grant` and `revoke` methods implement the access control commands of the model (see Section 2.2). The space of the object that invokes either operation is the grantor or revoker space of the operation.

The `newInstance` method creates an object within the specified space. This is how objects are initially created inside of a space. The implementation verifies that the class specified extends `IOSObject`, so that subsequently created objects have a means to obtain a reference to their `Space`. Further, only a parent space may execute this method. The goal is to prevent spaces injecting malicious code into a space in the aim of forcing that space to execute a `grant` that would allow the malicious object space gain an access right to the attacked space. The `setParent` method is executed by the system when initiating a space; the `checkAccess` method that consults the security policy. These two methods are only used by the object space model implementation.

The third of the classes in the object space API is `RemoteSpace`:

```
public final class RemoteSpace implements SpaceObject
{
  public RemoteSpace(Space sp);
}
```

This represents a handle on another space. The only user-visible (`public`) operation is the constructor that allows an object to generate a remote space pointer from the pointer to its enclosing space. This enables a space to transfer a pointer to itself to other spaces and thus allow other spaces to grant it access rights.

It is important to note that an object can only possess a `Space` reference to its enclosing space, and never to other spaces. In this way, the system assures that an object in one space does not force another space to grant it an access right since the `grant` and `revoke` operations are only defined in `Space`, meaning that the system can always identify the space of the invoking object and thus authorize the call. Note also that the `Space` and `RemoteSpace` classes are `final`, meaning that a malicious program cannot introduce Trojan Horse versions of these classes into the system.

3.2 Example code extracts

The first example continues the program isolation discussion of Section 2.3.1, and is taken from a newspaper system for the production and distribution of articles [19]. Here we concentrate on a program that compiles an article. For security reasons, we wish to isolate this program from the rest of the system - in particularly from the `Storage` and graphical `Editor` objects. This requires being able to meditate all method calls between the `client` program and the services. These security requirements are summarized in Figure 3.

The is a typical example of the need to isolate user programs from the rest of the system. Section 4 gives a performance comparison of an implementation of this example

using Java loader spaces with copy-by-value semantics, and the object spaces implementation.

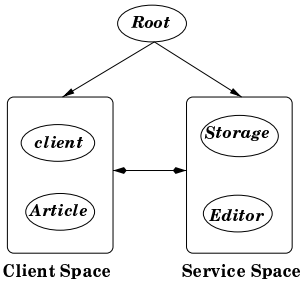

Figure 3 The article packager example.

In the code below, `Root` is the application start-up program that creates two object spaces, and instantiates the objects in each domain. This is the only class of the application that uses the object space model API methods. The `Editor` class uses several `Swing` components to offer a front-end user interface; this exchanges request messages and events with the client program.

```
public class Root extends IOSObject{

  public void start(){
    // Create the client and server Spaces
    RemoteSpace child1 = mySpace.createChildSpace();
    RemoteSpace child2 = mySpace.createChildSpace();

    // Allocate access rights;
    mySpace.grant(child1,child2);
    mySpace.grant(child2,child1);

    // Create the services ....
    Editor E = (Editor)mySpace.
                  newInstance("GUI.Editor", child1);
    Storage A = (Storage)mySpace.
                  newInstance("Kernel.Storage", child1);
    // .... and create the client
    client c = (client)mySpace.
                  newInstance("Kernel.client", child2);

    // Start things running
    Editor ed = E.init(c); A.init();c.init(ed, A);
  }

  public static void main( String[] args ){
    Root R = new Root();
    RemoteSpace space1 = Space.createRootSpace(R);
    R.start();
    }
  }
```

In this example, the application `main` starts an instance of `Root`. This creates two child spaces, `child1` and `child2`, grants a right to each space to invoke object methods in the other. An `Editor` object and a `Storage` object are created in space `child1` and the client program is installed in `child2`. The editor is given a reference to the client object (so that it can forward events from the GUI interface) and the client is given a reference to the two service objects.

The `Root` class here is almost identical to that used in an implementation of the guarded object model of Figure 2b. A `Guard` object and `client` are created in distinct spaces. In the extract of this example below, the guard has a string (of class `IOSString`) as guarded object. The class `IOSString` is our own version of `String`; the motivation for this class is given in Section 4.

```
import InternetOS.*;
import InternetOS.lang.*;

public class Guard extends IOSObject{
  IOSString guardedObject;
  RemoteSpace guardedSpace;

  public void init() {
    guardedSpace = mySpace.createChildSpace();
    guardedObject = (IOSString)mySpace.
           newInstance("InternetOS.lang.IOSString",
                       guardedSpace);
    guardedObject.
         set(new IOSObject("The secret text."));
    }

  public Object getObject(IOSString password,
                          RemoteSpace caller){
    // if checkAccess(password) {...}
    mySpace.grant(caller, guardedSpace);
    return guardedObject;
  }
}
```

The `Guard` object has an `init` method that is called by the `Root`. This method creates a child space (`guardedSpace`), instantiates the guarded object in this space, and initializes it using its `set` method (defined in `IOSString`). The role of the guard is to mediate access requests on `getObject`. A client must furnish a password string and the guard verifies the password using the guard object's `checkAccess` method. If the check succeeds, the guard grant's access to the client space and returns the object reference.

```
public class client extends IOSObject{

 Guard G;

 public void init(Guard G){
   this.G = G;
   IOSString password =
       new IOSString("This is my password");

   IOSString s =
     (IOSString)G.getObject(password,
                            new RemoteSpace(mySpace));
   System.out.println("String is "); s.print();
 }
}
```

The `client` is a program that requests access from the guard by supplying the password and a pointer to its space to `getObject`.

4 The Object Space Implementation

In this section we describe the implementation of our model. We first describe the notion of *bridge*, which is the mechanism that separates spaces at the implementation level.

For portability and prototyping reasons, the current implementation is made over the Java 2 platform, so no modifications to the virtual machine or language were made. A future implementation could integrate the model into the JVM; in this way other aspects of protection domains such as resource control and safe termination can also be treated.

We begin in Section 4.1 by describing the basic role of bridge objects. Section 4.2 describes how they are interposed on method calls between spaces, and Section 4.3 explains how bridge classes are generated. Section 4.4 describes in more detail how the object space model interacts and in some cases conflicts with features of the Java language. Section 4.5 presents a performance evaluation of the implementation.

4.1 Bridge Objects

So far, we have seen that objects belong to spaces and that they interact either locally inside the same space or issue method calls across space boundaries. Interactions between objects of different spaces are allowed only if the security policy permits.

To implement the object space model, a bridge object is interposed between a caller and a callee object when these are located in different spaces. If the caller has the authorization to issue the call, then the bridge forwards the call to the callee, otherwise a security violation exception is raised. We call the callee the *protected object*, since it is protected from external accesses by the bridge, and we use the term *possessor* to refer to the caller. This is illustrated in Figure 4, where real references are denoted by arrows; the dashed line arrow denotes a protected reference whose use is meditated by a security policy. The security policy is represented by an access matrix accessible to all bridges; this encodes the authorization relation $\mathcal{R}$ defined in Section 2.2.

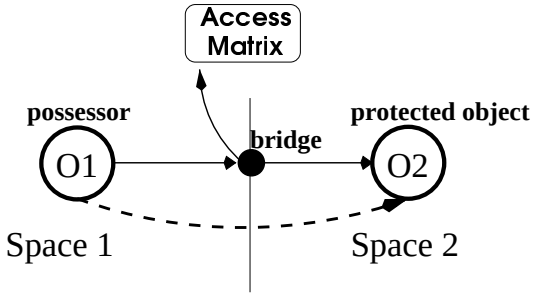

Figure 4 A bridge object interposed between object spaces.

Bridges are hidden from application programmers. They are purely an implementation technique and do not appear in the API. Assuming security permits, a program behaves as if it has a direct reference to objects in remote spaces. The main exception to this rule are array references which always refer to local copies of arrays, even if the entries in an array can refer to remote objects. We return to the question of arrays in Section 4.4. Consider class `Root` in Section 3.2; its `start` method contains the call `c.init(ed, A)` to transfer references for the editor and storage objects to the client program. The three objects are all in a different space to the `Root` object; the references used are in fact references to bridge objects even though the programmer of the `Root` class does not see this.

Bridges are implemented using instances of Java `Bridge` classes, where `Bridge` is an interface that we provide. Each class C has a bridged class B_C constructed for it. The interface of B_C is the same as that of C. Further, B_C is defined as a subclass of C, which makes it possible to use instances of B_C (i.e., the bridges) anywhere that instance of C are expected.

The role of a bridge is three-fold:

1. It verifies that the caller can issue a call to the protected object. To be more precise, this results in verifying that the space of the caller can access the space of the callee according to the security policy, consulted using the `checkAccess` method of class `Space`. This method is shown in Figure 7.

2. It forwards the request from the possessor to the protected object, if the possessor has the right to access the protected object.

3. It ensures that objects exchanged as parameters between the possessor and the protected object do not become directly accessible from outside their spaces.

   The protection model is broken if an object obtains a direct reference to an object in another space (a reference is direct if no bridge is interposed between the objects). This can happen during a call if the arguments are directly passed to the callee. Therefore, a bridge can be interposed between the callee and the arguments it receives. Similarly, this wrapping can occur on the object returned from the method call on the protected object. To avoid reference leaks exploiting the Java exception mechanism, bridges are also responsible for catching exceptions raised during the execution of the protected object's method, and for throwing bridged versions of the exceptions to the caller.

4.2 Interposition of Bridges

Bridges are introduced into the system when an application object creates an object in a child space using the `newInstance` method. This method is furnished by the system (in `Space`) and cannot be redefined by users since it is defined in a `final` class. In addition to creating the required object and assigning it to the space, `newInstance` creates a bridge for the new object. A reference to this bridge is returned to the object that initiates the object creation, making the new object accessible to its creator only through the bridge. For instance, in the example 3.2, references `E, A, c` point to a bridge instead of pointing directly to an `Editor`, a `Storage` or a `client` object since these objects are created using `newInstance`.

The other way that bridge objects appear in the system is during cross-domain calls where the need for protection for arguments and returned objects arise. By default, when a reference to an object is passed through a bridge, a bridge

object for the referenced object is generated in the destination space. Nevertheless, if a bridge object for the protected object already exists in the destination space, then a reference to this bridge is returned instead of having a new bridge object generated. This is implemented using a map that maps protected object and space pairs to the bridge used by objects in that space to refer to the protected object. An advantage of this solution is that the same bridge is shared among objects of the same space referring to the same protected object. However, if objects reside in different spaces, they cannot share the same bridge. A second advantage of this is that the time needed to consult the bridge cache is inferior to the time needed to generate a new bridge object. A final advantage concerns equality semantics: the == operator applied to two bridge references to the same protected object always evaluates to true.

However, there are cases where bridge interposition is not necessary. For instance, if an object creates another object which resides in the same space as its creator, then a direct reference to the new object is allowed. This is the case when the Java `new` operator is used, i.e., an object created with `new` belongs to the same space as its creator and direct invocations are allowed. Further, if a bridge receives a bridge reference as an argument to a call and observes that the protected object of that bridge is actually in the destination space, then the protected object reference is returned in place of the bridge.

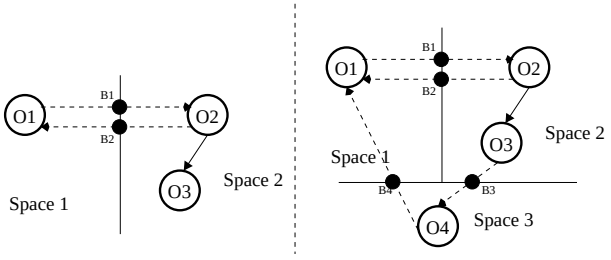

Figure 5 The creation of bridges between spaces.

An example of the interposition of bridge objects is shown in Figure 5. Space 1 possesses an object O1 that creates an object O2 in Space 2 using `newInstance`; a bridge B1 is created for this reference. O1 then passes a reference for itself to O2; B1 detects that this reference is remote and creates a bridge B2. O2 creates an object O3 locally in its space using `new`, and obtains a direct reference to it. O3 then creates an object O4 in Space 3, and a bridge B3 is created. Finally, O2 passes a reference for O1 to O4 (via O3); a new bridge B4 is created for this that notes that the reference is from Space 3 to Space 1.

The only exception to the above is the exchange of `RemoteSpace` objects. Objects of this type can be freely passed to foreign spaces without bridge intervention. Consequently, direct access to these objects is allowed. These objects are used by other spaces for granting or revoking accesses to their children. Allowing direct sharing of `RemoteSpace` objects does not lead to reference leaks, since no methods or instance fields of `RemoteSpace` are accessible to user programs, as can be seen from its API.

A `Space` object transmitted through a bridge is converted to a `RemoteSpace`. This is needed to ensure the invariant that a `Space` object can only be referenced by an object enclosed by that space.

4.3 Generating Bridge Classes

This subsection describes the generation by our system of the bridge class B_C which mediates accesses to instances of class C.

Bridge generation always starts from a call to the `getBridge` method of the `BridgeFactory` class. This method expects three arguments, the object the bridge has to guard (the protected object), the space of the protected object and the space of the possessor. The method `getBridge` returns a bridge whose class is a subclass of the protected object's class. If the class file of the bridge does not exist at the time the method is called, construction of the class file is started and instantiation of a new object follows. This method is also responsible for the management of the map that caches bridges interposed between a given space and a protected object and returns a cached bridge if another object in the given space refers to the protected object. An outline of the code of class `BridgeFactory` is shown in Figure 10.

Bridge classes are placed in the same package as the object space implementation. Their constructors are `protected`, which prevents user code from directly creating instances.

The main task behind the generation of a bridge is to produce, for each method m defined in class C as well as in its superclasses, a corresponding method m_B in B_C that implements the expected functionality of the bridge as described in Section 4.1.

The structure of each m_B method generated for m is uniform. First, a piece of code is inserted at the beginning of the method to consult the security policy. If the access is granted, a bridge, instead of the argument itself, is passed to the protected object when forwarding the call. This has to be done for each argument (except if the argument is primitive or of class `RemoteSpace`, in which case the value of the argument is copied) and this ensures that the protected object cannot possess a direct access to arguments. Once the arguments are converted, the method forwards the call to the protected object. If the call returns a non-primitive value, then as for the arguments, a bridge instead of the returned value is returned to the caller object. Figure 9 presents the bridge generated from the user class `FileUpdater` shown in Figure 8.

To avoid exceptions leaking out internal objects, bridges catch exceptions thrown in the protected object and generate a bridge that encapsulates the exception, before throwing this exception back to the caller.

The code that checks whether access to the protected object is allowed is performed in the static method `checkAccess` defined in class `Space`. This method takes the space of a protected object as well as the space of its caller as input and consults the access matrix stored in the two dimensional array called `authorizations` for deciding whether access can be granted or not.

The code that interposes a bridge between the arguments of the call and the protected object is present in method `getBridgeForArg`, whereas the code that interposes a bridge between the returned value and the possessor of the bridge

is located inside method `getBridgeForReturn`. Both methods are defined inside class `BridgeFactory` as shown in Figure 10. Notice that these methods handle several cases; either the argument (respectively the returned object) is a bridge, a `RemoteSpace`, a `Space` or an instance of a user class. If it is a bridge, then the object protected by the bridge is extracted and an appropriate bridge is interposed between the protected object and the callee (respectively the caller).

```
public class BridgeInternal
{
  Object protectedObj;
  Space protectedObjSpace;
  Space callerSpace;

  //initialize fields.
  initialize( Object go, Space goSpace, Space pSpace)
  {...}
}
```

Figure 6 Class BridgeInternal

```
class Space{
  // the access matrix
  static boolean[][] authorizations;
  ...
  static boolean checkAccess( Space protectedObjSpace,
                              Space callerSpace ){
     return Space.authorizations[callerSpace.spaceID]
                            [protectedObjSpace.spaceID];
  }
}
```

Figure 7 Method checkAccess controls access.

Each bridge contains a `BridgeInternal` object. The role of this object is to store the information related to the state of the bridge, i.e, its protected object, the latter's space, and the space of the possessor. The class `BridgeInternal` is shown in Figure 6. It is not possible to reserve a field for this information inside a user bridge class because the generic methods `getBridgeForArg` and `getBridgeForReturn` need to access this information when they receive any bridge as argument or returned object.

Soot is the framework for manipulating Java bytecode [24] that we used for generating the bridge classes.

```
public class FileUpdater
{
  public File concatFiles(File file1 , File file2)
                throws FileNotFoundException
  {
      if( !file1.exists() || !file2.exists() )
          throw new
              FileNotFoundException("File Not Found!");

      file1.append(file2);

      return file1;
  }
}
```

Figure 8 A user class example

4.4 Caveats for Java

This section looks in more detail at the implications of the object space implementation for Java programs. In particular, several features of the language, such as `final` classes and methods, are incompatible with the implementation approach. Dealing with these issues means imposing restrictions on the classes of objects that can be referenced across space boundaries.

Final and private clauses are important software engineering notions for controlling the visibility of classes in an application. For the object space implementation to work, each class C of which an object is transfered through a bridge has a class B_C generated that subclasses C. Final classes thus cannot have bridges generated. In the current implementation, the bridge generator complains if an object is passed whose class contains `final` clauses, though

```
public class FileUpdaterBridge extends FileUpdater
                                 implements Bridge {
  BridgeInternal bi = new BridgeInternal();

  FileUpdaterBridge(){}

  FileUpdaterBridge( Object obj,
                     Space protectedObjSpace,
                     Space callerSpace ) {
    bi.initialize( obj , protectedObjSpace ,
                   callerSpace );
  }

  BridgeInternal getBridgeInternal(){return bi;}

  public File concatFiles(File arg1 , File arg2)
                  throws FileNotFoundException {
    if( Space.checkAccess( bi.protectedObjSpace,
                           bi.callerSpace ) ) {
      try {
        File arg1Bridge = (File)BridgeFactory.
                    getBridgeForArg( arg1 , bi );
        File arg2Bridge = (File)BridgeFactory.
                    getBridgeForArg( arg2 , bi );

        File returnedObj=((FileUpdater)bi.protectedObj).
                concatFiles( arg1Bridge , arg2Bridge );

        return (File)BridgeFactory.
          getBridgeForReturn(returnedObj , bi );
      }
      catch (FileNotFoundException e) {
        throw (FileNotFoundException)BridgeFactory.
                           getBridgeForReturn(e , bi);
      }
      catch (Throwable e) {
        throw (RuntimeException)BridgeFactory.
                           getBridgeForReturn(e , bi);
      }
    }
    else
      throw new AccessException("Unauthorized Call");
  }
}
```

Figure 9 Example bridge class generated.

this restriction does not apply to `java.lang.Object` (see below). In order to handle `final` methods and classes, the object space system loader could remove `final` modifiers from class files before linking. To prevent illegal subclassing, the loader must record the `final` modifiers in each class already loaded, and verify that further classes loaded do not violate `final` constraints. The loader must also remove `private` modifiers from classes B_C. This rewriting approach was used by loaders in the JavaSeal [6] system to remove `catch`s of `ThreadDeath` exceptions, since catching these exceptions would allow an applet to ignore terminate signals from its parent. The re-writing approach does not work for system classes, as these are loaded and linked by the basic system loader.

```
class BridgeFactory {
  // maps a pair (objectToProtect , callerSpace) to
  // to the bridge interposed between them
  Map objAndCallerSpaceToBridge;
  ...
  static Bridge getBridge( Object object ,
                           Space protectedObjSpace,
                           Space callerSpace ) {
     // This method first checks if the map already
     // contains a bridge interposed
     // between the object and callerSpace.
     // If so, it returns the bridge.
     // If not, it checks whether the bridge's class
     // file exists.
     // If the class file does not exist, this method
     // asks Soot to build one.
     // Finally, it creates and returns a new instance
     // of the bridge.
  }

  static Object getBridgeForArg(Object arg,
                            BridgeInternal currentBI) {
    if( arg instanceof Bridge ) {
      BridgeInternal argBI = ((Bridge)arg).
                            getBridgeInternal();

      // If the call argument is a bridge on a object
      // located in the same space as the callee,
      // return a direct reference to the object.
      if( argBI.protectedObjSpace ==
          currentBI.protectedObjSpace )
        return argBI.protectedObj;

      // The call argument is located in another space.
      // Get a handle on it.
      return BridgeFactory.
        getBridge( argBI.protectedObj ,
                   argBI.protectedObjSpace ,
                   currentBI.protectedObjSpace );
    }
    else if( arg instanceof RemoteSpace ) {
      // No bridges required around RemetoSpace
      return arg;
    }
    else if( arg instanceof Space ) {
      // Do not allow transfer of space objects.
      return new RemoteSpace((Space)arg);
    }
    else
      // The call argument lives in the caller's space.
      // Get a handle on it.
      return BridgeFactory.
        getBridge( arg ,
                   currentBI.callerSpace,
                   currentBI.protectedObjSpace );
  }

  // Same idea as getBridgeForArg but this time
  // the returned object is protected from the caller
  static Bridge getBridgeForReturn( Object returnedObj,
                       BridgeInternal currentBI ){...}
}
```

Figure 10 Class BridgeFactory

System classes These classes include the `java.lang`, `java.util` and `java.io` classes. The problem with these classes is that they can be final, e.g., `java.lang.String`, or they contain `final` methods that cannot be overridden.

The class `java.lang.Object` must be permitted since every class sub-classes it. The only final methods of this class are `notifyAll`, `notify`, `wait`, and `getClass`. These methods cannot be overridden, and so invocation of these methods on objects cannot be controlled. The former three methods are used for thread synchronization. However, locking is out of the scope of our access control model; it is an issue for a fully-fledged protection domain model but this requires modification to the virtual machine in any case. Concerning the method `getClass`, a program that calls `getClass` on a bridge class gets a class object for the bridged object. However, the constructor of a bridge class is protected, so the program can do nothing with the object.

Special treatment is also given to system classes like `String` and `Integer` which are final classes in Java 2. Our implementation provides tailored versions of these classes to represent strings and integers exchanged across boundaries. The class `IOSString` for instance is simply a wrapper around a `String` object, and can be exchanged between spaces. The reader may have noticed the class `IOSString` in the paper's examples. The object space implementation also provides a bridged class for `IOSString`. This class contains a copy of the wrapped `String` object in `IOSString`, and is used to transfer the value of the string across spaces in the `set` method. The API of IOSString is given below. The second constructor takes a `String` object; this allows a space to create an `IOSString` from a `String` locally and to transmit that string value to another domain.

```
public class IOSString implements Serializable{
  protected String myString;
  public IOSString(){};
  public IOSString(String s);
  public IOSString getString();
  public void set(IOSString s);
  public void print();
}
```

Lastly, since `String` is final and cannot have a bridge defined for it, bridge classes define the `toString` method to return `null` in order to avoid direct references to `String`s in remote domains. In cases where strings need to be exchanged for convenience, like in exceptions for instance, the user class should define a `getMessage` method that returns an `IOSString`.

Field accesses Access to fields is also a form of inter-object communication and must be controlled for security. The current implementation however does not yet cater for this. A solution would be for the loader to instrument the bytecode with instructions that consult the access matrix before each field access, or for field accesses to be converted into method calls. The former approach is applied to Java in [21]. In the current implementation of bridges, access to fields in remote objects become access to fields in bridges. These fields do not reflect the corresponding fields in the protected object.

Static fields and methods Static methods pose two problems. First, they cannot be redefined in subclasses. Second, objects referenced by static variables could be shared between protection domains without an access control check taking place. In a fully fledged implementation of protection domains, classes should not be shared between domains [6] to avoid undetected sharing between domains. In the object space implementation, the bridge generator signals an error when an it receives an object of a user class that contains static methods.

The problem of static variables is looked at in [8]. This proposal strengthens isolation between loader spaces by keeping a different copy of objects referenced by static variables for each copy of the class used by a loader. Unfortunately, we cannot use this solution for the object space implementation since classes are shared across domains.

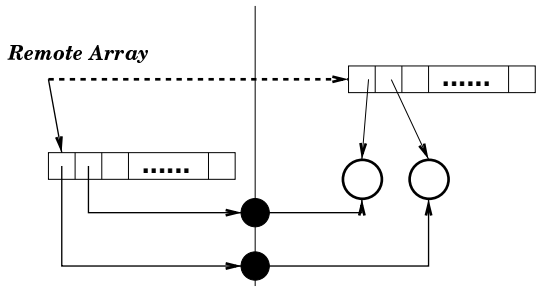

Figure 11 Treatment of arrays in object space implementation.

Arrays Arrays in Java are objects in the sense that methods defined in `Object` can be executed on array objects. Unfortunately, an array is not an object in the sense that element selection using "[ ]" is not a method call, and this requires special treatment. Our approach is outlined in Figure 11. Whenever a reference to an array object is copied across a space boundary (i.e., through a bridge), the array is copied locally. The copy is even made if the array contains primitive types like `int` or `char`. The implication of this approach is that method calls on array objects do not traverse space boundaries and that array selection is done locally. In effect, copy by value is being used for array objects; an array entry can be modified in one space without a corresponding change in another domain. Entries in copied arrays for objects become bridges if not already so. This means that non-array objects are always named using the same bridge object within a space, even though array objects may be copied. If sharing of arrays is required, then the programmer must furnish an array class that has entry selectors as methods.

This solution has an interesting repercussion regarding the example of the bug in Java cited in Section 2.3.2. The `signers` object was in fact represented by an array "`Identity signers[]`". If the object space model were used to implement this, then a copy of the signer array would be returned to the caller, whose modifications to the array would remain innocuous.

Synchronization on objects is intricately influenced by the interposition of bridges between objects. Two objects located in different spaces and willing to synchronize on the same protected object would experience undesired behavior since they are implicitly performing their operations on two different bridges protecting the protected object instead of acting on the protected object itself. This problem arises if `synchronized` statements are used instead of relying on solutions that exploit `synchronized` methods. The latter is perfectly valid since bridges forward calls to protected objects and consequently, locking occurs on the protected objects themselves.

Native methods can also lead to security flaws since they could be used to leak object references between spaces and there is no way to control this code. Our current implementation for Java does not allow bridges for classes that possess native methods, except for `Object`'s methods, e.g., `hashCode`.

4.5 Performance evaluation

Efficiency is one of the goals of the object space model. In particular, the cost of mediation of inter-space calls by bridge should be generally inferior to the cost of copy-by-value (of which "serialization" is an example) and the exchange of the byte array over a communication channel.

We conducted performance measures for the implementation running over SunOS 5.6 on a 333 MHz Ultra-Sparc-IIi processor using Sun's VM for JDK1.2.1. All measures were obtained after averaging over a large number of iterations.

One of the basic measures taken was to compare the cost of a method call between protection domains using the space (bridge) model and the loader (Java serialization) model. We also made comparisons with J-Kernel [25]. The latter allows domains to exchange parameters either by using the Java serialization mechanism, or by using a faster serialization tool developed for J-Kernel or by passing capabilities. A J-Kernel capability is an object that denotes an object in a remote domain; this is J-Kernel's equivalent of a bridged object.

The table below shows comparisons for: A) a method call with no parameters, B) for a method call with a string as parameter, and C) for a method call with an article object as a parameter. The `Article` class is used in the application of Section 3.2, and consists of a hash-table of files representing the article contents, as well as strings for the article attributes. Times are shown in micro-seconds. For A, we estimated that a basic method call without arguments (and serialization) was around 5 nano-seconds.

A cross-domain call without arguments is faster in our approach than with J-Kernel. For such a basic call, J-Kernel's overhead can mainly be explained by the thread context switch that has to be performed when crossing domains. In our implementation, the only overhead resides in the security policy check required during cross-domain calls. This cost is quite low since this check reduces to a lookup in an access matrix implemented as a static two dimensional array stored in class `Space`. Even though accessing the matrix is fast, the trade-off is that space required is $O(N^2)$ in the number of `Spaces` present in the system.

Mechanisms that use copy for passing parameters are as expected slower that their counterpart that do not (J-Kernel with capability and our object space model). Further, they do not scale well with the size of arguments.

The cost of parameter passing with the object space model is approximately two times slower than passing parameters with capabilities in J-Kernel. The overhead comes from the dynamic creation and lookup of bridges in our model. However, this cost comes with a benefit. Our model has stricter controls on access to objects. In J-Kernel, once a capability is released into the environment, it is not possible to control its spread among domains whereas in our model, we can selectively `grant` or `revoke` access to certain domains.

	Space	Serial.	J-Kernel		
			Serial.	fast copy	capability
A	0.2	–	0.8		
B	2.5	91.8	264.7	7.1	1.4
C	2.5	363.2	1004.2	587.2	1.4

The figures give an estimate of the basic mechanism. To get a more general overview, we implemented the article packager example of Section 3.2 using both the object space model and the Java loader model. The space implementation was described earlier. In the loader implementation, a class loader object is created to load the client and article classes. The service classes (`Storage` & `Editor`) are loaded with the system loader. A communication `channel` object is installed between the client and service objects for the exchange of serialized messages.

The application is highly interactive, so a direct comparison is not obvious; we therefore compared two types of communication: the cost of saving an article stored within a program on disk, and the cost of sending an event message from the GUI Editor to the client.

In the loader version, the time to save a small article is approximately 5617 micro-seconds; this cost includes the time to serialize the article. In the space version, the figure was slightly less (5520 micro seconds). This also has an article serialization since the article must be serialized to be saved on disk. The figure includes a creation of a bridge object for the article being passed to the `Storage` object. The time to send a `message` from the GUI to the program is about 143 micro-seconds in the space implementation. In the loader implementation, this figure is around 1511 micro-seconds due to serialization. The cost of serialization can be reduced by making the classes of the objects exchanged sharable (have them loaded by the system loader). However, the result of this is to weaken isolation because there is greater scope for aliasing between domains.

Regarding space usage, a bridge requires 4 words: a reference to a `BridgeInternal` object which contains 3 words (reference to guarded object, and references to guarded and possessor spaces). A `Space` object requires 3 words (an internal Integer identifier, a reference for the parent space, a reference to a hash-map object containing the children spaces); the pointer to the access matrix is static, so is shared by all `Space` objects. If there are N spaces active in a system, then the overhead of a space is N^2 access matrix entries and NM entries in the hash-map maintained by `BridgeFactory` that maps object and space pairs to bridges. M denotes the number of objects in the space referenced by objects in other spaces. If all spaces contains M objects, then the maximum number of bridges in the system is N^2M; this represents the case where all objects in all spaces are referenced by objects in all other spaces.

The measures were taken for installed bridge classes. In our implementation, a bridge class is generated on the fly if the class cannot be found on disk. This is a costly process. For instance, a bridge for the `Editor` class takes around 3.67 seconds to generate (due to parsing of the class file). On startup of the article packager application, the root, service and client spaces are created; this necessitates the creation of 10 bridges, which takes around 6.24 seconds.

5 Related Work

This section compares our object space model with related work; it is divided into Java related work, and more general work on program security.

5.1 Java Security

Java has an advanced security model that includes protection domains, whose design goal was to isolate applets from each other. The basic mechanism used is the class loader. Each applet in Java is assigned its own class loader which loads a distinct and private version of a class for its protection domain [17]. Java possesses the property that a class of one loader has a distinct type to the same class loaded by another loader. Typing is therefore the basis for isolation since creating a reference from one *loader space* to another is signaled as a type error.

A problem with this model is that dynamic typing can violate the property that spaces do not reference each other. This is because all classes loaded by the basic system loader are shared by all other loader spaces - they are never reloaded. The system loader loads all basic classes (e.g., `java.lang.*`) so sharing between loader spaces is endemic. This sharing is enough to lead to aliasing between loader spaces. Consider a class `Password` which is loaded by two loader spaces i and j; the resulting class versions are $\texttt{Password}_i$ and $\texttt{Password}_j$. This class implements the interface `PasswordID` with methods `init` and `value`. Suppose that the interface `PasswordID` is loaded by the system loader. In this case, the following program allows one password to read the value of the other, that is, the password object of loader space (UID 2) can directly invoke the password object in the other space.

```
public final class Password implements PasswordID{
```

```
 private int UID;
 private PasswordID sister;
 private String password;

 public static void main(String[] args)
                          throws Exception{
   // Create two loader spaces
   MyLoader cl1 = new MyLoader();
   MyLoader cl2 = new MyLoader();
   // Root leaks references to each space
   PasswordID child1 = (PasswordID)cl1
             .loadClass("Password").newInstance();
   PasswordID child2 = (PasswordID)cl2
             .loadClass("Password").newInstance();
   child1.init(1, "hth3tgh3", child2);
   child2.init(2, "tr54ybb", child1);
 }

 public void init(int i, String s, PasswordID R){
   sister = R; password = s; UID = i;
   if(UID == 2)
      System.out.println("Here's No 1's password: "
                         + sister.value());
  }

  public String value(){
      return password;
  }
}
```

This program starts by creating two loaders of class `MyLoader`. This loader reads files from a fixed directory. It *delegates* loading of all basic Java classes and of the `PasswordID` interface to the parent (system) loader. The program then creates an instance of `Password` in each loader space (by asking each loader to load and instantiate an instance of the class). The program grants each password a reference for each other. Despite the fact that each domain has a distinct loader, the call on `value` by the second password on the first succeeds.

Loader spaces are used to implement protection domains in several Java-based systems, e.g., [3, 14, 6, 25]. Isolation is obtained only if the shared classes do not make leaks such as that in the above example. In the object space approach, the model at least guarantees that if ever a reference to an object escapes or is leaked to another space, use of that reference is nevertheless mediated by a security policy. The security policy prohibits calls between spaces by default: an access right for a space must be explicitly granted, and this grant can be undone by a subsequent revocation.

One advantage of the loader space model over the object space model is that the former allows a program to control the classes that are loaded into its protection domain. This is important for preventing code injection attacks, where malicious code is inserted into a domain in an attempt to steal or corrupt data. In the object space model, only a parent can force a child space to execute code not foreseen by the program through the `newInstance` method. However, there is no way to control the classes used by a particular space.

Section 4 compared the implementation of the object space model with J-Kernel [25]. Recall that protection domains in J-Kernel are made up of selected shared system classes, user classes loaded by a domain loader, as well as instances of these classes. A `capability` object is used to reference an object in a remote domain. A call on a capability object transfers control to the called domain; parameters in the call are copied by value unless they are capability objects, which are copied directly.

In comparison to the object space model, J-Kernel uses copy-by-value by default, whereas the object space model uses copy-by-reference. J-Kernel must explicitly create a `capability` for an object to transfer it by reference; the object space model must explicitly serialize an object to copy it by value. The latter approach is a more natural object-oriented choice. Access control is based on capabilities in J-Kernel. A problem with capabilities is that their propagation cannot be controlled: once a domain exports a capability for one of its objects, it can no longer control what other domain receives a copy of the capability. Revocation exists but this entails revoking all copies of a capability, meaning that the distribution of access rights for an object must start again from scratch. In the object space model, an owner can grant and revoke rights for spaces selectively to other spaces. Another difference between the two systems is the presence of the hierarchy in the object space model and the absence of multiple class loaders and class instances.

The goal of the JavaSeal kernel [6] is to isolate mobile agents from each other and from the host platform. A protection domain in JavaSeal is known as a *seal*, and is also implemented using the Java loader mechanism. The set of seals is organized into a hierarchy. A message exchanged between two seals is routed via their common parent seal, which can suppress the message for security reasons. It was in fact our experience with programming a newspaper application [19] over JavaSeal that first motivated the object space model. Many objects such as environment variables and article objects needed to be distributed to several seals. This meant copy-by-value semantics, which we found to be cumbersome for mutable objects like key certificates and article files. We wanted a safe form of object sharing to simplify programming.

Interesting similarities exist between the object space model and memory management in real-time Java [5]. The latter has `ScopedMemory` objects that act as memory heaps for temporary objects. A newly created real-time thread can be assigned a `ScopedMemory`; alternatively, threads can enter the context of a `ScopedMemory` by executing its `enter()` method. The memory object contains a reference counter that is incremented each time that a thread enters it. An object created by a thread in a `ScopedMemory` is allocated in that memory object. An object (in a `ScopedMemory`) may create other `ScopedMemory` objects, thus introducing a hierarchy. The goal of the scoped memory model is to avoid use of a (slow) garbage collector to remove objects. When the reference counter of a `ScopedMemory` object becomes 0, the objects it contains can be removed. To prevent dangling references, an object cannot hold a reference to an object in a sibling `ScopedMemory`; the JVM dynamically checks all reference assignments to verify this constraint.

Compared with the object space model, both approaches use a hierarchy with accesses between spaces being dynamically checked. However, the access restrictions in the object space model can be dynamically modified, and accesses between non-related spaces are possible. On the other hand, the object space model does not deal with resource termination.

5.2 Program Security Mechanisms

There has been much work on integrating *access control* into programs. Some approaches annotate programs with calls to a security policy checker [21, 9]. In Java for instance [9], a system class contains a method call to a `SecurityManager` object that checks whether the calling thread has the right to pursue the call. Another approach to program security uses programming language support. For instance, the languages [13, 18] include the notion of access rights; programs can possess rights for objects and access by a program to an object can only progress if it possesses the access right. Language designers today tend to equate security to type correctness. In this way, security is just another "good behavior" property of a program, that can be verified using static analysis or dynamic checking [20, 16]

Leroy and Rouaix use typing to enforce security in environments running applets [16]. Security in this context means that an applet cannot gain access to certain objects (such as those private to an environment function), and that objects which are accessible can only be assigned a specified set of permitted values. Each system type τ has special versions t that each define a set of permitted values. For instance, τ may be `String`, and t be `CLASSPATH` with possible values being `/applet/public` and `/bin/java`. Each conversion from τ to t on an object entails verifying that the object respects the permitted values. Environment functions available to applets are bridge-like in the sense that each incoming reference of type τ is cast to t. This is similar to the object space model implementation in Java since for each class τ, a bridge class t is constructed that contains code to verify the system's access control policy. Access permissions are specified by an access matrix in the object space model, rather than by permitted object values.

The goal of JFlow [20] is *information flow security*; this deals with controlling an attacker's ability to infer information from an object rather than with controlling access to the object's methods. JFlow extends the Java language by associating security labels with variables. A security label denotes the sensitivity of an object's information. JFlow has a static analyzer that ensures that an object does not transfer a reference or data to an object with an inferior security label, as this would constitute a leak. The complexity of the mechanism comes from ensuring that information about objects used in an conditional expression evaluation is not implicitly leaked to objects modified in the scope of the conditional expression.

In comparison to these works, the object space implementation relies on typing to ensure that each object access is made using a secure version of the class (i.e., one that includes access control checks). Annotation of classes with checks could be included to check field accesses between objects [21], as we mentioned in Section 4.

A related topic to access control is *aliasing control*, e.g., Confined types [4], Balloons [1] or Islands [11]. Confined types is a recent effort to control the visibility of kernel objects by controlling the visibility of class names. A confined type is a class whose objects are invisible to specific user programs. The advantage of this approach is that the confinement of a type is verified by the compiler. On the other hand, classes cannot be confined and non-confined at the same time. It is important that one can designate some objects of a class as protected, and other objects as public. For instance, the visibility of `String`s that represent passwords must be confined, though the class `String` is a general class that should be accessible to all programs. Another problem with aliasing control is that object visibility can vary during the object's lifetime. For instance, an object given to a server must remain accessible to that server during the server's work-time. Once the server has completed its task, the object should be removed from the server's visibility; this is the server containment property [7]. The object space model controls access on an object basis, and the visibility constraints can be dynamically altered.

6 Conclusions

This paper has presented an access control model for an object-oriented environment. The model API is strongly influenced by Java and its loader spaces programming model, though aims to overcome weaknesses in Java access control caused by aliasing. We evaluated our proposition for an implementation over Java 2. Though the implementation has the advantage of portability, it means that we cannot address resource control and domain termination issues. These issues must be treated if object spaces are to become fully-fledged protection domains. Virtual machine support could also be useful to overcome the limitations of the model in Java, e.g., the prohibition of field accesses and the work around of `final` modifiers.

Our results show that interposition of access control programs between objects of different domains can be more efficient than a simple copy-by-value of data between loader spaces. We believe that the object space model is a more natural object programming style than copy-by-value, especially for objects that need to be accessed by many programs and whose value can change often, e.g., environment variables. The model also has the advantage that any leak of a reference between spaces is innocuous if the receiving space has not been explicitly granted the right to use the space of the referenced object. And even if access has been granted, this right may always be removed.

Acknowledgments The authors are grateful to the anonymous referees and to Jan Vitek for very valuable comments on the content and presentation of this paper. This work was support by the Swiss National Science Foundation, under grant FNRS 20-53399.98.

References

[1] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, New York, NY, June 1997.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[4] B. Bokowski and J. Vitek. Confined Types. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34 of *ACM Sigplan Notices*, pages 82–96, N. Y., Nov. 1–5 1999. ACM Press.

[5] G. Bollella, B. Brosgol, P. Dribble, and et. al. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, Reading, MA, 2000.

[6] C. Bryce and J. Vitek. The Javaseal Mobile Agent Kernel. In D. Milojevic, editor, *Proceedings of the 1st International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents (ASAMA'99)*, pages 176–189, Palm Springs, May 9–13, 1999. ACM Press.

[7] E. Cohen and D. Jefferson. Protection in the Hydra Operating System. *ACM SIGOPS*, 9(5):141–160, Nov. 1975.

[8] G. Czajkowski. Application Isolation in the Java Virtual Machine. In *Proceeings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'200)*, ACM Sigplan Notices, N. Y., Oct. 15–19 2000. ACM Press.

[9] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, Reading, MA, USA, 1999.

[10] L. Gong and R. Schemers. Signing, Sealing, and Guarding Java Objects. *Lecture Notes in Computer Science*, 1419:206–218, 1998.

[11] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[12] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2):11–16, Apr. 1992.

[13] A. K. Jones and B. H. Liskov. A Language Extension for Controlling Access to Shared Data. *IEEESE*, SE-2(4):277–285, Dec. 1976.

[14] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. Research Report RZ 2996, University of Minnesota, May 1999.

[15] B. W. Lampson. A Note on the Confinement Problem. *Communications of the Association for Computing Machinery*, 16(10):613–615, Oct. 1973.

[16] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 Jan. 1998.

[17] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, Oct. 1998.

[18] J. R. McGraw and G. R. Andrews. Access Control in Parallel Programs. *IEEE Transactions on Software Engineering*, 5(1):1–9, Jan. 1979.

[19] J.-H. Morin and D. Konstantas. Commercialization of Electronic Information. *Journal of End User Computing*, 12(2):20–32, Apr.-June 2000.

[20] A. C. Myers. JFlow: practical mostly-static information flow control. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 228–241, New York, NY, USA, 1999. ACM Press.

[21] R. Pandey and B. Hashii. Providing Fine-Grained Access Control for Java Programs. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 449–473, Lisbon, Portugal, June 1999. Springer-Verlag.

[22] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The Use of Name Spaces in Plan 9. *Operating System Review*, 27(2):72–76, Apr. 1993.

[23] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[24] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode using the Soot framework: Is it feasible? In D. Watt, editor, *CC2000-International Conference on Compiler Construction*, 2000.

[25] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. J-Kernel: A Capability-Based Operating System for Java. *Lecture Notes in Computer Science*, 1603:369–394, 1999.