Implementing Multiple Protection Domains in Java

Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski,
Deyu Hu, and Thorsten von Eicken

Technical Report 97-1660
Department of Computer Science
Cornell University

Abstract

Safe language technology can be used for protection within a single address space. This protection is enforced by the language's type system, which ensures that references to objects cannot be forged. A safe language alone, however, lacks many features taken for granted in more traditional operating systems, such as rights revocation, thread protection, resource management, and support for domain termination. This paper describes the J-Kernel, a portable Java-based protection system that addresses these issues. A number of micro-benchmarks are presented to characterize the costs of language-based protection, and an extensible web server based on the J-Kernel demonstrates the use of safe language techniques in a large application.

1 Introduction

Traditional operating systems use virtual memory to enforce protection between processes. A process cannot directly read and write other processes' memory, and communication between processes requires traps to the kernel. In the past decade of operating systems research, a large number of fast inter-process communication mechanisms have been proposed [2][23][7]. Nevertheless, the cost of passing through the kernel and of switching address spaces remains orders of magnitude larger than that of calling a procedure.

With the increasing adoption of extensible applications and component software, the cost of inter-process communication is leading to a difficult trade-off between robustness and performance. For example, the Netscape browser allows plug-ins to be loaded directly into the browser process to extend Netscape's functionality. However, an error in a plug-in can corrupt the entire browser. Although a separate process could be used for each plug-in, this would be both cumbersome to program and slow, because of the amount of communication between the plug-ins and the browser. Most web servers support plug-ins as well and in this case, the robustness issue is even more important – a browser crash may be annoying, but a server crash can be disastrous.

The robustness versus performance tradeoff is pervasive in component software (e.g., OLE, JavaBeans [14], ActiveX, OpenDoc). Microsoft's COM [25], for example, provides two different models for composing components: each component can run in its own process for protection, or multiple components can share a process (often termed *in-proc*) for performance. With more and more applications on the desktop being composed of "reusable" components the protection issue is becoming pressing: if not properly isolated, the failure of any component could cause large portions of the user's desktop environment to crash. As an example, consider a chart object embedded in a word processor document. If the charting component fails, the word processor should be able to continue operating. Ideally, a new chart component could be instantiated to replace the failed component, without restarting the word processor or reloading the document that contains the chart object.

This paper explores the use of safe language technology to offer high performance as well as protection in a software component environment. Safe languages such as Java [10], Modula-3 [30], and CAML [20] use type safety and controlled linking to enforce protection between multiple components without relying on hardware support. In a safe language environment, calls across protection boundaries could potentially be as cheap as simple function calls[1], enabling as much communication between components as desired without performance drawbacks.

While many extensible applications and component environments can benefit from protection, the features required in different settings vary. In this paper, we assume that applications are composed of independently developed software components that communicate through well-structured interfaces. We assume that the protection mechanism should enforce this structure, just like modern languages enforce module or class structures. Thus communication should only be possible through well-defined interfaces, and not through side effects. In all settings, we strive to enable failure isolation: a bug in one component should not crash other components. However, the required degree of failure isolation varies: in an application suite produced by a group of developers, the primary concern is accidental effects of one component on another. On the other hand, a web server allowing arbitrary users to upload extensions requires bulletproof protection assumed.to guard against malicious behavior.

Several projects [1, 4, 8, 11, 37] have recently described how to build protection domains around components in a safe language environment, where a protection domain specifies the resources to which a software component has access. The central ideas are to use the linker to create multiple namespaces and to use object references (i.e., pointers to objects) as capabilities for cross-domain communication. The multiple namespaces ensure that the same variable, procedure, or type names can refer to different instances in different domains. Object references in safe languages are unforgeable and can thus be used to confer certain rights to the holder(s). In an object-oriented language, the methods applicable to an object are in essence call gates. This paper argues in Section 2 that this straight-forward approach, while both flexible and fast, is ultimately unsatisfactory: using objects references as capabilities leads to severe problems with revocation, resource management, and inter-domain dependency analysis.

In order to overcome the limitations of the straight-forward approach, we introduce additional mechanisms borrowed from traditional capability systems. The result is a system described in Section 3, called the J-Kernel. The J-Kernel is written entirely in Java, and provides sophisticated capability-based protection features. We choose Java for practical reasons – Java is emerging as the most widely used general-purpose safe language, and dependable Java virtual machines are widespread and easy to work with. While Java does allow for multiple protection domains within a single Java Virtual Machine (JVM), the current sandbox model for applets is very restrictive. It lacks many of the characteristics that are taken for granted in more traditional systems and, in particular, does not provide a clear way for different protection domains to communicate with each other.

We concentrated our efforts on developing a general framework to allow multiple protection domains within a single JVM. We provide features found in traditional operating systems, such as support for rights revocation and domain termination. In addition, we support flexible protection policies between components, including support for communication between mutually suspicious components. The main benefits of our system are highly flexible protection model, low overheads for communication between software components, and operating system independence. Our current J-Kernel implementation runs on standard JVMs.

Language-based protection does have drawbacks. First, code written in a safe language tends to run more slowly than code written in C or assembly language, and thus the improvement in cross-domain communication may be offset by an overall slowdown. While much of this slowdown is due to current Java just-in-time compilers optimizing for fast compile times at the expense of run-time run-time,performance, even with sophisticated optimization it seems likely that Java programs will not run as fast as C programs. Second, all current language-based protection systems are designed around a single language, which limits developers and doesn't handle legacy code. Software fault isolation [36] and verification of assembly language [28, 29, 27] may someday offer solutions, but are still an active area of research [31].

Section 4 describes an extensible web server based on the J-Kernel. Section 5 discusses related work, and section 6 concludes.

[1] In fact, in the case of Java, just-in-time compilers have the opportunity to perform optimizations that would be impossible in a system based on hardware protection. If a cross-domain call is merely a function call, then the compiler could potentially inline the call for even faster performance.

2 Language-based protection background

In an unsafe language, any code running in an address space can potentially modify any memory location in that address space. While, in theory, it is possible to prove that certain pieces of code only modify a restricted set of memory locations, in practice this is very difficult for languages like C and assembly language [3, 28], and cannot be fully automated. In contrast, the type system and the linker in a safe language restrict what operations a particular piece of code is allowed to perform on which memory locations.

The term *namespace* can be used to express this restriction: a namespace is a partial function mapping names of operations to the actions taken when the operations are executed. For example, the operation "read the field `out` from the class `System`" may perform different actions depending on what class the name `System` refers to.

Protection domains around software components can be constructed in a safe language system by providing a separate namespace for each component. Communication between components can then be enabled by introducing sharing among namespaces. Java provides three basic mechanisms for controlling namespaces: selective sharing of object references, static access controls, and selective class sharing.

Selective sharing of object references

Two domains can selectively share references to objects by simply passing each other these references. In the example below, `method1` of class A creates two objects of type A, and passes a reference to the first object to `method2` of class B. Since `method2` acquires a reference to `a1`, it can perform operations on it, such as incrementing the field `j`. However, `method2` was not given a reference to `a2` and thus has no way of performing any operations on it. Java's safety prevents `method2` from forging a reference to `a2`, e.g., by casting an integer holding `a2`'s address to a pointer.

```
class A {                               class B {
    private int i;                          public static void method2(A arg) {
    public int j;                               arg.j++;
    public static void method1() {          }
        A a1 = new A();                 }
        A a2 = new A();
        B.method2(a1);
    }
}
```

Static access control

The preceding example demonstrated a very dynamic form of protection – methods can only perform operations on objects to which they have been given a reference. Java also provides static protection mechanisms that limit what operations a method can perform on an object once the method has acquired a reference to that object. A small set of modifiers can change the scope of fields and methods of an object. The two most common modifiers, `private` and `public`, respectively limit access to methods in the same class or allow access to methods in any class. In the classes shown above, `method2` can access the `public` field `j` of the object `a1`, but not the `private` field `i`.

Selective class sharing

Domains can also protect themselves through control of their *class namespace*. To understand this, we need to look at Java's class loading mechanisms. To allow dynamic code loading, Java supports user-defined *class loaders* which load new classes into the virtual machine at run-time. A class loader fetches Java bytecode from some location, such as a file system or a URL, and submits the bytecode to the virtual machine. The virtual machine performs a verification check to make sure that the bytecode is legal, and then integrates the new class into the machine execution. If the bytecode contains references to other classes, the class loader is invoked recursively in order to load those classes as well.

Class loaders can enforce protection by making some classes visible to a domain, while hiding others. For instance, the example above assumed that classes A and B were visible to each other. However, if class A were hidden from class B (i.e. it did not appear in B's class namespace), then even if B obtains a reference to an object of type A, it will not be able to access the fields `i` and `j`, despite the fact that `j` is public.

2.1 Straight-forward protection domains: the *share anything* approach

The simple controls over the namespace provided in Java can be used to construct software components that communicate with each other but are still protected from one another. In essence, each component is launched in its own namespace, and can then share any class and any object with other components using the mechanisms described above. While we will continue to use the term *protection domain* informally to refer to these protected components, we will argue that it is impossible to define protection domains precisely when using this approach.

The example below shows a hypothetical file system component that gives objects of type `FileSystemInterface` to its clients to give them access to files. Client domains make cross-domain invocations on the file system by invoking the `open` method of a `FileSystemInterface` object. By specifying different values for `accessRights` and `rootDirectory` in different objects, the file system can enforce different protection policies for different clients. Static access control ensures that clients cannot modify the `accessRights` and `rootDirectory` fields directly, and one client cannot forge a reference to another client's `FileSystemInterface` object.

```
class FileSystemInterface {
      private int accessRights;
      private Directory rootDirectory;
      public File open(String fileName) {...}
}
```

The filesystem example illustrates an approach to protection in Java that resembles a capability system. Several things should be noted about this approach. First, this approach does not require any extensions to the Java language - all the necessary mechanisms already exist. Second, there is very little overhead involved in making a call from one protection domain to another, since a cross-domain call is simply a method invocation, and large arguments can be passed by reference, rather than by copy. Third, references to any object may be shared between domains since the Java language has no way of restricting which references can be passed through a cross-domain method invocation and which cannot.

When we first began to explore protection in Java, this *share anything* approach seemed the natural basis for a protection system, and we began developing on this foundation. However, as we worked with this approach a number of problems became apparent.

Revocation

The first problem is that access to an object reference cannot be revoked. Once a domain has a reference to an object, it can hold on to it forever. Revocation is important in enforcing the principle of least privilege: without revocation, a domain can hold onto a resource for much longer than it actually needs it. The most straightforward implementation of revocation uses extra indirection. The example below shows how a revocable version of the earlier class `A` can be created. Each object of `A` is wrapped with an object of `AWrapper`, which permits access to the wrapped object only until the `revoked` flag is set.

```
class A {
      public int method1(int arg1, int arg2) {...}
}
class AWrapper {
      private A a;
      private boolean revoked;
      public int meth1(int arg1, int arg2) {
            if(!revoked) return a.meth1(arg1, arg2);
            else throw new RevokedException();
      }
      public void revoke() { revoked = true; }
}
```

In principle, this solves the revocation problem and is efficient enough for most purposes. However, our experience shows that programmers often forget to wrap an object when passing it to another domain. In particular, while it is easy to remember to wrap objects passed as arguments, it is common to forget to wrap other objects to which the first one points. In effect, the default programming model ends up being an unsafe model where objects cannot be revoked. This is the opposite of the desired model: safe by default and unsafe only in special cases.

Inter-domain Dependencies and Side Effects

As more and more object references are shared between domains, the structure of the protection domains is blurred, because it is unclear from which domains a shared object can be accessed. For the programmer, it becomes difficult to track which objects are shared between protection domains and which are not, and the Java language provides no help as it makes no distinction between the two. Yet, the distinction is critical for reasoning about the behavior of a program running in a domain. Mutable shared objects can be modified at any time in by other domains that have access to the object, and a programmer needs to be aware of this possible activity. For example, a malicious user might try to pass a byte array holding legal bytecode to a class loader (byte arrays, like other objects, are passed by reference to method invocations), wait for the class loader to verify that the bytecode is legal, and then overwrite the legal bytecode with illegal bytecode which would subsequently be executed. The only way the class loader can protect itself from such an attack is to make its own private copy of the bytecode, which is not shared with the user and is therefore safe from malicious modification.

Domain Termination

The problems associated with shared object references come to a head when we consider what happens when a domain must be terminated. Should all the objects that the domain allocated be released, so that the domain's memory is freed up? Or should objects allocated by the domain be kept alive as long as other domains still hold references to them? From a traditional OS perspective, it seems natural that when a process terminates all of its objects disappear, because the address space holding those objects ceases to exist. On the other hand, from a Java perspective, objects can only be deallocated when there are no more reachable references to them.

Either solution to domain termination leads to problems. Deallocating objects when the domain terminates can be extremely disruptive if objects are shared at a fine-grained level and there is no explicit distinction between shared and non-shared objects. For example, consider a Java String object, which holds an internal reference to a character array object. Suppose domain 2 holds a String object whose internal character array belongs to domain 1. If domain 1 dies, then the String will suddenly stop working, and it may be beyond the programmer's ability to deal with disruptions at this level.

On the other hand, if a domain's objects do not disappear when the domain terminates, other problems can arise. First, if a server domain fails, its clients may continue to hold on to the server's objects and attempt to continue using them. In effect, the server's failure is not propagated correctly to the clients. Second, if a client domain holds on to a server's objects, it may indirectly also hold on to other resources, such as open network connections and files. A careful server implementation could explicitly relinquish important resources before exiting, but in the case of unexpected termination this may be impossible. Third, if one domain holds on to another domain's objects after the latter exits, then any memory leaks in the terminated domain may be unintentionally transferred to the remaining one. It is easy to imagine scenarios where recovery from this sort of shared memory leak requires a shutdown of the entire VM. An example of a shared memory leak in the current Java API is the method `String.intern()`, which inserts String objects into an internal table of the shared system class `String`. Unless the table is implemented with weak references (which most virtual machines do not provide), these objects are never removed. This means that the size of the internal table only increases over time until the entire virtual machine is shut down.

Threads

By simply using method invocation for cross-domain calls, the caller and callee both execute in the same thread, which creates several potential hazards. First, the caller must block until the callee returns – there is no way for the caller to gracefully back out of the call without disrupting the callee's execution. Second, Java threads support methods such as `stop`, `suspend`, and `setPriority` that modify the state of a thread. A malicious domain could call another domain and then suspend the thread so that the callee's execution gets blocked, perhaps while holding a critical lock or other resource. Conversely, a malicious callee could hold on to a `Thread` object and modify the state of the thread after execution returns to the caller.

Resource Accounting

A final problem with the simple protection domains is that object sharing makes it difficult to hold domains accountable for the resources that they use, such as processor time and memory. In particular, it is not clear how to define a domain's memory usage when domains share objects. One definition is that a domain is held accountable for all of the objects that it allocates, for as long as those objects remain alive. However, if shared objects aren't deallocated when the domain exits, a domain might continue to be charged for shared objects that it allocated, long after it has exited. Perhaps the cost of shared objects should be split between all the domains that have references to the object. However, because objects can contain references to other objects, a malicious domain could share an object that looks small, but actually contains pointers to other large objects, so that other domains end up being charged for most of the resources consumed by the malicious domain.

In summary, the simple approach to protection in Java outlined in this section is both fast and flexible, but it runs into trouble because of its lack of structure. In particular, it fails to clearly distinguish between the ordinary, non-shared object references that constitute a domain's internal state, and the shared object references that are used for cross-domain communication. Nevertheless, this approach is useful to examine, because it illustrates how much protection is possible with the mechanisms provided by the Java language itself. It suggests that the most natural approach to building a protection system in Java is to make good use of the language's inherent protection mechanisms, but to introduce additional structure to fix the problems. The next section presents a system that retains the flavor of the simple approach, but makes a stronger distinction between non-shared and shared objects.

3 The J-Kernel

The J-Kernel is a capability-based system that supports multiple, cooperating protection domains which run inside a single Java virtual machine. Capabilities were chosen because they have several advantages over access lists: (i) they can be implemented naturally in a safe language, (ii) they can enforce the principle of least privilege more easily, and (iii) by avoiding access list lookups, operations on capabilities can execute quickly.

The primary goals of the J-Kernel are:

- a precise definition of protection domains, with a clear distinction between objects local to a domain and *capability objects* that can be shared between domains,
- well defined, flexible communication channels between domains based on capabilities,
- support for revocation for all capabilities, and
- clean semantics of domain termination.

To achieve these goals, we were willing to accept higher cross-domain communication overheads when compared to the share anything approach. In order to ensure portability, the J-Kernel is implemented entirely as a Java library and requires no native code or modifications to the virtual machine. To accomplish this, the J-Kernel defines class loaders that examine and in some cases modify user-submitted bytecode before passing it on to the virtual machine. These class loaders also generate bytecode at run-time for stub classes used for cross-domain communication. Finally, the J-Kernel's class loader substitutes safe versions for some problematic standard classes. With these implementation techniques, the J-Kernel builds a protection architecture that is radically different from the `SecurityManager`-based protection architecture that is the default model on most Java virtual machines.

Protection in the J-Kernel is based on three core concepts – capabilities, protection domains, and cross-domain calls:

- Capabilities are implemented as objects of the class `Capability` and represent handles onto resources in other domains. A capability can be revoked at any time by the domain that created it. All uses of a revoked capability throw an exception, ensuring the correct propagation of failure.
- Protection domains are represented by the Java class `Domain`. Each protection domain has a namespace that it controls as well as a set of threads. When a domain terminates, all of the capabilities that it created are revoked, so that all of its memory may be freed, thus avoiding the domain termination problems that plagued the share anything approach.
- Cross-domain calls are performed by invoking methods of capabilities obtained from other domains. The J-Kernel's class loader interposes a special calling convention[2] for these calls as follows. Arguments and return values are passed by reference if they are also capabilities, but they are passed by copy if they are primitive types or non-capability objects. When an object is copied, these rules are applied recursively to the data in the object's fields, so that a deep copy of the object is made. The effect is that only capabilities can be shared between protection domains and references to regular objects are confined to single domains.

3.1 J-Kernel implementation

The J-Kernel's implementation of capabilities and cross-domain calls relies heavily on Java's *interface classes*. An interface class defines a set of method signatures without providing their implementation. Other classes that provide corresponding implementations can then be declared to *implement* the interface. Normally interface classes are used to provide a limited form of multiple inheritance (properly called interface inheritance) in that a class can implement multiple interfaces. However, Sun's remote method invocation (RMI) specification [15] "pioneered" the use of interfaces as compiler annotations. Instead of using a separate interface definition language (IDL), the RMI specification simply uses interface classes that are flagged to the RMI system in that they extend the class `Remote`. Extending `Remote` has no language effect, rather, it directs the RMI system to generate appropriate stubs and marshalling code.

Because of the similarity of the J-Kernel's cross-domain calls to remote method invocations, we have integrated much of Sun's RMI specification into the capability interface. The example below shows a simple *remote interface* and a class that implements this remote interface, both written in accordance with Sun's RMI specification.

```
interface ReadFile extends Remote { // interface class shared with (visible from) other domains
       byte readByte() throws RemoteException;
       byte[] readBytes(int nBytes) throws RemoteException;
}
class ReadFileImpl implements ReadFile { // implementation hidden from other domains
       public byte readByte() {...}
       public byte[] readBytes(int nBytes) {...}
       ...
}
```

To create a capability in the J-Kernel, a domain calls the `create` method of the class `Capability`, passing as an argument a target object that implements one or more remote interfaces. The `create` method returns a new capability, which extends the class `Capability` and implements all of the remote interfaces that the target object implements. The capability can then be passed to other domains, which can cast it to one of its remote interfaces, and invoke the methods this interface declares. In the example below domain 1 creates a capability and adds it to the system-wide repository (the repository is a service allowing domains to publish capabilities under a name). Domain 2 retrieves the capability from the repository, and makes a cross-domain invocation on it.

Domain 1:
```
ReadFileImpl target = new ReadFileImpl(); // instantiate new ReadFileImpl object
Capability c = Capability.create(target); // create a capability for the new object
```

[2] The standard Java calling convention passes primitive data types (int, float, etc.) by copy and object data types by reference.

```
Domain.getRepository().bind("Domain1ReadFile", c); // add it to repository under some name
```

Domain 2:

```
Capability c = Domain.getRepository().lookup("Domain1ReadFile"); // extract capability
byte b = ((ReadFile) c).readByte(); // cast it to ReadFile, and invoke remote method
```

Essentially, a capability object is a wrapper object around the original target object. The code for each method in the wrapper switches to the domain that created the capability, makes copies of all non-capability arguments according to the special calling convention, and then invokes the corresponding method in the target object. When the target object’s method returns, the wrapper switches back to the caller domain, makes a copy of the return value if it is not a capability, and returns.

Local-RMI stubs

The simple looking call to `Capability.create` in fact hides most of the complexity of traditional RPC systems. Internally, `create` automatically generates a stub class at run-time for each target class. This avoids off-line stub generators and IDL files, and it allows the J-Kernel to specialize the stubs to invoke the target methods with minimal overhead. Besides switching domains, stubs have three roles: copying arguments, supporting revocation, and protecting threads.

By default, the J-Kernel uses Java’s built-in serialization features [15] to copy an argument: the J-Kernel serializes an argument into an array of bytes, and then deserializes the byte array to produce a fresh copy of the argument. While this is convenient because many built-in Java classes are serializable, it involves a substantial overhead. Therefore, the J-Kernel also provides a fast copy mechanism, which makes direct copies of objects and their fields without using an intermediate byte array. The fast copy implementation automatically generates specialized copy code for each class that the user declares to be a fast copy class. For cyclic or directed graph data structures, a user can request that the fast copy code use a hash table to track object copying, so that objects in the data structure are not copied more than once (this slows down copying, though, so by default the copy code does not use a hash table).

Each generated stub contains a revoke method that sets the internal pointer to the target object to `null`. Thus all capabilities can be revoked and doing so makes the target object eligible for garbage collection, regardless of how many other domains hold a reference to the capability. This prevents domains from holding on to the garbage of other domains.

In order to protect the caller’s and callee’s threads from each other, the generated stubs provide the illusion of switching threads. Because most virtual machines map Java threads directly onto kernel threads it is not practical to actually switch threads: as shown in the next subsection this would slow down cross-domain calls by an order of magnitude. A fast user-level threads package might solve this problem, but would require modifications to the virtual machine, and would therefore limit the J-Kernel’s portability. The compromise struck in the J-Kernel uses a single Java thread for both the caller and callee but prevents direct access to that thread to avoid security problems.

Conceptually, the J-Kernel divides each Java thread into multiple segments, one for each side of a cross-domain call. The J-Kernel class loader then hides the system `Thread` class that manipulates Java threads, and interposes its own with an identical interface but an implementation that only acts on the local thread segment. Thread modification methods such as `stop` and `suspend` act on thread segments rather than Java threads, which prevents the caller from modifying the callee’s thread segment and vice-versa. This provides the illusion of thread-switching cross-domain calls, without the overhead for actually switching threads. The illusion is not totally convincing, however – cross-domain calls really do block, so there is no way for the caller to gracefully back out of one if the callee doesn’t return.

Class Name Resolvers

In the standard Java applet architecture, applets have very little access to Java’s class loading facilities. In contrast, J-Kernel domains are given considerable control over their own class loading. Each domain has its own class namespace that maps names to classes. Classes may be local to a domain, in which case they are only visible in that domain’s namespace or they may be shared between multiple domains, in which case they are visible in many namespaces. A domain’s namespace is controlled by a user-defined *resolver*, which is queried by the J-Kernel whenever a new class name is encountered. A domain can use a resolver to load new bytecode into the system, or it can make use of existing shared classes. After a domain has loaded new classes into the system, it can share these classes with other domains if it wants, by making a `SharedClass` capability available to other domains[3].

Shared classes are the basis for cross-domain communication: domains must share remote interfaces and fast copy classes to establish common methods and argument types for cross-domain calls. Allowing user-defined shared classes makes the cross-domain communication architecture extensible; standard Java security architectures only allow pre-defined “system classes” to be shared between domains, and thus limit the expressiveness of cross-domain communication. Ironically, the J-Kernel needs to *prevent* the sharing of some system classes. For example, the file system and thread classes present security problems. Others contain resources that need to be defined on a per-domain basis: the class `System`, for example, contains static fields holding the standard input/output streams. In other words, the “one size fits all” approach to class sharing in most Java security models is simply not adequate, and a more flexible model is essential to make the J-Kernel safe, extensible, and fast.

3.2 J-Kernel Micro-Benchmarks

To evaluate the performance of the J-Kernel mechanisms we measured a number of micro-benchmarks on the J-Kernel as well as on a number of reference systems. Unless otherwise indicated, all micro-benchmarks were run on 200Mhz Pentium-Pro systems running Windows NT 4.0 and the Java virtual machines used were Microsoft’s VM (MS-VM) and Sun’s VM (Sun-VM) with Symantec’s JIT compiler. All numbers are averaged over a large number of iterations.

Null LRMI

Table 1 dissects the cost of a null cross-domain call (null LRMI) and compares it to the cost of a regular method invocation, which takes a few tens of nanoseconds. The J-Kernel null LRMI takes 50x to 100x longer than a regular method invocation. With MS-VM, a significant fraction of the cost lies in the interface method invocation necessary to enter the stub. Additional overheads include the synchronization cost when changing thread segments (two lock acquire/release pair per call) and the overhead of looking up the current thread. Overall, these three operations account for about 70% of the cross-domain call on MS-VM and about 80% on Sun-VM. Given that the implementations of the three operations are independent, one could hope for significantly better performance in a system that includes the best of both VMs.

Operation	MS-VM	Sun-VM
Regular Method invocation	0.04	0.03
Interface method invocation	0.54	0.05
Thread info lookup	0.55	0.29
Acquire/release lock	0.20	1.91
J-Kernel LRMI	2.22	5.41

Table 1. Cost of null method invocations (in µs)

To compare the J-Kernel LRMI with traditional OS cross-domain calls, Table 2 shows the cost of several forms of local RPC available on NT. *NT-RPC* is the standard, user-level RPC facility. *COM out-of-proc* is the cost of a null interface invocation to a COM component located in a separate process on the same machine. The communication between two fully protected components is at least a factor of 3000 from a regular C++ invocation (shown in the *COM in-proc* entry). In the J-Kernel, cross-domain communication is within a factor of 200 from a regular Java invocation (about 60 in MS-VM and 180 in Sun-VM).

[3] Shared classes (and, transitively, the classes that shared classes refer to) are not allowed to have static fields, to prevent sharing of non-capability objects through static fields. In addition, to ensure consistency between domains, two domains that share a class must also share other classes referenced by that class.

Form of RPC	Time
NT-RPC	109
COM out-of-proc	99
COM in-proc	0.03

Table 2. Local RPC costs using standard NT mechanisms (in µs)

Threads

Table 3 shows the cost of switching back and forth between two Java threads in MS-VM and Sun-VM. The base cost of two context switches between NT kernel threads (*NT-base*) is 8.6 µs, and Java introduces an additional 1-2 µs of overhead. This confirms that switching Java threads during cross-domain calls would add a significant cost to J-Kernel LRMI.

NT-base	MS-VM	Sun-VM
8.6	9.8	10.2

Table 3. Cost of a double thread switch using regular Java threads (in µs)

Argument Copying

Table 4 compares the cost of copying arguments during a J-Kernel LRMI using Java serialization and using the J-Kernel's fast-copy mechanism. By making direct copies of the objects and their fields without using an intermediate Java byte-array, the fast-copy mechanism improves the performance of LRMI substantially — more than an order of magnitude for large arguments. The performance difference between the second and third rows (both copy the same number of bytes) is due to the cost of object allocation and invocations of the copying routine for every object.

Number of objects and size	MS-VM		Sun-VM	
	LRMI w/ Serialization	LRMI w/ Fast-copy	LRMI w/ Serialization	LRMI w/ Fast-Copy
1 x 10 bytes	104	4.8	331	13.7
1 x 100 bytes	158	7.7	509	18.5
10 x 10 bytes	193	23.3	521	79.3
1 x 1000 bytes	633	19.2	2105	66.7

Table 4. Cost of Argument Copying (in µs)

In summary, the micro-benchmark results are encouraging in that the cost of a cross-domain call is 50x lower in the J-Kernel than in NT. However, the J-Kernel cross-domain call still incurs a stiff penalty over a plain method invocation due to the lack of optimizations in Java.

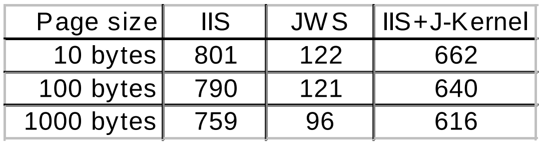
4 An Extensible Http Server

One of the driving applications for the J-Kernel is an extensible HTTP server. The goal is to allow users to dynamically extend the functionality of the server by uploading Java programs, called *servlets* [17], that customize the HTTP request processing for a subset of the server's URL space.

Instead of building (or porting) an entire HTTP server in Java, we integrated the J-Kernel into the off-the-shelf Microsoft server (IIS 3.0). The J-Kernel runs within the same process as IIS (as an in-proc ISAPI extension) and includes a system servlet with access to native methods that allows it to receive HTTP requests from IIS and return corresponding replies. This HTTP system servlet forwards each request to the appropriate user servlet, each of which runs in its own J-Kernel domain. The implementation of the bridge between IIS and the J-Kernel is multithreaded to allow multiple outstanding HTTP requests and it allows the Java code to run in the same thread as IIS uses to invoke the bridge.

Server throughput measurements

To quantify the impact of the J-Kernel overheads in the performance of the HTTP server, several simple experiments measure the number of documents per second that can be served by Microsoft’s IIS, Sun’s Java Web Server 1.0.2 (JWS) [18], and J-Kernel running inside IIS. The hardware platform consists of a quad-processor 200MHz Pentium-Pro (results obtained on one- and two-processor machines are similar). The parameter of the experiments is the size of document being served. All three tests follow the same scenario: eight multithreaded clients repeatedly request the same document. IIS serves documents in a traditional way – by fetching them from NT’s file cache, while JWS and J-Kernel utilize servlets to return in-memory documents.



Page size	IIS	JWS	IIS+J-Kernel
10 bytes	801	122	662
100 bytes	790	121	640
1000 bytes	759	96	616

Table 5. HTTP server throughput (in pages/second)

As Table 5 shows, the overhead of passing requests into and out of the J-Kernel decreases IIS’s performance by 20%. Additional measurements show that the ISAPI bridge accounts for about half of that performance gap and only the remainder is directly attributable to the J-Kernel. The order-of-magnitude gap between J-Kernel and JWS is due to the fact that JWS is written entirely in Java and is executed without a JIT compiler. At the time of this writing the implementation of JWS as an IIS plug-in with JIT was not fully functional.

CS314 servlets

Part of the motivation for the J-Kernel came from a set of servlets developed for an undergraduate computer architecture course (CS314) taught at Cornell. The course staff wrote compiler, assembler, and linker components in Java, which students used for course homeworks and projects. For a number of reasons, the course staff chose a web-based solution and implemented the components as servlets running in an extensible web server, Jigsaw.

Jigsaw [38] is a web server written in Java by the W^{3}C, and servlets are really just dynamically loaded classes without robust protection. Since these servlets are developed by the trusted course staff, malicious attack is not a source of concern. However, the lack of failure isolation made the introduction of new features during the course very difficult. In addition, the lack of clean servlet termination semantics made it impossible to replace servlets without restarting the entire server. Experience with this set of servlets had a strong influence on the J-Kernel design. The servlets are now being re-implemented using the J-Kernel. By using a protection domain for each component, the problems of the Jigsaw version are avoided.

5 Related Work

Several major vendors have proposed extensions to the basic Java sandbox security model for applets [8, 16, 31, 26]. However, all of these proposals have focused on protecting trusted system resources from untrusted applets, and have not really addressed applet-to-applet communication and protection. By only addressing protection of trusted system resources, they have avoided the difficult questions about object sharing, class sharing, and thread protection. It is not obvious how these approaches can extend to a more general model allowing communication between mutually suspicious domains.

A number of related safe-language systems are based on the idea of using object references as capabilities. Wallach et. al. [37] describe three models of Java security: type hiding (making use of dynamic class loading to control a domain’s namespace), stack introspection, and capabilities. They recommended a mix of these three techniques. The E language from Electric Communities [4] is an extension of Java targeted towards distributed systems. E’s security architecture is capability based; programmers are encouraged to use object references as the fundamental building block for protection. Odyssey [8] is a system that supports mobile agents written in Java. It supports capabilities defined with special IDL files. All three of these systems allow non-capability objects to be passed directly between domains, and generally correspond to the share anything approach described in Section 2. They do not address the issues of revocation, domain termination, thread protection, or resource accounting.

The SPIN project [1] allows safe Modula-3 code to be downloaded into the operating system kernel to extend the kernel's functionality. SPIN has a particularly nice model of dynamic linking [35] to control the namespace of different extensions. Since it uses Modula-3 pointers directly as capabilities, the limitations of the share anything approach apply to it.

Several recent software-based protection techniques do not rely on a particular high level language like Java or Modula-3. Typed assembly language [27] pushes type safety down to the assembly language level, so that code written at the assembly language level can be statically type checked and verified as safe. Software fault isolation [36] inserts run-time "sandboxing" checks into binary executables to restrict the range of memory that is accessible to the code. With suitable optimizations, sandboxed code can run nearly as fast as the original binary on RISC architectures. However, it is not clear how to extend optimized sandboxing techniques to CISC architectures, and sandboxing cannot enforce protection at as fine a granularity as a type system. Proof carrying code [28, 29] generalizes many different approaches to software protection – arbitrary binary code can be executed as long as it comes with a proof that it is safe. While this can potentially lead to safety with no overhead, generating the proofs is nontrivial.

The J-Kernel enforces a structure that is similar to traditional capability systems [19, 21]. Both the J-Kernel and traditional capability systems are founded on the notion of unforgeable capabilities. In both, capabilities name objects in a context-independent manner, so that capabilities can be passed from one domain to another. The main difference is that traditional capability systems used virtual memory or specialized hardware support to implement capabilities, while the J-Kernel uses language safety. The use of virtual memory or specialized hardware led either to slow cross-domain calls, to high hardware costs, or to portability limitations. Using Java as the basis for the J-Kernel simplifies many of the issues that plagued traditional capability systems. First, unlike systems based on capability lists, the J-Kernel can store capabilities in data structures, because capabilities are implemented as Java objects. Second, rights amplification [19] is implicit in the object-oriented nature of Java: invocations are made on methods, rather than functions, and methods automatically acquire rights to their *self* parameter. In addition, selective class sharing can be used to amplify other parameters. Although many capability systems did not support revocation, the idea of using indirection to implement revocation is certainly not new. The issue of resource accounting was also known to implementers of capability systems – Wulf et. al. [39] point out that "No one 'owns' an object in the Hydra scheme of things; thus it's very hard to know to whom the cost of maintaining it should be charged", and that the accounting issue would need to be addressed in a "production operating system".

Similarities exist between the J-Kernel and single-address operating systems (SASOS), like Opal [4] and Mungi [13]. SASOS remove the address space borders, allowing for cheaper and easier sharing of data between processes. Opal and Mungi were implemented on architectures offering large address spaces (64-bit) and used password capabilities as the protection mechanism. Password capabilities are protected from forgery by a combination of encryption and sparsity.

Several research operating systems support very fast inter-process communication. Recent projects, like L4, Exokernel, and Eros, provide fine-tuned implementations of selected IPC mechanisms, yielding an order of magnitude improvement over traditional operating systems. The systems are carefully tuned and aggressively exploit features of the underlying hardware.

The L4 μ–kernel [11] rigorously aims for minimality and is designed from scratch, unlike first-generation μ–kernels, which evolved from monolithic OS kernels. The system was successful at dispelling some common misconceptions about μ–kernel performance limitations. Exokernel [7] shares L4's goal of being an ultra-fast μ–kernel, but is also concerned with untrusted loadable modules (similar to the SPIN project). Untrusted code is given efficient control over hardware resources by separating management from protection. The focus of the EROS [34] project is to support orthogonal persistence and real-time computations. Despite quite different objectives, all three systems manage to provide very fast implementations of IPC with comparable performance, as shown in Table 6. A short explanation of the 'operation' column is needed. Round-trip IPC is the time taken for a call transferring one byte from one process to another and returning to the caller; Exokernel's protected control transfer installs the callee's processor context and starts execution at a specified location in the callee.

The results are contrasted with a 3-argument method invocation in the J-Kernel. The J-Kernel's performance is comparable with the three very fast systems. It is important to note that L4, Exokernel and Eros are implemented as a mix of C and assembly language code, while J-Kernel consists of Java classes without native code support. Improved implementations of JVMs and JITs are likely to enhance the performance of the J-Kernel.

System	Operation	Platform	Time (us)
L4	Round-trip IPC	P5-133	1.82
Exokernel	Protected control transfer (r/t)	DEC-5000	2.40
Eros	Round-trip IPC	P5-120	4.90
J-Kernel	Method invocation with 3 args	P5-133	3.77

Table 6. Comparison with selected kernels.

6 Conclusion

This paper explores the use of safe language technology to construct robust protection domains. The advantages of using language-enforced protection are portability and good cross-domain performance. The most straightforward implementation of protection in a safe language environment is to use object references directly as capabilities. However, problems of revocation, domain termination, thread protection, and resource accounting arise when non-shared object references are not clearly distinguished from shared capabilities. We argue that a more structured approach is needed to solve these problems: only capabilities can be shared, and non-capability objects are confined to single domains.

We developed the J-Kernel system, which demonstrates how the issues of object sharing, class sharing, and thread protection can be addressed. As far as we know, the J-Kernel is the first Java-based system that integrates solutions to these issues into a single, coherent protection system. Our experience using the J-Kernel to extend the Microsoft IIS web server leads us to believe that a safe language system can achieve both robustness and high performance. Simple servlets downloaded into the web server achieve a performance close to that of the server running stand-alone.

Because of its portability and flexibility, language-based protection is a natural choice for a variety of extensible applications and component-based systems. From a performance point of view, safe language techniques are competitive with fast microkernel systems, but do not necessarily achieve their promise of making cross-domain calls as cheap as function calls. Implementing a stronger model of protection than the straightforward share anything approach leads to thread management costs and copying costs, which increase the overhead to much more than a function call. Fortunately, there clearly is room for improvement. We found that many small operations in Java, such as allocating an object, invoking an interface method, and manipulating a lock were slower than necessary on current virtual machines. Java just-in-time compiler technology is still evolving. We expect that as virtual machine performance improves, the J-Kernel's cross-domain performance will also improve. In the meantime, we will continue to explore optimizations possible on top of current off-the-shelf virtual machines, as well as to examine the performance benefits that customizing the virtual machine could bring.

7 References

1. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

2. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. *Lightweight Remote Procedure Call*. In Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP), pages 102-113, Arizona, December 1989.

3. R. S. Boyer, and Y. Yu. *Automated proofs of object code for a widely used microprocessor*. J. ACM 43, 1 (Jan. 1996), 166-192.

4. J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. *Lightweight Shared Objects in a 64-Bit Operating System.* In Proceedings of the ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1992.

5. S. Drossopoulou, S. Eisenbach. *Java is Type Safe - Probably*. In Proceedings of the 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997.

6. Electric Communities. *The E White Paper*. Available at http://www.communities.com/products/tools/e.

7. R. Engler, M. F. Kaashoek, and J. James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

8. General Magic. Odyssey. Available at http://www.genmagic.com/agents.

9. L. Gong. *Java Security: Present and Near Future*. IEEE Micro, 17(3):14--19, May/June 1997.

10. J. Gosling, B. Joy, and G. Steele. *The Java language specification.* Addison-Wesley, 1996.

11. D. Hagimont, and L. Ismail. *A Protection Scheme for Mobile Agents on Java.* In Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking, Budapest, Hungary, September 26-30, 1997.

12. H. Härtig, et. al. *The Performance of μ-Kernel-Based Systems*. In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97), October 5-8, 1997, Saint-Malo, France.

13. G. Heiser, et. al. *Implementation and Performance of the Mungi Single-Address-Space Operating System*. Technical Report UNSW-CSE-TR-9704, June 1997, the Univeristy of New South Wales, Sydney, Australia.

14. JavaSoft. JavaBeans, *Version 1.01 Specification.* Available at http://java.sun.com.

15. JavaSoft. *Remote Method Invocation Specification.* Available at http://java.sun.com.

16. JavaSoft. *New Security Model for JDK1.2.* Available at http://java.sun.com

17. JavaSoft. *Java Servlet API*. Available at http://java.sun.com.

18. JavaSoft *JavaServer Documentation*. Available at http://java.sun.com

19. A. K. Jones and W. A. Wulf. *Towards the Design of Secure Systems*. Software Practice and Experience, Vol. 5, No. 4.

20. X. Leroy. *Objective Caml*. Available at http://pauillac.inria.fr/ocaml/.

21. H. M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.

22. J. Liedtke. *On μ-kernel Construction.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, December 1995.

23. J. Liedtke, et. al. *Achieved IPC Performance.* In Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS), May 5-6, Chatham, MA.

24. T. Lindholm, and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

25. Microsoft Corporation and Digital Equipment Corporaton. *The Component Object Model Specification.* Redmond, WA, July 1996.

26. Microsoft Corporation. *Microsoft Security Management Architecture White Paper*. Available at http://www.microsoft.com/ie/security.

27. G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. To appear in the 1998 Symposium on Principles of Programming Languages.

28. G. Necula and P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* In Proceedings of the 2nd Operating Systems Design and Implementation (OSDI), Seattle, WA, October 1996.

29. G. Necula. *Proof-carrying code*. In 24th ACM Symposium on Principles of Programming Languages, pages 106-119, Paris, 1997.

30. G. Nelson, ed. *System Programming in Modula-3.* Prentice Hall, 1991.

31. Netscape Corporation. *Java Capabilities API*. Available at http://www.netscape.com.

32. V. Saraswat. *Java is not type-safe*. Available at http://www.research.att.com/~vj/bug.html.

33. Z. Shao. *Typed Common Intermediate Format.* 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, October 1997.

34. J. S. Shapiro, D. J. Farber, and J. M. Smith. *The Measured Performance of a Fast Local IPC*. In the 5th International Workshop on Object-Orientation in Operating Systems, Seattle, Washington. 1996

35. E. G. Sirer, M. Fiuczynski, P. Pardyak, and B. Bershad. *Safe Dynamic Linking in an Extensible Operating System.* 1st Workshop on Compiler Support for Systems Software, February 1996.

36. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation*. In Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC, December 1993.

37. D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), Saint-Malo, France, October 1997.

38. World Wide Web Consortium, *Jigsaw 1.0*, http://www.w3.org/Jigsaw.

39. W. A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C. mmp: An Experimental Computer System*, McGraw-Hill, New York, NY (1981).