USENIX Association

Proceedings of the 9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

© 2000 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

A Secure Java™ Virtual Machine

Leendert van Doorn
leendert@watson.ibm.com
Global Security Analysis Laboratory
IBM T.J. Watson Research Center
Yorktown, NY

Abstract

The Java™ Virtual Machine is viewed by many as inherently insecure despite all the efforts to improve its security. In this paper we take a different approach to Java security and describe the design and implementation of a system that provides operating system style protection for Java code. We use hardware protection domains to separate Java classes, provide access control on cross domain method invocations, efficient data sharing between protection domains, and memory and CPU resource control. These security measures, when they do not violate the policy, are all transparent to the Java programs, even when a subclass is in one domain and its superclass is in another. To reduce the performance impact we group classes and share them between protection domains and map data on demand as it is being shared.

1. Introduction

Java™ [21] is a general-purpose programming language that has gained popularity as the programming language of choice for mobile computing. The language is used for World Wide Web programming [2], smart card programming [22], embedded device programming [16], and even for providing executable content for active networks [42]. Three reasons for this popularity are Java's portability, its security properties, and its lack of explicit memory deallocation.

Java programs are compiled into an intermediate representation called byte codes and run on a Java Virtual Machine (JVM). This JVM contains a byte code verifier that is essential for Java's security. Before execution begins the verifier asserts that the byte codes do not interfere with the execution of other programs by assuring it uses valid references and control transfers. Byte codes that successfully pass this verification are executed but still subject to number of other security measures implemented in the Java runtime system.

All of Java's security mechanisms depend on the correct implementation of the byte code verifier. In our opinion this is a flawed assumption and past experience has shown a number of security problems with this approach [11, 17, 35]. More fundamental is that from software engineering research it is known that every 1000 lines of code contain 35-80 bugs [7]. Even very thoroughly tested programs still contain on average about 0.5-3 bugs per 1000 lines of code [30]. Given that JDK 2 contains ~1.6M lines of code it is reasonable to expect 56K to 128K bugs. Granted, not all of these bugs are in security critical code but all code is security sensitive since it runs within a single protection domain.

Other unsolved security problems with current JVM designs are its vulnerability to denial of service attacks and its discretionary access control mechanisms. Denial of service attacks are possible because the JVM lacks proper support to bound the amount of memory and CPU cycles used by an application. The discretionary access control model is not always the most appropriate one for executing untrusted mobile code.

Interestingly, exactly the same security problems occur in operating systems. There they are solved by introducing hardware separation between different protection domains and controlled access between them. This hardware separation is provided by the memory management unit (MMU), an independent hardware component that controls all accesses to main memory. To control the resources used by a process an operating system limits the amount of memory it can use, assigns priorities to bias its scheduling, and can enforce mandatory access control. However, unlike programming language elements, processes are coarse grained and have primitive sharing and communication mechanisms.

An obvious solution to Java's security problems is to integrate the JVM with the operating system's process protection mechanisms. How to adapt the JVM efficiently and transparently (*i.e.,* such that multiple

Java applets can run on the same JVM while protected by the MMU) is a non-obvious problem. It requires a number of hard operating system problems to be resolved. These problems include: uniform object naming, object sharing, remote method invocation, thread migration, and protection domain and memory management.

The central goal of our work was the efficient integration of operating system protection mechanisms with a Java runtime system to provide stronger security guarantees. A subgoal was to be transparent with respect to Java programs. Where security and transparency conflicted they were resolved by a separate security policy. Using the techniques described in this paper we have build a prototype JVM that contains the following features:

- The transparent hardware assisted separation of Java classes, provided that they do not violate a preset security policy.
- The control over memory and CPU resources used by a Java class.
- The enforcement of mandatory access control for Java method invocations, class inheritance, and system resources.
- The employment of the *least privilege* concept and the introduction of a *minimal trusted computing base* (TCB).
- The JVM does not depend on the correctness of the Java byte code verifier for inter-domain protection.

In our opinion, a JVM using these techniques is much more amenable to an ITSEC or a Common Criteria evaluation than a pure software protection based system.

Our JVM consists of a small trusted component, called the *Java Nucleus*, which acts as a reference monitor and manages and mediates access between different protection domains (see figure 1). These protection domains contain one of more Java classes and their object instances. References to objects are capabilities [12] which are managed by the Java Nucleus.

For an efficient implementation of our JVM we depend on low-level operating system the functionality provided by *Paramecium* [41], an extensible operating system. The Java Nucleus uses its low-level protection domain and memory management facilities and its IPC for cross domain method invocations. The data is shared on demand using virtual memory remapping. When the data contains pointers to other data elements they are transparently shared as well. The garbage collector, which is a part of the Java Nucleus, handles runtime data relocation, sharing and revocation of data elements, protection, and the reclaiming of unused memory cells over multiple protection domains.

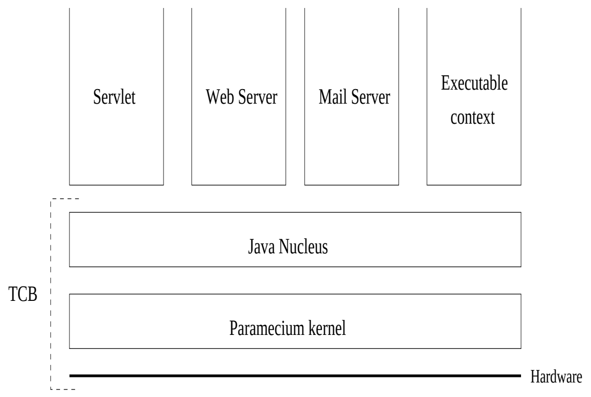

Figure 1. Secure JVM overview.

In the next section of this paper we will describe the problems involved in language and operating system integration. Section 3 discusses the separation of concerns when designing a JVM architecture with a minimal TCB. It focuses on the security guarantees offered at run time and the corresponding threat model. Since our system relies on Paramecium it is described separately in section 4. Section 5 describes some of the key implementation details of our JVM. It discusses the memory model used by our JVM, its IPC mechanism, its data sharing techniques, and its garbage collector. Section 6 discusses some early experiences with our JVM, including a performance analysis and some example applications. Section 7 discusses related work and is followed by our conclusions in section 8.

2. Operating and Run Time System Integration

Integration of an operating system and a language runtime system has a long history (*e.g.*, Mesa/Cedar [39], Lisp Machines [29], Oberon [43], JavaOS [32], *etc.*), but none of these systems use hardware protection to supplement the protection provided by the programming language. In fact, most of these systems provide no protection or depend on a trusted code generator. For example, the Burroughs B5000 [9] enforced protection through a trusted compiler. It did not provide an assembler since it could be used to circumvent this protection.

Over the years these integrated systems have lost popularity in favor of time-shared systems with a process protection model. These systems provide better security and fault isolation by using hardware separation between untrusted processes and controlling the communication between them. A side effect of this separation is that sharing is much harder and more inefficient.

The primary reasons why the transparent integration of a process protection model and a programming language are difficult are summarized in table 1. The key problem is their lack of a common naming scheme. In a process model each process has its own virtual address space, requiring techniques like pointer swizzling to translate addresses between different domains. Aside from the naming issues, the sharing granularity is different. Processes can share coarse grained pages while programs share many small variables. Reconciling the two as in distributed shared memory systems [27] leads to the undesirable effects of false sharing or fragmentation. Another distinction is the unit of protection. For a process this is an protection domain, for programs it is a module, class, object, *etc.* Finally, processes use rudimentary IPC facilities to communicate that can send and receive blocks of data. Programs on the other hand use procedure calls and memory references.

In order to integrate a process protection model and a programming language we need to adapt some of the key process abstractions. Adapting them is hard to do in a traditional operating system because they are hardwired into the system. Extensible operating systems on the other hand provide much more flexibility (*e.g.*, Paramecium, OSKit [19], L4/LavaOS [28], ExOS [15], and SPIN [6]). For example, in our system the Java Nucleus acts as a special purpose kernel for Java programs. It controls the protection domains that contain Java programs, creates memory mappings, handles all protection faults for these domains and controls cross protection domain invocations. These functions are hard to implement on a traditional system but straightforward on an extensible operating system. A second enabling feature of extensible operating systems is the dramatic improvement in cross domain transfer cost by eliminating unnecessary abstractions [5, 24, 28, 33]. This makes the tight integration of multiple protection domains feasible. Another advantage of using an extensible kernel is that they tend to be several orders smaller than traditional kernels. This is a desirable property since the kernel is part of the TCB.

	Process Protection Model	**Programming Language**
Name space	disjoint	single
Granularity	pages	variables
Unit	protection domain	class/object
Communication	IPC	call/memory

Table 1. Process protection model *vs.* programming language.

For a programming language to benefit from hardware separation it has to exhibit a number of requirements. The first one is that the language must contain a notion of a unit of protection. These units form the basis of the protection system. Examples of these units are classes, objects, and modules. Each of these units must have one or more interfaces to communicate with other units. Furthermore there needs to be non-language reasons to separate these units, like running multiple untrusted applets simultaneously on the same system. The last requirement is that the language needs to use a typed garbage collection system rather than programmer managed dynamic memory. This requirement allows a third party to manage, share and relocate the memory used by a program.

In this paper we concentrate on the integration of Paramecium and Java. While both fulfill the requirements listed above, the techniques are applicable to other operating systems and programming languages.

3. Separation of Concerns

The goal of our secure JVM is to minimize the trusted computing base (TCB) for a Java run-time system. For this it is important to separate security concerns from language protection concerns and establish what type of security enforcement has to be done at compile time, loading time, and run time.

At compile time the language syntax and semantic rules are enforced by a compiler. This enforcement ensures valid input for the transformation process of source code into byte codes. Since the compiler is not trusted the resulting byte codes cannot be trusted and therefore we can not depend on the compiler for security enforcement.

At load time a traditional JVM loads the byte codes and relies on the byte code verifier and various run-time checks to enforce the Java security guarantees. As discussed in the introduction, we do not rely on the Java byte code verifier for security for its size, complexity, and track record. Instead we aim at minimizing the TCB and use hardware fault isolation between groups of classes and their object instances and control access to methods and state shared between them. A separate security policy defines which classes are grouped together in a single protection domain and which methods they may invoke on different protection domains. It is important to realize that all classes within a single protection domain have the same trust level. Our system provides strong protection guarantees between different protection domains, *i.e.*, inter-domain protection. It does not enforce intra-domain protection, this is left to the run-time system if desirable. This does not constitute a breakdown of security of the system. It is the policy that defines the security. If two classes that are in the same domain, *i.e.*, have the same trust level, misbehave with respect to one another this clearly constitutes a failure in the policy specification. These two classes should not have been in the same protection domain.

The run-time security provided by our JVM consists of hardware fault isolation among groups of classes and their object instances by isolating them into multiple protection domains and controlling access to methods and state shared between them. Each security policy, a collection of permissions and accessible system resources, defines a protection domain. All classes with the same security policy are grouped into the same domain and have unrestricted access to the methods and state within it. Invocations of methods in other domains pass through the Java Nucleus. The Java Nucleus is a trusted component of the system and enforces access control based on the security policy associated with the source and target domain.

The Java Nucleus consists of four components: a class loader, a garbage collector, a thread system, and an IPC component. The class loader loads a new class, translates the byte codes into native machine codes, and deposits them into a specified protection domain. The garbage collector allocates and collects memory over multiple protection domains, assists in sharing memory among them, and implements memory resource control. The thread system provides the Java threads of control and maps them directly onto Paramecium threads. The IPC component implements cross protection domain invocations, access control, and CPU resource usage control.

The JVM's trust model (*i.e.*, what is included in the minimal trusted computing base) depends on the correct functioning of the garbage collector, IPC component, and thread system. We do not depend on the correctness of the byte code translator. When the byte code translator is trusted to separate executable content from data certain optimizations are possible. These are described in section 5.2.

References to memory cells (primitive types or objects) act as capabilities [12] and can be passed to other protection domains as part of a cross domain method invocation (XMI) or object instance state sharing. Passing an object reference results in passing the full closure of the reference. That is, all cells that can be obtained by dereferencing the pointers that are contained in the cell of which the reference is passed. Capabilities can be used to implement the notion of least privilege but suffer from the classical confinement and revocation problem. Solving these is straightforward since the Java Nucleus acts as a reference monitor. However, this violates the Java language transparency requirement (see section 8).

Our system does not depend on the Java security features such as byte code verification, discretionary access control through the security manager, and its type system. We view these as language security measures that assist the programmer during program development and they should not be confused or combined with system security measures. The latter isolates and mediates access between protection domains and resources; these measures are independent of the language. However, integrating an operating system style protection with the semantic information provided by the language runtime system does allow finer grained protection and sharing than is possible in contemporary systems.

The security provided by our JVM is defined in a security policy. The elements that comprise this policy are listed in table 2. They consist of a set of system resources available to each protection domain, classes whose implementation is shared between multiple domains, object instance state that is shared, and access control for each cross domain method invocation.

The first policy element is a per method access control for cross protection domain invocations. Each method has associated with it a list of domains that can invoke it. If the invocation target is not in this set, access is denied. Protection is between domains, not within domains, hence there is no access control for method invocations within the same domain.

To reduce the number of cross protection domain calls (XMIs) the class text (instructions) can be shared between multiple domains. This is analogous to text sharing in UNIX, where the instructions are loaded into

Granularity	**Mechanism**
Method	Invocation access control
Class	Instruction text sharing between domains
Class	Object sharing between domains
Reference	Opaque object handle
System	Paramecium name space per domain

Table 2. Security policy elements.

memory only once and mapped into each domain that uses it. This reduces memory requirements. In our case it eliminates the need for expensive XMIs. The object instance state is still private to each domain.

Object instance state is transparently shared between domains when references to it are passed over XMIs or when an object inherits from a class in a different protection domain. Which objects can be passed between domains is controlled by the Java programs and not by the JVM. Specifying this as part of the security policy would break the Java language transparency requirement. Per-method access control gives the JVM the opportunity to indirectly control which references are passed.

In circumstances where object instance state sharing is not desirable a class can be marked as non-sharable for a specified domain. Object references of this class can still be passed to the domain but cannot be dereferenced by it. This situation is similar to client/server mechanisms where the reference acts as an opaque object handle. Since Java is not a real object-oriented language, it allows clients to directly access object state, this mechanism is not transparent for some Java programs.

Fine grained access control over the system resources is provided by the Paramecium name space mechanism. If a service name is not in the name space of a protection domain, that domain cannot get access to the service. The name space for each protection domain is constructed and controlled by our Java Nucleus.

To reduce the number of XMIs the classes with the same security policy are grouped into the same protection domain. The number of XMIs can be further reduced by sharing the instruction text of class implementations between different domains.

Table 3 summarizes the potential threats our JVM can handle, together with their primary protection mechanism. Some threats, such as covert channels, are not handled in our system. Other threats, such as denial of service attacks caused by improper locking behavior are considered policy errors. The offending applet should not have been given access to the mutex.

Threat	**Protection mechanism**
Fault isolation	Protection domains
Denial of service	Resource control
Forged object references	Garbage collector
Illegal object invocations	XMI access control

Table 3. Threat model.

4. Paramecium

Paramecium [41] is an extensible object-based operating system for building application-specific operating systems. It consist of a protected and trusted nanokernel which implements only those services that cannot be moved into the application protection domain without jeopardizing the system's integrity. All other system components, like thread packages, device drivers, and virtual memory implementations reside outside this nucleus.

The kernel provides three basic services which all use a protection domain as their unit of granularity. These services are: event management, memory management, and name space management. Each resource managed by these services is identified by a capability.

The first basic service provided by the kernel is event management. Paramecium uses preemptive events for handling interrupts, traps, and interprocess communication. Associated with each event is a handler which is executed when the event is raised. A handler consists of a protection domain identifier, the address of a call-back function, and a stack pointer. Raising an event causes control to be transfered to the handler specified by the protection domain identifier and call-back function using the specified handler stack. The event service also has provisions for the handler to determine the caller's domain.

The second basic service provided by the kernel is memory management. This service manages physical and virtual memory. The physical memory service allocates physical pages which are then mapped onto a virtual memory address using the virtual memory service. Each physical page is identified by a generic system-wide resource identifier. Shared memory is implemented by passing this physical page resource identifier to another protection domain and having it map it into its virtual address space.

Paramecium implements multiple protection domains. Each protection domain is a mapping of virtual to physical pages together with a set of domain specific events. These domain events are raised on, for example, division by zero traps when this particular domain is in control.

The protection domain's virtual memory space is managed by the virtual memory service. This service implements functions to map physical pages onto virtual addresses, set virtual page attributes (*e.g.*, read-only, read-write, execute-only), and unmap them. Each virtual page has associated with it a fault event that is raised, if set, when a fault occurs on an address within that page. These faults include among others: an instruction access fault when a page is marked as non-executable or a data access fault when a write occurs to a read-only page. A generic fall back event is raised when no event is associated with a virtual page.

Event handlers for a virtual page fault do not need to reside in the same protection domain as the one where the fault occurs. These can be handled by an external process that might, for example, implement demand paging or, as in our secure JVM, maintain full control over the protection domains that are executing Java programs.

The last basic service provided by the kernel is name space management. Paramecium organizes its components into objects and interface references to instantiated objects. These are stored in a hierarchical name space. Each protection domain has a view of its own subtree of the name space, the kernel address space has a view of the entire tree including all the subtrees of different protection domains (see figure 2).

Standard operations exist to bind to an existing object reference, to load an object from the repository, and to obtain an interface from a given object reference. Binding to an object happens at runtime. One would reconfigure a particular service by overriding its name. A search path mechanism exists to control groups of overrides. When an object is owned by a different

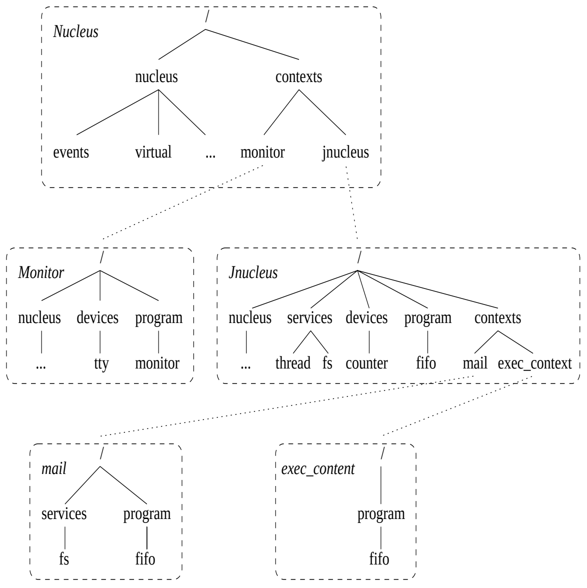

Figure 2. Paramecium name spaces.

protection domain the name service automatically instantiates proxy interfaces.

When a protection domain is created it is passed the root of its name space. Depending on the names in its space, it can contact external services. For example, the file server is known as ‘‘/services/fs’’. Binding to this name and obtaining a file system interface enables the process to create and delete files by invoking the methods from the interface. When the name is not present in the name space no file system operations are possible. By default protection domains are created with an empty name space.

Applications that use this kernel are implemented as and constructed from separate components. One example of a component is Paramecium’s thread package. This package provides a priority scheduler and support for migratory threads [18]. Migratory threads can migrate from one protection domain to another without actually switching threads or changing the thread identifier. This saves a number of context switches which are required by systems that hand off work from a thread in one domain to a thread in another domain.

5. Secure Java Virtual Machine

The Java Nucleus forms the minimal trusted computing base (TCB) of our secure JVM. This section describes the key techniques and algorithms used by the Java Nucleus.

In short, the Java Nucleus provides a uniform naming scheme for all protection domains, including the Java Nucleus. It provides a single virtual address space where each protection domain can have a different protection view. All cross protection domain method invocations (XMIs) pass through our Java Nucleus which controls access, CPU and memory resources. Data is shared on demand between multiple protection domains, *i.e.* whenever a reference to shared data is dereferenced. Our Java Nucleus uses shared memory and runtime reallocation techniques to accomplish this. Only references passed over an XMI or object instances whose inherited classes are in different protection domains can be accessed, others will cause security violations. These protection mechanisms depend on our garbage collector to allocate and deallocate typed memory, relocate memory, control memory usage, and keep track of ownership and sharing status.

It is possible to use the techniques described below to build a secure JVM using an interpreter rather than a compiler. Each protection domain would then have a shared copy of the interpreter interpreting the Java byte codes for that protection domain. We have not explored such an implementation because of the obvious performance benefits of executing generated machine code.

The next subsections describe the key techniques and algorithms in greater detail.

5.1. Memory Organization

The Java Virtual Machine assumes a single address space in which references can be passed between method invocations. This and Java's dependence of garbage collection dictated our memory organization.

Inspired by single address space operating systems [10], we organized memory into a single virtual address space. Each protection domain has, depending on its privileges, a view onto this address space. This view includes a set of physical memory pages to virtual mappings together with their corresponding access rights. A small portion of the virtual address space is reserved by each protection domain to store per domain specific data.

Central to the protection domain scheme is the Java Nucleus (see figure 3). The Java Nucleus is analogous to an operating system kernel. It manages a number of protection domains and has full access to all memory mapped into these domains and their corresponding access permissions. The protection domains themselves cannot manipulate the memory mappings or the access rights of their virtual memory pages. The Java Nucleus handles both data and instruction access (*i.e.*, page) faults for these domains. Page faults are turned into appropriate Java exceptions when they are not used by the system.

For convenience all the memory available to all protection domains is mapped into the Java Nucleus with read/write permission. This allows it to quickly access the data in different protection domains. Because memory addresses are unique and the memory pages are mapped into the Java Nucleus protection domain, the Java Nucleus does not have to map or copy memory as an ordinary operating system kernel.

The view different protection domains have of the address space depends on the mappings created by the Java Nucleus. Consider figure 3. A mail reader application resides in the context named mail. For efficiency reasons, all classes constituting this application reside in the same protection domain; all executable content embedded in an e-mail message is executed in a separate domain, say executable content. In this example memory region 0 is mapped into the context executable content. Part of this memory contains executable code and has the execute privilege associated with it. Another part contains the stack and data and has the read/write privilege. Region 0 is only visible to the executable content context and not to the mail context. Likewise, region 2 is not visible to the executable content context. Because of the hardware memory mappings these two contexts are physically separated.

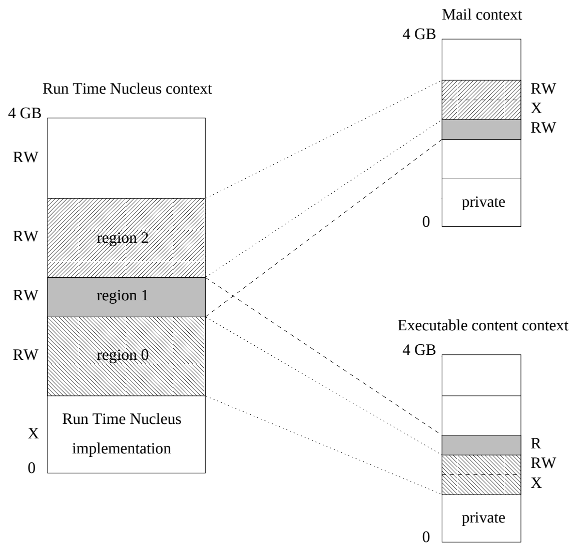

Figure 3. Java nucleus memory map.

Region 1 is used to transfer data between the two contexts and is set up by the Java Nucleus. Both contexts have access to the data, although the executable content context has only read access. Violating this access privilege causes a page (data access) fault to be generated which is handled by the Java Nucleus. It will turn the fault into a Java exception.

5.2. Cross Domain Method Invocations

A cross domain method invocation (XMI) mimics a local method invocation except that it crosses a protection domain boundary. A vast amount of literature exists on low latency cross domain control transfer [5, 24, 28]. Our XMI mechanism is loosely based on Paramecium's system call mechanism which uses events. The following example illustrates the steps involved in an XMI.

Consider the protection domains A and B and a method M which resides in domain B. When A calls method M an instruction fault will occur since M is not mapped into context A. The fault causes an event to be raised in the Java Nucleus. The event handler for this fault is passed two arguments: the fault address (*i.e.*, method address) and the fault location (*i.e.*, call instruction). Using the method address, the Java Nucleus determines the method information which contains the destination domain and the access control information. Paramecium's event interface is used to determine the caller domain. Based on this information, an access decision is made. If access is denied, a security exception is raised in the caller domain.

Using the fact that method information is static and that domain information is static for code that is not shared, we can improve the access control check process. Rather than looking up this information, the Java Nucleus stores a pointer to it in the native code segment of the calling domain. The information can then be accessed quickly using a fixed offset and fault location parameter. Method calls are achieved through special trampoline code that embeds these two values. More precisely, the call trampoline code fragment in context A for calling method M appears as (in SPARC [38] assembly):

```
    call  M                ! call method M
      mov   %g0, %i0       ! nil object
    b,a   next_instr       ! branch over
    .long <caller domain> !   JNucleus data
    .long <method info>   !   JNucleus data
next_instr:
```

The information stored in the caller domain must be protected from tampering. This is achieved by mapping all executable native code as execute only; only the Java Nucleus has full access to it.

When access is granted for an XMI, an event is associated with the method if one is not already present. Then the arguments are copied into the registers and onto the event handler stack as dictated by the calling frame convention. No additional marshalling of the parameters is required. Both value and reference parameters are passed unchanged. Using the method's type signature to identify reference parameters, we mark data references as exported roots (*i.e.*, garbage collection roots). Instance data is mapped on demand as described in the next section. Invoking a method on an object reference causes an XMI to the method implementation in the object owner's protection domain.

Virtual method invocations, where a set of specific targets is known at compile-time but the actual target only at runtime, require a lookup in a switch table. The destinations in this table refer to call trampolines rather than the actual method address. Each call trampoline consists of the code fragment described above.

Using migratory threads, an XMI extends the invocation chain of the executing thread into another protection domain. Before raising the event to invoke the method, the Java Nucleus adjusts the thread priority according to the priority of the destination protection domain. The original thread priority is restored on the method return. Setting the thread priority enables the Java Nucleus to control the CPU resources used by the destination protection domain.

Local method invocations use the same method call trampoline as the one outlined above, except that the Java Nucleus does not intervene. This is because the method address is available locally and does not generate a fault. The uniform trampoline allows the Java Nucleus to share class implementations among multiple protection domains by mapping them in. For example, simple classes like the java.lang.String or java.lang.Long can be shared by all protection domains without security implications. Sharing class implementations reduces memory use and improves performance by eliminating XMIs. XMIs made from a shared class do not have their caller domain set, since there can be many caller domains, and require the Java Nucleus to use the system authentication interface to determine the caller.

5.3. Data Sharing

Passing parameters, as part of a cross domain method invocation (XMI), requires little more than copying them by value and marking the reference variables as exported roots. Subsequent accesses to these references will cause a protection fault unless the reference is already mapped in. The Java Nucleus, which handles the access fault, will determine whether the faulting domain is allowed access to the variable referenced. If allowed, it will share the page on which the variable is located.

Sharing memory on a page basis traditionally leads to false sharing or fragmentation. Both are clearly undesirable. False sharing occurs when a variable on a page is mapped into two address spaces and the same page contains other unrelated variables. This clearly violates the confinement guarantee of the protection domain. Allocating each variable on a separate page results in fragmentation with large amounts of unused physical memory. To share data efficiently between different address spaces, we use the garbage collector to reallocate the data at runtime. This prevents false sharing and fragmentation.

Consider figure 4 which shows the remapping process to share a variable *a* between the mail context and the executable content context. In order to relocate this variable we use the garbage collector to update all the references. To prevent race conditions the threads within or entering the contexts that hold a reference to *a* are suspended (step 1). Then the data, *a*, is copied onto a new memory page (or pages depending on its size) and referred to as *a′*. The other data on the page is not copied, so there is no risk of false sharing. The garbage collector is then used to update all references that point to *a* into references that point to *a′* (step 2). The page holding *a′* is then mapped into the other context (step 3) Finally, the threads are resumed, and new threads are allowed to enter the unblocked protection domains (step 4). The garbage collector will eventually delete *a* since it does not have any references to it.

Other variables that are shared between the same protection domains are tagged onto the already shared pages to reduce memory fragmentation. The process outlined above can be applied recursively. That is, when a third protection domain needs access to a shared variable the variable is reallocated on a page that is shared between the three domains.

In order for the garbage collector (see section 5.4) to update the cell references it has to be exact. That is, it must keep track of the cell types and of references to each cell to distinguish valid pointers from random integer values. The updating itself can either be done by a full walk over all the in-use memory cells or by arranging each cell to keep track of the objects that reference it. The overhead of the relocation is amortized over subsequent uses.

Besides remapping dynamic memory, the mechanism can also be used to remap static (or class) data. Absolute data memory references can occur within the native code generated by the just-in-time compiler. Rather than updating the native code on each relocation, the just-in-time compiler generates an extra indirection to a placeholder holding the actual reference.

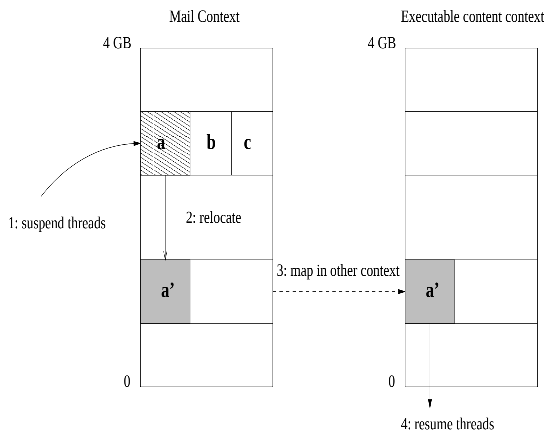

Figure 4. Data remapping between address spaces.

This placeholder is registered with the garbage collector as a reference location.

Data remapping is not only used to share references passed as parameters over an XMI, but also to share object instance data between sub and superclasses in different protection domains. Normally, object instances reside in the protection domain in which their class was loaded. Method invocations on that object from different protection domains are turned into XMIs. In the case of an extended (*i.e.*, inherited) class the object instance state is shared between the two protection domains. This allows the sub and superclass methods to directly access the instance state rather than capturing all these accesses and turning them into XMIs. To accomplish this our JVM uses the memory remapping technique outlined above.

The decision to share object instance state is made at the construction time of the object. Construction involves calling the constructor for the class followed by the constructors for its parent classes. When the parent class is in a different protection domain the constructor invocation is turned into an XMI. The Java Nucleus performs the normal access control checks as for any other XMI from a different protection domain. The object instance state, that is passed implicitly as the first argument to the constructor call, is marked as an exportable root. The mechanisms involved in marking memory as an exportable root are discussed in the next section.

Java uses visibility rules (*i.e.*, *public* and *protected*) to control access to parts of the object instance state. Enforcing these rules through memory protection is straightforward. Each object’s instance state is partitioned into a shared and non-shared part. Only the shared state can be mapped.

An example of state sharing between super and subclass is shown in figure 5. Here the class BitMap and all its instances reside in protection domain A. Protection domain B contains all the instances of the class Draw. This class is an extension of the BitMap class which resides in a different protection domain. When a new instance of Draw is created the Draw constructor is called to initialize the class. In this case the constructor for Draw is empty and the constructor for the superclass BitMap is invoked. Invoking this constructor will cause a transfer into the Java Nucleus.

The Java Nucleus first checks the access permission for domain B to invoke the BitMap constructor in domain A. If granted, the object pointer is marked as an exportable root and passed as the first implicit parameter. Possible other arguments are copied as part of the XMI mechanism and the remote invocation is

```
class BitMap { // Domain A
    private static int N = 8, M = 8;
    protected byte bitmap[][];

    protected BitMap() {
        bitmap = new byte[N/8][M];
    }

    protected void set(int x, int y) {
        bitmap[x/8][y] |= 1<<(x%8);
    }
}

class Draw extends BitMap { // Domain B
    public void point(int x, int y) {
        super.set(x, y);
    }

    public void box(int x1, int y1,
                    int x2, int y2) {
        for (int x = x1; x < x2; x++)
            for (int y = y1; y < y2; y++)
                bitmap[x/8][y] |= 1<<(x%8);
    }
}
```

Figure 5. Simple box drawing class.

performed. The BitMap constructor then assigns a new array to the bitmap field in the Draw object. Since the assignment is the first dereference for the object it will be remapped into domain A. When the creator of the Draw object calls box and dereferences bitmap it will be remapped into domain B (because the array is reachable from an exported root cell to domain A; see next section). Further calls to box do not require this remapping. A call to point results in an XMI to domain A where the superclass implementation resides. Since the Draw object was already remapped by the constructor it is does not require any remapping.

Whenever a reference is shared among address spaces all references that are reachable from it are also shared and will be mapped on demand when referred to. This provides full transparency for Java programs which assume that a reference can be passed among all its classes. A potential problem with on-demand remapping is that it dilutes the programmers' notion of what is being shared over the life-time of a reference. This might obscure the security of the system. To strengthen the security, an implementation might decide not to support remapping of objects at all or provide a proactive form of instance state sharing. Not supporting instance state sharing prevents programs that use the object oriented programming model from being separated into multiple protection domains. For example, it precludes the isolation and sharing of the AWT package in a separate protection domain.

The implementation has to be conservative with respect to references passed as arguments to cross domain method invocations and has to unmap them whenever possible to restrict their shared access. Rather than unmapping at the invocation return time, which would incur a high call overhead, we defer this until garbage collection time. The garbage collector is aware of shared pages and determines whether they are reachable in the context they are mapped in. If they are unreachable, rather than removing all the bookkeeping information the page is marked invalid so it can be remapped quickly when it is used again. This does not work very well if the page contains two variables of which only one is passed by reference. In that case a new remapping is required. To reduce the amount of remapping, separate pages are used for shared instance state and state passed as a reference to a cross domain invocation.

5.4. Garbage Collection

Java uses garbage collection [25] to reclaim unused dynamic memory. In our design we use a non-collecting incremental traced garbage collector which is part of the Java Nucleus. The garbage collector is responsible for collecting memory in all the address spaces it manages. A centralized garbage collector has the advantage that it is easier to share memory between different protection domains and to enforce central access and resource control. An incremental garbage collector has better real time properties than non-incremental collectors.

More precisely, the garbage collector for our secure Java machine must have the following properties:

(1) Collect memory over multiple protection domains and protect the bookkeeping information from the potentially hostile domains.

(2) Relocate data items at runtime. This property is necessary for sharing data across protection domains. Hence, we use an exact garbage collector rather than a conservative collector [8].

(3) Determine whether a reference is reachable from an exported root. Only those variables that can be obtained via a reference passed as an XMI argument or instance state are shared.

(4) Maintain, per protection domain, multiple memory pools with different access attributes. These are execute only, read-only, and read-write pools that contain native code, read-only and read-write data segments respectively.

(5) Enforce resource control limitations per protection domain.

As discussed in the previous section all protection domains share the same virtual address map albeit with different protection views of it. The Java Nucleus protection domain, which contains the garbage collector, has full read-write access to all available memory. Hence the ability to collect memory over different domains is confined to the Java Nucleus.

A key feature of our garbage collector is that it integrates collection and protection. Classical tracing garbage collection algorithms assume a single address space in which all memory cells have the same access rights. In our system cells have different access rights depending on the protection domain accessing it and cells can be shared among multiple domains. Although access control is enforced through the memory protection hardware, it is the garbage collector that has to create and destroy the memory mappings.

The algorithm we use (see the pseudo-code in figure 6) is an extension of a classic mark-sweep algorithm which runs concurrently with the mutators [13]. The original algorithm uses a tricolor abstraction in which all cells are painted with one of the following colors: *black* indicates that the cell and its immediate descendents have been visited and are in use; *grey* indicates that the cell has been visited but not all of its descendents, or that its connectivity to the graph has changed; and *white* indicates untraced (*i.e.*, free) cells. The garbage collection phase starts with all cells colored white and terminates when all traceable cells have been painted black. The remaining white cells are free and can be reclaimed.

To extend this algorithm to multiple protection domains we associate with each cell its owner domain and an export set. An export set denotes to which domains the cell has been properly exported. Garbage collection is performed on one protection domain at a time, each keeping its own color status to assist the marking phase. The marking phase starts by coloring all the root and exported root cells for that domain as grey. It then continues to examine all cells within that domain. If one of them is grey it is painted black and all its children are marked grey until there are no grey cells left. After the marking phase, all cells that are used by that domain are painted black. The virtual pages belonging to all the unused white cells are unmapped for that domain. When the cell is no longer used in any domain it is marked free and its storage space is reclaimed. Note that the algorithm in figure 6 is a simplification of the actual implementation, many improvements (such as [14, 26, 36]) are possible. A correctness proof of the algorithm follows from [13].

```
COLLECT():
    for (;;) {
      for (d in Domains)
        MARK(d)
      SWEEP();
    }

MARK(d: Domain): // marker phase
    color[d, (exported) root set] = grey
    do {
      dirty = false
      for (c in Cells) {
        if (color[d, c] == grey) {
          color[d, c] = black
          for (h in children[c]) {
            color[d,h] = grey
            if (EXPORTABLE(c, h))
              export[d,h] |= export[d,c]
          }
          dirty = true
        }
      }
    } while (dirty)

SWEEP(): // sweeper phase
    for (c in Cells) {
      used = false
      for (d in Domains) {
        if (color[d, c] == white) {
          export[d, c] = nil
          UNMAP(d, c)
        } else
          used = true
        color[d, c] = white
      }
      if (used == false)
        DELETE(c)
    }

ASSIGN(a, c): // pointer assignment
    *a = c
    d = current domain
    export[d,c] |= export[d,a]
    if (color[d, c] == white)
      color[d, c] = grey

EXPORT(d: Domain, c: Cell): // export object
    color[d,c] = grey
    export[d,c] |= owner(c)
    export[owner(c),c] |= d
```

Figure 6. Multiple protection domain garbage collection.

Cells are shared between other protection domains by using the remapping technique described in the previous section. In order to determine whether a protection domain d has access to a cell c the Java Nucleus has to examine the following three cases: The trivial case is where the owner of c is d. In this case the cell is already mapped into domain d. In the second case the owner of c has explicitly given access to d as part of an XMI parameter or instance state sharing or is directly reachable from such an exported root. This is reflected in the export information kept by the owner of c. Domain d has also access to cell c if there exists a transitive closure from some exported root r owned by the owner of c to some domain z. From this domain z there must exist an explicit assignment which resulted in c being inserted into a data structure owned by d or an XMI from the domains z to d passing cell c as an argument. In the case of an assignment the data structure is reachable from some other previously exported root passed by d to z. To maintain this export relationship each protection domain maintains a private copy of the cell export set. This set, usually nil and only needed for shared memory cells, reflects the protection domain's view of who can access the cell. A cell's export set is updated on each XMI (*i.e.*,export) or assignment as shown in figure 6.

Some data structures exist prior to, for example, an XMI passing a reference to it. The export set information for these data structures is updated by the marker phase of the garbage collector. It advances the export set information from a parent to all its siblings taking the previously mentioned export constraints into account.

Maintaining an export set per domain is necessary to prevent forgeries. Consider a simpler design in which the marker phase advances the export set information to all siblings of a cell. This allows the following attack where an adversary forges a reference to an object in domain d and then invokes an XMI to d passing one of its data structures which embeds the forged pointer. The marker phase would then eventually mark the cell pointed to by the forged reference as exported to d. By maintaining for each cell a per protection domain export set forged pointers are impossible.

Another reason for keeping a per protection domain export set is to reduce the cost of a pointer assignment operation. Storing the export set in the Java Nucleus would require an expensive cross protection domain call for each pointer assignment, by keeping it in user space this can be eliminated. Besides, the export set is not the only field that needs to be updated. In the classic Dijkstra algorithm the cell's color information needs to be changed to *grey* on an assignment (see figure 6). Both these fields are therefore kept in user space.

The cell bookkeeping information consists of three parts (see figure 7). The public part contains the cell contents and its per domain color and export information. These parts are mapped into the user address space, where the color and export information is stored in the per domain private memory segment (see 5.1). The nucleus part is only visible to the Java Nucleus. A page contains one or more cells where for each cell the content is preceded by a header pointing to the public information. The private information is obtained by hashing the page frame number to get the per page information which contains the private cell data. The private cell data contains pointers to the public data for all protection domains that share this cell. When a cell is shared between two or more protection domains the pointer in the header of the cell refers to public cell information stored in the private domain specific portion of the virtual address space. The underlying physical pages in this space are different and private for each protection domain.

To amortize the cost of garbage collection, our implementation stores one or more cells per physical memory page. When all the cells are free the page is added to the free list. As stated earlier, each protection domain has three memory pools: an execute-only pool, a read-only pool, and a read-write pool. Cells are allocated from these pools depending on whether they contain executable code, constant data, or volatile data. When memory becomes really tight, pages are taken from their free lists, their virtual pages are unmapped, and their physical pages returned to the system physical page pool. This allows them to be re-used for other domains and pools.

Exposing the color and export set fields requires the garbage collector to be very conservative in handling these user accessible data items. It does not,

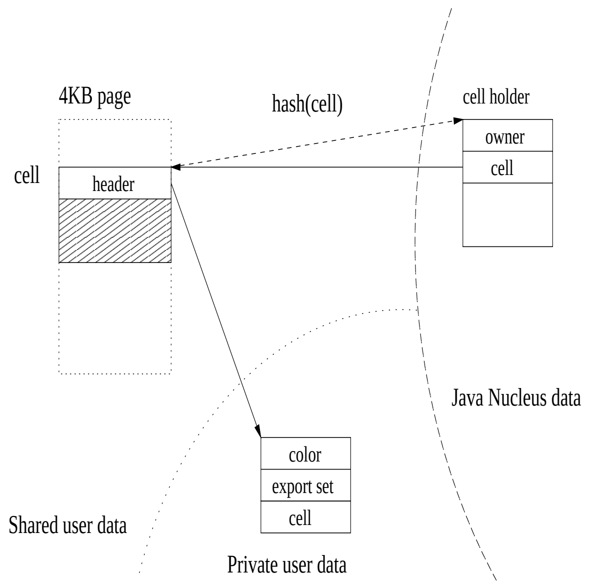

Figure 7. Garbage collection cell structure.

however, reduce the security of our system. The user application can, at most, cause the marker phase to loop forever, cause its own cells that are still in use to be deallocated, or hang on to shared pages. These problems can be addressed by bounding the marker loop phase by the number of in-use cells. Deleting cells that are in use will cause the program to fail eventually, and hanging on to shared pages is not different from the program holding on to the reference.

When access to a cell is revoked, for example as a result of an XMI return, its color is marked grey and it is removed from the receiving domain's export set. This will cause the garbage collector to reexamine the cell and unmap it during the sweep phase when there are no references to it from that particular domain.

To relocate a reference the Java Nucleus forces the garbage collector to start a mark phase and update the appropriate references. Since the garbage collector is exact it only updates actual object references. An alternative design for relocation is to add an extra indirection for all data accesses. This indirection eliminates the need for explicit pointer updates. Relocating a pointer consists of updating its entry in the table. This design, however, has the disadvantage that it imposes an additional cost on every data access rather than the less frequent pointer assignment operation and prohibits aggressive pointer optimizations by smart compilers.

The amount of memory per protection domain is constrained. When the amount of assigned memory is exhausted an appropriate exception is generated. This prevents protection domains from starving other domains of memory.

6. Experience

Our prototype implementation is based on Kaffe, a freely available JVM implementation [40]. We used its class library implementation and JIT compiler and we reimplemented the IPC, garbage collector, and thread subsystems. Our prototype implements multiple protection domains and data sharing. For convenience, the Java Nucleus contains the JIT compiler and all the native class implementations. It does not yet provide support for text sharing of class implementations and has a simplified security policy description language. Currently, the security policy defines protection domains by explicitly enumerating the classes that comprise it and the access permissions for each individual method. The current garbage collector is not exact for the evaluation stack and uses a weaker form to propagate export set information.

The trusted computing base (TCB) of our system is formed by the Paramecium kernel and the Java Nucleus. The size of our Paramecium kernel is about 11000 lines of commented header files, and C++/assembler code. The current Java Nucleus is about 22000 lines of commented header files and C++ code. This includes the JIT component, threads, and much of the Java run-time support. In a system that supports text sharing the Java Nucleus can be reduced considerably.

A typical application of our JVM is that of a web server written in Java that supports servlets, like W3C's JigSaw. Servlets are Java applets that run on the web server and extend the functionality of the server. They are activated in response to requests from a web browser and act mainly as a replacement for CGI scripts. Servlets run on behalf of a remote client and can be loaded from a remote location. They should therefore be kept isolated from the rest of the web server.

Our test servlet is the SnoopServlet that is part of the Sun's Java servlet development kit [37]. This servlet inherits from a superclass HttpServlet which provides a framework for handling HTTP requests and turning them into servlet method calls. The SnoopServlet implements the GET method and returns a web page containing a description of the browser capabilities. This page is served to the client by a simple web server which is implemented by the HttpServlet superclass. For our test the web server and all class libraries are loaded in protection domain WS, the servlet implementation is confined to Servlet.

The WS domain makes 2 calls into the Servlet domain, one to the constructor for SnoopServlet object and one to the doGet method implementing HTTP GET. This method has two arguments, the servlet request and reply objects. Invoking methods on these causes XMIs back into the WS domain. In this test a total of 217 XMIs occurred. Many of these calls are to runtime classes such as java/io/PrintWriter (62) and java.lang.StringBuffer (101). In an implementation that supports text sharing these calls would be local procedure calls and only 33 calls would require an actual XMI to the web server. Many of these XMIs are the result of queries from the servlet to the browser.

The number of objects that are shared and therefore relocated between the WS and Servlet domains are 47. Most of the relocated objects are static strings (45) which are used as arguments to print the browser information. These too can be eliminated by using text sharing since the underlying implementation of print uses a single buffer. In that case only a single buffer needs to be relocated. The remaining relocated objects are the result of the the HttpServlet class keeping state information.

The cost of an XMI from the WS domain to the Servlet domain is about 11 μsec. This high cost can be purely attributed to the cost of a Paramecium's IPC on a 50 MHz SPARC, which is 5 μsec. The overhead for Java XMIs is negligible.

7. Related Work

Our system is the first to use hardware fault isolation on commodity components to supplement language protection by tightly integrating the operating system and language runtime system. In our design we concentrated on Java, but our techniques are applicable to other languages as well (*e.g.*, SmallTalk [20] and Modula3 [31]) provided they use garbage collection, have well defined interfaces, and distinguishable units of protection. A number of systems provide hardware fault isolation by dividing the program into multiple processes and use a proxy based system like RMI or CORBA, or a shared memory segment for communication between them. Examples of these systems are the J-Kernel [23] and cJVM [1]. This approach has a number of drawbacks that are not found in our system:

(1) Most proxy mechanisms use marshalling to copy the data. Marshalling provides copy semantics which are incompatible with the shared memory semantics required by the Java language.

(2) The overhead involved in marshalling and unmarshalling the data is significant compared to on demand sharing of data.

(3) Proxy techniques are based on interfaces and are not suited for other communication mechanisms such as instance state sharing. The latter is important for object oriented languages.

(4) Proxy mechanisms usually require stub generators to generate proxy stubs and marshalling code. These stub generators use interface definitions that are defined outside the language or require language modifications to accommodate them.

(5) It is harder to enforce centralized resource control within the system because proxy mechanisms encourage many independent instances of the virtual machine.

The work by Back *et. al.* [3] and Bernadat *et. al.* [4] focuses on the resource control aspects of competing Java applets on a single virtual machine. Their work is integrated into a JVM implementation while our method of resource control is at an operating system level. For their work they trust the byte code verifier.

8. Conclusions

The security provided by our JVM consists of separate hardware protection domains, controlled access between them, and system resource usage control. An important goal of our work was to maintain transparency with respect to Java programs. Our system does not, however, eliminate covert channels or solve the capability confinement and revocation problem.

The confinement and revocation problem are inherent to the Java language. A reference can be passed from one domain to another and revocation is entirely voluntary. These problems can be solved in a rather straightforward manner, but they do violate the transparency requirement. For example, confinement can be enforced by having the Java Nucleus prohibit the passing of references to cells for which the calling domain is not the owner. This could be further refined by requiring that the cell owner should have permission to call the remote method directly when its data is passed over it by another domain. Alternatively, the owner could mark the cells it is willing to share or maintain exception lists for specific domains. Revocation is nothing more that unmapping the cell at hand.

In the design of our JVM we have been very careful to delay expensive operations until they are needed. An example of this is the on-demand remapping of reference values, since most of the time reference variables are never dereferenced. Another goal was to avoid cross-protection domain switches to the Java Nucleus. The most prominent example of this is pointer assignment which is a tradeoff between memory space and security. By maintaining extra, per protection domain, garbage collector state we perform pointer assignments within the same context, thereby eliminating a large number of cross domain calls due to common pointer assignment operations. The amount of state required can be reduced by having the compiler produce hints about the potential sharing opportunities of a variable.

In our current JVM design, resources are allocated and controlled on a per protection domain basis, as in an operating system. While we think this is an adequate protection model, it might prove to be too coarse grained for some applications and might require techniques as suggested by Back *et. al.* [3].

The current prototype implementation shows that it is feasible to build a JVM with hardware separation whose Java XMI overhead is small. Many more optimizations, as described in this paper, are possible but have not been implemented yet. Most notable is the lack of instruction sharing which can improve the performance considerably since it eliminates the need for XMIs. When these additional optimizations are factored in, we believe that a hardware assisted JVM compares quite well to JVM's using software fault isolation.

The security of our system depends on the correctness of the shared garbage collector. Traditional JVMs rely on the byte code verifier to ensure heap integrity and a single protection domain garbage collector. Our garbage collector allocates memory over multiple protection domains and cannot depend on the integrity of the heap. Especially the latter requires careful analysis of all the attack scenarios. In our design the garbage collector is very conservative with respect to addresses it is given. Each address is checked against tables kept by the garbage collector itself and the protection domain owning the object to prevent masquerading. The instance state splitting according to the Java visibility rules prevents adversaries from rewriting the contents of a shared object. Security sensitive instance state that is shared, and therefore mutable, is considered a policy error or a programming error.

Separating the security policy from the mechanisms allows the enforcement of many different security policies. Even though we restricted ourself to maintaining transparency with respect to Java programs, stricter policies can be enforced. These will break transparency, but provide higher security. An example of this is the opaque object reference sharing. Rather than passing a reference to shared object state, an opaque reference is passed. This opaque reference can only be used to invoke methods on, the object state is not shared and can therefore not be inspected.

The garbage collector, and consequently runtime relocation, have a number of interesting research questions associated with them that are not yet explored. For example, the Java Nucleus is in a perfect position to make global cache optimization decisions because it has an overall view of the data being shared and the XMIs passed between domains. Assigning a direction to the data being shared would allow fine grained control of the traversal of data. For example, a client can pass a list pointer to a server applet which the server can dereference and traverse but the server can never insert one of its own data structures into the list. This is reminiscent of Shapiro's *diminish-grant* model for which confinement has been proven [34].

The Java Nucleus depends on user accessible low-level operating system functionality that is currently only provided by extensible operating systems (*e.g.*, Paramecium, OSKit [19], L4/LavaOS [28], ExOS [15], and SPIN [6]). Implementing the Java Nucleus on a conventional operating system would be considerably harder since the functionality listed above is intertwined in hard coded abstractions that are not easily adapted.

9. Acknowledgments

We would like to thank Paul Karger and Dan Wallach for many helpful discussions; and Charles Palmer, David Safford, Trent Jaeger, Andy Tanenbaum, Rajesh Bordewaker, Matthias Kaiserswerth, Wietse Venema, Jonathan Shapiro, Peter Gutmann, Larry Koved, and the anonymous referees for helpful comments on the paper.

References

1. Y. Aridor, M. Factor and A. Teperman, "cJVM: a Single System Image of a JVM on a Cluster", *Proc. of the 1999 IEEE International Conference on Parallel Processing (ICPP'99)*, Aizu-Wakamatsu City, Japan, Sep. 1999, 4-11.

2. K. Arnold and J. Gosling, *The Java Programming Language Second Edition*, Addison Wesley, Reading, MA, 1997.

3. G. Back, P. Tullman, L. Stoller, W. C. Hsieh and J. Lepreau, "Java Operating Systems: Design and Implementation", Tech. Rep. UUCS-98-015, Aug. 1998.

4. P. Bernadat, D. Lambright and F. Travostino, "Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments", *Proc. of the 19th IEEE Real-time Systems Symposium (RTSS'98)*, Madrid, Spain, Dec. 1998.

5. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, "Lightweight Remote Procedure Call", *Proc. of the 12th Symposium on Operating System Principles, ACM SIGOPS 23*, 5 (Dec. 1989), 102-113.

6. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker and C. Chambers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. of the 15th Symposium on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 267-284.

7. B. W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

8. H. Boehm and M. Weiser, "Garbage Collection in an Uncooperative Environment", *Software—Practice & Experience 18*, 9 (1988), 807-820.

9. Burroughs, *The Descriptor − a Definition of the B5000 Information Processing System*, Burroughs Corporation, Detroit, MI, 1961.

10. J. S. Chase, H. M. Levy, M. J. Feeley and E. D. Lazowska, "Sharing and Protection in a Single-address-space Operating System", *ACM Transactions on Computer Systems 12*, 4 (Nov. 1994), 271-307.

11. D. Dean, E. W. Felten and D. S. Wallach, "Java Security: From HotJava to Netscape and Beyond", *Proc. of the IEEE Security & Privacy Conference*, Oakland, CA, May 1996, 190-200.

12. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", *Comm. of the ACM 9*, 3

(Mar. 1966), 143-155.

13. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten and E. F. Steffens, “On-the-fly Garbage Collection: An Excercise in Cooperation”, *Comm. of the ACM 21*, 11 (Nov. 1978), 965-975.

14. D. Doligez and G. Gonthier, “Portable Unobtrusive Garbage Collection for Multiprocessor Systems”, *Proc. of the 21st Annual ACM SIGPLAN Notices Symposium on Principles of Programming Languages*, Jan. 1994, 70-83.

15. D. R. Engler, M. F. Kaashoek and J. O’Toole Jr., “Exokernel: An Operating System Architecture for Application-Level Resource Management”, *Proc. of the 15th Symposium on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 251-266.

16. Esmertec, “Jbed Whitepaper: Component Software and Real-Time Computing”, White paper, Esmertec, 1998. (available as http://www.jbed.com/).

17. E. Felten, Java’s Security History, (available as http://www.cs.princeton.edu/sip/history.html), 1999.

18. B. Ford and J. Lepreau, “Evolving Mach 3.0 to a Migrating Thread Model”, *Proc. of the Usenix Winter ’94 Conference*, San Francisco, CA, Jan. 1994, 97-114.

19. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers, “The Flux OSKit: A Substrate for Kernel and Language Research”, *Proc. of the 16th Symposium on Operating System Principles, ACM SIGOPS 31*, 5 (Oct. 1997), 38-51.

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

21. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison Wesley, Reading, MA, 1996.

22. S. B. Guthery and T. M. Jurgensen, *Smart Card Developer’s Kit*, Macmillian Technical Publishing, Indianapolis, IN, 1998.

23. C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu and T. Von Eicken, “Implementing Multiple Protection Domains in Java”, *Proc. of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998, 259-270.

24. W. C. Hsieh, M. F. Kaashoek and W. E. Weihl, “The Persistent Relevance of IPC Performance: New techniques for Reducing the IPC Penalty”, *Proc. Fourth Workshop on Workstation Operating Systems*, Napa, California, Oct. 1993, 186-190.

25. R. Jones and R. Lins, *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, John Wiley & sons, New York, 1996.

26. H. T. Kung and S. W. Song, “An efficient parallel garbage collection system and its correctness proof”, *IEEE Symp. on Foundations of Computer Science*, 1977, 120-131.

27. K. Li and P. Hudak, “Memory coherence in shared virtual memory systems”, *ACM Transactions on Computer Systems 7*, 4 (Nov. 1989), 321-359.

28. J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam and T. Jaeger, “Achieved IPC Performance (Still The Foundation For Extensibility)”, *Proc. of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham (Cape Cod), MA, May 1997, 28-31.

29. D. A. Moon, “Genera Retrospective”, *Proc. of the International Workshop on Object Orientation in Operating Systems, IEEE CS*, Palo Alto, CA., Oct. 1991, 2-8.

30. G. J. Myers, “Can Software for SDI Ever be Error-free?”, *IEEE computer 19*, 10 (1986), 61-67.

31. G. Nelson, *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

32. T. Saulpaugh and C. A. Mirho, *The Inside JavaOS Operating System*, Addison Wesley, Reading, MA, 1999.

33. J. S. Shapiro, D. J. Farber and J. M. Smith, “The Measured Performance of a Fast Local IPC”, *Proc. of the Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct. 1996, 89-94.

34. J. S. Shapiro and S. Weber, “Verifying Operating System Security”, MS-CIS-97-26, University of Pennsylvania, Philadelphia, PA, July 1997.

35. E. G. Sirer, Security Flaws in Java Implementations, (available as http://kimera.cs.washington.edu/flaws/index.html), 1997.

36. G. L. Steele, “Multiprocessing compactifying garbage collection”, *Comm. of the ACM 18*, 9 (Sep. 1975), 495-508.

37. SunSoft, Java Servlet Development Kit, (available as http://java.sun.com/products/servlet/index.html), 1999.

38. Sun Microsystems Inc., *The SPARC Architecture Manual*, Prentice Hall, Englewood Cliffs, NJ, 1992.

39. W. Teitelman, “A tour through Cedar”, *IEEE Software 1*, 2 (1984), 44-73.

40. Transvirtual Technologies Inc., Kaffe OpenVM, (available as http://www.transvirtual.com/), 1998.

41. L. Van Doorn, P. Homburg and A. S. Tanenbaum, “Paramecium: An extensible object-based kernel”, *Proc. of the Fifth Hot Topics in Operating Systems (HotOS) Workshop,*, Orcas Island, WA, May 1995, 86-89.

42. D. Wetherall and D. L. Tennenhouse, “The ACTIVE IP Option”, *Proc. of the Seventh SIGOPS European Workshop, ACM SIGOPS*, Connemara, Ireland, Sep. 1996.

43. N. Wirth and J. Gütknecht, *Project Oberon, The Design of an Operating System and Compiler*, ACM Press, 1992.