Termination in Language-based Systems

Algis Rudys
arudys@cs.rice.edu

John Clements
clements@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

Language runtime systems are increasingly being enbedded in systems to support runtime extensibility via mobile code. Such systems raise a number of concerns when the code running in such systems is potentially buggy or untrusted. While sophisticated access controls have been designed for mobile code and are shipping as part of commercial systems such as Java, there is no support for terminating mobile code short of terminating the entire language runtime. This paper presents a concept called "soft termination" which can be applied to virtually any mobile code system. Soft termination allows mobile code threads to be safely terminated while preserving the stability of the language runtime. In addition, function bodies can be permanently disabled, thwarting attacks predicated on system threads eventually calling untrusted functions. We present a formal design for soft termination and an implementation of it for Java, built using Java bytecode rewriting, and demonstrating reasonable performance (5-40% slowdowns on benchmarks).

1 Introduction

In recent years, many systems have turned to language runtime systems for enforcement of security. Language-based enforcement of security was popularized by Java [26, 33, 41] and the Java Virtual Machine (JVM), which were adopted by Netscape for its Navigator 2.0 browser in 1995. Java promised an environment where untrusted and malicious code (hereafter called a "codelet"[1]) could run safely inside the Web browser, enhancing the user's Web experience without jeopardizing the user's security. Rather than using kernel-based protection, the JVM would run inside the same address space as the browser, providing protection and separation as a side-effect of enforcing its type system. A combination of static and dynamic type checking would serve to prevent a malicious codelet from forging a reference to an arbitrary memory location and subverting the system. In addition to its applications within Web browsers, codelets have also been touted for OS kernel extensions, active networking, extensible databases, agent-based negotiating systems, and presumably other problem domains.

The *promise* of Java may be attractive, but a large number of security flaws have been discovered in Java since its release [12, 13, 36, 42]. Significant strides have been made at understanding the type system [1, 22, 46, 16, 17, 10, 11, 43] and protecting the Java system classes from being manipulated into violating security [51, 52, 25, 18, 19], but efforts to control resource exhaustion have lagged behind. A simple infinite loop will still freeze the latest Web browsers. The most successful systems to date either run the JVMs in separate processes or machines [34, 14, 44], surrendering any performance benefits from running the JVM together with its host application, or create a process-like abstraction inside the JVM [4, 49, 29, 5]. These process abstractions either complicate memory sharing or make it completely impossible.

This paper describes a new language runtime-based mechanism called *soft termination*. While it is not specific to Java, soft termination can be deployed on Java, and we present a Java-based implementation. Soft termination is intended to be invoked either by an administrator or by a system resource monitor which has concluded that a codelet is exceeding its allotted resources and may no longer be allowed to run. Soft termination provides semantics similar to the UNIX `ps` and `kill` commands while preserving system integrity after termination, yet requires neither process-like struc-

[1] The term "codelet" is also used in artificial intelligence, numerical processing, XML tag processing, and PDA software, all with slightly different meanings. When we say "codelet," we refer to a small program meant to be executed in conjunction with or as an internal component of a larger program.

tures nor limits on memory sharing. Our implementation of soft termination is defined as a code-to-code transformation, and is thus more easily portable across languages and implementations of the same language. Soft termination supports two kinds of program termination: safe thread termination and safe code disabling. Safe thread termination must deal with cases where the target thread is currently executing critical system code that may not necessarily be designed to respond to an asynchronous signal. Safe codelet disabling must deal with cases where future threads may invoke functions of the disabled codelet, yet the codelet should not be able to "hijack" the thread and continue execution.

In the following sections, we discuss the concept of soft termination, and present our implementation. Section 2 discusses how prior work has addressed these concerns. Section 3 formalizes and describes what we mean by termination. Section 4 describes our Java-based implementation of soft termination, and mentions a number of Java-specific issues that we encountered. We present performance measurements in section 5. Finally, section 6 describes our experience integrating soft termination with a Java-based Web server.

2 Related work

Systems such as Smalltalk [24], Pilot [40], Cedar [48], Lisp Machines [6], and Oberon [53] have taken advantage of language-based mechanisms to provide OS-like services. Perhaps as early as the Burroughs B5000 series computers [38], language based mechanisms were used for security purposes. More recently, language-based enforcement of security was popularized by Java, originally deployed by Netscape for its Navigator 2.0 browser in 1995 to run untrusted applets.

However, these systems provide little or no support for resource management on the programs they run. A number of projects have been developed to address this. A recent Scheme system called MrEd [21] supports thread termination and management of resources like open files but has no way of disabling code from running in future threads. Some systems, such as PLAN [30], restrict the language to guarantee that programs will terminate. In general, many language systems support interactive debugging, which includes the ability to interrupt a running program and inspect its state. This can be performed with operating-system services or by generating inline code to respond to an external debugger.

Much of the recent research in this area has been focused on the Java programming language. PERC [37], for instance, is a Java-derived language that extends Java to support asynchronous exceptions. A programmer may specify blocks with provable limits on their runtime and asynchronous exceptions are deferred until the block completes execution.

J-Kernel [29, 45] is a system for managing multiple Java codelets running in the same JVM. It is written entirely in Java, giving it the advantage of working with multiple JVMs with minimal adjustment. It is implemented as a transformation on Java bytecode as the bytecodes are loaded by the system. While these papers mention termination, no details are presented for how termination might be supported.

JRes [9] is a resource management system for Java. Bytecode rewriting is used to instrument memory allocation and object finalization in order to maintain a detailed accounting of memory usage. Again, termination is mentioned, but no significant details are provided.

KaffeOS [2, 3] provides an explicit process-like abstraction for Java codelets. KaffeOS is implemented as a heavily customized JVM with significant changes to the underlying language runtime system and system libraries. Code termination is supported in the same manner as a traditional operating system: user codelets are strongly separated from the kernel by running in separate heaps. Memory references across heaps are heavily restricted. Bernadat *et. al.* [5] and van Doorn [50] describe similar systems that customize a JVM in order to support better memory account and security. van Doorn takes advantage of lightweight mechanisms provided by an underlying micro-kernel. KaffeOS provides a style of termination we call hard termination (see section 3.2). It's not clear how termination is supported in the other systems.

3 System Design

A large space of possible designs exist for supporting termination in language runtimes. We first consider the naïve solutions and explain the hard problems raised by their failings. We then discuss how operating systems perform termination and finally, describe our own system.

3.1 Naïve termination

One naïve solution to termination would be to identify undesired threads and simply remove them from the thread scheduler. This technique is used by Java's deprecated `Thread.destroy()` operation [47]. Unfortunately, there are numerous reasons this cannot work in practice.

Critical sections A thread may be in a critical section of system code, holding a lock, and updating a system data structure. Descheduling the thread would either leave the system in a deadlock situation (if the lock is not released) or leave the system data structures in an undefined state, potentially destabilizing the entire system (if the lock is forcibly released).

Boundary-crossing threads In an object-oriented system, a program wishing to inspect or manipulate an object invokes methods on that object. When memory sharing is unrestricted between the system and its codelets or among the codelets, these method invocations could allow a malicious codelet to "hijack" the thread from its caller and perhaps never release it. This is especially a concern if the thread in question is performing system functions, such as finalizing dead objects prior to garbage collection.

Blocking calls The language runtime system has support for making native OS system calls. A thread should not be descheduled while it is blocking on a system call such as an I/O call.

Another naïve solution is to force an asynchronous exception, as done by Java's deprecated `Thread.stop()` operation [47]. While this exception will wait for blocking calls to complete, it may still occur inside a critical section of system code. In addition, blocking calls could potentially never return, resulting in a nonterminable thread. Finally, a workaround is needed to prevent user-level code from catching the exception.

3.2 Hard termination

Operating systems like Unix support termination by carefully separating the kernel from the user program. When a process is executing in user space, the kernel is free to immediately deschedule all user threads and reclaim the resources in use. If the process is executing in the kernel, termination is normally delayed; a flag is checked when the kernel is about to return control to the user process. In cases where the kernel may perform an operation that could potentially block forever (*e.g.*, reading from the network), the kernel may implement additional logic to interrupt the system call. Other system calls, those that complete in a guaranteed finite time, need not check whether their user process has been terminated, as the kernel will handle the termination signal on the way out. We call such a mechanism a *hard termination system* because once termination is signaled, user-level code may be terminated immediately with no harmful side-effects.

3.3 Soft termination

Unlike a traditional operating system, the boundary between user and system code in a language runtime is harder to define. While all code within the system is generally tagged with its protection domain, there is nothing analogous to a system call boundary where termination signals can be enforced. Furthermore, because a thread could easily cross from user to system code and back many times, there may never be a correct time at which it becomes safe to terminate a thread.

This section introduces a design we call *soft termination* and describes the properties we would find desirable. We present an implementation of soft termination based on code rewriting for a simplified language and prove that all programs will terminate when signaled to do so.

3.3.1 Key ideas

Soft termination is based on the idea that a codelet may be instrumented to check for a termination condition during the normal course of its operation. Our goal is to perform these checks as infrequently as necessary — only enough to ensure that a codelet may not execute an infinite loop. Furthermore, as with the Unix kernel, we would like the termination of a codelet to not disturb any system code it may be using at the time it is terminated, thus preserving system correctness.

The soft termination checks are analogous to *safe points* – used by garbage collection systems and some debugging environments. In the same way as we would like to check for a termination flag, these systems build checks for low-memory conditions or debugger interrupts directly into the application.

$$
\begin{array}{rcl}
P & = & \Gamma\, M \\
\Gamma & = & D \ldots D \\
D & = & (\textbf{define}\ \ (f\ x\ \ldots)\ M) \\
M & = & (f\ M\ \ldots) \mid (\textbf{if}\ \ M\ M\ M) \mid (\textbf{let}\ \ (x\ M)\ \ldots\ M) \mid (\textbf{try}\ \ M\ M) \mid (\textbf{throw}) \mid V \\
V & = & x \mid \mathit{true} \mid \mathit{false} \mid c \\
E & = & [] \mid (f\ V\ \ldots\ E\ M\ \ldots) \mid (\textbf{if}\ \ E\ M\ M) \mid (\textbf{try}\ \ E\ M) \mid (\textbf{let}\ \ (x\ V)\ \ldots\ (x\ E)\ (x\ M)\ \ldots\ M) \\
\text{final states} & = & V \mid \textbf{error}
\end{array}
$$

Figure 1: Simple language used for our analysis.

$$
\begin{array}{rcll}
E[(f\ V\ \ldots)] & \mapsto & E[V_0] & \text{if } f \in \text{ built-in primitives} \\
E[(f\ V \ldots)] & \mapsto & \textbf{error} & \text{if } f \notin \text{ built-in primitives,} \\
 & & & (f\ V\ \ldots) \text{ is undefined, or} \\
 & & & (\textbf{define}\ \ (f\ x\ \ldots)\ M) \notin \Gamma \\
E[(f\ V\ \ldots)] & \mapsto & E[[V\ \ldots/x\ \ldots]M] & \text{if } (\textbf{define}\ \ (f\ x\ \ldots)\ M) \in \Gamma \\
E[(\textbf{if}\ \ \mathit{true}\ M_1\ M_2)] & \mapsto & E[M_1] & \\
E[(\textbf{if}\ \ \mathit{false}\ M_1\ M_2)] & \mapsto & E[M_2] & \\
E[(\textbf{if}\ \ V\ M_1\ M_2)] & \mapsto & \textbf{error} & \text{if } V \neq \mathit{true},\mathit{false} \\
E[(\textbf{try}\ \ V\ M)] & \mapsto & E[V] & \\
E[(\textbf{let}\ \ (x\ V)\ \ldots\ M)] & \mapsto & E[[V\ \ldots\ /\ x\ \ldots]\ M] & \\
E[(f\ V\ \ldots\ (\textbf{throw})\ M\ \ldots)] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{if}\ \ (\textbf{throw})\ M_1\ M_2)] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{try}\ \ (\textbf{throw})\ M)] & \mapsto & E[M] & \\
E[(\textbf{throw})] & \mapsto & \textbf{error} & \\
E[(\textbf{let}\ \ (x\ V)\ \ldots\ (x\ (\textbf{throw}))\ (xM)\ \ldots M)] & \mapsto & E[(\textbf{throw})] & \\
\mathit{eval}(\Gamma,\ M) & = & \left\{ \begin{array}{ll} V & \text{if } \Gamma \vdash M \mapsto^{*} V \\ \textbf{error} & \text{if } \Gamma \vdash M \mapsto^{*} \textbf{error} \end{array} \right. &
\end{array}
$$

Figure 2: An operational semantics for our language.

$$
\begin{array}{lll}
(1) & \mathcal{X}[\![\Gamma\ M]\!] & = \mathcal{X}[\![\Gamma]\!]\ \mathcal{X}[\![M]\!] \\
(2) & \mathcal{X}[\![D\ \ldots\ D]\!] & = \mathcal{X}[\![D]\!]\ \ldots \mathcal{X}[\![D]\!] \\
(3) & \mathcal{X}[\![(\textbf{define}\ \ (f\ x\ \ldots)\ M)]\!] & = (\textbf{define}\ \ (f\ x\ \ldots)\mathcal{X}[\![M]\!]) \\
(4) & \mathcal{X}[\![(f\ M\ \ldots)]\!] & = (\textbf{let}\ \ (t\ \mathcal{X}[\![M]\!])\ \ldots\ (\textbf{if}\ \ (\textbf{CheckTermination})\ (\textbf{throw})\ (f\ t\ \ldots))) \\
(5) & \mathcal{X}[\![(\textbf{if}\ \ M\ M\ M)]\!] & = (\textbf{if}\ \ \mathcal{X}[\![M]\!]\ \mathcal{X}[\![M]\!]\ \mathcal{X}[\![M]\!]) \\
(6) & \mathcal{X}[\![(\textbf{try}\ \ M\ M)]\!] & = (\textbf{try}\ \ \mathcal{X}[\![M]\!]\ \mathcal{X}[\![M]\!]) \\
(7) & \mathcal{X}[\![(\textbf{throw})]\!] & = (\textbf{throw}) \\
(8) & \mathcal{X}[\![(\textbf{let}\ \ (xM)\ \ldots M)]\!] & = (\textbf{let}\ \ (x\ \mathcal{X}[\![M]\!])\ \ldots \mathcal{X}[\![M]\!]) \\
(9) & \mathcal{X}[\![V]\!] & = V
\end{array}
$$

Figure 3: The soft termination transformation.

3.3.2 Formal design

For our analysis, we begin with a simple programming language having methods or functions, conditional expressions, and simple exceptions (see figures 1 and 2). In our language, a program is a collection of function definitions followed by an expression to be evaluated. A function body contains function/method calls as well as primitive operations, conditionals, and exceptions. A (**CheckTermination**) expression is assumed to return a boolean value, indicating whether termination for the current codelet has been externally (and asynchronously) requested. We write the semantics of our language using the same style as Felleisen and Hieb [20].

The soft termination transformation, $\mathcal{X}$, is described in figure 3. Rule 4 of this transformation inserts the check for termination before every function invocation. The other rules describe how the transformation continues recursively on expressions.

Proving that a transformed program terminates in a finite time, given that (**CheckTermination**) returns true, is a straightforward exercise. First, we state without proof that our language semantics are consistent and well-formed and that there are no collisions or unbound names in the name-space (*i.e.*, every reference to a function or variable can be bound to a single definition).

Next, we start with all possible expressions M from figure 1. After applying $\mathcal{X}$ to M, and assuming **CheckTermination** is true, we wish to show the program enters a "locked" state, where termination is guaranteed. We call this M_{lock}.

$$
\begin{array}{rcll}
M_{lock} & = & (\textbf{let}\ (t\ M_{lock})\ \ldots & (a)\\
 & & \quad (\textbf{if}\ (\textbf{CheckTermination}) & \\
 & & \quad\quad (\textbf{throw})\ (f\ t\ \ldots)\ ))\ | & \\
 & & (\textbf{if}\ (\textbf{CheckTermination}) & (b)\\
 & & \quad (\textbf{throw})\ (f\ V\ \ldots)\ )\ | & \\
 & & (\textbf{if}\ \mathit{true}\ (\textbf{throw})\ (f\ V\ \ldots)\ )\ | & (c)\\
 & & (\textbf{if}\ M_{lock}\ M_{lock}\ M_{lock})\ | & (d)\\
 & & (\textbf{let}\ (x\ M_{lock})\ \ldots\ M_{lock})\ | & (e)\\
 & & (\textbf{try}\ M_{lock}\ M_{lock})\ | & (f)\\
 & & (\textbf{throw})\ | & (g)\\
 & & V & (h)
\end{array}
$$

To prove termination, we must show that, once (**CheckTermination**) returns *true*, M_{lock} is closed under program stepping, and that the syntactic length of the program will be strictly decreasing.

Closure may be stated as follows: if $M \mapsto M'$, and $M \in M_{lock}$, then $M' \in M_{lock}$. By inspection, for all possible expressions in M_{lock}, we observe that our semantics preserves closure.

The syntactic length property may be stated as follows: starting with an initial expression $M_0 \in M_{lock}$, we wish to prove that $\forall M, M'$ where $M_0 \mapsto^* M \mapsto M : |M'| < |M|$. For cases (d) through (h), the relevant semantic rules in figure 2 clearly guarantee $|M'| < |M|$. Likewise, while (**CheckTermination**) is true, $(a) \mapsto^* (b) \mapsto (c) \mapsto (g)$, reducing the program's syntactic length on each step.

When a program is executing when (**CheckTermination**) is initially *false* and becomes *true* at an arbitrary time, the expression being evaluated may not be in M_{lock}. The case where this matters is when termination is requested just after stepping case (b). (b) would step outside M_{lock}: $(b) \mapsto (\textbf{if}\ \mathit{false}\ (\textbf{throw})\ (f\ V\ \ldots)) \mapsto (f\ V\ \ldots) \mapsto b'$. The final step to b' expands the program's syntactic length. However, it also brings the program back to M_{lock} as the body of f was subject to the $\mathcal{X}$ transformation. Thereafter, the program length will be strictly decreasing and the program will clearly terminate when $|M'| = 1$ (*i.e.*, a value or **error**).

Codelet, system, and blocking code Termination becomes trickier when blocking calls are introduced and when we distinguish between **codelet** and **system** code. Figure 4 introduces an extension to our little language where functions are labeled as to the origin of their code, and whether or not they block. The **blocking** label would normally be applied to system functions known to block, such as I/O operations. By convention, every blocking function $f_{blocking}$ is defined to have some roughly equivalent $f_{nonblocking}$ which returns a tuple containing a boolean and a value. The boolean indicates whether the call succeeded and if so, the value is the result. Polling $f_{nonblocking}$ is intended to be semantically equivalent to calling $f_{blocking}$.

The transformation in figure 5 is more complex than that of figure 3 so it bears some explanation.

Rules 3a through 3c describe the transformation on function definitions. Rules 3a and 3b say that system code and codelets have separate transformations, $\mathcal{X}_{system}$ and $\mathcal{X}_{codelet}$, respectively. Rule 3c says a non-blocking wrapper is created for every blocking function. In this case, the wrapper invokes the non-blocking call equivalent to the original. If the call succeeds, the value is returned. Otherwise, the wrapper polls to see if termination has been indicated and throws an exception if it has. Otherwise, the non-blocking call is recursively polled again.

$$
\begin{array}{rcl}
P & = & \Gamma\, M \\
\Gamma & = & D \ldots D \\
L & = & \textbf{system} \mid \textbf{codelet} \\
D & = & (\textbf{define}\ L\ (f\ x\ \ldots)\ M) \mid \\
 & & (\textbf{define}\ \textbf{blocking}\ (f_{blocking}\ x\ \ldots)\ f_{nonblocking}\ M) \\
M & = & (f\ M\ \ldots) \mid (\textbf{if}\ M\ M\ M) \mid (\textbf{let}\ (x\ M)\ \ldots\ M) \mid (\textbf{try}\ M\ M) \mid (\textbf{throw}) \mid V \\
V & = & x \mid \mathit{true} \mid \mathit{false} \mid c \\
E & = & [] \mid (f\ V\ \ldots\ E\ M\ \ldots) \mid (\textbf{if}\ E\ M\ M) \mid (\textbf{try}\ E\ M) \mid (\textbf{let}\ (x\ V)\ \ldots\ (x\ E)\ (x\ M)\ \ldots\ M) \\
\text{final states} & = & V \mid \textbf{error}
\end{array}
$$

Figure 4: An extended language for analysis distinguishing codelets from system code.

$$
\begin{array}{lllll}
(1) & \mathcal{X}[\![\Gamma\, M]\!] & = & \mathcal{X}[\![\Gamma]\!]\ \mathcal{X}_{codelet}[\![M]\!] & \\
(2) & \mathcal{X}[\![D\ \ldots\ D]\!] & = & \mathcal{X}[\![D]\!]\ \ldots \mathcal{X}[\![D]\!] & \\
(3a) & \mathcal{X}[\![(\textbf{define}\ \textbf{codelet}\ (f\ x\ \ldots)\ M)]\!] & = & (\textbf{define}\ (f\ x\ \ldots)\ \mathcal{X}_{codelet}[\![M]\!]) & \\
(3b) & \mathcal{X}[\![(\textbf{define}\ \textbf{system}\ (f\ x\ \ldots)\ M)]\!] & = & (\textbf{define}\ (f\ x\ \ldots)\ \mathcal{X}_{system}[\![M]\!]) & \\
(3c) & \mathcal{X}[\![(\textbf{define}\ \textbf{blocking}\ (f_{blocking}\ x\ \ldots) & = & (\textbf{define}\ (f_{wrapper}\ x\ \ldots) & \\
 & \quad f_{nonblocking}\ M)]\!] & & \quad (\textbf{let}\ (b,t\ (f_{nonblocking}\ x\ \ldots)) & \\
 & & & \qquad (\textbf{if}\ b\ t\ (\textbf{if}\ (\textbf{CheckTermination})\ (\textbf{throw})\ (f_{blocking}\ x\ \ldots)))\ )) & \\
 & & & (\textbf{define}\ (f_{blocking}\ x\ \ldots)\ M) & \\
(4a) & \mathcal{X}_{codelet}[\![(f\ M\ \ldots)]\!] & = & (\textbf{let}\ (t\ \mathcal{X}_{codelet}[\![M]\!])\ \ldots & \\
 & & & \quad (\textbf{if}\ (\textbf{CheckTermination})\ (\textbf{throw})\ (\mathcal{X}[\![f]\!]\ t\ \ldots))\ ) & \\
(4b) & \mathcal{X}_{system}[\![(f\ M\ \ldots)]\!] & = & (\mathcal{X}[\![f]\!]\ \mathcal{X}_{system}[\![M]\!]\ \ldots) & \\
(5a) & \mathcal{X}[\![f_{blocking}]\!] & = & f_{wrapper} & \text{if } (\textbf{define}\ \textbf{blocking}\ (f_{blocking}\ \ldots)\ f_{nonblocking}\ \ldots) \in \Gamma \\
(5b) & \mathcal{X}[\![f]\!] & = & f & \text{otherwise} \\
(6a) & \mathcal{X}_{codelet}[\![(\textbf{if}\ M\ M\ M)]\!] & = & (\textbf{if}\ \mathcal{X}_{codelet}[\![M]\!]\ \mathcal{X}_{codelet}[\![M]\!]\ \mathcal{X}_{codelet}[\![M]\!]) & \\
(6b) & \mathcal{X}_{system}[\![(\textbf{if}\ M\ M\ M)]\!] & = & (\textbf{if}\ \mathcal{X}_{system}[\![M]\!]\ \mathcal{X}_{system}[\![M]\!]\ \mathcal{X}_{system}[\![M]\!]) & \\
 & \ldots & & &
\end{array}
$$

Figure 5: The soft termination transformation with codelets and blocking calls.
Note the domain of this transformation is in language of figure 1 with a small extension to allow multiple return values. The semantics of figure 2 apply.

When soft termination is implemented in a concrete language, other mechanisms may be available to create these wrappers.

Rules 4a and 4b describe how function calls are handled for $\mathcal{X}_{system}$ and $\mathcal{X}_{codelet}$. For codelets, the termination checks are added, just as in figure 3. For system code, no termination checks are added. The purpose of this is to avoid destabilizing system code. When termination is desired, it will only change the control flow of the codelet, not the system. If system code makes an up-call to a codelet, it becomes possible for a terminated codelet to throw an exception. System code would be responsible for catching this exception and proceeding appropriately. In a mobile code environment, where codelets are untrusted, system code must already be prepared for such up-calls to throw exceptions, but system code now need not be concerned with asynchronous termination.

Rules 5a and 5b describe how blocking function calls are replaced with calls to their non-blocking wrappers. The remainder of the rules describe how the transformation continues recursively on expressions.

Despite the domain of the transformation in figure 5 being within the language of figures 1 and 2, we can no longer prove termination; system code may have infinite loops. It is now the programmer's responsibility to label all code that may potentially have such infinite loops and provide some kind of non-blocking wrappers. It's important to point out that a program consisting strictly of functions labeled $\mathcal{X}_{codelet}$ will be transformed to precisely the same result as if it were written in the simpler language, and thus can be terminated. When control flow is executing in a system function, the termination checks will be deferred. When control flow is executing in a codelet function, termination will happen normally. This also addresses the the boundary-crossing thread concern (see section 3.1): if system code calls a function within a terminated codelet, that function will be guaranteed to terminate in a finite time, and thus a system thread cannot be "hijacked."

4 Java Implementation

In an effort to understand the practical issues involved with soft termination, we implemented it for Java as a transformation on Java bytecodes. Our implementation relies on a number of Java-specific features. We also address a number of Java-specific quirks that would hopefully not pose a problem in other language systems.

4.1 Termination check insertion

Java compilers normally output Java bytecode. Every Java source file is translated to one or more class files, later loaded dynamically by the JVM as the classes are referenced by a running program. JVMs know how to load class files directly from the disk or indirectly through "class loaders," invoked as part of Java's dynamic linking mechanism. A class loader, among other things, embodies Java's notion of a name space. Every class is tagged with the class loader that installed it, such that a class with unresolved references is linked against other classes from the same source. A class loader provides an ideal location to rewrite Java bytecode, implementing the soft termination transformation. A codelet appears in Java as a set of classes loaded by the same class loader. System code is naturally loaded by a different class loader than codelets, allowing us to naturally implement the codelet transform ($\mathcal{X}_{codelet}$) separately from the system code transform ($\mathcal{X}_{system}$).

Our implementation uses the freely available CFParse and JOIE packages [32, 7], which provide convenient bytecode file parsing and manipulation.

The basic structure of our bytecode modification is exactly as described in section 3.3.2. A static boolean field is added to every Java class, initially set to false. The CheckTermination operation, implemented in-line, tests if this field is true, and if so, calls a handler method that decides whether to throw an exception. As an extension to the semantics of figure 5, we allow threads and thread groups to be terminated as well as specific codelets, regardless of the running thread. The termination handler, when invoked, compares its caller and its current thread against a list of known termination targets. Note that, if the boolean field is set to false, the runtime overhead is only the cost of loading and checking the value, and then branching forward to the remainder of the method body.

Figure 6 shows how the codelet soft termination transform ($\mathcal{X}_{codelet}$) would be applied to a Java method invocation.

4.2 Control flow

Java has a much richer control flow than the little language introduced earlier. First and foremost, Java bytecode has a general-purpose branch instruction. We do

```
          getstatic termination_signal Z
          ifeq SKIP
          invokestatic termination_handler()V
SKIP:     original_invoke_instruction
```

Figure 6: Bytecodes inserted into Java for the soft termination check.

nothing special for forward branches, but we treat backward branches as if they were method invocations and perform the appropriate code transformation. An additional special case we must handle is a branch instruction which targets itself.

Java bytecode also supports many constructions that have no equivalent Java source code representation. In particular, it is possible to arrange for the `catch` portion of an exception handler to be equal to the `try` portion. That means an exception handler can be defined to handle *its own exceptions*. Such a construction allows for infinite loops without any method invocation or backward branching. While such code should most likely be rejected by the Java bytecode verifier, as it is not allowed in the JVM specification [33], the bytecode verifier currently treats such constructions as valid. We specifically check for and reject programs with overlapping `try` and `catch` blocks.

Suppose a `try` block includes the first instruction of a method, and the first instruction of the respective `catch` block recursively invokes the method. This situation could result in an infinite recursive loop. To avoid this problem, we specify that the checks we insert at the beginning of a method are not covered by any exception block. Then, if the class has been signaled to terminate, any method we enter will immediately fail.

Lastly, Java bytecode supports a notion of subroutines within a Java method using the `jsr` and `ret` instructions. `jsr` pushes a return address on the stack which `ret` consumes before returning. The Java bytecode verifier imposes a number of restrictions on how these instructions may be used. In particular, a return address is an opaque type which may be consumed at most once. The verifier's intent is to insure that these instructions may be used only to create subroutines, not general-purpose branching. As such, we instrument `jsr` instructions the same way we would instrument a method invocation and we do nothing for `ret` instructions.

4.3 Blocking calls

To address blocking calls, we follow the transformation outlined in section 3.3.2. Luckily, all blocking method calls in the Java system libraries are `native` methods (implemented in C) and can be easily enumerated and studied by examining the source code of the Java class libraries.

Java provides a mechanism for interrupting blocking calls: `Thread.interrupt()`. This method causes the blocking method to throw a `java.lang.InterruptedException` or `java.io.InterruptedIOException` exception. While the X transformation is defined to poll a non-blocking version, we would prefer to take advantage of the interruption support already inside the JVM.

To accomplish this, we must track which threads are currently blocking and the codelets on behalf of whom they are blocking. The wrapper classes now get the current thread and save it in a global table for later reference. In order to learn the codelet on whose behalf we are about to block, we take advantage of the stack inspection primitives built into modern Java systems [52, 25].

Stack inspection provides two primitives that we use: `java.security.AccessController.doPrivileged()` and `getContext()`. `getContext()` returns an array of `ProtectionDomains` that map one-to-one with codelets. The `ProtectionDomain` identities are then saved alongside the current thread before the blocking call is performed.

When we wish to terminate a codelet, we look up whether it is currently waiting on a blocked call, and only then do we interrupt the thread.

Taking advantage of another property of Java stack inspection, we can distinguish between blocking calls being performed on behalf of system code and those being performed indirectly by a codelet. Generally, we would rather not interrupt a blocking call if system code is depending on its result and system state could become corrupted if the call were interrupted. On the other hand, we have no problem interrupting a blocking call if only a codelet is depending on its result. Java system code already uses `doPrivileged()` to mark regions of privileged system code and `getContext()` to get dynamic traces for making access control checks. These regions are exactly the same regions where preserving system integrity upon termination is important; if system code is using its own security privileges, it wants the operation to succeed regardless of its caller's privileges. Thus, we overload the semantics of these existing security primitives to include whether blocking calls should be interrupted.

4.4 Invoking termination

Our system supports three kinds of termination: termination of individual threads, termination of thread groups, and termination of codelets.

To terminate a thread or thread group, we must map the threads we wish to terminate to the set of codelets potentially running those threads and set the termination signal on all classes belonging to the target codelet. Furthermore, we must check if any of these threads are currently blocking and interrupt them, as appropriate (see section 4.3). At this point, the thread requesting termination performs a `Thread.join()` on the target thread(s), waiting until they complete execution. Once the target thread has completed, the termination signals are cleared and execution returns to its normal performance.

If multiple threads are executing concurrently over the same set of classes and only one is terminated, the other threads may experience degraded performance while the target is still running because the termination signal will be true, and the termination handler will be invoked, only to return shortly thereafter.

In the case where we wish to terminate a specific codelet, disabling all its classes forever, we simply set the termination signal on all classes in the codelet and immediately return. Any code that invokes a method on a disabled class will receive an exception indicating the class has been terminated.

Note that termination requests can be handled concurrently. A potential for deadlock occurs where a thread could request its own termination, or where a cycle of threads may request each others' termination. In production, where a user is manually terminating threads or codelets, this would not be an issue. The termination operation itself should not be provided to untrusted codelets. Instead, it is protected using the same security mechanisms as other Java privileged calls.

4.5 Optimizations

If a Java method contains a large number of method invocations, the transformed method may be significantly larger than the original, causing performance problems. We address this concern by observing that we get similar termination semantics by adding the soft termination check at the entry points to methods rather than at the call sites. So long as a soft termination check occurs either before or immediately after a method invocation, the resulting program will execute the same.

A more aggressive optimization, which we have not yet performed, would be an interprocedural analysis of methods that can be statically determined to always terminate in a finite time. If a method has no backward branches or method invocations, we can statically label it as known-terminating. Then, when a known-terminating method is to be invoked, no termination check would need to be inserted. This could have a significant performance benefit for "getter" and "setter" methods. If termination checks are performed at method entry points as described above, then known-terminating methods can simply omit the termination check at their entry point, allowing for a completely local optimization.

Globally, it would be possible to generalize known-terminating methods. If a method only invokes known-terminating methods and has no backward branches, then it may also be defined to be known-terminating. A global analysis is not generally possible with Java dynamic loading (where classes are not available until runtime), although it could be possible in Java systems that support dynamic class reloading such as HotSpot [27].

4.6 Synchronization

A particularly tricky aspect of supporting soft termination in a Java system is supporting Java's synchronization primitives.

The Java language and virtual machine specifications are not clear on how the system behaves when a deadlock is encountered [26, 33]. With Sun's JDK 1.2.1, the only way to recover from a deadlock is to terminate the JVM. Obviously, this is an unsatisfactory solution. Ideally, we would like to see a modification to the JVM where locking primitives such as the `monitorenter` bytecode are interruptible as are other blocking calls in Java. We could then apply standard deadlock detection techniques and chose the threads to interrupt.

Additionally, it is possible to construct Java classes where the the `monitorenter` and `monitorexit` operations are not properly balanced. Despite the fact that there exist no equivalent Java source programs, current JVM bytecode verifiers accept such programs. As such, it is possible for a malicious program to acquire a series of locks and terminate without those locks being released until the JVM terminates.

Our current system makes no attempt to address these issues.

4.7 Thread scheduling

Our work fundamentally assumes the Java thread system is preemptive. This was not the case in many early Java implementations. Without a preemptive scheduler, a malicious codelet could enter an infinite loop and no other thread would have the opportunity to run and request the termination of the malicious thread. This would defeat soft termination.

4.8 System code interruptibility

Our work fundamentally assumes that all system methods that may be invoked by a codelet will either return in a finite time or will reach a blocking native method call which can be interrupted. It may be possible to construct an input to system code that will cause the system code itself to have an infinite loop. Addressing this concern would require a lengthy audit of the system code to guarantee there exist no possible inputs to system functions that may infinitely loop.

4.9 Memory consistency models

The Java language defines a relaxed consistency model where updates need not be propagated without the use of locking primitives. In our current prototype, we use no synchronization primitives when accessing this variable. Since external updates to the termination signal could potentially be ignored by the running method, this could defeat the soft termination system.

Instead, we take advantage of Java's `volatile` modifier. This modifier is provided to guarantee that changes must be propagated immediately [26]. On the benchmark platform we used, the performance impact of using `volatile` versus not using it is negligible. However, on other platforms, especially multiprocessing systems, this may not be the case.

4.10 Defensive security

Our prototype implementation makes no attempt to protect itself from bytecode designed specifically to attack the termination system (*e.g.*, setting the termination flag to *false*, either directly or through Java's reflection interface). Such protection could be added as a verification step before the bytecode rewriting.

5 Performance

We measured the performance of our soft termination system using Sun Microsystems Ultra 10 workstations (440 MHz UltraSparc II CPU's, 128 MB memory, Solaris 2.6), and Sun's Java 2, version 1.2.1 build 4, which includes a JIT. Our benchmarks were compiled with the corresponding version of `javac` with optimization turned on.

We used two classes of benchmark programs: microbenchmarks that test the impact of soft termination on various Java language constructs (and also measuring worst case performance), and macrobenchmarks which represent a number of real-world applications.

5.1 Microbenchmarks

We first measured a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, exception handling, synchronization, and I/O. We used a microbenchmark package developed at University of California, San Diego, and modified at University of Arizona for the Sumatra Project [28, 39]. The results are shown in figure 7.

As one would expected, tight loops suffered the worst slow-downs: roughly a factor of two. When we rewrite the loop, the termination check costs roughly the same as the original loop termination check, so it's sensible to see a factor of two performance degradation.

For other microbenchmarks, we see much smaller overheads. The overhead of handling exceptions, performing synchronization, or doing I/O operations dominates the cost of checking for termination. The largest overhead of these was 14% for the synchronization microbenchmark. This benchmark is a tight loop with a small synchronized block within it. The additional overhead can be attributed to rewriting the backward branch at the end of the loop.

For the I/O and exception-handling microbenchmarks, the performance figures are much better. As expected, our modifications should have a negligible effect in these areas. Since I/O and exception handling are relatively costly operations, modifications don't have a significant impact on the performance.

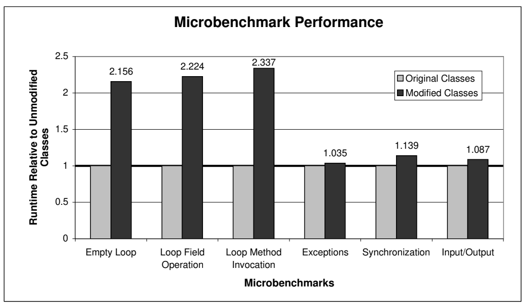

Figure 7: Performance of rewritten microbenchmark class files relative to the performance of the corresponding original class files.

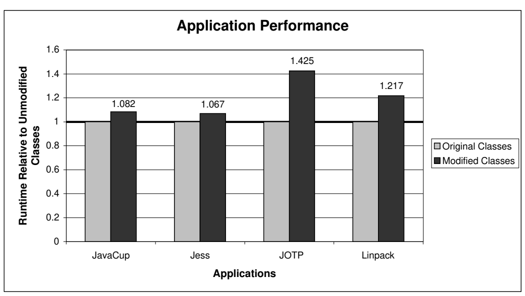

Figure 8: Performance of rewritten application class files relative to the performance of the corresponding original class files.

5.2 Application benchmarks

We benchmarked the real-world applications JavaCup [31], Linpack [15], Jess [23], and JOTP [35]. These programs were chosen to provide sufficiently broad insight into our system's performance.

JavaCup, a parser-generator, was chosen to demonstrate how the rewritten classes perform in handling text processing. Jess, an expert system, was chosen to demonstrate the performance of the rewritten classes in handling symbolic data and solving logic problems. Linpack is a loop-intensive floating-point benchmark. JOTP is a one-time password generator which uses a cryptographic hash function. The results are shown in figure 8.

For the JavaCup test, we generated a parser for the Java 1.1 grammar. There was a modest 8% increase in execution time. For the Jess test, we ran several of the sample problems included with Jess through the system, and calculated the cumulative runtimes. There was a 7% increase in execution time. Both JavaCup and Jess represent applications whose performance is not dependent on tight loops. Instead, these applications spend more of their time performing I/O and symbolic computations. Their performance closely tracks the performance of the I/O microbenchmark.

For the Java OTP generator, we generated a one-time password from a randomly-chosen seed and password, using 200,000 iterations. There was a 43% increase in runtime. For the Linpack benchmark, there was a 22% increase in runtime. Both JOTP and Linpack are loop-intensive programs, so we would expect their performance to more closely track the loop-based microbenchmarks. Indeed, the slowdown for JOTP and Linpack is much worse than JavaCup and Jess, but it's

Microbenchmarks		Application Benchmarks	
Microbenchmark	*Checks per Second*	*Application Benchmark*	*Checks per Second*
Empty Loop	1.417×10^8	JavaCup	2.771×10^7
Loop Field Operation	1.054×10^8	Jess	1.013×10^7
Loop Method Invocation	1.197×10^8	JOTP	4.102×10^7
Exceptions	8.551×10^7	Linpack	3.128×10^8
Synchronization	1.674×10^8		
Input/Output	3.557×10^6		

Figure 9: Average number of termination checks performed per one second increase in runtime for the micro and application benchmarks.

still much better than the worse case represented by the microbenchmarks.

5.3 Termination check overhead

Finally, to gauge the actual impact of our class file modifications, we counted the number of times we checked the termination flag for each benchmark. Using this, we calculated the number of termination checks being performed per every second increase in runtime overhead. The results for all benchmarks are listed in figure 9.

For the three looping microbenchmarks, We found that over $100,000,000$ checks are performed for every second of runtime overhead. This evaluates to less than 5 CPU cycles for each check performed. For the exception, synchronization, and input/output microbenchmarks, the cost of performing these original operations far outweighs the overhead of the checks, so the number of termination checks per second is not as significant in determining the performance impact. An extreme example of this is the input/output test. While we measured approximately $3,500,000$ checks per second, the overhead of performing the actual I/O operations overshadow this.

The application benchmarks reflect the results of the microbenchmarks. The loop-intensive JOTP and Linpack register approximately $41,000,000$ and $310,000,000$ checks per second, respectively. The results for JavaCup and Jess, of over $27,000,000$ and $10,000,000$ checks per second, respectively, are more in line with the I/O microbenchmark results.

These performance figures seem to indicate that for real-world applications, the showdown will be roughly proportional to how much the application's performance is dependent on tight loops. Applications which have tight loops may experience at worst a factor of two slowdown and more commonly $20-40\%$. Applications without tight loops can expect more modest slowdowns, most likely below 10%. The number of termination checks the system can perform per second seems not to be a limiting factor in system performance.

6 Real-World Test Case

In order to demonstrate our soft termination system, we integrated it into the Jigsaw web server. Jigsaw is a free, open Java-based web server which supports servlets [8]. We integrated our bytecode-rewriting system into the servlet loader for Jigsaw. Thus, every servlet that is loaded into Jigsaw is first rewritten to support soft termination.

We also wrote an administrative screen similar to the `top` utility in UNIX. It provides a list of all active servlets on the system. It also gives an option to terminate a servlet. Selecting this option activates the terminate signal in the specified servlet. We observed that servlets would take at most ten seconds to terminate.

7 Conclusion

While Java and other general-purpose language-based systems for codelets have good support for memory protection, authorization, and access controls, there is little support for termination. Without termination, a system can be vulnerable to denial-of-service attacks or even to bugs where a faulty codelet has an infinite loop.

We have introduced a concept we call soft termination, along with a formal design and an implementation for Java that allows for asynchronous and safe termination of misbehaving or malicious codelets. Soft termination can be implemented without making any changes to the underlying language or runtime system. Our Java implementation relies solely on class-file bytecode rewriting, making it portable across Java systems and easier to consider applying to non-Java systems. In real-world benchmarks, our system shows slow-downs of $10-40\%$, although we expect that this could be significantly lowered if we were willing to tie our system more closely to a specific Java implementation.

A larger research area remains: building language runtimes that support the general process-management semantics of operating systems. Because language runtimes allow and take advantage of threads and memory references that easily cross protection boundaries, traditional operating system processes may not be the appropriate in this new setting. Opportunities exist to design new mechanisms to add these semantics to programming language runtimes.

8 Acknowledgments

Jiangchun "Frank" Luo and Liwei Peng helped implement an early prototype of this system. Matthias Felleisen and Shriram Krishnamurthy provided many helpful comments. This work is supported by NSF Grant CCR-9985332.

References

[1] ALVES-FOSS, J., Ed. *Formal Syntax and Semantics of Java*. No. 1523 in Lecture Notes in Computer Science. Springer-Verlag, July 1999.

[2] BACK, G., AND HSIEH, W. Drawing the Red Line in Java. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems* (Rio Rico, Arizona, Mar. 1999). `http://www.cs.utah.edu/flux/papers/redline-hotos7.ps`.

[3] BACK, G., HSIEH, W. C., AND LEPREAU, J. Processes in KaffeOS: Isolation, resource management, and sharing in Java. Tech. Rep. UUCS-00-010, University of Utah, Apr. 2000. `http://www.cs.utah.edu/flux/papers/kaffeos-tr00010-base.html`.

[4] BACK, G., TULLMAN, P., STOLLER, L., HSEIH, W. C., AND LEPREAU, J. Java operating systems: Design and implementation. Tech. Rep. UUCS-98-015, University of Utah, Aug. 1998. `http://www.cs.utah.edu/projects/flux/papers.html`.

[5] BERNADAT, P., LAMBRIGHT, D., AND TRAVOSTINO, F. Towards a resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, Dec. 1998).

[6] BROMLEY, H. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers, 1986.

[7] COHEN, G., CHASE, J., AND KAMINSKY, D. Automatic program transformation with JOIE. In *Proceedings of the 1998 Usenix Annual Technical Symposium* (New Orleans, Louisiana, June 1998), pp. 167–178.

[8] CONSORTIUM, W. W. W. *Jigsaw - The W3C's Web Server*, 2000. `http://www.w3.org/Jigsaw/`.

[9] CZAJKOWSKI, G., AND VON EICKEN, T. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 1998), pp. 21–35.

[10] DEAN, D. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security* (Zurich, Switzerland, Apr. 1997).

[11] DEAN, D. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, New Jersey, Nov. 1998.

[12] DEAN, D., FELTEN, E. W., AND WALLACH, D. S. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Oakland, California, May 1996), pp. 190–200.

[13] DEAN, D., FELTEN, E. W., WALLACH, D. S., AND BALFANZ, D. Java security: Web browsers and beyond. In *Internet Besieged: Countering Cyberspace Scofflaws*, D. E. Denning and P. J. Denning, Eds. ACM Press, New York, New York, Oct. 1997, pp. 241–269.

[14] DIGITIVITY CORP. *The Digitivity Cage*, 1997. `http://www.digitivity.com`.

[15] DONGARRA, J., WADE, R., AND MCMAHAN, P. *Linpack Benchmark – Java Version*. `http://netlib2.cs.utk.edu/benchmark/linpackjava/`.

[16] DROSSOPOULOU, S., AND EISENBACH, S. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)* (Jyväskylä, Finland, June 1997).

[17] DROSSOPOULOU, S., WRAGG, D., AND EISENBACH, S. What *is* Java binary compatibility? In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 1998), pp. 341–358.

[18] EDJLALI, G., ACHARYA, A., AND CHAUDHARY, V. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS '98)* (San Francisco, California, Nov. 1998), ACM Press, pp. 38–48.

[19] ERLINGSSON, U., AND SCHNEIDER, F. B. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop* (Caledon Hills, Ontario, Canada, Sept. 1999), ACM Press.

[20] FELLEISEN, M., AND HIEB, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102* (1992), 235–271.

[21] FLATT, M., FINDLER, R. B., KRISHNAMURTHY, S., AND FELLEISEN, M. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP '99)* (Paris, France, Sept. 1999). `http://www.cs.rice.edu/CS/PLT/Publications/icfp99-ffkf.ps.gz`.

[22] FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 1998), pp. 171–183.

[23] FRIEDMAN-HILL, E. J. *Jess, The Java Expert System Shell*. Distributed Computing Systems, Sandia National Laboratories, 1997. `http://herzberg1.ca.sandia.gov/jess/`.

[24] GOLDBERG, A., AND ROBSON, D. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.

[25] GONG, L. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, June 1999.

[26] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[27] GRISWOLD, D. *The Java HotSpot Virtual Machine Architecture*. Sun Microsystems, Palo Alto, California, Mar. 1998. `http://java.sun.com/products/hotspot/whitepaper.html`.

[28] GRISWOLD, W. G., AND PHILLIPS, P. S. Bill and paul's excellent ucsd benchmarks for java (version 1.1), 1997. `http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html`.

[29] HAWBLITZEL, C., CHANG, C.-C., CZAJKOWSKI, G., HU, D., AND VON EICKEN, T. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, June 1998), USENIX.

[30] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages* (1998), ACM, pp. 86–93. `http://www.cis.upenn.edu/~switchware/papers/plan.ps`.

[31] HUDSON, S., FLANNERY, F., AND ANANIAN, C. S. *CUP LALR Parser Generator for Java*, 1999. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[32] IBM. *CFParse*, 1999. `http://www.alphaworks.ibm.com/tech/cfparse`.

[33] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[34] MALKHI, D., REITER, M., AND RUBIN, A. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, California, May 1998), pp. 40–51.

[35] MANTAKOS, H. *Jotp: The Java OTP Calculator*. `http://www.cs.umd.edu/~harry/jotp/`.

[36] McGraw, G., and Felten, E. W. *Security Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, New York, New York, 1999.

[37] Nilsen, K., Mitra, S., Sankaranarayanan, S., and Thanuvan, V. Asynchronous Java exception handling in a real-time context. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, Dec. 1998), NewMonics, Inc. `http://www.newmonics.com/news/conferences/rtss/plrtia/async.exception.pdf`.

[38] Organick, E. I. *Computer System Organization: The B5700/B6700 Series*. Academic Press, New York, 1973.

[39] Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, T., and Watterson, S. A. Toba: Java for applications: A *way ahead of time* (WAT) compiler. Tech. Rep. 97-01, The University of Arizona, 1997. `ftp://ftp.cs.arizona.edu/reports/1997/TR97-01.ps`.

[40] Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., and Purcell, S. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb. 1980), 81–92.

[41] Saulpaugh, T., Clements, T., and Mirho, C. A. *Inside the JavaOS Operating System*. Addison-Wesley, Reading, Massachusetts, Feb. 1999.

[42] Sirer, E. G. Kimera: A Java system architecture. `http://kimera.cs.washington.edu`, 1997.

[43] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J., and McDirmid, S. Distributed virtual machines: A system architecture for network computing. In *Eighth ACM SIGOPS European Workshop* (Sept. 1998).

[44] Sirer, E. G., Grimm, R., Gregory, A. J., and Bershad, B. N. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles* (Kiawah Island Resort, South Carolina, Dec. 1999), ACM, pp. 202–216.

[45] Spoonhower, D., Czajkowski, G., Hawblitzel, C., Chang, C.-C., Hu, D., and von Eicken, T. Design and evaluation of an extensible web & telephony server based on the J-Kernel. Tech. Rep. 98-1715, Cornell University, Nov. 1998. `http://www2.cs.cornell.edu/slk/papers/tr98-1715.pdf`.

[46] Stata, R., and Abadi, M. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages* (Jan. 1998), ACM, pp. 149–160.

[47] Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit deprecated?, 1999. `http://www.javasoft.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html`.

[48] Swinehart, D. C., Zellweger, P. T., Beach, R. J., and Hagmann, R. B. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (Oct. 1986), 419–490.

[49] Tullman, P., and Lepreau, J. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop* (Sept. 1998).

[50] van Doorn, L. A secure Java virtual machine. In *Ninth USENIX Security Symposium Proceedings* (Denver, Colorado, Aug. 2000).

[51] Wallach, D. S., Balfanz, D., Dean, D., and Felten, E. W. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles* (Saint-Malo, France, Oct. 1997), pp. 116–128.

[52] Wallach, D. S., and Felten, E. W. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, California, May 1998), pp. 52–63.

[53] Wirth, N., and Gutknecht, J. *Project Oberon*. ACM Press, 1992.