PROBLEMS RUNNING UNTRUSTED SERVICES AS JAVA THREADS

Almut Herzog
Dept. of Computer and Information Science
Linköping University, Sweden
almhe@ida.liu.se

Nahid Shahmehri
Dept. of Computer and Information Science
Linköping University, Sweden
nahsh@ida.liu.se

Abstract A number of Java environments run untrusted services as Java threads. However, Java threads may not be suitably secure for this task because of its problem with safe termination, resource control and thread isolation. These problem areas have been recognised by the research community and are comprehensively addressed in the not yet implemented Java Isolate API. Meanwhile, Java threads continue to be used for running untrusted code.

This paper examines the risks associated with Java threads that run untrusted code and presents existing research solutions. Requirements for a secure execution environment are presented. The requirements are contrasted by recommendations and problems when using Java threads for running untrusted code.

Keywords: Java, server, security, thread, untrusted code, termination, requirements for a secure execution environment

1. Introduction

There are a number of server environments or *containers* that run untrusted Java code: Applets in web browsers, servlets in web servers, Enterprise Java Beans (EJB) in their EJB containers, OSGi bundles in their framework—to name a few. In all the examples, the untrusted code must adhere to a certain API. In all scenarios the code comes from different sources and was developed without knowledge about other code that runs in the same container. Normally all named containers make use of Java threads to run the untrusted code.

E. Nardelli et al. (eds.), *Certification and Security in Inter-Organizational E-Service*

In operating systems, protection between threads of the same process is usually low because the design idea of threads is that they are co-operative and friendly—they are meant to achieve a common goal. Whereas processes may originate from different users, and may be hostile to one another, only a single user is meant to own an individual task with multiple threads [SG97].

In Java containers, this is not necessarily so and Java designers have only lately built protection mechanisms that prevent one thread from modifying and specifically from stopping other threads [Oak01, OW99]. In Java 1.1, this was not done yet, and consequently threads could terminate other threads within the Java virtual machine. An attack applet demonstrated that it could destroy all other applets in a browser [LaD97]; but the vulnerability was also used to check which applets were currently executing in a browser and to kill unwanted applets [HT98].

But there is not only the problem of directly manipulating another thread. Untrusted code should be prevented by its container to starve other threads. For this, resource management is needed which is not currently provided by any Java implementation.

For some containers (e.g. the OSGi framework) it is important to reliably perform life cycle management of the untrusted code it runs. Life cycle management comprises stopping the untrusted code (called bundle), updating it with a new bundle uploaded by a provider and restarting it. If the bundle cannot be stopped in a timely fashion (or not at all) the life cycle management fails and at the worse leaves the container in a hung state that may also affect applications it runs.

This paper explores the suitability of Java threads for running untrusted code and points to earlier and on-going research that deals with problems encountered. We arrive at recommendations that should be considered when building a Java container that runs untrusted code as Java threads when using an existing Java Virtual Machine.

The paper is structured as follows. In section 2, we examine existing Java container problems and refer to attempts to their solution. Section 3 states requirements that a secure container should fulfill. Section 4 is the main part and shows how a technical implementation of the stated requirements are supported or thwarted by the current Java implementaton. Section 5 concludes the paper.

2. Problems

The problem areas that are not well covered by Java threads are safe termination and resource control, and to a lesser extent also isolation. In the following sections we elaborate these problems and point to existing research solutions.

Safe Termination

In 2002, there was a rather inflamed debate in the Java newsgroups about running untrusted code as threads within a container. The discussion started with a message that complained that there was no recommended, safe way to stop a Java thread [Gal02]. The stop()-method within the Thread class is deprecated because it is not safe to use [Sun99, Hag99]. Consequently, there is no way to enforce that a thread terminates. Because of this, malicious threads can keep alive as long as they want. Their not-terminating may be enough to bring the execution environment into a hung state when it wants to shut down or uninstall the application to which this thread belongs (and waits indefinitely for the thread to terminate).

Example: An OSGi framework implementation (SGADK 3.1 (Service Gateway Application Developers' Kit) by Gatespace (`www.gatespace.com`), which is based on the OSGi specification 2.0) addresses the problem of termination by starting all threads created by bundles as daemon threads. A daemon thread is a thread that can be disregarded when the Java Virtual Machine wants to exit. It is meant to be used for helper threads like the garbage collector. This solution can simply be circumvented by a malicious thread by setting itself to be a non-daemon thread in its constructor:

```
public class MaliciousThread extends Thread {
    public MaliciousThread() {
        setDaemon(false);
    }
}
```

If a bundle in the used OSGi implementation specifically starts a non-daemon thread, the framework will hang on its shutdown command because it waits for the non-daemon thread to terminate (the System.exit()-method is not explicitly invoked). If that thread is looping or waiting forever, it effectively prevents the framework from shutting down and impairs its availability, because the framework is in a hung state.

Rudys and others [LPR99, RW02] have addressed safe thread termination by bytecode rewriting of every untrusted class. In the modified bytecode, it is checked at certain places in the code whether the code should terminate, much like the solution mandated by other literature [Sun99, Hag99]. The termination flag is checked on every backwards branch, at the beginning of every catch block and before every method call. However, it remains unclear if the solution can interrupt in a timely way a long-lasting native call such as writing a big chunk of a file over a slow network. The authors are aware that their solution encompasses a heavy performance penalty up to 100% and that they have not addressed the problem that an attacker could try to write code that explicitly attacks the rewriting algorithm.

Resource Control

Even though Java has good support for access control to resources through the Security Manager, once access is granted the resource is handed unrestrictedly to the client code. If for example the code in *malicious.jar* is allowed to write to `/tmp/a`, it may well fill the disk by writing an enormous amount of data to it. If *malicious.jar* is allowed to read certain files, it may well keep them open "for ever", thus effectively blocking them for other users or possibly reaching the operating system resource constraint of maximum number of open files per user or user process.

This lack of *release control* or *resource control* has been identified as a serious problem by the research community and solutions abound [CvE98, BCS01, BLT98, BHV01, CB02, CSN00, GL02, HS02, LP99, SBB+01] (see table 1 for a summary of resource-aware Java Virtual Machines). The most influential work is JRes, the resource-accounting interface for Java [CvE98]. The basic idea is to add resource accounting and limiting facilities to Java as a class library that replaces other core Java classes by bytecode rewriting. Due to a lack of resource management Java can neither prevent malicious code from using too many resources nor can it charge the code for the resources it is using. JRes is designed to address this problem. Both Web servers (running servlets) and Web browsers (running applets) are affected. JRes introduces a resource manager class that co-exists with the Java class loader and the security manager. The resource manager contains native code for CPU and memory accounting. Overuse callbacks are generated and throw exceptions when resource limits are exceeded. As JRes is not built into the JVM, there are a number of restrictions on what it can do. It specifically can not change the thread scheduling algorithm, take control of the scheduler or access the memory allocator and garbage collector. This work has eventually lead to the defintion of a resource management interface for the Java platform [CHS+03] that overcomes many shortcomings of JRes and may be incorporated into the Java platform in the future.

Isolation

Isolation among classes in the same runtime environment is achieved by the class loading mechanism [Gon98, LB98]. The Java class loader ensures that different implementations of classes with the same fully qualified name can exist in different class loaders. Apart from the name space advantage, the mechanism also prevents that a malicious class loader corrupts the classes used by another class loader by purposely loading a malicious class into that class loader's name space [JP00].

However, class loading is not a perfect isolation mechanism. Interference among classes that execute in the same Java virtual machine can still happen.

Name	Resources	Policy	Implementation
JRes [CvE98]	CPU, memory, network	Policy must be programmed as Java code but is otherwise versatile.	Use of the Java native interface (JNI) for CPU and memory accounting, bytecode rewriting.
– [BLT98]	CPU, memory	In a policy file with unknown syntax, permissions did not exist in Java 1.1	Based on a modified 1.1 Java virtual machine, mapping of Java threads to POSIX threads with realtime behaviour, modified Java heap manager allocates a fixed memory quota per protection domain.
Aroma [SBB+01]	CPU, disk, network	The resource control mechanisms allow limits to be placed on the rate and quantity of resources used by Java threads. Syntax unknown.	Java virtual machine based on C++-classes using native threads, Java 1.2.2 compatible; four to 11 times slower than a Sun or IBM Java implementation.
SOMA [BCS01]	CPU, memory, (bandwidth, disk space)	Unknown. Support for the policy language Ponder [CML+00] is future work.	Based on the Java VM Profiler Interface and JNI; bandwidth and disk space are not controlled only monitored.
Real-time Java www.rtj.org	CPU	Hard-coded	Choice of different thread schedulers that implement different policies.
– [LP99]	CPU	A language for the specification of resource limits is meant to be part of this work but a description is omitted due to lack of space	Based on a modified 1.1 Java virtual machine through modification of the thread scheduler.
Spout [CSN00]	CPU, memory	In a configuration file with unknown syntax.	Based on a modified 1.1 Java virtual machine. Resource accounting is updated for each method call. Little information is provided as resource control is only a side effect of Spout.
J-SEAL2 [BHV01]	CPU, memory, threads	In a Java object	Bytecode rewriting
JRAF [CB02]	CPU, memory, more resources may be plugged in	In an XML configuration file	Bytecode rewriting, a framework for resource accounting
Raje [GL02]	CPU, memory, sockets	Contract-based. The component comes with a policy.	Modified Kaffe JVM 1.0.6 with Linux C code
- [HS02]	Java-mediated resources	Extended Java policy file syntax.	Based on a modified 1.3 Java virtual machine. Uses the existing access controller to make resource control decisions.

Table 1. Resource-aware Java virtual machines

Czajkowski and Daynès [CD01] name the following remaining problems: Mutable parts of classes, typically static fields and static synchronised methods of system classes, can leak object references and can allow one application to prevent others from invoking certain methods. Internalised strings introduce shared, easy to capture monitors. Sharing event and finalisation queues can block or hinder the execution of some applications. Also, class loading is wasteful because of the code replication that is happening. Czajkowski says [Cza00]: "Typically, class loaders *do not share enough*: they replicate the code of application classes. Second, class loaders *share too much*: they share static fields of system classes."

Better isolation is achieved by the Multitasking Virtual Machine (MVM) [CD01] that particularly addresses the problem of executing different applications in the same Java virtual machine. MVM is a modified Java HotSpot virtual machine that shares application code wherever possible and replicates code where that is needed. In addition, native libraries, usually forbidden in safe Java environments, are contained by running them in a different process [CDW01] than the MVM.

MVM is closely related to the new Isolate API [JSR01, PCDV02]. An isolate is "a handle to an isolated computation" and allows controlled handling of that computation such as starting, suspending, resuming, and terminating. This safe life-cycle management is achieved by a strict isolation of the tasks and duplication of shared classes. The Isolate API is proposed but any implementation is currently dormant [Sop].

3. Requirements for a Secure Container

In previous work [Her02, HS04b], we have defined a secure container as an environment built in such a way that an application cannot harm other applications within the container, the container itself, the underlying operating system, hardware or connected network nodes. The more elaborate requirements are the following.

1 An application shall not use local or connected resources (CPU, RAM, temporary disk, bandwidth, etc.) in such a way that it would prevent other software from executing.

2 An application shall not explore or modify the local node (other applications, the container, the operating system, hardware) unless authorised.

3 An application shall not explore or modify connected nodes (local or remote) unless authorised.

The implementation of a Java container that enforces these requirements using vanilla Java threads to run the application is not possible with the current Java implemententation, partly due to isolation problems among Java threads

No.	Requirement	Recommendations and caveats
1	An application shall not use local or connected resources (CPU, RAM, temporary disk, bandwidth, etc.) in such a way that it would prevent other software from executing.	Thwarted by *unsafe thread termination*, *insufficient isolation* and *lack of resource control*.
2	An application shall not explore or modify the local node (other applications, the container, the operating system, hardware) unless authorised.	Addressed by the security manager's *existing access control* but possibly thwarted by *default policy*, *default thread security*, *coarse-grained thread access*, and *stepping out of the sandbox*.
3	An application shall not explore or modify connected nodes (local or remote) unless authorised.	Solved partially by the security manager's *existing access control* but only socket based network access is subject to access control.

Table 2. Requirements for a secure execution environment and existing recommendations and caveats. Refer to the items listed in section 4 for more details on the keywords in italics.

and partly because of the missing resource control features. In the following section, we put up recommendations and caveats that should be considered by container designers.

4. Recommendations and Caveats

Having described a number of problems that occur when multiple, untrusted applications execute in the same Java virtual machine, this section contains recommendations that should be considered by container designers in the absence of existing, reliable solutions. The recommendations show the current limits of Java but also how to efficiently use existing security features. For each recommendation or caveat we state which of the above requirements it solves or thwarts (see table 2 for a summary).

There is a technical and non-technical solution. The non-technical solution consists of a contract between container and application.

- *Contract* (solves the requirements but does not enforce them) EJB containers for example have a contract with the Enterprise JavaBeans that they execute. The contract is mandated by the Enterprise JavaBeans specification (see *Chapter 25: Runtime Environment* of the specification [Suna]). The beans are restricted in many ways (they must not start threads, must not use I/O, etc.) but the beans can get access to that functionality through the container. It is then up to the container to manage security aspects, e.g. to safely stop a thread or manage I/O. There is no

enforcement of that contract, but the bean that violates it is not compliant and may not run in compliant containers. Such a contract may be useful for other containers as well.

A technical solution is difficult because of the earlier mentioned shortcomings of the existing Java Virtual Machine. There is no perfect way for running untrusted code as Java threads yet but the following recommendations give insight in problematic areas and address possible solutions, extensions or work-arounds.

- *Unsafe thread termination* (thwarts requirement 1) Container designers need to be aware that there is no guaranteed way to stop threads unless the Java virtual machine exits, too. Only the java.lang.System.exit()-method will effectively terminate all running threads, regardless of their state. Any clean-up is then done by the operating system as a result of the death of the Java virtual machine process.

  If the container needs to stop threads and needs to rely on the fact that the thread actually stops, then this is not possible in a safe way in Java (and probably all other thread APIs). The reason for this is that threads do not keep track of their resources, which is one of their distinctions from heavyweight processes. With processes, the operating system can perform the clean-up because there is information about process states.

  Consequently, if reliably stopping threads is an issue for the container, one should consider running the untrusted code as a separate process in its own Java virtual machine (until the Isolate API has been implemented). This achieves good isolation from other processes and allows resource control on the operating system level. Applications are still able to communicate with each other—Java has support for inter-process communication through the java.lang.Runtime and java.lang.Process classes and through sockets. On the other hand, separate processes imply a great memory and startup overhead because of the many replications of the runtime environment and maybe also application classes.

  A research solution by Rudys and others [LPR99, RW02] addresses safe thread termination (see section 2.1).

- *Insufficient isolation* (thwarts requirement 1) Different applications are usually loaded by different class loaders into the runtime environment. However, class loaders do not offer perfect isolation between applications. System classes are still shared. A malicious application may use access to system classes to stage denial-of-service attacks for other applications. It may e.g. hijack the finaliser queue (which is shared), it

may explicitly or implicitly invoke the garbage collector (which suspends all other threads while it runs), it may monopolise the standard input and standard output stream, etc. This is addressed by the Isolate API [JSR01, PCDV02].

- *Lack of resource control* (thwarts requirement 1) Access to the CPU should be controlled, so that one application cannot starve other applications. This is possible with resource-aware Java virtual machines as mentioned in section 2.2 or by running each application in its own virtual machine and operating process.

- *Existing access control* (solves requirements 2 and 3) What resources application threads are allowed to access can be regulated with Java's security manager. The security manager should be activated by default, and the default policy for applications should be restrictive. By carefully tuning garbage collection and certain reordering of permissions in the policy file, the performance of a Java virtual machine with a security manager is usually acceptable [HS04a].

- *Default policy* (complicates requirement 2) The system default policy, hidden in the Java virtual machine installation directory, should be revised and probably overruled by a more rigid policy file for the execution environment. For instance, Sun's Java implementation comes with a system default policy that allows any thread to stop other threads.

- *Default thread security* (complicates requirement 2) Java's default security manager offers little protection for user threads from other threads. Any thread in the Java virtual machine can find any other non-system thread. Threads cannot stop arbitrary threads, but they can call the interrupt()-method of non-system threads. This can result in arbitrary behaviour because the interrupt()-method behaves differently depending on the state of the interrupted thread. The interrupt()-method is meant to signal to the executing thread that it should terminate. Thus, applications that correctly and politely respond to an interrupt can be shut down by a malicious application.

  Also, in the standard security manager, the permission to modify a thread is only checked if the thread to be modified or stopped belongs to the system thread group. Consequently, the Java virtual machine protects itself and its system threads, but does not offer the same access control for application threads. Any application thread, although run under a standard security manager, can still find out about all threads in the Java virtual machine, except for threads in the system group. The motivation for this behaviour is unclear. The Java documentation [Sunb] is aware of this

and suggests the implementation of a proprietary security manager that implements a stricter access control if that is needed by the application.

The container should use a security manager that implements access control for all threads and not only for system threads. Different policies for such access control are discussed e.g. in the section *Implementing Thread Security* of [Oak01].

- *Coarse-grained thread access* (complicates requirement 2) A more fine-grained policy for thread access may be needed, because a permission is granted for all threads or for no threads. A granularity of thread group may be more appropriate. For instances, it may be perfectly fine for a thread to manipulate other threads within its own group; at worst, it causes disarray among its own threads. However, manipulating threads outside the thread group of that application, especially system threads, is much more critical because this manipulation potentially affects other applications and the container.

- *Stepping out of the sandbox* (complicates requirement 2) Containers need to be careful that the code they run does not have permission to step out of its contained environment. A complete listing of all permissions and their implications can be found at [Sunc]. If the execution environment is meant to decide if it allows code that needs certain permissions, then knowledge about permission implications are vital. Also the contract between Enterprise JavaBeans and their containers [Suna] is a useful starting point for a reasonable default policy in a Java server environment. Some examples:

  - No application should be given permission to exit the Java virtual machine. That would effectively kill both the current thread and any other Java thread within the Java virtual machine as well as the operating system process of the Java virtual machine.
  - An application that comes with RuntimePermissions modifyThread, modifyThreadGroup or stopThread should be considered as potentially malicious. This application has the ability to manipulate all threads within the Java virtual machine including the threads that make up the execution environment and the Java virtual machine. Unfortunately, these permissions are required for code that uses RMI (remote method invocation).
  - No untrusted application should have permission to run native code in the same operating system process as the container. Native code executes outside the control of the Java security manager and interacts directly with the operating system. Native code is only restricted by limitations implemented by the operating system.

From these recommendations and the problems mentioned in section 2 it becomes clear that Java threads (as of J2SDK 1.5) are not suitable for running untrusted code. The only currently available resort is to use processes or to build on research prototypes. None of the three alternatives is attractive. For the time being, a solution based on a contract between the untrusted application and the container may be most suitable. However, a contract is not at all appropriate if the container executes completely unknown code such as applets in a web browser. In environments where more trust can be put on the application, where the origin of the application is known (as in JavaBeans, servlets and OSGi bundles) a contract solution may be an interim solution until the Isolate API is implemented.

5. Conclusion

To summarise, the Java thread model is currently not suitable for running untrusted code as threads. Operating system processes offer better isolation but with the obvious drawbacks of startup overhead, high memory consumption and heavyweight communication between applications. If threads must be used, the Java security manager enhanced with stronger thread control can control initial access to resources. A resource-aware Java virtual machine ensures that untrusted code does not hog CPU or memory. However, there is currently no support for isolation of threads from each other or a method for safely terminating a thread.

Research solutions exist and have even made it into a Java Community Process with the goal to introduce the Isolate API. This API will solve the shortcomings discussed in this paper. For the time being, a non-technical solution based on a contract between application and container may be most feasible.

References

[Gal02] <galaygalay@hotmail.com>. Sun's thread retardedness. `http://comp.lang.java.programmer` (visited 7-Apr-2004), 4-Apr-2002.

[BCS01] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. How to monitor and control resource usage in mobile agent systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, pages 65–75. IEEE, 2001.

[BHV01] Walter Binder, Jarle G. Hulaas, and Alex Villazón. Portable resource control in Java: The J-SEAL2 approach. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 139–155. ACM Press, October 2001.

[BLT98] Philippe Bernadat, Dan Lambright, and Franco Travostino. Towards a resouce-safe Java for service guarantees in uncooperative environments. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications*. IEEE, December 1998.

[CB02] Vladimir Calderon and Walter Binder. JRAF–the Java resource accounting facility. In *Proceedings of the Workshop on Resource Managment for Safe Languages (ECOOP'02)*. `http://www.ovmj.org/workshops/resman/` (visited 22-Apr-2004), June 2002.

[CD01] Grzegorz Czajkowski and Laurent Daynès. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 125–138. ACM Press, 2001.

[CDW01] Grzegorz Czajkowski, Laurent Daynès, and Mario Wolczko. Automated and portable native code isolation. In *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 298–307. IEEE, 2001.

[CHS+03] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce. A resource management interface for the Java platform. Technical Report TR-2003-124, Sun Microsystems Laboratories, May 2003.

[CML+00] Antonio Corradi, Rebecca Montanari, Emil Lupu, Morris Sloman, and Cesare Stefanelli. A flexible access control service for Java mobile code. In *Proceedings of the 16th Annual Conference on Computer Security Applications (ACSAC'00)*, pages 356–365. IEE, December 2000.

[CSN00] Tzicker Chiueh, Harish Sankaran, and Anindya Neogi. Spout: A transparent distributed execution engine for Java applets. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, pages 394–401. IEEE, April 2000.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 21–35. ACM Press, October 1998.

[Cza00] Grzegorz Czajkowski. Application isolation in the JavaŹ virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 354–366. ACM Press, October 2000.

[GL02] Frédéric Guidec and Nicolas LeSommer. Towards resource consumption accounting and control in Java: a practical experience. In *Proceedings of the Workshop on Resource Managment for Safe Languages (ECOOP'02)*. `http://www.ovmj.org/workshops/resman/` (visited 22-Apr-2004), June 2002.

[Gon98] Li Gong. Secure Java class loading. *IEEE Internet Computing*, 2(6):56–61, November 1998.

[Hag99] Peter Haggar. *Practical Java—Programming Language Guide*. Addison Wesley, 1999.

[Her02] Almut Herzog. Secure execution environment for Java electronic services. Licentiate Thesis No. 991. Linköping University, Sweden. `http://www.ida.liu.se/~almhe/publications/tek-lic-991.pdf` (visited 23-Apr-2004), December 2002.

[HS02] Almut Herzog and Nahid Shahmehri. Using the Java sandbox for resource control. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec'02)*, pages 135–147. Karlstad University, November 2002.

[HS04a] Almut Herzog and Nahid Shahmehri. Performance of the Java security manager. Submitted for publication, 2004.

[HS04b] Almut Herzog and Nahid Shahmehri. Requirements for a secure Java application container. Submitted for publication, 2004.

[HT98] Vesna Hassler and Oliver Then. Controlling applets' behavior in a browser. In *Proceedings of the 14th Annual Conference on Computer Security Applications (ACSAC'98)*, pages 120–125. IEEE, 1998.

[JSR01] Java Community Process. JSR 121: Application isolation API specification. `http://jcp.org/en/jsr/detail?id=121` (visited 22-Apr-2004).

[JP00] Jamie Jaworski and Paul J. Perrone. *Java Security Handbook*. Sams Publishing, November 2000.

[LaD97] Mark D. LaDue. Java insecurity. *Computer Security Journal*, 13(1):63–68, 1997.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 36–44. ACM Press, October 1998.

[LP99] Manoj Lal and Raju Pandey. CPU resource control for mobile programs. In *Proceedings of the 3rd International Symposium on Mobile Agents*, pages 74–88. IEEE, 1999.

[LPR99] Jiangchun Frank Luo, Liwei Peng, and Algis Rudys. Safe termination of Java classes. Project report, Dept. of Computer Science, Rice University, Houston, TX, November 1999.

[Oak01] Scott Oaks. *Java Security*. O'Reilly & Associates, Inc., 2nd edition, 2001.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., 2nd edition, 1999.

[PCDV02] Krzysztof Palacz, Grzegorz Czajkowski, Laurent Daynès, and Jan Vitek. Incommunicado: Efficient communication for isolates. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 262–274. ACM Press, November 2002.

[RW02] Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):138–168, May 2002.

[SBB+01] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Kenneth M. Ford, Paul T. Groth, Gregory A. Hill, and Raul Saavedra. State capture and resource control for Java: The design and implementation of the aroma virtual mach. In *Proceedings of the JavaŹ Virtual Machine Research and Technology Symposium*. Usenix, April 2001.

[SG97] Avi Silberschatz and Peter Galvin. *Operating System Concepts*. Addison Wesley, 1997.

[Sop] Pete Soper. [Isolate-interest] JSR 121 status? `http://altair.cs.oswego.edu/pipermail/isolate-interest/2004-March/00009%9.html` (visited 22-Apr-2004).

[Suna] Sun Microsystems, Inc. Enterprise JavaBeans specification 2.1. `http://java.sun.com/products/ejb/docs.html` (visited 16-Apr-2004).

[Sunb] Sun Microsystems, Inc. Java 2 platform, standard edition, v.1.5.0 API specification. `http://java.sun.com/j2se/1.5.0/docs/api/index.html` (visited 7-Apr-2004).

[Sunc] Sun Microsystems, Inc. Permissions in the Java 2 SDK. `http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html` (visited 15-Apr-2004).

[Sun99] Sun Microsystems, Inc. Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit deprecated. `http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecati%on.html` (visited 7-Apr-2004), 1999.