Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments

Philippe Bernadat — The Open Group Research Institute, 2 Avenue Vignates, F38610 Gières - FRANCE, bernadat@gr.opengroup.org

Dan Lambright — EMC corporation, 171 South St., Hopkinton, MA 01748, dlambrig@emc.com

Franco Travostino — Bay Networks, 600 Technology Park Drive, Billerica, MA 01821, travos@Baynetworks.COM

Abstract[1]

Denial-of-Service (DoS) might only be an annoyance while browsing the Web. In the emerging breed of mission-critical Internet systems, however, service guarantees are mandatory and DoS attacks can be fatal. Increasingly, such systems are being built in Java, for which vulnerability to DoS attacks is inherent in the JavaSoft Java Virtual Machine (JVM) specification. For instance, objects with different levels of trust are required to withdraw memory from a unique heap; furthermore, their threads must compete for CPU cycles with scheduling and synchronization policies that are undefined. In this paper, we describe JVM extensions and new APIs for partitioning memory and CPU resources among untrusted, mutually suspicious applications coexisting within the same JVM. Non-interference properties apply to a new formulation of name-space that includes: exclusive access to a resource budget, an API to negotiate such a budget, real-time thread control, an independent garbage collector, and the system's capability to asynchronously terminate the name-space itself. We survey the solution space, discuss our approach, describe the implementation, and finally report on our experience in applying our technology to a Java-based environment for Active Networks.

Keywords: Java, system resources, security, denial-of-service, real-time

1. This work was supported by the Defense Advanced Research Projects Agency (DARPA) when the three authors were with The Open Group Research Institute, Cambridge, MA.

1 Introduction

The increasing popularity of Java as a platform for control systems calls for security models that explicitly target service guarantees besides system integrity. In a Java-based control system—e.g., a system that controls physical devices and hosts transient mobile agents with network management duties—two or more heterogeneous functions typically come to coexist within the same Java Virtual Machine (JVM) [11]. Some functions are real-time sensitive and trusted (e.g., to control physical devices), others are not real-time sensitive and possibly untrusted (e.g., mobile agent code); hybrid combinations are also possible. Java's type-safety and its three-pronged approach—class-loader, byte-code verifier, and security manager—to securing sandboxes are not sufficient to guarantee that, for example, a visiting agent will not derail a real-time sensitive control loop by (perhaps inadvertently) mounting a memory consumption attack (according to the JavaSoft specification, a JVM's heap and garbage collector are shared by all the threads in the JVM) [19].

Attacks based on consumption of resources—even the fundamental resources, such as CPU and memory—are well-known to anybody familiar with Java [21][8]. What is far less obvious is that the impact of these attacks can rapidly escalate, from the level of mere annoyance in Web browsers to fatal errors and catastrophic black-outs in the emerging breed of Java-based control systems that demand service guarantees.

Which levels of *non-interference* can we guarantee between mutually suspicious functions collocated within a single JVM? Which types of denial-of-service can we thwart, with or without modifications to JVM or run-time implementations? Our research program is actively looking for answers to these questions; some of the answers can be found in this paper, with limitations on the types of resources and attacks being addressed. Our solution space is only limited by the requirement not to change the Java language and its bytecode representation (thus, we still deliver on the write-once-run-anywhere promise of Java; guarantees may vary depending upon local JVM and underlying OS implementations).

We classify resources as either explicitly mediated by the Java runtime (e.g., GUI windows available through AWT), implicitly mediated by the JVM (e.g., memory available through heap), or directly mapped to OS resources (e.g., a socket available through an OS interface and a Java wrapper). Mainstream Java[2] supports access controls for resources only in the first and last categories; augmenting these controls with the notions of principal and maximum allowed quotas is a fairly straightforward proposition. Conversely, the problem of thwarting harmful consumption of JVM-mediated resources—CPU and memory— has been hitherto ignored in its entirety by mainstream Java [26]. Secure partitioning of these resources is the thrust of the work described in the paper. To enable service guarantees, our defense mechanisms must be activated before consumption of CPU or memory can result in a denial-of-service attack via depletion of the resource.

For convenience, we classify harmful consumption of CPU and memory into two broad groups, based on how the resource is consumed. *First order attacks* are carried out by hackers with limited efforts, or by non-malicious run-away code; the code capitalizes on overt loopholes (e.g., a forever loop wherein more resources are referenced at each iteration). *Second order attacks* employ more subtle strategies, like the ones that an attacker may elaborate after analyzing the internals of a particular implementation (e.g., a fragmentation attack against an algorithm for garbage collecting memory).

In this paper, we describe our effort to partition Java's CPU and memory, and to thwart first and second order attacks on these resources. To this extent, we present a novel incarnation of **name-space** with some unique properties:

- Exclusive access to a resource budget;
- An API to negotiate such a budget with the system;
- Real-time thread control;
- A dedicated, independent garbage collector;
- Asynchronous termination of the whole name-space with resources being reclaimed by the system;

We demonstrate that this new name-space is a proper abstraction to reason about non-interference properties among untrusted, mutually suspicious code.

The paper proceeds with a motivating scenario where first and second order denial-of-service attacks can be fatal. We introduce typical policies that will drive the mechanisms for partitioning Java's CPU and memory. We continue with the evaluation of various approaches to realizing these mechanisms, their scope, limits, and level of intrusiveness on the JVM. We motivate our choice of a particular approach, and we describe how we have proceeded with its actual implementation; empirical data from micro-benchmarks follow. We then present an example wherein policy and mechanisms combine to realize a secure, Java-based network node for Active Networks.

2 An example

In traditional networks, transient packets carry data for routers to forward according to protocols already resident on routers. With Active Networks [29], researchers have blurred the dichotomy between transient data packets and resident protocols; *capsules* carry data and optionally some architecture-neutral code to be executed at participating routers. The use of capsules enables a host of new opportunities, including advanced QoS, self-adaptive connections, efficient data dissemination, network upgrades, etc. [9][15][28][39]

The logical step from Active Networks to routers providing Java execution services is easy to make. Some of the capsules in transit must execute Java byte code in their pay-load, other capsules may also execute code or peruse state that was left behind by predecessor capsules, and yet other capsules may also need to consult node-wide information such as routing tables, etc. In all cases, capsules are mutually suspicious and untrusted by the router; furthermore capsules may have deadlines and be sensitive to delays caused by other capsules.

A capsule is evaluated within a type of Java sandbox specifically designed for active networking (i.e., with its dedicated byte-code verifier, class-loader and security manager); other than that, we begin with an off-the-shelf Java platform and we investigate its shortcomings.

First order attacks on CPU and memory consumption lead to results that are both unpredictable and dependent on the JVM implementation. In the case of a memory shortage, for instance, most JVMs will attempt to throw an OutOfMemory exception to the thread in progress, which may or may not be the one responsible for the shortage. In some cases, the JVM might simply terminate itself.

The JVM can be modified to impose a quota for CPU and memory within each sandbox (e.g., a name-space). The processing of capsules would still be vulnerable to more subtle second order attacks such as:

- triggering the garbage collector.
- memory fragmentation attacks.
- exploiting conservative based garbage collectors.
- stalling finalization of objects.

One possible line of defense is to only evaluate capsules whose originator we recognize and trust, and whose authenticity and integrity can be verified. This method, however,

2. As in Sun's JDK 1.2.

does not address the problems of non-malicious but harmful code (e.g., bugs) [21]; nor does it address problems or attacks coming from the composition of elemental units that are individually authenticated and certified to be harmless, but are harmful in their combined state.

Another coarse line of defense is to establish conventions on untrusted code executed by a capsule, e.g., forbidding iterative constructs whose total elapsed time cannot be determined statically [15]. Methods such as this greatly limit the expressive power of the language [16] and thus the growth potential of Active Networks.

3 Typical policies

The mechanisms for partitioning resources and the policies driving these mechanisms must be clearly separated. This allows us to employ multiple policies (customized to various environments) without duplicating the engineering effort to implement the enforcement mechanisms [12].

The notion of principal — an entity in the system to which authorizations and accountability apply — is an invariant for all policies. The mechanisms, whether inside or outside the JVM, must be able to unambiguously distinguish among principals. How a principal is represented is entirely orthogonal to our work; one possibility is strings that represent public keys derived from a specific crypto-system.

We anticipate using at least two policies. The first one grants principals a *fair access* to target resources; a principal will never be starved regardless of the trusted or untrusted workload within the system.

The second policy realizes the QoS-centric model of *you pay for what you use* [6]: a principal may rightfully count on a disproportionate share of resources after successfully negotiating these resources with the system. Billing and other models inspired by economics rely upon the correct enforcement of this policy. There is no systemic prevention of starvation, and it is up to the policy builder to control whether starvation should or should not happen.

It is indeed possible to realize the fair access policy as a special case of the latter policy (the contrary is not true). For convenience, however, we prefer to distinguish these as two separate policies because we want to itemize the implications of each policy on the underlying mechanisms.

In all cases of violations, we require that only those principals responsible for excessive resource consumption be penalized and brought back to conforming behaviors. This is a significant departure from mainstream Java where any code that happens to execute may pay all the consequences of harmful resource consumption.

4 Types of mechanisms

We survey several mechanisms to partition CPU and memory resources among principals, to achieve non-interference goals, and to support *fair access* and *you pay for what you use* policies.

Multiple instantiation. Coercing multiple principals to coexist within a unique JVM results in a radical departure from the original Java model (i.e., there is one principal who owns the whole Virtual Machine, permissions and restrictions apply to classes).

In many cases, there is an easier way. For instance, it may be possible to achieve non-interference goals via multiple instantiation of JVMs, one per principal or per group of functions with high affinity with one another. Each JVM would be encapsulated into a *wrapper*—a software module that mediates between an unmodifiable or misbehaving system and the external system.

An existing, well understood wrapper is a UNIX or NT process. The kernel enforces memory limits, and schedules each process [18]. To isolate mutually suspicious functions, one could instantiate them in separate JVMs, which in turn execute in different processes.

A distinct advantage to using multiple instantiation and wrappers is simplicity (i.e., we can continue to use off-the-shelf JVMs). There are three major shortcomings. First, inter-JVM communication for coordination and sharing of information among principals (e.g., routing tables) become problematic. To remain platform neutral, one could adopt solutions available in the Java space, such as RMI [29]; alternately, one could use platform specific IPC mechanisms. In both cases, the need for rich interaction among JVMs can very well erode the whole simplicity argument.

Second, at the rate of one JVM and wrapper per principal, scalability can quickly become a problem. Without considering the costs of instantiating a JVM, the wrapper itself may come with a cost. In most UNIX systems, a process requires several non-pageable and thus precious resources to be instantiated [18][1]. One could object to the heavy-weight nature of mapping between wrapper and UNIX process, and create a more lightweight wrapper, with two or more JVMs coexisting within a UNIX process; in this case, however, the simplicity argument is eroded again.

Third, multiple instantiation of JVMs and wrappers may not be an option at all. A JavaOS system is, for example, a system which only provides a Java run-time [20]; such a run-time cannot be cloned nor encapsulated in a wrapper using off-the-shelf technology.

In summary, the applicability of multiple instantiation techniques combined with wrappers is limited at best; when they do apply, however, the simplicity argument makes them a very strong candidate.

We now proceed with the analysis of more active solutions to partitioning CPU and memory.

Byte-code editing.

With *load-time byte-code editing* [4], we modify byte-code prior to execution so that it conforms to particular coding conventions. These conventions—e.g., place a yield() within every loop—are far too pedantic and impractical for a Java programmer to manually comply to; furthermore, we could not trust the programmer for compliance, and we would have to write a byte-code verifier to perform validation. We prefer a byte-code modifier that piggybacks on a standard byte-code verifier and removes the burden from the programmer.

These byte-code editing techniques draw their strength from the richness of the Java byte-code; there is no dependence on the JVM implementation. On the other hand, these techniques exhibit severe limits in accuracy (e.g., what is the memory footprint of an object? what is the cost of a code path?). They are unable to thwart second order attacks like heap fragmentation attacks. Neither are these techniques neutral with respect to performance; there is a trade-off between performance and accuracy, and each system can dynamically pursue its optimal trade-off based upon local knowledge (e.g., attack history, location within the network, etc.) [4].

Explicit code conventions.

With this technique, we explicitly impose coding restrictions on the Java programmer; adherence to the conventions must be statically verifiable by an extended byte-code verifier. For instance, a code convention can mandate that memory allocations cannot be performed in loops, and that the number of iterations in a loop must be statically known.

Like byte-code editing, this approach avoids JVM dependencies. The disadvantage is that accuracy must be balanced against code versatility; to reach even a modest level of accuracy, the expressive power of the language must be greatly limited. For example, all iterators have to be considered harmful and thus unacceptable. When used in combination with byte-code editing, this approach acts as an accuracy booster, in that it further restricts cases where byte-code analysis would be circumvented.

Direct JVM modifications.

So far, we have analyzed techniques that are not intrusive with respect to the JVM and are also independent of any particular JVM implementation.

We now consider thwarting resource attacks by binding resources to the sandbox throughout the JVM. First, we can monitor resource consumption as described in the byte-code editing section, but with much more accuracy and fewer performance problems. Secondly, we can aim at much stronger non-interference among sandboxes. For this, we map a name space to a private partition of the memory heap, with its own garbage collector; a principal can debit and credit memory from the private heap until its exhaustion, but without implications on other heaps. Likewise, threads can be scheduled within the JVM such that a sandbox always receives, for example, a given percentage of CPU cycles.

Direct JVM modifications have the distinct advantage of accuracy due to internal knowledge of the JVM. Depending on how tightly we bind resources to the sandbox abstraction in the JVM, we can counter both first and second level attacks; thus, we can aim at a strong non-interference among principals. Performance degradation is marginal with respect to the above mentioned mechanisms.

The disadvantage to this approach is that it demands a substantial engineering effort. The risks of compromising compliance to Java standards, opening new security holes, or introducing subtle OS dependencies make any intervention on the JVM particularly challenging.

The four approaches described in this section are not mutually exclusive. It is conceivable to use multiple instantiations of JVM for some high-level classification—e.g., paying customers vs. by-standers. For further classifications that lead to stricter QoS classes—e.g., one item buyers vs. large quantity buyers—one could use a JVM with direct modifications for memory control only and byte-code editing techniques for CPU control only.

5 Our solution

We decided to adopt **direct JVM modifications** as our core partitioning mechanism for the strongest non-interference, because we are interested in thwarting first and second level attacks on CPU and memory, and we are interested in policies more complex than *fair share*.

5.1 CPU control

In order to realize the two policies relevant to us, Java threads must be preemptible (e.g., otherwise a thread that is expected to run to completion could instead never yield the processor). To manage the CPU resource effectively, we must also require the availability of thread scheduling policies that track CPU accrual [22] (e.g., fixed priority round-robin, time-sharing), with some control over the scheduling attributes being exposed through the Java API.

We adapt our target JVM to map Java threads and synchronization primitives to the POSIX standard (1003.1c-1995). This standard defines a number of desirable real-time features, including support for multiple scheduling policies and priority inheritance. In turn, the layer that provides POSIX compliance may use user-space or native threads, internally. Native threads are a desirable feature for responsive systems in general because all scheduling decisions are centralized into a single locus of control—the kernel scheduler—which is thus well positioned to assess relative merit of each individual thread in the system. As POSIX threads provide multi-priority scheduling policies other than run-to-completion, adapting a JVM to POSIX threads requires that the JVM be thread-safe.

To measure CPU accrual, we can rely upon round-robin scheduling (SCHED_RR for POSIX threads) or time-share scheduling (often provided as SCHED_OTHER for POSIX threads); both these policies track CPU accrual and re-schedule events upon accrual reaching some limits.

For our *fair access* policy, we have the choice between using SCHED_RR, limiting all untrusted code to a single priority level, or using time-sharing, and trusting the system for floating the priority values according to general system workload.

Conversely, the *pay for what you use* policy is best supported by a multi-priority schema. We note that a multi-priority schema does not directly imply threads with multiple priority; for example, one could structure an event-driven application to apply weight fair queueing (WFQ [40]) to input events being processed by threads at the same priority.

In the most general case where threads with multiple priorities exist, we use SCHED_RR, and control both the priority and quantum dimensions in SCHED_RR. We define *thread speed* as the number of quanta expended per unit of time. In order to reason about relative progress among threads, we normalize all threads to use the same quantum. The *pay for what you use* policy results in the requirement to constrain individual threads to the *speed limit* that they have negotiated with the system. We believe that SCHED_RR augmented with speed control gives us superior control than the time-share policy, particularly when there are threads entitled to significantly different speed limits.

The POSIX thread standard does not define interfaces to either negotiate or measure thread speed; the standard only gives us a way to affect thread speed by setting a new thread priority. Thus, we establish our own extension to the POSIX thread library; through this extension, the modified JVM maintains an up-to-date <maximum speed, actual speed> tuple for each thread that needs to be controlled. In turn, Java runtime API extensions allow the programmer to negotiate maximum speed, or simply query the current value of the tuple. The way and the rate at which actual speed is sampled depend upon thread implementation; for native threads, it also depends upon the kernel interface that the local OS offers. The measure of actual speed must also take into account cases of extra cycles being legitimately acquired when the system would otherwise become idle, or when mutex priority inheritance requires a thread to run.

The extent of speed control upon threads is also highly platform dependent. While for some platforms we can extend the JVM to perform proactive CPU control—i.e., thread cannot violate speed limit by design [22]—in other platforms only feedback-based CPU control is viable—i.e., a thread is inflicted a penalty time at depressed priority after being charged with one or more speed limit violations. In all cases, speed limit violations are meant to be a crude security device, and not a scheduling feature; the non-malicious developer is much better off by structuring the code and negotiating resources in ways that do not trigger speed control at all (i.e., by voluntarily yielding the processor at appropriate times).

5.2 Memory control

For memory control, we wish to allocate and debit memory blocks to principals, and prevent any one principal from taking more memory than the quota associated with it allows; should a principal hit its quota limit, it will be thrown an OutOfMemory exception. Additionally, we wish to prevent a principal from executing second order denial of service attacks through the garbage collector as described in section 2.

To this end, we have modified the JVM's heap manager to allow the developer to instantiate a "private", independently garbage-collected heap that is bound to a name-space. This binding fits elegantly within the Java model, because classes loaded into one name-space are not visible in other name-spaces. Therefore, there is no need for objects in different name-spaces to be in the same heap [19].

There is an important exception to this rule. Java's system classes (e.g. java.lang.*), loaded by the "null" class-loader, are shared; thus, static variables in system classes are visible to every name-space on the system [34]. Thus, we put system classes into a "system heap", visible to all name-spaces[3]. As a first level approximation, classes in the system heap are trusted and loaded off the local file system or any other trusted path; classes loaded in the private heap are untrusted and their memory resources are strictly bounded.

In our model, an invocation of the *class-loader* class constructor[4] leads not only to the creation of a new name-space but also instantiates a new heap, which we refer to as the *private* heap. We refer to the heap used when the JVM is started up as the *system* heap (the class-loader associated with this heap is sometimes referred to as the null class-loader).

3. Note we could load system classes separately into each name-space. In general, however, this practice is of limited utility in that it prevents sharing data (through static variables) between name-spaces.

4. To be more precise, since the java.lang.ClassLoader class is abstract, the instantiated class must be an extension to the ClassLoader class.

Figure 1 graphically depicts the components of the memory system. In the diagram, each shaded rectangle represents a unique heap; ovals represent name-spaces. The large oval labeled L0 represents system classes that are visible to every heap. The diagram shows three examples of object instantiations. Object α is an instantiation of a private class which is allocated out of heap H3. Object β is an instantiation of a system class; it is allocated out of the private heap labeled H1. Object δ depicts an object instantiated out of the system heap (H0). Only system classes may do this.

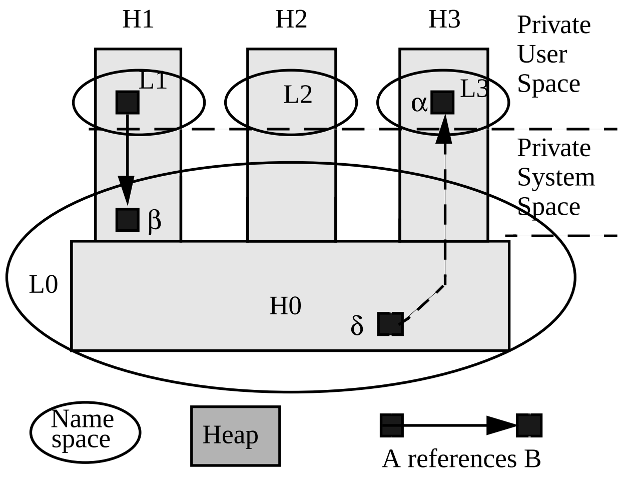

Figure 1: In this example, dashed lines represent illegal crossreferences; they must be detected and prevented. Solid lines represent legal references.

We now proceed to examine the system in greater detail. Whenever the heap manager needs to allocate memory, it determines **which** heap to allocate memory from according to the following rules:

In Java, classes are loaded with the class-loader *loadClass()* method. A *loadClass()* invocation is either made explicitly from the application, or implicitly as a consequence of executing, for instance, the *new* operation on an unknown class within the new name-space. A class is loaded in the same heap as its class-loader. For example, if code executing within a class loaded in the private heap called *new* on an unloaded non-system class, we would load the class into the private heap.

As would be expected, instantiations of classes loaded in the private heap are also allocated from the private heap. However, instances of *system* classes are handled differently. When a system class, such as java.util.Vector, is instantiated, we desire the private heap to be used. Otherwise, a malicious entity could deplete the system heap by allocating all of the memory from it. Thus, we bind a heap to the currently executing thread, which we refer to as the *current* heap. The current heap is the heap that the thread was instantiated in (aside from two important exceptions described below). Thus, if the thread was created in private heap H1, the current heap would be H1, and java.util.Vector would be instantiated within H1.

There are two special cases where the current heap is different than the one bound to the thread:

- First, system classes are always loaded and initialized into the system heap regardless of the current heap value.
- Second, the current heap switches from the system heap to the private heap when a system class creates a new class loader (this is how new private heaps are created).

During execution of a Java program, object references (pointers to memory in the heap) will exist either on the stack or in the heap. It is crucial that we prevent references from one heap—be it private or system heap—to another (a *cross-reference*). If the referred object was collected by the garbage collector, this event would not be visible to the reference in the referring heap. In that case, when a reference was made after the collection, the results would be unpredictable.

A cross-reference could be established via a *bridge class* [34]. A bridge class is a class accessible from more then one name-space. Any system class could potentially play the role of bridge classes. In Figure 2, a reference is stored in a system class shared by all name-spaces, and a reference to an object in name-space A is copied into name-space B.

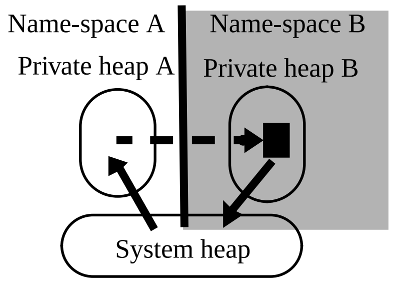

Figure 2: Example of cross-reference among heaps. If the object in B was collected, heap A and the system heap would reference garbage.

When a new name-space is created, the class-loader constructor method returns a reference to itself. This could be a cross-reference to the new heap, and the creator could maliciously use this reference to obtain other references which would not be visible to the new heap's garbage collector. In our model, we assume that the instantiated name-spaces *trust* their creator. This is analogous to saying downloaded code trusts the machine it runs on.

We detect cross-references by verifying that for every assignment, the heaps of both the destination and source in the assignment are identical. These verification steps may be performed either statically (when a class is loaded) or at runtime (while code is being interpreted). In principle, static verification would be preferable because it would not incur any runtime performance overhead. But because of Java's inheritance language features, it is impossible to predict whether an assignment generates a cross-reference or not, given that any object can be cast to the Object class (which in turn could be a bridge class). A static verifier would have to conservatively assume that such code generated a cross-reference even if at runtime it did not. The developer would be greatly restricted in how he wrote his code to avoid such ambiguities.

The JVM specification requires that heap storage for objects be reclaimed by an automatic storage management system (e.g. a garbage collector); objects are never explicitly deallocated. An object is considered garbage if it is not reachable by any thread of execution, statically or via any path of object references.

To garbage collect the system heap, a portion of the stacks of threads which originate from private heaps must be scanned for references to system objects. This is because these threads may have references to system objects on their stacks which were obtained from static variables or methods in system classes.

System classes are never unloaded, and thus are not garbage collected. This allows private heaps to store references to static system classes in their stack or heap. Relative to the developer, the environment appears to be normal, unmodified Java.

When the last non daemon thread running within a private heap terminates (voluntarily or due to an exception), the JVM reclaims all the associated objects and classes and the heap is freed back into the system pool. Note that the code must also check the stacks of system threads to ensure no bridge classes remain.

5.3 Conformance to the Java API

Several cross-references are a direct consequence of the Java API specification—for these, it does not really matter which Java API implementation we decide to start from.

A noteworthy example is related to the ThreadGroup class. Thread and ThreadGroup classes are defined such that a conformant implementation must have a list of all the Threads in a ThreadGroup and a link to its parent ThreadGroup (by following the parent link, it must be possible for some authorized code to reach a root ThreadGroup whose parent is NULL). The ThreadGroup hierarchy spans the whole JVM and makes it hard to partition ThreadGroups across our name-spaces without incurring cross-references. Our solution is to establish a new ThreadGroup whose parent is NULL any time we create a name-space with its private heap. This solution implies a departure from the Java API, in that it requires that our own code be able to create a ThreadGroup whose parent is NULL; our solution is otherwise seamless to code executing within the name-space.

We change the implementation of java.lang.Runtime methods to tie in with information or actions whose scope is the name-space or its private heap rather than the whole JVM (e.g., java.lang.Runtime.totalMemory must now reflect the memory extent of private heaps); in this, we do not see any departure from the Java semantics.

5.4 Resource Control APIs

In the previous sections, we have established new parameters such as current and maximum thread speed, current and maximum heap size for a given principal. We now look into the problem of exposing control of these parameters to code acting on behalf of a principal. We allow the untrusted code to affect the current values of resources while the trusted code retains exclusive control of the maximum values.

Our ambition to effectively control many resources in the near future—and not only CPU and memory—prompt us to invest in a resource framework, within which multiple resources can be named in a platform independent format, and can be listed in budgets of heterogeneous resources on a per-principal basis. Our resource naming schema is reminiscent of the one used by the SNMP MIB standard [27], and actually uses MIB names whenever there is a match between the resources that we deal with and known MIBs.

The values associated with a particular resource may be inherently dependent on the architecture of the host system; for example, values associated with the CPU resource typically depend upon CPU speed. In these cases, we allow the code to specify its reference platform, and we give the API implementation the task of performing the approximate adjustments that may be necessary.

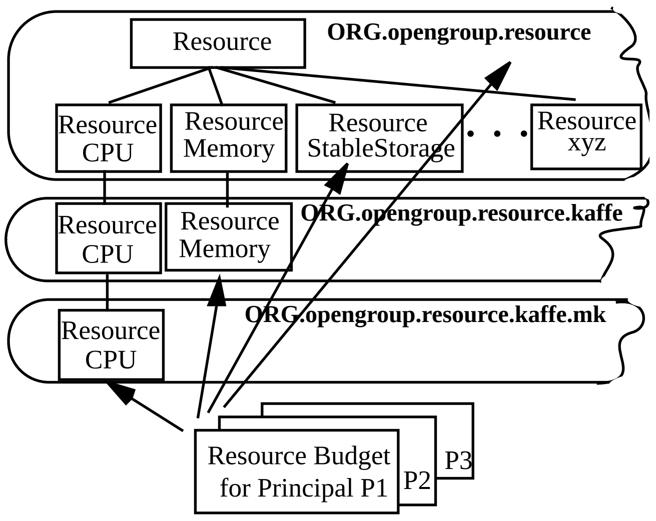

Figure 3: A resource framework for controlling extensible sets of heterogeneous resources; sets are defined on a per principal basis.

Resources are represented through Java classes that inherit from a root resource class. By applying single inheritance multiple times, we start with an unspecified, abstract resource root class and we eventually reach the desired level of abstraction for a given resource, as shown in Figure 3. The API verbs that apply to a resource are introduced and refined throughout the whole inheritance chain.

We bootstrap the resource framework from a static policy file that provides default initial/maximum values; per principal initial/maximum values are also allowed.

6 Implementation

We have realized a Java platform within which CPU and memory resources are partitioned. We have developed our JVM extensions within Kaffe 0.9.2 [38], a clean-room and freely available JVM that implements Java 1.1.

6.1 CPU control

We have made the connection between Java threads and POSIX threads. A complication of the JVM specification is that knowledge of each thread's stack pointer is required, to detect stack overflows and to perform garbage collection on the thread's stack. This information must be derived somehow even if the JVM itself is not performing context switches.

Our mapping to POSIX threads in Kaffe maintains a shadow register of the stack pointer. When a new thread is created, the register is initialized to take the address of an automatic variable stored in the first procedure that the thread executes. Special prologue and epilogues were written (for both JIT and interpreting mode) that decrease and increase the register by the size of an activation record. The prologue routine also checks for stack overflows.

For configurations required to apply a *fair access* policy, we set the threads' scheduling policy to round-robin (SCHED_RR), we assign a common priority value, and we use the default quantum.

To demonstrate the enforcement of the *pay for what you use* policy, we have realized a feedback-based control system. We have operated with native threads and with the MK kernel [36], a real-time derivative of Mach [1]; in the kernel, we have established a new system call—task_threads_get_progress—to periodically retrieve the actual speed (e.g., quanta per second) of the active threads in a given task. Kernel extensions to realize proactive control in the form of CPU reservations are underway.

A JVM internal thread running at the highest priority periodically samples actual speed, and matches it to maximum speed on a per thread basis. If actual speed is greater than maximum speed, the thread priority is temporarily depressed (i.e., the thread will still run if the machine were to become idle).

To determine the time to be spent at depressed priority, we *smooth* our speed samples by looking at the total amount of time a thread has taken since it started. In fact, it would be very hard to achieve a stable system if our observations and consequent actions were limited to an individual observation period of the speed of threads.

Figure 4 shows a stable system within which several competing threads meet their goals in CPU accrual. These data points were obtained with competing threads that are all exclusively CPU bound and never spontaneously yield the processor.

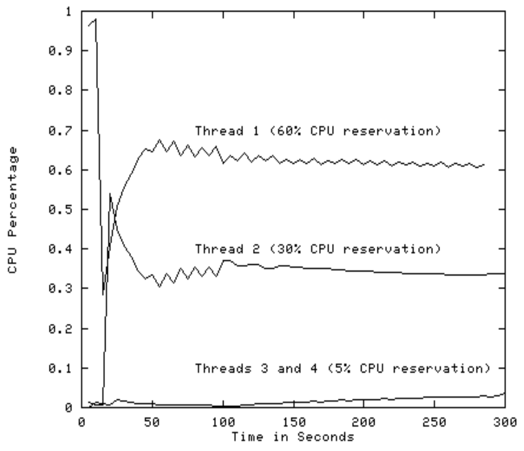

Figure 4: Constraining four threads to their individual CPU quota. CPU quota vary from thread to thread.

6.2 Memory control

Our heap management system guarantees bounded memory pools to the code. In our implementation, a new heap is created whenever a class-loader class is instantiated. Classes loaded into the new name-space, and objects instantiated from code loaded into that name-space, will draw memory from the new heap. Our APIs (Section 5.4) sets the size of the heap. Note that creating a new heap is optional; our interface allows the developer to create a new name-space which shares the system heap rather than creating a new heap.

Only a system class may instantiate a new heap. If a private class instantiates a new name-space, a new heap is not created. This is because our implementation distinguishes objects created in the system heap from objects created in the private heap, but not private objects from one another. Our crossreference checks cannot detect malicious cross-references between private heaps. We did not feel that the usefulness of hierarchical heaps was worth the extra engineering effort and complexity required to make them work.

Our implementation of heap partitioning binds heaps to objects by adding a field in the memory block that the object belongs to. This makes checks for cross-references fast; the fields of the source and destination need only be compared for equality. Table 2 reports the results of micro-benchmarks wherein a particular byte-code is executed in a tight loop and as such constitutes a worst case scenario. Alternatively, we could have derived a reference to the heap from the object's data structure; however, this would incur additional space overhead.

Micro-benchmark	JIT	interpreted
putstatic	43%	85%
putfield	53%	88%
aastore	53%	85%

Table 1: Performance implications of runtime cross reference checks, as a percentage of the original speed.

Rather than restricting the expressive ability of Java, we opted to do runtime checks for cross-references. We augmented the JVM[5] such that whenever a bytecode writes to a reference (i.e. `putstatic`, `aastore` or `putfield`) or loads a reference to its stack (i.e. `getstatic, aaload, getfield, areturn`) an exception is thrown if an attempt to cross-reference is being made.

For a "fair access" type policy, the private heaps would be created with equal sizes. Otherwise the resource constraints would be propagated through the resource control APIs.

6.3 Core Java API

As our work has also prompted changes to the implementation of the Java API classes (see 5.3), we have adopted the Kore technology [33], a clean-room and freely available version of Sun's core API packages.

The ThreadGroup case was discussed in Section 5.4. Other system classes, such as java.util.Hashtable, were implemented in such a way that cross-references are incurred. In all these circumstances we have modified the Kore classes.

7 Example

In Section 2, we have introduced a router for active networks as a motivating scenario for service guarantees and non-interference properties among mutually suspicious code in Java. In this section, we describe our experience in building a prototype of such a router. To this extent, we have used ANTS [37], a Pure-Java framework for composing and evaluating Active Network capsules.

We have analyzed the behavior of ANTS version 1.1 under classes of denial-of-service attacks on CPU and memory such as the ones in Section 2. We have found that attacks may be made on it which are fatal to the whole JVM. For instance, a malicious or buggy capsule that carries a forever loop prevents any other capsule from making forward progress (i.e., a capsule being evaluated holds a global monitor that never gets released, regardless of preemptive versus non-preemptive implementation of Java threads).

5. We have also modified the just-in-time compiler.

In the off-the-shelf ANTS, we have also found attacks that may be countered or be fatal, depending upon a particular run. For instance, a thread evaluating a malicious capsule can be thrown an OutOfMemory exception, the monitor be released, and any other capsule make forward progress afterwards. The same thread, however, can be caught in a memory shortage while extending the stack, and this results in a fatal failure.

Furthermore, ANTS' full or partial defenses to resource consumption attacks rests on working assumptions—no persistent state and LRU caches only, single threaded capsule evaluation—that we want to remove to further explore QoS capabilities and *pay for what you use* policies built into active nodes. We have indeed extended ANTS to support persistent state and multi-threaded capsule evaluation; the resulting system is vulnerable to all the attacks described in Section 2. This is also our starting point for applying and evaluating our CPU and memory partitioning techniques.

In our modified ANTS, we evaluate capsules in the multiple, non-interfering name-spaces that we have built. We had the choice of defining what is in or out of these name-spaces; that is, we had to choose among several alternatives for fitting the existing ANTS code into the new domains depicted in table 2. For this proof-of-concept demonstration, we have chosen to make our name-spaces accessible through different port numbers (whereas unmodified ANTS handles a single port number). Thus, the port number decides which name-space a capsule ought to be evaluated in. It is appropriate to think of the port number as a preliminary, very crude notion of principal identifier (we will introduce principals associated with end-to-end flows in the near future).

Shared	Exclusive
runtime	caches
bootstrap protocol	protocol
demand loading protocol	transient data
routing tables	channel, thread stack, class-loader

Table 2: ANTS entities classified w.r.t. their scope.

Practically, we have made one ANTS channel serve one port number, and we have associated a name-space to each ANTS channel. Table 2 lists the ANTS' entities that are shared across name-spaces, and the entities that are instead sandboxed into a name-space.

In our modified ANTS environment, the capsules carrying attacks like those in Section 2 are systematically aborted once they exceed their resource limit, and the other capsules are unaffected.

8 Related work

Development efforts are underway in both industry and academic circles to improve service reliability in Java.

PERC, a product developed by Newmonics Inc., is a re-implementation of the JVM designed to give hard real-time guarantees [26]. The product allows the developer to supply bounded execution times (deadlines) and memory utilization for code segments. PERC does this by making syntactic extensions to the Java language. Code blocks that have deadlines are bracketed by two new keywords. The "timed" statement allows the developer to establish a time for a given block of code. The "atomic" statement allows the developer to define a block of code that will not be interrupted or preempted. Newmonics provides a special compiler that translates these statements into a new attribute in the classfile format (thus they modify the language syntax and classfile representation, but not the bytecode).

PERC has a configuration phase that analyzes byte-code to determine resource requirements for downloaded code. The analysis does not work for arbitrary code, but is restricted to code segments which adhere to strict coding conventions (e.g. code blocks with deadlines shall not contain unbounded loops).

In this report, we have only addressed our effort to partition resources and have not described how our system can meet real-time deadlines. Thus, we contrast our work to PERC and other real-time systems [23] only along the dimension of resource partitioning.

The level of granularity to which our system binds resources is much coarser than in PERC—we allocate CPU and memory to principals rather than small code segments. Our system is more effective at providing non-interference properties in uncooperative environments, in that we have independently garbage collected heaps. Conversely, PERC uses copy collection and requires a system manager to reserve CPU time for garbage collection worst case scenarios. Additionally, our system does not make any changes to the Java language (i.e., our APIs are written in Java and may be compiled using standard tools).

The J-Kernel [14] libraries realize protection domains and capability-based communication across protection domains. The J-Kernel libraries could enhance our Java platform with rich semantics to share objects among name-spaces.

Inferno, designed for telephony systems in the embedded systems space, has a richer interface to resources than Java does [7]. For example, developers may explicitly set a scheduling policy for a thread, and the garbage collector may be configured to run in constant time. However, Inferno's trust model is different from Java in that applications are considered to be trustworthy if they can be authenticated. There is no name-space concept in Inferno to limit the set of objects visible to an application, and there is no access control mechanism. We desire stronger security guarantees in our target system.

The resource partitioning concepts presented in this document are similar to the ones that inspired CORDS Paths[31], Scout's Paths[24], and resource channels in Nemesis[25]. In these systems the code is native, trusted, and statically compiled. Our resource partitions in Java are meant to map onto such Paths [32] or resource channels to exploit bottom-up control of resources throughout the various layers (from native and trusted system layers to layers for mobile and untrusted code).

9 Conclusion

Through extensions to the Java API and a JVM implementation, we have realized a Java platform wherein untrusted and mutually suspicious code can execute without interference. This platform can support both a *fair access* policy and a *pay for what you use* policy for resource consumption. Furthermore, this platform is robust to what we classified as first level and second level denial-of-service attacks on CPU and memory consumption.

We have demonstrated the effectiveness of our platform in the context of Active Networks. Using our platform and a modified version of ANTS, we were able to build a network node which is resilient to classes of denial-of-service attacks mounted by untrusted, transient capsules; the same attacks would otherwise be fatal to either the off-the-shelf ANTS, or the JVM.

No changes to the Java language nor its intermediate byte-code representation were necessary. In the specific case of the ANTS application, our changes did not impact wire format and thus interoperability with other installations.

We plan to evolve our work towards providing real-time behaviors, pervasive support for multiple principals, and control of resource types other than CPU and memory.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. "Mach: A New Kernel Foundation for UNIX Development". Proceedings of the Summer 1986 USENIX Conference. July, 1986. pages. 93-112.

[2] O Agesen, D Delefs. "Finding References in Java Stacks". OOPSLA'97 Workshop on Garbage Collection and Memory Management.

[3] A. V. Aho, R. Sethi, J. D. Ullman. Compilers -- Principles, Techniques and Tools. Addison-Wesley,1986.

[4] P Bernadat, L Feeney, D. Lambright, F Travostino. "Making Service Guarantees in a Java-Based Uncooperative Environment". Work-In-Progress (WIP) Session in the 4th IEEE Real-

Time Technology and Applications Symposium, Denver, CO, 1998.

[5] H Boehm, M Weiser. “Garbage Collection in an Uncooperative Environment”. Software - Practive and Experience. Vol 18(9), 807-820 (September 1988)

[6] R. Clark, F. Travostino, and D.Wells, “Management of Resources in CONVERSANT” Position paper at the DARPA ActiveNet Workshop, March ‘98; http://www.dyncorp-is.com/darpa/meetings/anets98mar/position.html

[7] P. David. “Inferno Security.” Proceedings of IEEE COMPCON 1997} pages 97-102. February 1997.

[8] D. Dean, E. Felten, D. Wallach.“Java Security: From HotJava to Netscape and Beyond.” IEEE Symposium on Security and Privacy}. May 1996.

[9] D. Feldmeier, A. McAuley & J. Smith, D. Bakin, W. Marcus, and T Raleigh. “Protocol Boosters” IEEE JSAC Special Issue on “Protocol Architectures for the 21st Century.”

[10] D. Flanagan. “Java in a Nutshell.” O’Reilly & Associates, Inc.

[11] Gong L, Balfanz, D. “Experience with Secure Multi-Processing in Java”. Technical Report 560-97, Department of Computer Science, Princeton University, September 1997

[12] R Grimm, B Bershad. “Providing Policy-Neutral and Transparent Access Control in Extensible Systems” University of Washington Technical Report UW-CSE-98-02-02.

[13] J. Hartman, U. Manber, L. Peterson, T. Proebsting. “Liquid Software: A New Paradigm for Networked Systems.” Technical Report 96-11. Dept. of Computer Science, The University of Arizona.

[14] C. Hawblitzel, C. Chang, G. Czaikowski, D. Hu, and T. von Eicken. “Implementing Multiple Protection Domains in Java,” Department of Computer Science, Cornell University, Technical Report TR97-1660, December 1997.

[15] Hicks, M. Kakkar, P. Moore, J. Gunterm C, Nettles, S. “PLAN: A Programming Language for Active Networks”. http://www.cis.upenn.edu/~switchware/PLAN/

[16] D. Lambright. “Automated Verification of Mobile Code”. Technical Report TR97-14. The University of Arizona, August 1997.

[17] M. Ladue. “A Collection of Increasingly Hostile Applets”. URL: http://www.math.gatech.edu/~mladue/HostileApplets.html

[18] S Leffler, M McKusick, M Karels, J Quaterman. “The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, 1989.

[19] T. Lindholdm, F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[20] P. Madany. JavaOS(tm): “A Standalone Java Environment”. http://www.javasoft.com/docs/white/index.html

[21] G. McGraw, Edward W. Felten. “Java Security: Hostile Applets, Holes and Antidotes.” John Wiley and Sons, New York, 1996.

[22] C. W. Mercer, S. Savage, H. Tokuda. “Processor Capacity Reserves: Operating System Support for Multimedia Applications”. in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[23] A Miyoshi, T. Kitayama. “Implementation and Evaluation of Real-Time Java Threads”. *Proceedings of the Real-Time Systems Symposium*. December 2-5, 1997. San Francisco CA.

[24] D. Mosberger, and L.Peterson, “Making paths explicit in the Scout operating system,” Proceedings of the Second Symposium on Operating Systems Design and Implementation, pages 153-168, Oct. 1996.

[25] Nemesis, http://www.cl.cam.ac.uk/Research/SRG/pegasus/nemesis.html.

[26] K Nilsen. “Java for Real-Time”. *Real-Time Systems Journal*, 11(2) 1996.

[27] M Rose, K. McCloghrie. “Management Information Base for Network Management of TCP/IP-based internets: MIB-II”. RFC 1213.

[28] B Schwartz, W Zhou, Alden W. Jackson, W.Timothy Strayer, Dennis Rockwell, Craig Partridge. Smart Packets for Active Networks (January, 1998). http://www.net-tech.bbn.com/smtpkts/smtpkts-index.html

[29] Sun Microsystems. “RMI - Remote Method Invocation”. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html

[30] D. L. Tennenhouse, J. M. Smith, W. Sincoskie, D. J. Wetherall, G. J. Minde. “A Survey of Active Network Research” IEEE Communications Magazine. Vol. 35, No. 1, pp.80-86. January 1997.

[31] F.Travostino, E.Menze, and F.Reynolds, “Paths: Programming with System Resources in Support of Real-Time Distributed Applications,” Proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Laguna Beach, Ca, February 1996, pp.36-45.

[32] F Travostino. “Conversant: An Environment for Real-time, Secure Active Networks” http://www.opengroup.org/RI/PubProjPgs/CONVERSANT.htm

[33] P. Tullmann. “Kaffe and the Kore Class Libraries”. http://www.cs.utah.edu/~tullmann/kore/

[34] V Saraswat. “Java is Not Type-Safe”. URL: http://www.research.att.com/~vj/main.html. 1997.

[35] R. Wahbe, S. Lucco, T.E. Anderson, S. Graham. “Efficient Software-Based Fault Isolation.” in *Proceedings of the Fourteenth Symposium on Operating Systems Principles*. 1993.

[36] D. Wells, “A Trusted, Scalable, Real-Time Operating System Environment,” 1994 Dual-Use Technologies and Applications Conference Proceedings, pp.II-262/270, Utica, NY, May, 1994.

[37] D Wetheral, G Guttag, D Tennenhouse, “ANTS: A toolkit for building and dynamically deploying network protocols”. *IEEE OPENARCH 98*, San Francisco, CA, Apr. 1998.

[38] T Wilkinson and Associates. “Kaffe: a free virtual machine to run Java code”. URL: http://www.kaffe.org, 1997.

[39] “DynaCache Overview”. http://www.infolibria.com/products/f-over.htm

[40] Zhang, L. “Virtual Clock: a New Traffic Control Algorithm for Packet Switching Networks”. ACM Transactions on Computer Systems 9(2):101-124. May 1991.