I-JVM: a Java Virtual Machine for Component Isolation in OSGi

Nicolas Geoffray[†]
nicolas.geoffray@lip6.fr

Gaël Thomas[†]
gael.thomas@lip6.fr

Gilles Muller[±]
gilles.muller@emn.fr

Pierre Parrend[*]
parrend@fzi.de

Stéphane Frénot[‡]
stephane.frenot@insa-lyon.fr

Bertil Folliot[†]
bertil.folliot@lip6.fr

[†]UPMC/LIP6 [±]EMN
INRIA-Regal
104 avenue du Président Kennedy
75016 Paris, France

[*]FZI Karlsruhe
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany

[‡]INSA Lyon CITI
INRIA-Ares
20 avenue Albert Einstein
69621 Villeurbanne Cedex, France

Abstract

The OSGi framework is a Java-based, centralized, component oriented platform. It is being widely adopted as an execution environment for the development of extensible applications. However, current Java Virtual Machines are unable to isolate components from each other. For instance, a malicious component can freeze the complete platform by allocating too much memory or alter the behavior of other components by modifying shared variables.

This paper presents I-JVM, a Java Virtual Machine that provides a lightweight approach to isolation while preserving compatibility with legacy OSGi applications. Our evaluation of I-JVM shows that it solves the 8 *known OSGi vulnerabilities that are due to the Java Virtual Machine and that the overhead of I-JVM compared to the JVM on which it is based is below* 20%.

1. Introduction

The OSGi framework [26] is a Java-based component platform that is being widely adopted as an execution environment for the development of extensible applications, such as Eclipse [2] or Java Enterprise Servers [3]. Extensibility in the OSGi platform is provided through a deployment unit called a *bundle*, which groups together a set of components that are loaded through a specific Java class loader. The OSGi platform is popular because it provides modularity while still providing efficient communication through direct method calls between components.

Initially, the OSGi platform was designed for environments where all bundles trust each other. Nowadays, it is also being promoted for systems such as next generation Internet home gateways where third party services can be downloaded dynamically [31]. However, the OSGi platform cannot protect a bundle against another malicious or buggy bundle. First, `java.lang.Class` objects, strings and static variables are shared in the Java Virtual Machine (JVM). The corruption of any one of these entities by a malicious or buggy bundle will impact all bundles. Second, a thread can freeze the JVM and deny service by exhausting memory or monopolizing the CPU. Third, it may be impossible to terminate a bundle that denies service, which makes a shut down of the entire platform the only solution. Not surprisingly, a recent work has identified 25 different vulnerabilities in current implementations of the Java/OSGi platform that may either lead to a violation of data integrity or a freeze of the platform [29]. While 17 of these vulnerabilities are due to a weak implementation of the OSGi framework itself and can be solved by adding suitable security checks, the remaining 8 originate in isolation issues and need to be solved at the JVM level.

Several approaches to providing isolation in a single JVM, through *Isolates* (or Java processes) have been recently introduced [5, 9, 17]. These solutions duplicate the `java.lang.Class` objects, strings and static variables between isolates. However, to ensure full isolation, they confine a thread to a single isolate. As a consequence, a communication between two isolates must be done using an RPC-like mechanism, which involves parameter copying and in some cases thread synchronization. Since the OSGi platform uses communication between bundles heavily, using RPCs would induce a significant overhead. For example, despite using a highly optimized virtual machine, the developers of Singularity [12] report that a simple local remote procedure call through a shared heap takes about 2500

cycles while a direct call takes only about 10 cycles. Additionally, copying parameters implies modifying legacy bundles either at the source or bytecode level [34], that could compromise compatibility with legacy bundles.

This paper presents I-JVM, a Java Virtual Machine with lightweight isolates that is specifically designed to support the needs of the OSGi platform by associating each bundle with a separate isolate. The key contribution of I-JVM is to permit thread migration between isolates in order to keep the cost of an inter-isolate method call low. This enables complete bytecode compatibility with legacy OSGi bundles by avoiding the need to rewrite inter-bundle method calls. The main features of I-JVM are:

- Memory isolation. As shown by the Multi-Tasking Virtual Machine (MVM) [9], making `java.lang.Class` objects, strings and static variables private to an isolate is sufficient to ensure memory isolation in a single JVM. Therefore, an isolate cannot access an object of another isolate unless a reference is given explicitly through method invocation.

- Resource accounting. I-JVM keeps track of the current isolate in which a thread is running. This allows recording the amount of memory and CPU time spent within an isolate. These statistics provide sufficient information for an administrator or for resource policies to detect denial of service attacks from malicious bundles.

- Termination of isolates. When an isolate terminates, its classes should not be invoked anymore. In case a thread returns back to the terminating isolate, I-JVM modifies the stack so that an exception is raised and trapped at a lower stack level. All the objects referenced by the terminating isolate are reclaimed by the garbage collector, with the exception of objects shared with other bundles.

I-JVM has been developed by modifying the VMKit Java Virtual Machine [15, 16], a JVM specifically designed for easing experiments in Java. We have used I-JVM to run two legacy OSGi platforms: Felix [1] of the Apache Community and Equinox [2] of the Eclipse Project.

Overall the results of this study are:

- I-JVM solves the OSGi JVM-related weaknesses identified in [29]. We present 8 attacks that cover these weaknesses.

- I-JVM has a 16% overhead on inter-bundle calls. This is an order of magnitude better than the cost of an RPC call between two processes. Overall, the I-JVM slowdown is between 1% and 20% on a representative suite of macrobenchmarks.

- I-JVM requires the addition of only 650 lines of code to VMKit. Implementing I-JVM features in a legacy JVM should not be much more complex.

The rest of the paper is structured as follows. Section 2 describes the vulnerabilities of the OSGi platform. Section 3 explains and discusses the design and implementation of I-JVM. Section 4 provides performance measurements of I-JVM, and evaluates its robustness against denial of service attacks. Section 5 describes related work. Finally, Section 6 concludes the paper.

2. Vulnerabilities of OSGi

Vulnerabilities in OSGi have been identified at three sources [29]: (i) at the underlying operating system level, (ii) at the OSGi platform level and (iii) at the JVM level. The first kind of vulnerability is due to the possibility of running native code either inside the JVM process or as a separate process. These vulnerabilities are enabled by JNI or the `Runtime.exec` Java call. The second kind of vulnerability is related to weaknesses in the OSGi run-time and can be solved by adding security checks in the OSGi implementation [29]. In this paper, we attempt to solve the third kind of vulnerability which targets the JVM platform.

JVM vulnerabilities can themselves be subdivided into three categories: (i) lack of isolation, (ii) lack of resource accounting, (iii) failure to terminate a bundle. In the rest of this section, we present a list of 8 attacks that cover the previously reported JVM vulnerabilities in [28, 29]. Our experiments in Section 4 show that all attacks may corrupt, freeze or abort unprotected OSGi platforms.

Lack of isolation. As mentioned previously, *java.lang.Class* objects, strings and static variables are shared in the JVM by all bundles. A malicious bundle can alter static variables or lock shared objects, and therefore interfere with the execution of other bundles. We consider two representative attacks:

- A1 - Modification of a static variable: All bundles share static variables. Therefore a bundle can modify a public non-final static variable defined by either other bundles, the OSGi platform or the core Java System Library. For example, a malicious bundle can set a shared variable to null, thus preventing the correct execution of other bundles. Bundles can discover static variables from other bundles either at compilation, or at runtime with the reflection API of Java.

- A2 - Synchronized method or synchronized call block: a bundle can lock shared strings, *java.lang.Class* objects or static variables, which can eventually freeze the system.

Lack of resource accounting. JVMs implement a bundle by using a specific class loader. However, JVMs do not perform resource accounting on a per class loader basis. In case of the over-use of resources, it is impossible to identify the faulty bundle and stop execution of its code. Resource accounting would help detecting the following five denial of service attacks:

- A3 - Memory exhaustion: a malicious bundle consumes most of the memory by holding references to many or large objects. This leads to an *OutOfMemoryError* for other bundles.
- A4 - Standalone infinite loop: a malicious bundle consumes all CPU resources by entering an infinite loop.
- A5 - Excessive object creation: a malicious bundle repetitively allocates objects without referencing them, thus triggering garbage collection and object finalization, which monopolize the CPU.
- A6 - Excessive thread creation: a malicious bundle crashes the platform by exceeding the number of threads supported.
- A7 - Hanging thread: a malicious bundle blocks when being called, thus never returning to the caller.

Bundle termination. In some situations, the JVM is unable to unload a bundle and deallocate all its allocated objects. First, other bundles may continue to reference the bundle, thus preventing the JVM from unloading the bundle classes, because a call to methods defined by the classes of the bundle can still occur. Second, if the OSGi runtime recognizes a bundle as misbehaving and wants to stop its execution, methods of the bundle may be executing or be in the call stack of running threads.

- A8 - Lack of termination support: a malicious bundle continues execution even if the OSGi platform tries to unload it.

Finally, misuse of the Java `finalize` method is a vulnerability, not reported in [28], that targets any platform with dynamic class loading. A malicious code can run an infinite loop in the method and therefore prevent any memory reclamation. We do not discuss this vulnerability in this paper because we think the JVM should never allow an untrusted class to override the `finalize` method.

3. I-JVM Design and Implementation

The design goal of I-JVM is to provide bundle isolation while preserving the communication model of the OSGi platform, which relies on direct method calls. An OSGi application is composed of a set of dynamically loaded bundles and of the OSGi runtime itself. To implement isolation, each bundle is executed within a separate isolate. Additionally, the OSGi runtime runs in a specific isolate, Isolate0, which has higher rights than standard bundles.

In this section, we present I-JVM in detail. We focus on the main issues: isolation, resource accounting and termination of isolates. Then we report the implementation of I-JVM in VMKit [15, 16], and finally we describe how to run an OSGi platform using I-JVM.

3.1. Design

I-JVM permits explicit object sharing between isolates by passing an object reference in an inter-isolate method call. The key point of I-JVM is to provide thread migration between isolates in a single address space, which is a prerequisite for object sharing. Different isolates execute therefore on the same execution stack, which impacts isolation, resource accounting and termination.

Isolation and Thread Migration

I-JVM runs isolates in the same address space. Isolates provide a lightweight protection mechanism integrated in the JVM, so that the classes running in one isolate cannot crash classes running in another independent isolate. In I-JVM, an isolate is built from a class loader, so its scope is the classes loaded by the class loader. There is a specific isolate, Isolate0, which has higher rights on the platform than standard isolates. These rights are the permissions to start a new isolate, to terminate an isolate and to shut down the entire Java platform. The first Java class loader created becomes Isolate0. The subsequent class loaders are standard isolates.

An inter-isolate method call induces a thread migration. Each thread possesses a reference to the isolate in which it is currently running. This reference is updated on each cross-isolate call. A method defined in the Java System Library is not executed in a special isolate but in the isolate that called it.

To implement isolates, the main change from the JVM specification is to have a per-isolate private copy of static variables, strings and `java.lang.Class` objects. This is sufficient to ensure that an isolate does not have access to the internal state of another isolate since: (i) an isolate cannot construct a foreign reference thanks to the type safety of the Java bytecode; (ii) an isolate cannot access an isolate-private field or method thanks to the scope of fields and methods in the Java bytecode.

Isolation of static variables is done by associating an array of task class mirrors (defined by MVM in [9]) per class. The task class mirror of a class contains the initialization

state of the class, the static variables and the associated *java.lang.Class* object. Each isolate is assigned an index into the array. I-JVM uses the current isolate reference of the thread as an index into the array of task class mirrors of a class. Accessing a static variable requires fetching the execution environment, loading the isolate reference from the execution environment, loading the task class mirror and finally loading the static variable. Compared to simply loading a static variable, the task class mirror approach requires two additional loads. An isolate also always has to check the initialization state of a class before accessing one of its static variable or before calling a static method. Similarly to MVM, the just in time compiler cannot remove all of the class initialization checks, because the code compiled must be reentrant.

Resource Accounting

I-JVM monitors the execution of isolates. Resources consumed in the code of an isolate are charged to the isolate. Resources consumed in a method of the Java System Library are charged to the caller of the method. I-JVM counts the CPU time consumed, the memory used, the number of threads created, the number of connections used, the number of bytes read or written through I/O connections, and the number of garbage collection activations.

Memory and connections: The problem of memory accounting in OSGi is that multiple bundles share objects. I-JVM follows the garbage collector memory accounting algorithm of Price *et al.* [30], which specifically deals with different protection domains sharing a same object. The overall insight of their algorithm is that the garbage collector performs the memory accounting.

On a garbage collector activation, the collector scans the memory used by threads and charges an object to a task the first time the object is traversed. On subsequent execution of the collector, the shared object may be charged to another task. Although imprecise, this algorithm still offers a range of memory usages for a task, from maximum (all memory reachable from the task) to minimum (memory reachable only from the task). The author acknowledges that the approach opens a door to a resource Trojan horse problem [20] where a task allocates a large object that becomes shared, but the collector charges the object to another task. We discuss this issue in Section 6.

Threads: I-JVM counts the number of threads an isolate creates. I-JVM charges threads to their creator, even if they execute code from a random isolate via inter-bundle calls.

I/O reads and writes: Any read or write to a connection is instrumented in order to charge the isolate performing the operation. The approach is similar to accounting of network resources in JRes [10]: there are few classes that perform reads and writes on connections, and instrumenting them is straightforward.

CPU time: For CPU accounting, we have studied two solutions. The first one is to insert per-isolate time updates during an inter-isolate call. This induces a significant performance penalty because it requires: (i) two system calls to fetch the current time, one when entering and one when leaving the isolate and (ii) a lock acquisition to update the CPU time when leaving the call. Therefore, we chose a second solution in which I-JVM counts the CPU time spent in isolates by regularly sampling the value of the isolate reference of a running thread.

Garbage collection: I-JVM uses a single GC for all isolates. To detect attacks on garbage collection activations, I-JVM counts the number of times an isolate triggers the GC.

Isolate Termination

There are two main problems when terminating an isolate. First, threads migrate between isolates; therefore I-JVM cannot just kill the threads created by the isolate. Also, a thread created by another isolate may be executing code from the terminating isolate. Second, shared objects referenced by other isolates cannot be released.

When terminating an isolate, I-JVM stops the execution of all threads by sending them a signal. The handler of the signal is defined at startup and cannot be modified by isolates, thanks to the Java language. Upon receiving the signal, a thread inspects and modifies its stack as follows. For each frame, if it is called from a frame that belongs to the terminating isolate, the thread changes the return pointer to throw a `StoppedIsolateException` exception. The exception manager makes sure that all synchronized objects on the stack are released. Also, the terminating isolate cannot catch this exception: even if the isolate tries to catch it in the Java code, I-JVM will ignore it. Finally, I-JVM takes a special action for the last frame:

- If it belongs to the Java System Library, I-JVM sets the interrupted flag of the thread so that I/O or sleep calls are interrupted. This approach is similar to protection domain termination in Spring [18].
- If it belongs to the terminating isolate, the thread throws the `StoppedIsolateException`.

Moreover, code from the terminating isolate should not be called anymore. I-JVM prevents execution of the isolate by (i) not JIT compiling the methods not JITed yet and (ii) inserting at the beginning of all methods defined by the isolate an unconditional branch that jumps to a function that throws the `StoppedIsolateException`.

An isolate is only removed from memory when there is no remaining object whose class is defined by the isolate.

This approach is similar to the now deprecated *Thread.stop* mechanism. The *Thread.stop* method was deprecated because it may lead to objects with inconsistent states. Since only objects from the terminating isolate and shared objects used by the isolate may become inconsistent, we think the benefits of being able to terminate a bundle outweigh the inconsistency problems.

3.2. I-JVM Implementation

We have implemented I-JVM in the VMKit virtual machine [15, 16] which is specification compliant with the JVM. Overall, I-JVM required the addition of 650 lines of code to VMKit which are distributed as follows:

- Static variables, strings and *java.lang.Class* objects: 200 lines of code for implementing the task class mirror in each class. The changes are done at two levels: (i) in the Java class representation, which contains the task class mirror, (ii) in the bytecode translator to modify the accesses to static variables and *java.lang.Class* objects to reference the task class mirror.
- Method call: 150 lines of code for the update of the isolate reference in each isolate method compiled by VMKit.
- Resource accounting: 100 lines of code for accounting, for CPU, memory, I/O and threads.
- Isolate per bundle: 50 lines of code to create and attach a new isolate to a class loader when the latter first loads a class.
- Isolate termination: 150 lines of code for the termination of isolates.

These numbers are quite low and suggest that implementing I-JVM within a legacy JVM should be relatively easy.

3.3. Running OSGi on I-JVM

An OSGi application is made of a set of dynamically loaded bundles and the OSGi runtime itself. To implement protection, each bundle is executed within a separate isolate; the OSGi runtime runs in the Isolate0.

Isolate0 is associated with the applicative class loader that loads the `main` function of the OSGi framework. When OSGi loads a new bundle, it allocates a new class loader, and I-JVM associates a standard isolate to this class loader. This makes it possible to run a legacy OSGi runtime on I-JVM without any modification. The start method of a bundle receives an object that represents OSGi. This object is the first shared object between bundles. It is used in OSGi to register object references in a name service and to find foreign references. Hence, at startup, a bundle can only access foreign objects in this name service.

We additionally define a few rules that an OSGi runtime should follow:

1. It should create a new thread when calling the *start* and *stop* methods of a bundle, in order to prevent a malicious bundle from freezing the OSGi runtime.

2. It should use Java permissions to deny access of privileged resources to bundles. For example, the JVM already verifies that a class has the correct permissions to run non-Java code through the use of the JNI interface or the *Runtime.exec* call, or to call the *System.exit* method which shuts down the JVM.

3. It should send a `StoppedBundleEvent` to all bundles when a bundle is being killed. A bundle can then take any action it desires: it can ignore the event or may release the references it had on the terminating bundle objects. If the bundle does not release the references, I-JVM may charge the objects to the bundle. The key point is that resources from the terminating bundle will not be released until all bundles release their references to them.

We also define one rule for writing bundles. A bundle should be prepared to catch any kind of exception when calling a method from another bundle, to ensure that it will continue its execution even in the presence of faulty bundles. Like any regular JVM, I-JVM uses exceptions to signal an error during the execution of an isolate. Such errors include regular errors such as erroneous class files or null pointer exceptions, but also I-JVM specific errors when the isolate is being killed.

4. Evaluation

In this section, we first motivate the need for fast inter-bundle calls in OSGi by benchmarking a simple OSGi application. Then, we evaluate I-JVM, in terms of performance and memory overhead, and in terms of robustness against attacks.

4.1. Inter-bundle Calls

To motivate the approach of OSGi and the need for fast inter-bundle calls, we evaluated the application demo provided by Felix [1]. The application is a paint program architectured with bundles. The drawing area, as well as the shapes that can be drawn are implemented as bundles. The user can add shapes dynamically or remove them through the OSGi bundle manager.

We measured the number of inter-bundle calls when I-JVM executes the application. Each time a shape is dragged onto the drawing area, an inter-bundle call happens between

Local call	Local RMI	Incommunicado	I-JVM
20μs	90ms	9ms	24μs

Table 1. Cost of 200 inter-bundle calls, depending on the communication model. The benchmarks were measured on a Pentium D 3.0GHz with 3GB of memory.

the drawing area and the shape. Dragging and moving the shape from upper-left to the bottom-right makes roughly two hundred inter-bundle calls. Table 1 shows the time for performing two hundred inter-bundle calls depending on the communication implementation. We evaluate four kinds of implementations: (i) local call, (ii) RMI call, which is the standard inter-application communication in Java, (iii) Incommunicado [27], the communication model of Isolates, and (iv) I-JVM. Since we do not have access to an Incommunicado implementation, we rely on the number they report, which is ten times faster than RMI [27]. I-JVM is an order of magnitude faster than other approaches for inter-isolate communication.

4.2. Performance Overhead of I-JVM

Since VMKit has been implemented from scratch for easing experimentation with virtual machines, it does not compete with industrial JVMs. We thus report our experiments in terms of relative performance and memory overheads of I-JVM compared to VMKit.

All experiments were done on a Pentium D 3.0GHz with 3GB of memory running Mandriva Linux 2.6.23.

To evaluate the overhead of isolation and resource accounting in I-JVM, we have run the following set of micro-benchmarks: intra-isolate and inter-isolate calls, object allocation and access to static variables. We measured the overhead by performing the same operation a million times. We also ran the SPEC JVM98 benchmark in an isolate to measure the overall runtime cost of I-JVM. Finally, we evaluated the memory overhead induced by running two legacy implementations of OSGi, Felix [1] and Equinox [2] on top of I-JVM.

Figure 1 shows the relative performance of I-JVM compared to VMKit for the micro-benchmarks. I-JVM adds two test instructions when executing an intra-isolate method call. For an inter-isolate call, it also updates the current isolate of the thread, thus adding four store operations. Overall, an intra-bundle call induces a 14% overhead and an inter-bundle call induces a 16% overhead.

We also benchmarked the performance of object allocation by repetitively allocating a *java.lang.Object* object. In VMKit and I-JVM, the size of such an object is 28 bytes.

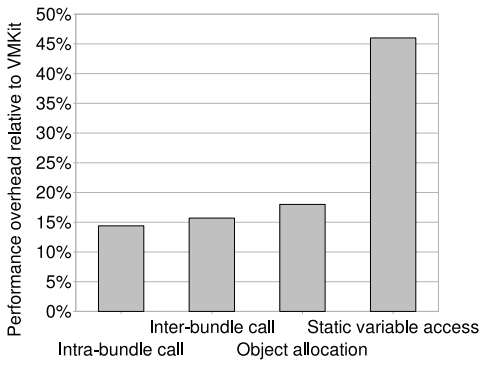

Figure 1. Performance of I-JVM for the micro-benchmarks, relative to VMKit.

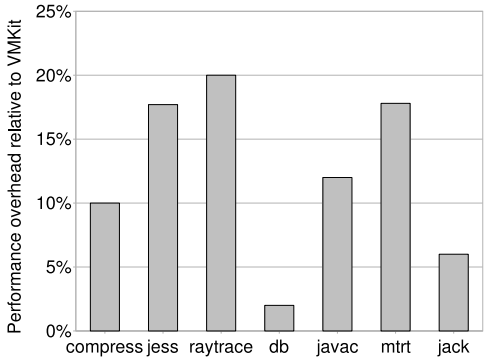

Figure 2. Overhead of I-JVM for the Spec JVM98 benchmarks, relative to VMKit.

The results show that there is an 18% overhead compared to VMKit, due to resource accounting, testing the memory limit when an isolate allocates an object and the intra-bundle cost of calling the *java.lang.Object* constructor. Finally, we measured static variable access. We removed all compilation optimizations in I-JVM to exhibit the cost of one access to the static variable. The benchmark shows that accessing a static variable gives a 46% overhead penalty on I-JVM. This is due to the task class mirror requiring two loads more than a simple load of a static variable plus an initialization check to verify that the static variable has been allocated. When I-JVM runs with all compilation optimizations, the overhead of accessing a static variable a million times is below 1% because the extra instructions execute only once.

We measured the execution time of SPEC JVM98 benchmarks [4] running within Isolate0, so as to evaluate the over-

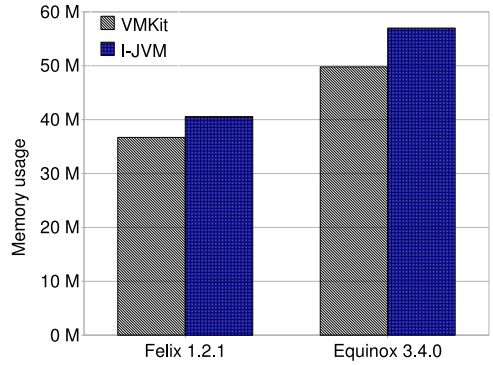

Figure 3. Memory consumption of I-JVM and VMKit on OSGi implementations.

head of I-JVM for standard Java programs. Figure 2 shows that the overhead of I-JVM is below 20% for all benchmarks. By comparison with MVM that reports a maximum overhead of 10% [9], I-JVM is less efficient. The main reason is that I-JVM induces an overhead for resource accounting and the test during intra-isolate calls. However, the cost of inter-isolate communication in MVM is an order of magnitude higher than a simple method call [27].

Finally, we measured the memory overhead induced by I-JVM when running an OSGi implementation. There are two places where I-JVM requires more memory than a standard JVM: (i) the array of task class mirrors for each class and (ii) a per-isolate set of strings and statistics information. Figure 3 shows the memory used when running the base configurations of Felix and Equinox. Felix runs the OSGi runtime and three management bundles (administration, shell, repository). Equinox runs the OSGi runtime and twenty-two management bundles. Overall, the memory overhead for both OSGi implementations is below 16%.

4.3. Robustness Evaluation

In Section 2, we presented a set of attacks to measure the robustness of JVM implementations running an OSGi platform. In this section, we compare the result of executing these attacks on I-JVM and on the Sun JVM, OpenJDK 6. I-JVM prevents the eight kinds of attacks by relying on an administrator. With the Sun JVM, the administrator loses control of the platform and has no possibility of stopping the execution of bundles, even if he detects the offending bundles. Thus, with the Sun JVM, the platform freezes or aborts under denial of service attacks.

A1 - Modification of a static variable. Bundle A defines an array as a static variable and works on the elements of the array. Bundle B finds the static variable either at compile-time, or at runtime through the Java reflection API. B sets the contents of the array to null.
`Result with Sun JVM`: Bundle A throws a *NullPointerException*.
`Result with I-JVM`: I-JVM isolates bundles so that they cannot access another bundle's static variables. The array is duplicated, therefore the modifications made by bundle B are local to bundle B. Bundle A continues to work on the array.

A2 - Synchronized method or synchronized call block. Bundle A calls a static *synchronize* method defined in a class that belongs to the bundle. Bundle A therefore synchronizes on the *java.lang.Class* object of the class. Another bundle B explicitly synchronizes on the object, holding the object forever.
`Result with Sun JVM`: Bundle A is blocked.
`Result with I-JVM`: I-JVM disallows the sharing of strings, *java.lang.Class* objects and static variables, therefore there is no interference between bundles that do not communicate with each other. Therefore, bundle A continues to run.

A3 - Memory exhaustion. A set of bundles are running on the platform. The OSGi runtime dynamically installs a new one that allocates many objects and stores them in an array, thus preventing the GC from deallocating the objects.
`Result with Sun JVM`: All bundles get a *OutOfMemoryError* when allocating a new object.
`Result with I-JVM`: I-JVM counts the memory used by each bundle. Based on this information, the administrator kills the offending bundle and all other bundles continue to run.

A4 - Standalone infinite loop. A set of bundles are running on the platform. The OSGi runtime dynamically installs a new one that runs an infinite loop.
`Result with Sun JVM`: The non-malicious bundles make progress slowly.
`Result with I-JVM`: I-JVM counts the CPU usage of each bundle. Based on this information, the administrator kills the offending bundle and all other bundles continue to run.

A5 - Excessive object creation. A set of bundles are running on the platform. The OSGi runtime dynamically installs a new one that allocates many objects but does not keep a reference to them, thus triggering the GC many times.
`Result with Sun JVM`: The JVM spends its time garbage collecting. The non-offending bundles make progess slowly.

`Result with I-JVM`: I-JVM counts the number of times a bundle runs a GC. Based on this information, the administrator kills the offending bundle and all other bundles continue to run.

A5 - Excessive thread creation. A set of bundles are running on the platform. The OSGi runtime dynamically installs a new one that endlessly creates threads.
`Result with Sun JVM`: All bundles get a *OutOfMemoryError* when allocating a new object or a new thread.
`Result with I-JVM`: I-JVM counts the number of threads a bundle creates. Based on this information, the administrator kills the offending bundle and all other bundles continue to run.

A7 - Hanging thread. Bundle A calls a method of bundle B and bundle B calls *Thread.sleep(0)*.
`Result with Sun JVM`: Execution never returns to bundle A.
`Result with I-JVM`: I-JVM inspects the current bundle of each thread and counts the number of sleeping threads in a bundle. Based on this information, the administrator kills the bundle that called *Thread.sleep*. If bundle A was prepared to catch the *StoppedIsolateException*, execution returns to A. Otherwise, the exception is caught at a lower stack level.

A8 - Lack of termination support. Bundle A calls bundle B and is waiting for an object value. Bundle B returns an object that points to the internal representation of bundle B in the OSGi platform. Bundle A stores the reference in one of its variable, thus the garbage collection can not delete bundle B until the reference is released. Bundle B then makes a denial of service attack. Therefore, the administrator tries to unload bundle B.
`Result with Sun JVM`: The OSGi platform is unable to unload the bundle, and the attack continues to run.
`Result with I-JVM`: All threads that execute code from bundle B throw an exception and execution never returns to bundle B.

These results show that I-JVM offers better isolation and resource control than a standard JVM. By isolating bundles and charging resource, I-JVM enables an administrator to find, locate and stop misbehaving bundles. While isolation and termination do not need any administration assistance, locating misbehaving bundles is still in the hands of the administrator. We plan as future work to write per bundle or per bundle group resource policies to automate denial of service detections.

5. Related Work

Our approach combines ideas from different research areas. In this section, we report how our work relates to (i) operating system structures for resource management and communications and (ii) resource accounting and isolation in JVMs.

5.1. Operating Systems

The OSGi architecture resembles that of a micro-kernel operating systems, where multiple modules run in their own protection domains and communicate by means of Inter-Process Communications (IPC) primitives. However, protection domains in micro-kernels are implemented as separate virtual address spaces, and must therefore cope with data copying during IPCs. While a number of projects achieve good performance in their IPCs implementation [7, 11, 13, 25], communications are still an order of magnitude less efficient than simple procedure calls.

Micro-kernels such as JX [17] or Singularity [21] use safe languages to run modules in a single address space. These systems enforce isolation by implementing communications through capability-like objects [23] or shared heaps [12]. Both cases break compatibility with existing OSGi applications.

The Scout operating system [33], Rialto [22], and resource containers [6] introduce new approaches for resource accounting in operating systems. Threads in these systems are not bound to a protection domain but migrate between protection domains while charging resources to a single resource management entity. These systems differentiate users and protection domains. In OSGi a protection domain (i.e. a bundle) *is* a user.

5.2. Isolation in Java Virtual Machines

The standard isolation mechanism in current JVMs is based on class loaders [24]. Class loaders provide name space isolation, i.e. there is no collision between classes with the same name but loaded by two different class loaders. Class loaders have weak isolation guarantees, as they still share static variables, interned strings and *java.lang.Class* objects. OSGi uses class loaders as well as Java permissions to enhance security and limit the rights of a bundle. A method that performs a privileged operation such as `System.exit` asks the JVM the protection level of its caller. The JVM inspects the stack trace to find which class loader loaded the class of the caller. A class loader can be given a security policy, which indicates what kind of privileged operations the classes loaded by this class loader can perform. Inspection of a stack is an expensive operation, which would dramatically reduce the performance of

our approach if it were performed on each access to a static variable, string or `java.lang.Class`.

There are many projects that attempt to add software-based processes to the JVM [5, 9, 10, 19]. While these systems provide isolation, termination and resource management, they change the programming model for communications and therefore break compatibility with existing OSGi applications.

The Secure JVM [35] is closer to our work. It is an implementation of a JVM on top of an extensible operating system. Modules run in their own address space. Programmers write inter-module communications like simple procedure calls. However, the operating system performs page remapping and object relocation during an inter-module call, which incurs performance penalties as compared to a simple procedure call. Moreover, the Secure JVM can not run on commodity systems.

Rudys *et al.* define *soft termination* for language-based systems [32]. Each Java class is annotated with a boolean value, indicating whether the class has to be stopped. The value is polled regularly by methods defined in the class. Because the polling code is inserted on backwards branches and after method calls, termination is sure to happen in a finite amount of time. They have implemented soft termination with bytecode rewriting. I-JVM implements *hard termination* in the core JVM: instead of inserting polling code, the compiler unwinder and machine code rewriting ensures that the code of the class will never be executed anymore and stops instantaneously. Overall, I-JVM is a refinement of three research projects: isolates in Java [9], soft termination [32] and garbage collector memory accounting [30] in language based systems.

Finally, in the context of OSGi, Service Coroner [14] is an administration tool to find stalled bundles. A stalled bundle is a terminating bundle that can not be unloaded because other bundles maintain unused references to it. Service Coroner does not modify the JVM or OSGi, but uses aspect oriented programming to instrument the creation of bundles. While adressing two different problems, both Service Coroner and our work attempt to give an administrator better control of the OSGi platform.

6. Conclusion and Future Work

We have described the design and implementation of I-JVM, a Java Virtual Machine extended with component isolation and termination in OSGi. I-JVM enables lightweight isolation of OSGi bundles while still providing fast communication through thread migration across bundles and direct sharing of objects. The isolate architecture of I-JVM allows a per-bundle resource accounting that an administrator can use to terminate a misbehaving bundle. Even though isolation and resource accounting has a small overhead compared to a regular JVM, our evaluation shows that I-JVM is able to inform an administrator of denial of service attacks.

In this paper, we considered resource accounting as an assistance for an administrator to locate possible resource problems and kill the bundles he thinks are malicious. However the administrator must be aware of what I-JVM measures: as Price *et al.* explain in [30], the memory usage statistics for a module are between a maximum (i.e. all shared objects used by the module are charged to the module) and a minimum (i.e. none of the shared objects are charged to the module). Also, bundles still have the possibility to create a resource Trojan horse problem [20]. Resource policies and bundles must be written with these problems in mind. Similarly, our CPU accounting mechanism is not as precise as a system where threads do not migrate. Precise techniques such as the one described in [8] incur a significant time overhead, reported to be between 67% and 279%, that we can not afford.

We plan as future work to implement a policy engine that will be able to handle such problems. Combined with the isolation and termination features presented in this paper, this will make I-JVM a more robust system to run OSGi applications.

Availability

I-JVM is publicly available via an open-source license at the URL:

`http://vmkit.llvm.org`

Aknowledgements

We would like to thank the anonymous reviewers for their extensive and helpful feedback on the paper. We would also like to thank Julia Lawall for her numerous insightful comments on the paper and Didier Donsez for the initial discussions on isolation in OSGi.

References

[1] Apache felix. http://felix.apache.org/site/index.html.

[2] Equinox. http://www.eclipse.org/equinox.

[3] Jonas J2EE Server. http://jonas.objectweb.org.

[4] SPECjvm98. http://www.spec.org/jvm98/.

[5] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 333–346, San Diego, USA, October 2000. USENIX Association.

[6] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*,

pages 45–58, New Orleans, USA, February 1999. USENIX Association.

[7] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *Transactions on Computer Systems*, 8(1):37–55, 1990.

[8] W. Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.

[9] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 125–138, Tampa Bay, USA, October 2001. ACM.

[10] G. Czajkowski and T. Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 21–35, Vancouver, Canada, October 1998. ACM.

[11] R. Draves, B. Bershad, R. Rashid, and R. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, USA, October 1991. ACM.

[12] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006. ACM.

[13] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–114, San Francisco, USA, January 1994. USENIX Association.

[14] K. Gama and D. Donsez. Using the service coroner tool for diagnosing stale references in the OSGi platform. In *Proceedings of the Middleware Conference Companion*, pages 58–61, Leuven, Belgium, December 2008. ACM.

[15] N. Geoffray. The VMKit project: JVM and .Net runtimes for LLVM. In *Proceedings of the LLVM Developer Meeting*, August 2008. http://vmkit.llvm.org.

[16] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: building a JVM with third party software. In *Proceedings of the International Symposium on Principles and Practice of Programming In Java*, pages 73–82, Modena, Italy, September 2008. ACM.

[17] M. Golm, M. Felsera, C. Wawersich, and J. Kleinoeder. The JX operating system. In *Proceedings of the Usenix Annual Technical Conference*, pages 45–58, Monterey, USA, June 2002. USENIX Association.

[18] G. Hamilton and P. Kougiouris. The Spring Nucleus: a microkernel for objects. In *Proceedings of the USENIX Summer Technical Conference*, pages 1–15, Cincinnati, USA, June 1993. USENIX Association.

[19] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX Annual Technical Conference*, pages 259–270, New Orleans, USA, June 1998. USENIX Association.

[20] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 391–401, Boston, USA, December 2002. ACM.

[21] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the Eurosys Conference*, pages 341–354, Lisboa, Portugal, April 2007. ACM.

[22] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Hot Topics in Operating Systems Workshop*, pages 12–17, Washington, USA, May 1995. IEEE Computer Society.

[23] H. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, USA, 1984.

[24] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 36–44, Vancouver, Canada, October 1998. ACM.

[25] J. Liedtke, K. Elphinstone, S. Schiinberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance. In *Proceedings of the Hot Topics in Operating Systems Workshop*, pages 28–31, Washington, USA, May 1997. IEEE Computer Society.

[26] OSGi Alliance. OSGi service platform, core specification release 4.1. Draft, 05 2007.

[27] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynès. Incommunicado: efficient communication for isolates. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 262–274, Seattle, USA, November 2002. ACM.

[28] P. Parrend and S. Frénot. Classification of component vulnerabilities in Java service oriented programming platforms. In *Proceedings of the Component-Based Software Engineering Symposium*, pages 80–96, Karlsruhe, Germany, October 2008. Springer-Verlag.

[29] P. Parrend and S. Frénot. Security benchmarks of OSGi platforms: toward hardened OSGi. *Software: Practice and Experience*, 39(5):471–499, April 2009.

[30] D. Price, A. Rudys, and D. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–274, Oakland, USA, May 2003. IEEE Computer Society.

[31] Y. Royon and S. Frénot. Multiservice home gateways: business model, execution environment, management infrastructure. *IEEE Communications Magazine*, 45(10):122–128, October 2007.

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

[33] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 59–72, New Orleans, USA, February 1999. USENIX Association.

[34] E. Tilevich and Y. Smaragdakis. J-Orchestra: automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 178–204, Malaga, Spain, June 2002. Springer-Verlag.

[35] L. van Doorn. A secure Java virtual machine. In *Proceedings of the USENIX Security Symposium*, pages 19–34, Denver, USA, August 2000. USENIX Association.