Detecting Malicious Java Code Using Virtual Machine Auditing

Sunil Soman Chandra Krintz Giovanni Vigna

Computer Science Department
University of California, Santa Barbara
{sunils,ckrintz,vigna}@cs.ucsb.edu

Abstract

The Java Virtual Machine (JVM) is evolving as an infrastructure for the efficient execution of large-scale, network-based applications. To enable secure execution in this environment, industrial and academic efforts have implemented extensive support for verification of type-safety, authentication, and access control. However, JVMs continue to lack intrinsic support for intrusion detection.

Existing operating system auditing facilities and host-based intrusion detection systems operate at the process level, with the assumption that one application is mapped onto one process. However, in many cases, multiple Java applications are executed concurrently as threads within a single JVM process. As such, it is difficult to analyze the behavior of Java applications using the corresponding OS-level audit trail. In addition, the malicious actions of a single Java application may trigger a response that disables an entire execution environment. To overcome these limitations, we have developed a thread-level auditing facility for the Java Virtual Machine and an intrusion detection tool that uses audit data generated by this facility to detect attacks by malicious Java code. This paper describes the JVM auditing mechanisms, the intrusion detection tool, and the quantitative evaluation of their performance.

1 Introduction

Java technology [18] was initially used by web page designers to embed active content. As a result of the widespread success and popularity of Java, developers now use the language for implementation of a wide range of large-scale systems, e.g., robust, mobile code systems [36, 19, 33] and complex server systems [43, 32]. In these systems, multiple applications and code components are uploaded by multiple (possibly untrusted) users for concurrent execution within a single Java Virtual Machine (JVM) [38]. The portability, flexibility, and security features of Java make it an ideal technology for the implementation of systems that support the execution of mobile code.

The use of Java enables portability because programs are encoded using an architecture-independent transfer format that can be executed without modification on any platform for which a JVM has been developed. Java supports flexibility by allowing applications to be upgraded and extended at run time through the use of dynamic code loading. In addition, the Java type system and verification mechanisms provide protection against some programming errors and malicious attacks.

The key areas that must be improved to enable continued wide-spread use of Java for Internet-scale server applications are performance and security. Recent advances in Just-In-Time (JIT) compilation and optimization techniques [35, 6, 8] offer significant improvements in Java program performance. Currently, to support security in server applications, Java provides mechanisms for authentication and access control [17, 2]. However, additional mechanisms are required to detect attacks that circumvent (or attempt to circumvent) the existing protections or abuse legitimate access.

Host-based Intrusion Detection Systems (HIDSs) provide a suitable solution to this problem [3, 12]. Existing HIDSs use process-level execution events, collected by an auditing facility in the operating system, to identify and respond to security threats and violations [13, 26, 52].

Unfortunately, since a JVM executes as a single, multi-threaded user process, we must presume that all suspicious activity reported by the auditing facility for the JVM process ID, is caused by the JVM itself. However, the culprit may be one of the *many* applications running within the JVM process. As users increasingly demand higher availability and reliability from the servers executing their applications, an intrusion response mechanism that simply terminates the JVM process that is executing malicious code becomes unacceptable. Therefore, novel auditing and intrusion detection techniques are needed, to provide finer-grained threat identification and response.

To this end, we have developed a JVM auditing facility and an intrusion detection system that detects attacks that are initiated by malicious Java code. To implement our system, We leveraged two existing technologies: a high-performance, open-source, Java Virtual Machine, called JikesRVM [1]; and an intrusion detection framework, called STAT, that we developed in prior work [50]. In this paper, we describe the JVM auditing mechanisms and the intrusion detection tool that we implemented, and provide a quantitative evaluation of the performance of our system.

In the next section, we place our work in the context of existing approaches to intrusion detection. In Section 3, we present an overview of our approach. In Section 4, we describe the JVM auditing system. Then, in Section 5, we present the intrusion detection system. Section 6 discusses the experimental evaluation of our system. Section 7 presents related work on Java security. Finally, Section 8 draws conclusions and outlines future work.

2 Extant Approaches to Intrusion Detection

Intrusion detection is performed by identifying the manifestation of a security violation in an input event stream. In host-based intrusion detection systems, the input event stream is usually represented by the audit records produced by the auditing facility of an operating system, such as the Solaris Basic Security Module [47]. Other possible input streams are system call traces and UNIX syslog messages.

The detection process can be performed according to different techniques. For example, it is possible to use statistical measures to characterize the normal behavior of users and applications [13, 23, 29]. Deviations from the established profiles are assumed to be evidence of an attack. The problem with these approaches is the difficulty to create a reliable model of the application behavior. An imprecise model may lead to both missed detections (called false negatives) and false alarms (called false positives).

Other approaches rely on the formal specification of the acceptable behavior of an application [34, 54, 52]. An execution history that does not conform to this behavior is considered malicious. The advantage of this approach is that the generation of false positives is greatly reduced. On the other hand, the generation of the specification requires access to the application source code and considerable effort, even when supported by tools. For this reason, these techniques are not in wide-spread use.

The most commonly used approach relies on models of attacks to analyze the input event stream. Systems based on this approach [20, 26, 39] are equipped with a number of attack signatures. These signatures are matched against the stream of audit data to identify the manifestation of an attack. The advantage of this approach is that it supports very effective intrusion detection and produces few false positives. In addition, signatures are not limited to modeling attacks against an application. For example, it is possible to describe attacks that represent abuses of legitimate access to the system. The disadvantage is that the signature set must be updated continuously as new ways of attacking a system are found.

In the work we present herein, we describe the design and implementation of a signature-based intrusion detection system. The system is an extension of previous work [49] in which we developed an auditing facility and an intrusion detection system for a mobile agent system, called Aglets [36]. This prior system detects malicious agent activity through the monitoring of the Aglets execution environment. In that case, the Java server application that was responsible for transferring and executing the mobile agents was instrumented to produce agent-related information.

The development of this agent monitoring system suggested a far more general approach in which the JVM itself is extended to produce the necessary information. Therefore, we developed a mechanism that collects

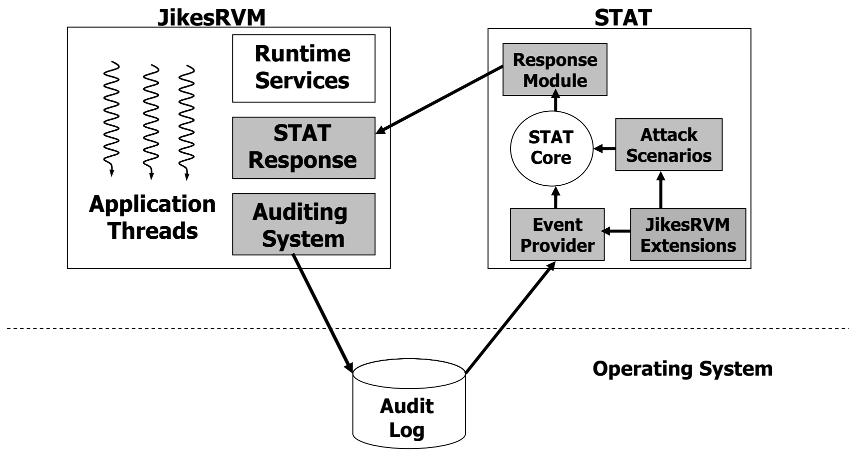

Figure 1: Architecture of the Java Virtual Machine auditing system and the STAT-based intrusion detection system

events that give information about the activity of threads within a JVM. The resulting auditing system can monitor the activity of any Java application, including various technologies supporting mobile code. We also developed an intrusion detection system that takes advantage of the finer-grained information produced by the JVM auditing system to detect attacks coming from malicious Java code.

To the best of our knowledge, no existing system performs auditing and threat detection at the Java thread level.

3 System Overview

Figure 1 shows a high-level overview of our intrusion detection architecture. The auditing system monitors executing Java threads (possibly associated with multiple, independent applications) and produces an audit log composed of records related to thread activity. The log is converted to an event stream by an event provider. The event stream is subsequently analyzed by an intrusion detection system for possible security threats or attacks. More precisely, the intrusion system compares patterns in the event stream against known attack scenarios. A match indicates that an intrusion or threat is in progress and that a response should be initiated. To do so, the intrusion detection system contacts the JVM which immediately terminates all offending threads. The JVM auditing system maps authenticated user IDs to threads so that malicious users can be identified and their threads selectively terminated.

Our system implementation couples and extends two existing frameworks: the JikesRVM, a high-performance Java virtual machine [30, 1], and STAT [51, 14], a general platform for the creation of intrusion detection sensors that we developed as part of prior work. The grayed boxes in the figure identify our extensions to these systems. In the sections that follow, we describe each of these components.

4 An Auditing System For A Java Virtual Machine

We implemented an auditing facility for the JikesRVM. The JikesRVM was designed with the goal of enabling high-performance Java server applications. As such, it represents the next generation of JVM technologies. The JikesRVM compiles Java bytecode programs into binary code (x86 or Power PC) at run time. The system operates at the method level and employs aggressive program optimization. The JikesRVM implements extensive run-time services including, garbage collection (GC), thread scheduling, synchronization, name-space separation, and class file verification. We implemented the latter two features ourselves as part of other projects [55]. Currently, the JikesRVM enables over 25% reduction in execution time over the Sun HotSpot JVM version 1.3.1 for x86/Linux (This value refers to a number of standard benchmark programs that we also used in Section 6 of this paper to evaluate the overhead introduced by our auditing facility).

All Java threads in the JikesRVM derive from a *virtual machine thread (VM_Thread)*, which is the basic unit of program execution. These threads are multiplexed onto a *virtual processor (VM_Processor)*, which is the abstraction of an underlying operating system thread. This implementation enables the JikesRVM to perform thread scheduling, independent to that performed by the operating system.

To monitor any suspicious activity performed by applications running in the JikesRVM, we extended the virtual machine with an event logging system. Each time an application or code component is uploaded into the executing JVM, a thread is created to execute the code. The thread is assigned a unique system identifier (SID) and a user identifier (UID). The SID enables the JikesRVM auditing system to identify a specific thread when logging execution events. UIDs associate users with individual threads. Both SIDs and UIDs are inherited by every thread created by the initial thread. Strong authentication mechanisms can be used to assign UIDs to threads, however, authentication and identification are not implemented natively by the JikesRVM (we are investigating such mechanisms as part of our current research).

In this first prototype, we simply map IP addresses to user IDs; this implementation is sufficient to effectively identify malicious threads. Since each application thread that executes in the system has an associated user ID, the threads of a user can be killed without affecting other user applications or the execution environment.

Figure 2 provides a graphic description of the JVM auditing facility. The auditing facility consists of an event driver, an event queue, and an event logger. The event driver adds thread-level execution events to the event queue. The logger processes events that are contained in the queue and writes them to an external log. We next describe the event driver. The implementation of the logger component is described in Section 4.2.

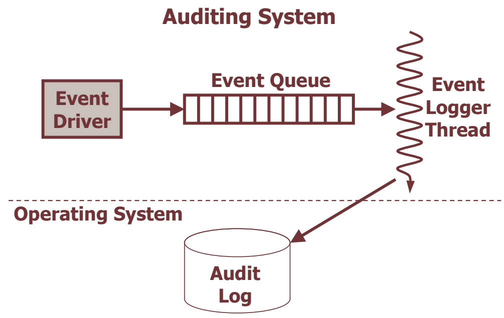

Figure 2: The JikesRVM auditing system.

4.1 Event Driver

The event driver provides an interface for inserting events in the event queue. The security-relevant operations in the JikesRVM are instrumented with calls to the event driver interface. For example, system calls invoked by the executing programs are instrumented in this way. We currently instrument operations that are of interest from a security perspective. However, any dynamic behavior can be instrumented using our system. The events we monitor in our prototype are: class events, system call events, JNI events, and thread interaction events.

4.1.1 Class Events

A class event occurs each time a thread loads a class. We instrumented the JikesRVM class loading code to record these events during execution. If a class is loaded from the network, the IP address of the code source is logged as part of the event.

We also record events associated with the creation of user-defined class loaders [37] and the classes that they load. User-defined class loaders are a powerful feature of the Java language that allows programs to define dynamically the way in which a class can be loaded and created. However, this functionality might also allow malicious applications to load classes from untrusted locations [41]. To record these events, we instrument the class loading code in user components that extend the Java ClassLoader library class [28].

4.1.2 System Call Events

System calls provide an interface through which an executing program can access operating system resources. For example, network communication, file I/O, and memory allocation are all accessed via system calls. By intercepting system calls, we are able to monitor and control such access.

The JikesRVM provides an abstraction of system calls via the VM system class. The system call routines in this class invoke the corresponding routines in the "Magic" (VM_Magic) class. The Magic class provides all the architecture-dependent, low-level functionality required by applications, e.g., raw memory access and calls to the operating system interface.

Methods in the Magic class are recognized and implemented as special system methods by both the baseline and optimizing compilers. These methods cannot be implemented by user application code. When the compiler encounters a call to a Magic method, it inlines code for the method into the caller routine. The Magic code for system calls consists of a number of calls to "wrapper routines" written in C. These routines perform the actual operating system call. To record system call events, we insert calls to the event driver interface at the VM abstraction layer (the VM system class) in the JikesRVM.

4.1.3 JNI Events

The JikesRVM implements most of the Java Native Interface (JNI) specification. The use of JNI is inherently "unsafe" since it allows user programs to call native methods, which can directly manipulate the host's file system, the process memory, and other resources. Therefore, the JNI might be used to bypass the security mechanisms of the JVM. However, the JNI offers greater control of system resources, e.g., for administration, resource accounting, and low-level device manipulation. In addition, the JNI offers the potential for improved execution performance since the code that is executed can be aggressively optimized and specialized statically for the underlying architecture, i.e., it does not require dynamic compilation. For such purposes, server systems may choose to support the use of the JNI functionality for a small, trusted subset of its users.

To monitor native method invocation, we instrument the JikesRVM JNI Compiler. The JNI compiler generates "glue code" that handles setting up the caller native method's frame for the transition from Java to C. Therefore, we inline a call to the event driver interface routine into this glue code.

4.1.4 Thread Interaction Events

In Java, an application thread can adversely affect another application thread by first obtaining a reference to the thread and then invoking one of the thread methods that could cause harmful thread-interaction, i.e., *suspend*, *interrupt*, *stop* and *kill*. In more recent versions of the Java language specification, these methods have been deprecated and it is recommended that thread interaction occur via shared variables only. However, many JVMs, including the JikesRVM, implement the direct access methods to maintain backward compatibility with legacy applications.

In Java, a thread could obtain illegal access to another thread directly, by ascending to the root thread group and recursively descending through the threads and thread groups below; or, indirectly, through the use of JNI methods that can access raw memory. Prevention of the former is straight-forward; however, the latter requires the monitoring of JNI events.

As described in the previous section, access to the JNI is granted to authorized users only. However, if the identity of a privileged user is stolen, the JNI can be used by an attacker to adversely interfere with other applications that are executing in the system. Therefore, we instrument all method invocations on thread objects that might cause interference in thread execution, including those that have been deprecated.

To record thread interaction, an instrumented method generates an event whenever the object on which it is invoked is of type Thread (*java.lang.Thread*). We modified the JikesRVM baseline and optimizing compilers so that a call to the event driver is inlined into each call of any thread method listed above, which might cause unwanted thread-interaction. In Java, an instance method is invoked as an "invokevirtual" call on an object reference. A reference to the instance resides on the stack when such a method is invoked. In our case, the object reference is a reference to the target application thread, i.e., the object on which the thread interaction method was invoked. To identify the source thread, i.e., the calling thread, we inline a call to the JikesRVM system method

VM_Thread.getCurrentThread, which gives us a reference to the thread in whose context the method call was made.

4.2 Event Logger

The event logger component runs as a JikesRVM system thread. The logger consumes events from an event queue, which the event driver interface populates. This design has the following advantages:

1. Implementing a separate event-logging thread allows us to identify and filter out events that are generated by the event logger itself (for example, system calls made by the event logger for disk I/O), by using the thread identifier.

2. Applications incur minimal delay since they do not need to wait for the disk I/O associated with the logging to complete. The application thread continues to run while the logger is processing events on the event queue.

We experienced a difficulty while instrumenting system calls. As described above, system calls are instrumented by inserting invocations to event driver interface routines, at the VM abstraction layer. The JikesRVM implementation requires that the routines in the VM class be uninterruptible. This is because an operating system call routine might directly modify the Java heap (e.g., using memcpy). Some system calls are not garbage-collection-safe (GC-safe), since they might modify the Java heap without the garbage collector's knowledge. The current JikesRVM implementation defines all system calls to be un-interruptible operations, whether or not they modify the heap. The VM stalls until the system call native code returns [25]. Hence, our event driver's event logging routines, which are called from the system call routines, cannot allocate memory to create event objects, as this might result in a garbage collection.
To solve this problem, we perform an initial static allocation of the event queue and associated event objects. Since the logger is decoupled from the driver, it can perform system calls that cause memory allocation. Therefore, the logger monitors the number of event objects in the queue, and, when a threshold is exceeded, it increases the size of the queue.

There are three drawbacks to this mechanism: (1) memory is wasted when the initial queue size is too large; (2) frequent reallocation can degrade performance if the queue size is too small; and (3) events will be missed if the queue fills before the logger thread has an opportunity to execute (and hence, increase the queue size). In our prototype, both the initial queue size and the queue allocation increment are application-dependent and can be set by the users of our system. Through empirical evaluation, using a large number of Java programs and different inputs, we determined that the queue should be doubled upon each increase and that a size of 16,000 events is sufficient to ensure that no event is ever lost. We use these values in our prototype for which we report results in Section 6.
The logger sleeps until the queue is sufficiently full. Then, the driver code wakes the logger. As with the initial queue size, this "wakeup" factor must be carefully chosen. If it is too small, the logger will be scheduled to run frequently, adversely affecting application throughput. If this factor is too large, the event driver might find the queue full and the event logger would miss events. Our empirical results for a wide range of programs and inputs indicate that this threshold should be 5/6 of queue capacity to ensure that no events are lost. The logger, when awakened, records all events that have been inserted into the queue since it was last put to sleep. It then clears the queue and goes back to sleep. Note that the event driver can continue to add events to the event queue as they are being processed by the logger. Thus, there are no extensive program interruptions due to the execution of the logger .

4.2.1 XML Encoding of Auditing Events

We encode the events that the logger consumes using an XML-based format. Encoding the events in XML supports inter-operability with other systems that may use the event stream generated by the auditing facility.
We developed a schema to encode all JikesRVM events, each of which consists of the event source, the action taken, and the result produced. Actual examples of our XML encoding are shown in Figure 3 for three JikesRVM events. The first encoding is an example of a thread interaction event in which a thread with ID 7 owned by a user with ID 23 kills a thread with ID 4 owned by the user with ID 10. The second encoding

```
<event timestamp=”Wed Oct 16
        17:17:14 2002”>
    <source>
        <thread id=”7” uid=”23” />
    </source>
    <action type=”THREAD KILL”/>
        <target>
            <thread id=”4” uid=”10” />
        </target>
    </action>
    <result status=”SUCCESS” />
    </result>
</event>
```

```
<event timestamp=”Mon Nov 25
        15:30:27 2002”>
    <source>
        <thread id=”2” uid=”17” />
    </source>
    <action type=”SYSCALL sysOpen” >
        <target>
            <file name=”/etc/passwd”
                mode=”WRITE” />
        </target>
    </action>
    <result returncode=”-1”
        status=”FAILED” />
    </result>
</event>
```

```
<event timestamp=”Mon Nov 25
        15:31:02 2002”>
    <source>
        <thread id=”8” uid=”11” />
    </source>
    <action type=”NET CONNECT”>
        <target>
            <server remoteaddress=”128.111.68.170”
                port=”25640” />
        </target>
    </action>
    <result returncode=”ECONNREFUSED”
        status=”FAILED”/>
    </result>
</event>
```

Figure 3: XML Encoding of example JikesRVM events.

is an example of a system call event in which a thread with ID 2, owned by a user with ID 17 attempts to open the `/etc/passwd` file for writing and fails. The final example in the figure shows the encoding of a network event in which a thread, owned by user with ID 11, tries to establish a connection to a host having IP address 128.111.68.170 on port 25640 and the connection is refused.

The `source` element describes the thread (and hence, user) that initiates the operation; the thread ID (attribute `id`) and the user ID (attribute `uid`) are both attributes of the `thread` element. The `action` element describes the operation performed by the source. The `type` attribute of this action element describes the action being recorded (for example, “JNI” to denote that a native method was invoked). The `target` element describes the target to which the action is being applied. The target of an operation can be a file, a server, a method, or a thread. Of these, the latter is described by the `id` attribute; others are described using the `name` attribute. The `result` element specifies the outcome of the operation. This element has two attributes, `returncode` and `status`, which are used to record return values, e.g., from system calls, and status values, e.g., errno values.

5 Detecting Malicious Java Code

The event stream produced by the JikesRVM event logger thread is used by an intrusion detection system to identify possible threats and attacks. The intrusion detection system was developed leveraging the STAT framework [26, 50]. The STAT framework provides a generic signature-based intrusion detection engine that can be extended to match a specific environment through a well-defined process.

The first step of the extension process includes the definition of a *language extension module*. This module extends STATL [14], the domain-independent attack modeling language provided by the framework, with the event types that are specific of a particular target domain. Therefore, we developed a language extension module that defines the event types that are produced by the JikesRVM auditing facility (e.g., the `JEvent` type). By doing this, it was possible to use the JVM-specific events when writing STATL scenarios. These scenarios represent state-transition models of attacks.

An example of a scenario is shown in Figure 4. The scenario models a two-step attack in which a malicious application uses the JNI to obtain a reference to another application’s thread, and then calls the “kill” method on that reference in an attempt to terminate the thread. This attack is detected by checking for a JNI method invocation (`transition_1`), followed by an attempt of the application to communicate with another user’s thread (`transition_2`). This and other scenarios are presented in detail in Section 5.1.

The second step of the framework extension process is the development of an *event provider module*. This module is responsible for collecting events from the environment and translating them in the format defined in the language extension module. We developed an event provider module that reads the events contained in the audit log produced by the JikesRVM event logger and generates events in the format specified by the language extension described above. These events are matched by the STAT analysis engine against the available attack sce-

```
scenario jikesRVM_jnithread () {
    transition transition_1 (s0 → s1) nonconsuming {
        [JEvent m1]: m1.targetType == "method" &&
            m1.actionString == "JNICALL";
        {
            log("JNI method %s invoked by thread %s",
            m1.target.name, m1.source.id);
        }
    }
    transition transition_2 (s1 → s2) nonconsuming {
        [JEvent m1]: m1.targetType == "thread" &&
            m1.actionString == "THREAD KILL" &&
            m1.source.uid == m1.target.uid;
            m1.source.uid == m1.target.uid;
        {
            log("THREAD KILL by user %s's thread %s
                to user %s's thread %s",
            m1.source.id, m1.source.uid,
            m1.target.id, m1.target.uid);
        }
    }
    initial state s0 {}

    state s1 {}

    state s2 {
        log("Suspicious thread communication
            after JNI method invocation!!");
    }
}
```

Figure 4: Example attack scenario.

narios.
The third and final step of the extension process is the creation of a *response module* that can be used to react to detected attacks in a specific environment. We developed a response module that initiates an appropriate response action when an attack or threat coming from a Java application is detected. The module sends to a dedicated thread in the JikesRVM a request for a particular response action. Currently, it is possible to request the termination of a particular thread or the termination of every thread belonging to a specific user. However, the system can be easily extended to implement other types of response, such as the dynamic modification of security restrictions. The communication between the response module and the JikesRVM is secured using encryption.
In summary, by extending STAT's generic analysis engine with the modules that we developed specifically for the JikesRVM, we obtained a complete signature-based intrusion detection system that is able to detect and respond to attacks coming from malicious Java applications. The following section describes the attack scenarios that we developed for this system.

5.1 Attack Scenarios

To evaluate the efficacy with which our intrusion detection system can detect suspicious activity, we developed mobile programs that exercise different attacks and the corresponding STATL scenarios. Although we only describe four such scenarios for brevity, the system can be extended easily to detect any number of different attacks. Examples of such attacks include email forging [41] and denial of service due to continuous resource allocation [7, 41], e.g., of heap memory, files, sockets.
The four scenarios that we developed implement a range of suspicious activities including attempts to access sensitive information, suspicious inter-thread communication, ping and port scans against hosts on an internal network, and transfer of privileged information outside the network. For these scenarios, we assume a server execution model in which users connect to the JVM server and upload code that the JVM executes.
In addition, we assume that the Java type system and class file verification together enable type-safe execution [38, 37]. Our auditing facility and intrusion detection system could be bypassed if type safety is somehow violated by an attacker's program, e.g., integers are converted to references or the program counter is modified to execute at an arbitrary memory location.

5.1.1 Unauthorized Access Detection

In this attack, an application attempts to access privileged information from the host operating system, such as the password file on a UNIX host. The attack is detected by monitoring system calls that operate on file system resources. The system intercepts and records all attempts to access privileged resources. The record contains the identity of the owner of the thread and the type of access requested. Following is an example of such an alert from the intrusion detection tool. The alert represents the attempt of an application to write to the privileged `/etc/shadow` file, which stores users' password hashes (on many UNIX systems).

```
TIME:  01/20/2003 13:44:04
ACTION: PRIVILEGED ACCESS
SOURCE UID: 0
SOURCE THREAD ID: 2
PRIVILEGED RESOURCE: /etc/shadow
ACCESS MODE: WRITE
MSG: Attempt to stat privileged file.
RESULT: FAILURE
SENSOR: jikesRVMstat@localhost
```

This scenario is an example of how the system can be used to detect malicious behavior even when an attack fails.

5.1.2 Harmful Inter-Thread Communication

A reasonably enterprising intruder might disrupt the functioning of other application threads in the system [41]. Consider the scenario in which an intruder has forged a legitimate user's identity, e.g., by stealing her user id, and has uploaded code using the stolen identity. In addition, the legitimate user is authorized to invoke native methods via the Java Native Interface (JNI). The intruder executes a native method that is designed to give her access to a thread object of another user. She then sends signals to that thread to terminate the thread. The system detects this scenario by detecting that a JNI invocation is performed followed by potentially harmful cross-thread communication between threads owned by two different users. The intrusion detection response module can communicate this information to the Java Virtual Machine, which can terminate the application uploaded by the intruder. Following is an example of such a detection alert:

```
TIME:   01/20/2003 20:44:04
ACTION: THREAD STOP after JNI Call.
SOURCE UID: 0
SOURCE THREAD ID: 2
TARGET UID: 8
TARGET THREAD ID: 1
MSG: JNI method "print" invoked by thread 2 (uid 0).
     Suspicious thread communication after
     JNI method invocation!!
RESULT: SUCCESS
SENSOR: jikesRVMstat@localhost
```

This scenario shows that in some cases malicious behavior can be detected only by using features that are internal to the JVM. Alternately, the attacker might obtain illegal access to an application thread by ascending to the thread's root thread group, and recursively descending through threads and thread groups below [41]. The intruder can then call *Thread.stop()* on the victim's thread. This single-step attack and can be handled by monitoring *Thread.stop()* calls.

5.1.3 Detecting Network Scans

With the next scenario, we show how the system can detect "bounce" attacks [24, 41] on internal servers. Using these well-known attacks, a malicious program may identify, manipulate, or discover vulnerabilities in hosts on an internal network.

In this scenario, the Java socket library is used by an attacker to perform network scans against hosts on a network behind a firewall. We assume that the JVM server system has access to these internal hosts. A malicious user uploads code that performs ping scans against internal subnets in an attempt to identify hosts that are alive. Similarly, the attacker can perform TCP and UDP scans to identify hosts with potentially vulnerable network services. To detect ping scans, we monitor the connection attempts made by an application to a range of hosts, or a range of ports on a host. Following is an example of an alert from the intrusion detection system that detects multiple connection attempts to a range of ports on host 128.111.68.170. The system identifies the thread and the user who performed the scan.

```
TIME:   01/23/2003 17:56:04
ACTION: MULTIPLE CONNECTS
SOURCE UID: 0
SOURCE THREAD ID: 2
SOURCE ADDR: 128.111.68.169
TARGET ADDR: 128.111.68.170
MSG: Connect attempts (1058) to multiple ports.
RESULT: FAILURE
SENSOR: jikesRVMstat@localhost
```

This scenario shows that it is possible to precisely identify the malicious code performing the attack using JVM-level audit data.

5.1.4 Detecting Transfer of Privileged Information

In the next scenario, we show how events that "leak" sensitive server information to the outside world [7] can be detected. The system is capable of detecting such events since we record system calls and network events.

Program	Description	Static Classes	Static Methods	Execution Time (sec)
compress	Spec JVM98 compression utility	12	44	7.11
db	Spec JVM98 database access program	3	34	16.22
jack	Spec JVM98 Java parser generator based on the Purdue Compiler Construction Tool set	56	315	4.40
java cup	LALR parser generator: A parser is created to parse simple mathematics expressions	36	385	0.38
javac	Spec JVM98 Java to bytecode compiler	176	1190	6.28
jess	Spec JVM98 expert system shell benchmark: Computes solutions to rule based puzzles	151	690	2.76
mpeg	Spec JVM98 audio file decompression tool Conforms to ISO MPEG Layer-3 spec.	55	322	5.84
mtrt	Spec JVM98 multi-threaded ray tracing implementation	26	180	3.59

Table 1: Description of the benchmarks used. The final column shows the execution performance in seconds of each benchmark executed using the JikesRVM.

An attempt to access server information, e.g., the `/etc/passwd` file, may not in itself be malicious behavior. The `/etc/passwd` file is commonly world-readable, since many UNIX utilities that run without super-user privileges depend on accessing this file to function correctly. In addition, most UNIX systems use separate shadow password files that contain the hashes of the actual passwords. However, the information contained in the */etc/passwd* file, such as account names and user information (names and addresses), might provide hints to an outside attacker, e.g., to mount a password guessing attack. As such, it may not be desirable for such information to leak out. To detect such an event, we implemented a two-step attack scenario in which an intruder accesses server information and then successfully connects to an external, untrusted machine. The alert produced for such an attempt is shown below (128.111.68.0 is our internal network) .

```
TIME:   01/23/2003 17:13:58
ACTION: PRIVILEGED TRANSFER
SOURCE UID: 0
SOURCE THREAD ID: 2
INTERNAL NETWORK: 128.111.68.0
REMOTE ADDR: 128.111.43.218
PRIVILEGED RESOURCE: /etc/passwd
ACCESS TYPE: READ
MSG: Attempt to open privileged file.
     Possible transfer of privileged information
     outside internal network!
     File: /etc/passwd     Host: 128.111.43.218
RESULT: SUCCESS
SENSOR: jikesRVMstat@localhost
```

This scenario shows how the intrusion detection system can be used to associate two apparently legitimate (and authorized) operations to produce evidence of suspicious behavior.

6 Evaluation

We collected the results that we present in this section by repeatedly executing benchmark applications on the JikesRVM with and without the auditing and response components. In both cases, the execution times include the time for both dynamic (Just-In-Time) compilation of the benchmark methods and their execution. We performed the timings on an Intel Xeon 2.4GHz processor (with Hyperthreading enabled) and a Seagate 15000 rpm SCSI hard disk. The operating system is Redhat Linux 7.3, kernel version 2.4.18. The JikesRVM version we used was 2.1.1, with the FastSemispace configuration. FastSemispace implements a semispace copying garbage collector and fully optimizes all methods Just-In-Time. In addition, the entire optimizing compiler is part of the JikesRVM image at startup, i.e., class files for this subsystem have been loaded, compiled, and optimized.

The benchmark programs that we used for result generation are listed in Table 1. This set of programs is commonly used for empirical evaluation of Java-based systems and includes the Spec JVM98 [45] benchmark

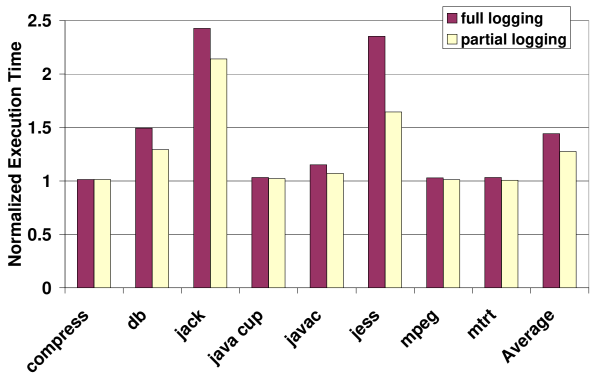

Figure 5: Execution times of our JikesRVM extension relative to the JikesRVM without auditing and response for a number of standard Java benchmarks. Full event logging implies that all system call, thread interaction, and JNI events are recorded. With partial event logging, we only log network use, file I/O, thread interaction, and JNI invocation events.

Program	Average Events per Second: Total Logging	Average Events per Second: Partial Logging
db	3995	956
compress	190	85
jack	12394	1894
java cup	420	209
javac	2442	400
jess	10075	3105
mpeg	1862	453
mtrt	474	240
Average	3982	918

Table 2: Event Rate (event count / execution time) for each benchmark. The first and second columns of data show the event rate for total logging and partial logging, respectively.

suite.

6.1 JikesRVM Auditing Overhead

To evaluate the efficacy of our system, we measured the overhead that the system imposes on the execution of programs for which no intrusions or threats are detected. Figure 5 shows the relative slowdown for each benchmark. The base case to which we are comparing is the JikesRVM without modification.

The delay experienced by the user application due to event logging depends on the number of events. The left-hand bar in the graph shows the impact of full logging, i.e., when all system calls, thread interaction, and JNI calls are logged. System call events generally dominate all other kinds of events, and applications with a large number of system calls incur in a substantial performance penalty. The right-hand bar, gives the performance of our system when partial logging is performed, that is when we only log events that we use in the attack scenarios presented in this paper, namely, network and file I/O, thread interaction, and JNI access. By reducing the number of events of interest, program performance improves. A user can configure our system to record any number of event types to manually adjust the trade-off between system performance and events audited.

The graph indicates that our logging introduces very little overhead in most cases (with or without full logging). For all but three benchmarks, the degradation is very small: For all benchmarks except *db*, *jack* and *jess*, performance is degraded by 5% with full logging and 2% for partial logging, on average. When we include these three benchmarks, performance is degraded by 44% for full logging and 26% for partial logging, on average.

Db, *jack* and *jess* have significant performance degradation because these benchmarks perform a large amount of file access operations, and consequently, produce a large number of events. This is clear from the average number of events per second shown in Table 2; data for both full and partial logging is shown. The relative performance degradation is greater for *jack* and *jess* than for *db* since they execute for a much shorter time (the execution times without auditing are shown in column 5 of Table 1).

Figure 6 shows the breakdown of the events generated by partial logging (100% denotes the total). These events are the number of JikesRVM system call interface routines ("wrapper routines") that are invoked by the JikesRVM Magic class. These, in turn, map onto the corresponding operating system calls: *sysStat* maps to the *stat* system call, *sysOpen* to *open*, *sysReadByte* and *sysReadBytes* to *read*, and *sysWriteByte* and *sysWriteBytes* to *write*. *Other* denotes all other non-system call events and is the only solid-colored segment. For all of the benchmarks, system call events clearly dominate other

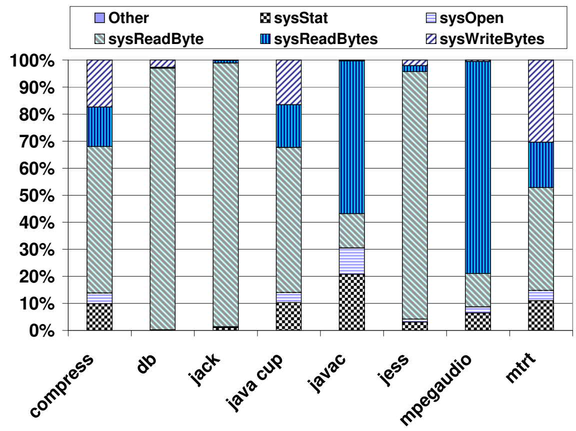

Figure 6: Breakdown of events for partial logging. These events are only those that we use for the attack scenarios that we present in this paper. Note that partial logging implies that in case of system call events, only network and file I/O events are recorded.

types of events. The number of system call events of each type varies across the benchmarks. However, file reads and writes are the most common in all cases. As part of future work, we plan to aggregate read and write system calls to reduce the overhead of instrumentation, so that events are generated only when a threshold is reached.

7 Related Work

Our work is primarily related to and complements three different areas of research. The first is extant intrusion detection research which we describe and contrast to our system in Section 2. Other related work includes alternate approaches to ensuring safe execution of mobile programs (including Java-based operating systems) and thread termination techniques.

The goal of much of the prior related research has been to develop operating systems and system management components using the Java language to enable resource management and process protection through the use of Java type-safety and load-time verification mechanisms [5, 31, 40, 22, 46, 48, 11, 21, 27, 4]. Other related work has focused on mechanisms that ensure that the execution of mobile code will not unintentionally or maliciously harm the underlying systems. Such techniques include stack inspection [16], proof-carrying code [42, 10, 9], software fault isolation [53], and code replacement [7].

The system that we propose is not an alternative to these existing approaches to program protection and system security. Instead, it offers a complementary technique (thread-level intrusion detection) that can be used to identify suspicious events or activities that may not be caught or detected by these existing approaches. For example, our system can be used within secure execution environments and Java-based operating systems to detect threads that continuously allocate memory in an attempt to cause the system to fail due to memory exhaustion or threads that perform bounce attacks on internal machines. In addition, our system is easily extensible and as such, administrators can add event detection of previously unforeseen attacks that arise but that are not handled by the underlying system.

A second area of related work is thread termination [44, 15]. In [44], the authors provide a formal specification and implementation of a technique for thread cessation called *soft termination*. Using this technique, a mobile code system can asynchronously and safely destroy the threads of a mobile program without termination of the execution environment. In our system, the response module receives messages from the STAT system when thread activity warrants its termination. The response module discontinues all threads in the system that were initiated by the user that spawned the ill-behaved thread. The module destroys non-running threads by removing them from the thread scheduling queue.

We could have implemented soft termination within the response module instead. Soft termination guarantees correct thread termination in the presence of all program and system activities, e.g., blocking system calls. As such, it is more robust and complete than our termination process. Since the JikesRVM currently only supports non-blocking system calls, we selected our simpler implementation for our initial prototype. As the JikesRVM evolves to include blocking system calls, we plan to consider the use of soft termination within our response module as part of future work.

8 Conclusions and Future work

Auditing at the Java Virtual Machine (JVM) level allows for fine-grained access to application execution events.

This information is necessary to perform effective intrusion detection and response for next-generation JVM server technologies in which multiple applications are uploaded from multiple (possibly untrusted) sites and execute concurrently within a single JVM. To this end, we developed an auditing facility for the JikesRVM and a host-based intrusion detection system that employs this audit data. As a result, attacks that exploit features internal to the JVM, such as the JNI, can now be detected. To our knowledge, this is the first system that performs auditing and intrusion detection at the thread-level within a JVM.

We also evaluated both the effectiveness of the detection process and the performance of the auditing systems. The results show that our approach introduces limited, adjustable, overhead while enabling many different attack scenarios to be detected.

Our future work will have two foci. First, we plan to extend and optimize the auditing system to handle additional events and to reduce the overhead of instrumentation. The extensions we plan include a publish-subscribe mechanism that allows intrusion detection systems to dynamically configure the auditing facility so that only the events that actually are necessary to the detection process are logged. In addition, we will use the experience that we gained with the prototype described herein, to reduce the overhead of instrumentation and audit collection within the JikesRVM.

As a second research direction, we plan to correlate traces collected at different abstraction levels to perform more effective intrusion detection. In particular, we plan to analyze the JVM-level traces with respect to application-level and OS-level traces. This integrated, multi-level approach will allow for more focused malicious code detection and a clearer evaluation of the impact of an attack on the underlying operating system.

Acknowledgments

We would like to thank the anonymous reviewers for providing extensive and useful comments on this paper.

Giovanni Vigna's work was supported by the National Science Foundation under grant CCR-0209065.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[2] A. Anderson. Java Access Control Mechanisms. Technical report, Sun Microsystems, March 2002.

[3] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, April 1980.

[4] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, October 2000.

[5] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[6] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[7] A. Chander, J. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *Proceedings of the 2001 DARPA Information Survivability Conference & Exposition (DISCEX II)*, pages 1027–1040, Anaheim, CA, June 2001.

[8] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[9] C. Colby, K. Crary, R. Harper, P. Lee, and F. Pfenning. Automated Techniques for Provably Safe Mobile Code. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume 1, pages 406–419, Hilton Head, SC, January 2000. IEEE Computer Society Press.

[10] C. Colby, P. Lee, G. Necula, F. Blan, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Program-*

ming Language Design and Implementation, pages 95–107, Vancouver, British Columbia, Canada, June 2000.

[11] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 21–35, Vancouver, Canada, October 1998.

[12] H. Debar, M. Dacier, and A. Wespi. Towards a Taxonomy of Intrusion Detection Systems. *Computer Networks*, 31(8):805–822, 1999.

[13] D.E. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[14] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[15] M. Flatt, R. Findler, S Krishnamurthy, and M. Felleisen. Programming Lanugages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proceedings of the ACM International Conference on Functional Programming (ICFP'99)*, pages 138–147, Paris, France, September 1999.

[16] C. Fournet and A.D. Gordon. Stack Inspection: Theory and Variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318, Portland, Oregon, 2002. ACM Press.

[17] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[20] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, pages 435–450, Toulouse, France, November 1992.

[21] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, June 1998.

[22] C. Hawblitzel and T. von Eicken. Luna: A Flexible Java Protection System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[23] P. Helman and G. Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. In *IEEE Transactions on Software Engineering*, volume Vol 19, No. 9, pages 886–901, 1993.

[24] Hobbit. The FTP Bounce Attack. Bugtraq, July 1995. `http://www.geocities.com/SiliconValley/1947/Ftpbounc.htm`.

[25] IBM Research. The Jikes Research Virtual Machine User's Guide. `http://www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html`.

[26] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[27] J. Hartman and L. Peterson and A. Bavier and P. Bigot and P. Bridges and B. Montz and R. Piltz and T. Proebsting and O. Spatscheckti. Experiences Building a Communication-oriented JavaOS. *Software – Practice and Experience*, 30(10), April 2000.

[28] Java 1.2 Library API. `http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html`.

[29] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[30] JikesRVM. Project Home Page. `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[31] JOS Developer Team. JOS: An Open, Portable, and Extensible Java Object Operating System. `http://cjos.sourceforge.net/archive/`.

[32] JRun. Project Home Page. `http://www.hallogram.com/jrun`.

[33] Jumping Beans. Project home page. `http://www.jumpingbeans.com`.

[34] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[35] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *Proceedings of the International Symposium on Code Generation and*

Optimization (CGO03), pages 69–78, San Francisco, CA, March 2003.

[36] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley Longman, 1998.

[37] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 36–44, Vancouver, British Columbia, Canada, 1998.

[38] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[39] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.

[40] M. Golm and M. Felser and C. Wawersich and J. Kleinoder. The JX Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

[41] G. McGraw and E.W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 2nd edition, 1999.

[42] G. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL97)*, pages 106–119, Paris, France, January 1997.

[43] NonStop Server for Java Software. Project Home Page. `http://nonstop.compaq.com/view.asp?IO=NSJAVAPD01`.

[44] A. Rudys and D.S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2):138–168, 2002.

[45] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[46] T. Stack, E. Eide, and J. Lepreau. Bees: A Secure, Resource-Controlled, Java-Based Execution Environment. In *Proceedings of the IEEE Conference on Open Architectures and Network Programming*, pages 97–106, San Francisco, CA, April 2003.

[47] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.

[48] P. Tullmann and J. Lepreau. Nested Java Processes: OS Structure for Mobile Code. In *Proceedings of the Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 111–117, Sintra, Portugal, 1998.

[49] G. Vigna, B. Cassel, and D. Fayram. An Intrusion Detection System for Aglets. In *Proceedings of the International Conference on Mobile Agents*, pages 64–77, Barcelona, Spain, October 2002.

[50] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, pages 1046–1055, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[51] G. Vigna, R.A. Kemmerer, and P. Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In W. Lee, L. Mè, and A. Wespi, editors, *Proceedings of the 4^{th} International Symposiun on Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 69–84, Davis, CA, October 2001. Springer-Verlag.

[52] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, Oakland, CA, May 2001. IEEE Press.

[53] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993. ACM Press.

[54] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[55] L. Zhang and C. Krintz. An Investigation of Concurrent Application Execution in the JikesRVM. Technical Report TR2003-02, Dept. Computer Science, UCSB, 2003.