Incinerator – Eliminating Stale References in Dynamic OSGi Applications

Koutheir Attouchi*, Gaël Thomas†, Gilles Muller‡, Julia Lawall‡ and André Bottaro*
*Orange Labs, Email: firstname.lastname@orange.com
†Telecom SudParis, Email: gael.thomas@telecom-sudparis.eu
‡Inria/LIP6/UPMC/Sorbonne University, Email: firstname.lastname@lip6.fr

***Abstract*—Java class loaders are commonly used in application servers to load, unload and update a set of classes as a unit. However, unloading or updating a class loader can introduce stale references to the objects of the outdated class loader. A stale reference leads to a memory leak and, for an update, to an inconsistency between the outdated classes and their replacements. To detect and eliminate stale references, we propose Incinerator, a Java virtual machine extension that introduces the notion of an outdated class loader. Incinerator detects stale references and sets them to null during a garbage collection cycle. We evaluate Incinerator in the context of the OSGi framework and show that Incinerator correctly detects and eliminates stale references, including a bug in Knopflerfish. We also evaluate the performance of Incinerator with the DaCapo benchmark on VMKit and show that Incinerator has an overhead of at most 3.3%.**

I. Introduction

A Java class loader is a container for a set of Java classes that makes it possible to load and unload the set of classes as a unit. Such a facility is convenient in the implementation of a middleware, that can use class loaders to organize the dynamic loading, unloading and updating of applications. Unloading or updating classes, however, introduces the possibility of stale references, *i.e.*, references to objects that instantiate the outdated classes. A stale reference amounts to a memory leak because it blocks garbage collection, not only of the referenced object but also of the outdated class loader, which stays reachable through the object's class. In this case, all classes loaded by the class loader remain reachable as well. A stale reference may also enable the execution of outdated code, which may render the system inconsistent.

Avoiding stale references is challenging. Since there is no support in the JVM to identify outdated class loaders, a stale reference is a Java reference like any other. Its only specificity is that it refers to an object that instantiates a class loaded by a class loader that is considered to be outdated. In order to avoid stale references, the developer thus has to manually track all inter-application references in the source code, and carefully insert code to release them when a class loader becomes outdated. This manual checking is difficult and error-prone. Moreover, because an application can be uninstalled or updated by a user, and this is not necessarily apparent in the application source code, the validity of inter-application references cannot be determined by static analysis. Thus, run-time analysis is required.

The OSGi framework [1] illustrates this problem of stale references. OSGi is a middleware that targets smart homes [2], a domain in which application updates are frequently necessary. OSGi loads each application, referred to as a *bundle*, in a separate class loader, and provides the ability to update a bundle by loading a new version of its code in a new class loader. In order to avoid costly remote procedure calls, bundles directly exchange Java references. These inter-bundle references can become stale when bundles are unloaded or updated.

In this paper, we propose to address the issue of stale references at the garbage collector level. Specifically, we propose a garbage collector extension, called *Incinerator*, which relies on the addition of an *outdated* flag to class loaders to identify, and then eliminate stale references. The flag is set by a middleware when the code of the application loaded by the class loader becomes outdated. Then, during the next garbage collection cycle, Incinerator checks each reference to know whether it references an object that instantiates a class loaded by an outdated class loader. If this is the case, the reference is identified as stale and Incinerator sets it to `null`. As a consequence, no stale object remains reachable and the associated memory is reclaimed by the garbage collector at the end of the collection cycle.

Incinerator induces low overhead since the garbage collection phase already traverses all live objects, and checking the staleness of a reference requires few operations. Incinerator is also independent of the specific garbage collection algorithm as it only requires the modification of the function that scans the objects. Regardless of the garbage collection strategy used, Incinerator guarantees that after a collection, outdated code cannot be used anymore. In the case of a stop-the-world collector, by explicitly triggering a collection just after the uninstallation of an application, the middleware can thus ensure that inconsistencies and memory leaks will never appear. In the case of a concurrent collector, which lets the application run during a collection, Incinerator provides a best effort behavior: it ensures that the outdated code cannot be used after the collection cycle, but the applications can still use this outdated code during the collection.

Incinerator does, however, change the behavior of the Java virtual machine, because it nullifies references that are found to be stale. Several compatibility issues could arise from this change. First, Incinerator does not change the behavior of a correctly written application that releases its stale references. Likewise, it does not change the behavior of a buggy application that retains stale references without using them. For an application that uses stale references, we distinguish two cases: (i) the stale reference is only accessed as part of a

cleanup operation, *i.e.*, a finalize method; Incinerator tries to execute this cleanup operation before nullifying in order to avoid other kinds of leaks, (ii) the stale reference is used elsewhere; in this case, the application that uses the stale reference contains a *bug* since using the stale object could lead to conflicting operations. Since the reference has been nullified, such a buggy application receives a `NullPointerException` which helps the developer track down the bug, by making it visible.[1] Thus, we reiterate that Incinerator only has an impact on the semantics of a Java application when the application contains stale reference bugs.

We have prototyped Incinerator in J3, a Java virtual machine based on VMKit [3]. Our implementation of Incinerator modifies the MMTk "Mark-Sweep" garbage collector [4] included in VMKit and adds 150 lines of code to J3. We have also used Incinerator together with Knopflerfish 3.5.0 [5], one of the main OSGi implementations, in order to solve the problem of stale references in the OSGi context. Preparing Knopflerfish for Incinerator required modifying only 6 lines of code, mainly to mark a class loader as outdated when the bundle becomes outdated.

We have evaluated the impact of Incinerator both in terms of the increase in robustness and the performance penalty with the following experiments:

- We have designed a test suite of seven micro-benchmarks that cover the possible sources of memory leaks caused by stale references. Incinerator identifies the stale references in all cases and prevents memory leaks, while memory leaks occur with a standard JVM.
- We have evaluated the overhead incurred by Incinerator using the DaCapo 2006 benchmark suite [6], which covers a wide range of application behaviors. The average overhead remains below 1.2% on a high-end desktop machine and below 3.3% on a smart-home PC, which shows that the overhead of Incinerator is reasonable in J3.
- We have used Incinerator to find a bug in a legacy OSGi bundle, the widely used `HTTP-Server` bundle of Knopflerfish. We have sent a bug report and a patch, which have been accepted by the Knopflerfish maintainers.

The rest of the paper is organized as follows. Section II describes Java class loaders and the problem of memory leaks. Section III presents the design and implementation of Incinerator. Section IV presents a use case of Incinerator in the context of OSGi. Section V evaluates the benefits of Incinerator. Section VI presents an overview of related work, and Section VII concludes.

II. Java class loaders

In order to understand the problem of stale references, we first briefly describe Java class loaders, and then describe how class loaders are used to load, unload and update applications.

A. Class loader

A Java virtual machine [7] (JVM) executes bytecode instructions that belong to Java classes. A class is loaded on-demand using a *Java class loader*, which is a Java object whose purpose is to return a Java class given a fully qualified class name. As shown in Figure 1, each Java object holds a reference to its class, and each class holds a reference to its class loader. Additionally, a class loader holds references to all the classes it has loaded.

O_1 is an object of class C_1. O_2 and O_3 are objects of class C_2. C_1 and C_2 were loaded by L.

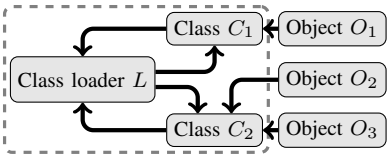

Fig. 1: References between class loaders, classes, and objects.

Like any other object, a class occupies memory. In the case of a class, memory is required to store the bytecode of the methods, their generated machine code, and the metadata of the class, e.g., the class's fully qualified name, hierarchy, and fields. The garbage collector [8] will free this memory when the class becomes unreachable. As a class references its class loader, a class loader L that has loaded classes $C_1, ..., C_N$ can only be collected when the graph of objects $\{L, C_1, ..., C_N\}$ becomes unreachable, which implies that, for each C_i, all objects that instantiate C_i are unreachable.

B. Using class loaders in a middleware

A number of Java middlewares, such as those defined by the OSGi [1] or the JEE [9] specifications, support the execution of multiple Java applications within a single Java Virtual Machine. These middlewares use class loaders to load, unload or update applications. In order to install an application, a middleware creates a class loader, and stores it in the middleware's metadata about the application. Subsequently, the middleware uses the class loader to load the code of the application into the Java virtual machine. To uninstall the application, the middleware simply releases the reference to the metadata about the application. To update the application, the middleware replaces the class loader associated with the application by a new class loader, which is used to load the new version of the application code.

We say that a class loader that was used to load either the code of a now uninstalled application or the code of an old version of an application is *outdated*. We also say that a reference is *stale* if it references an outdated class loader, a class loaded by an outdated class loader, or an object that instantiates one of these classes. Accordingly, we say that an object is stale if it is referenced by a stale reference.

Any stale reference prevents the garbage collector from reclaiming the memory of the outdated class loader, and thus all of the classes loaded by the outdated class loader. Thus, even a stale reference to a small object can have a dramatic impact on memory usage if many classes were loaded by the outdated class loader.

III. Incinerator Design and Implementation

The goal of Incinerator is to eliminate stale references by setting them to `null`. For this, Incinerator scans all live references in the Java memory space to determine whether

[1] It would be helpful to throw an exception that is specific to stale reference access, but this is not yet supported by our implementation.

any are stale. We have chosen to design Incinerator as an extension of the garbage collector, which already performs such a scan. This approach makes it possible to access private instance variables and method-local variables that would not be accessible to a Java application. Furthermore, this approach has a low performance penalty since in most cases the overhead is limited to the cost of checking the staleness of each reference during the heap traversal already performed by the garbage collector.

In this section, we first present how stale references are identified. Then, we discuss more specifically the other JVM features impacted by the nullification of stale references: synchronization and finalization. Finally, we introduce the most important implementation details.

A. Outdated class loaders and stale references

In the design of our approach, it is the middleware that causes references to become stale, by uninstalling or updating the classes of an application, but it is the JVM, specifically the garbage collector, that nullifies these stale references. To allow the middleware to communicate the current state of a class loader to the JVM, we introduce an "outdated" flag in the JVM's internal representation of a class loader and a native method for updating the flag's value. This flag is initially cleared when the class loader is created. The middleware must use the native method to set the flag when the associated application is uninstalled or updated.

During the graph traversal performed by the garbage collector, Incinerator inspects the outdated flag of the class loader of the class of each referenced object. If the flag is set, Incinerator considers that the reference is stale.

B. Stale references and synchronization

In Java, each object has an attached monitor, whose purpose is to provide thread synchronization. The list of the threads blocked while waiting for the monitor is stored in the monitor structure, which can only be retrieved through the object. Therefore, if a thread is holding the monitor at the time when the associated object becomes stale and Incinerator nullifies all references to the object, the holding thread will become unable to reach the monitor structure to unblock any blocked threads. These threads would remain blocked, leaking both their thread structures and any referenced objects.

Incinerator addresses this issue by waking up the blocked threads in a cascaded way. To allow each wakened thread to detect that the monitor is stale, we add a *stale* flag to the monitor structure. During a collection, when Incinerator finds a stale object with an associated monitor, it nullifies the stale reference, marks the monitor as stale, and then wakes up the first blocked thread. The thread wakes up at the point where it blocked, in the monitor acquiring function. We modify this function so that when a thread wakes up, it checks the stale flag. If the flag indicates that the monitor is stale, the monitor acquiring function wakes up the next blocked thread and throws a `NullPointerException` to report to the current thread that the object is stale. Note that there is no special treatment of the thread that is actually holding the monitor. It continues executing the critical section normally, and will receive a `NullPointerException` either when it tries to exit the critical section, or beforehand, if it tries to access the stale object.

Most modern JVMs allocate a monitor structure that is separate from the object and is managed explicitly [10]. This monitor structure is normally freed during a collection when the memory of its associated object is reclaimed. With Incinerator, when the memory of a stale object is reclaimed, its monitor structure has to survive the collection, if threads are blocked on it, so that each thread can wake up the next one. We thus further modify the monitor acquiring function so that it frees the monitor structure when it detects that there are no remaining blocked threads.

C. Stale references and finalization

In Java, a `finalize()` method defines clean-up code that is executed exactly once by the garbage collector before the memory associated with the object is reclaimed. If a `finalize()` method accesses a stale reference that was nullified by Incinerator, then it encounters a `NullPointerException` and is not able to complete the clean up. This may lead to other kinds of resource leaks, such as never closing a file descriptor or a network connection. As Incinerator is designed with the goal of preventing resource leaks, we have decided to treat finalizable objects specially, to give a `finalize` method that uses stale references a chance to execute correctly. There are two cases: a finalizable object, i.e., an object with a `finalize()` method, is either not alive at the end of a collection cycle, *i.e.,* it is either unreachable or stale, or it is.

If the finalizable object is not alive at the end of a collection, we defer the nullification of the stale objects reachable from the finalizable object by one collection cycle. As the finalizable object is not alive, the garbage collector immediately invokes its `finalize()` method after the collection cycle. By deferring the nullification by one cycle, Incinerator enables the `finalize()` method to execute correctly. This case is common and we have encountered it during our tests of Incinerator in the context of OSGi. Indeed, when an application is uninstalled or updated, all its finalizable objects become stale, thus not alive, and their `finalize()` methods are immediately invoked after the first collection cycle.

If the object is alive at the end of a collection, we do not defer the nullification of its stale references. Indeed, in this case, it is not known when and if the object will become unreachable, and thus when and if its `finalize()` method will be executed. Deferring the nullification of stale references until after the `finalize()` method is executed may indefinitely prevent Incinerator from performing nullification, and thus from eliminating the memory leaks caused by the stale references. In practice, we have not encountered reachable finalizable objects that use inter-application references in their `finalize()` methods in our tests. We thus think that nullifying the stale references in this case will only rarely prevent the execution of a `finalize()` method.

In order to distinguish the two cases, Incinerator has to identify the stale or unreachable finalizable objects. However, they are not known at the beginning of a collection cycle: stale or unreachable objects are only identified by Incinerator at the end of the collection cycle, when non-stale live objects

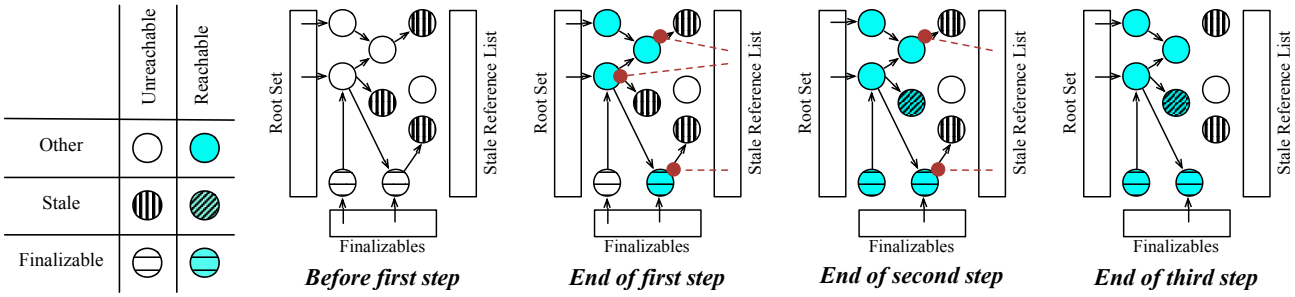

Fig. 2: Incinerator algorithm.

are identified. For this reason, Incinerator identifies the stale references that it will nullify in three steps (see Figure 2):

- During the first step, Incinerator identifies all the reachable stale references and adds their locations to a stale reference list.
- During the second step, Incinerator removes from the stale reference list the locations of the stale references reachable by the stale or unreachable finalizable objects.
- During the third step, Incinerator nullifies all the remaining stale references of the stale reference list.

Algorithm 1: Incinerator nullification algorithm

```
   Data:
     staleList: List of Reference *
     scanObject: Function
   /* First step: find the stale references                    */
1  Function scanObjectDuringMarking(obj: Object)
2      foreach Reference * pr ∈ obj do
3          if (*pr ≠ null) && (*pr).class.classloader.isOutdated
           then
4              staleList.add(pr);
5      originalScanObject(obj);
   /* Second step: remove the stale references reachable
      by an unreachable finalizable object                     */
6  Function scanObjectDuringFinalize(obj: Object)
7      foreach Reference * pr ∈ obj do
8          if (*pr ≠ null) && (*pr).class.classloader.isOutdated
           then
9              staleList.remove(pr);
10     originalScanObject(obj);
   /* Last step: nullifies the remaining references            */
11 Function nullify()
12     foreach Reference * pr ∈ staleList do
           /* If object locked, then starts the cascade        */
13         if (*pr).monitor.isLocked then
               /* Mark the monitor as outdated                 */
14             (*pr).monitor.isOutdated := true;
               /* Start the unlock cascade                     */
15             unlock((*pr).monitor);
           /* Nullify the stale reference                      */
16         *pr := null;
```

These three steps are presented in more detail in Algorithm 1. The first step (lines 1 to 5 of Algorithm 1) adds the locations of all of the reachable stale references to the stale reference list during the mark phase of the collector. The mark phase of the collector traverses the object graph from the root set and marks objects as *reachable*. The first step of Incinerator is thus implemented by modifying the scan function to use the function `scanObjectDuringMarking` to record any references to stale objects in the stale reference list, resulting in the dashed red lines in the diagram "End of first step" in Figure 2, before performing the normal scan of the object.

During the second step (lines 6 to 10 of Algorithm 1), Incinerator removes from the stale reference list those references that are reachable by the stale or unreachable finalizable objects. This step is performed during the final phase of the garbage collector, which handles finalizable objects. A Java garbage collector cannot reclaim the memory of an unreachable finalizable object and of its reachable sub-graph directly after a collection because the object can become reachable again during the execution of its `finalize()` method, e.g., by storing a reference to itself in a reachable object or in a static variable. For this reason, during the final phase of the collection, a Java garbage collector marks each unreachable finalizable object and its reachable sub-graph as reachable (hence the blue color for the balls on the lower left in Figure 2 in the diagram "End of second step"). To perform this, the collector simply starts a second traversal of the object graph, in which it considers the objects that are both finalizable and unreachable as the roots of the graph. We have modified the scan function `scanObjectDuringFinalize` that is used during this traversal to update the stale reference list and remove the stale reachable references.

The third step (lines 11 to 16 of Algorithm 1) nullifies the locations of the stale references contained in the stale reference list (red dashed lines in the diagram "End of second step" in Figure 2). This step is performed at the end of the collection cycle, just before reclaiming the memory of the dead objects. Incinerator takes care of starting the release cascade in case of an acquired monitor between the lines 13 and 15, and nullifies the locations of the references at line 16.

The algorithm presented so far is designed to protect against programmer bugs, but not a deliberate attack. As such, it is possible to construct a malicious application that prevents a stale object from ever being reclaimed by the garbage collector. As Incinerator defers the nullification of a stale reference reachable from an unreachable finalizable object, such a stale reference survives a collection cycle. A `finalize()` method of the malicious application could then force the stale reference to survive one more collection cycle by creating a new unreachable finalizable object that references the stale object. By repeating the same pattern, the malicious application could indefinitely defer the nullification of a stale

reference to the stale object. To protect against such attacks, Incinerator adds a second flag to the class loader to indicate whether its stale objects have already survived a collection cycle. Incinerator only defers the nullification of the stale references to the stale objects during the first collection and not during the second one.

D. Implementation details

The Incinerator prototype is based on the J3/VMKit [3] experimental JVM. Implementing Incinerator requires roughly 150 lines of C++ code. This suggests that Incinerator should be relatively easy to port to a different JVM. In the remainder of the section, we summarize the changes made in the JVM and discuss an incompatibility with the JVM specification [7] caused by the nullification of references.

a) JVM changes: We have seen that within the JVM, Incinerator requires changes in the garbage collector, in the support for class loading, and in the monitor implementation. The garbage collector is modified as presented in the previous section. The monitor implementation is also modified to support the algorithm described in Section III-B.

As an optimization, Incinerator is only enabled when stale references potentially exist. For this purpose, we have added a global flag that is set when a class loader is marked as outdated, since a new stale reference can only appear under this condition. Incinerator clears the flag at the end of a collection if all the stale references are eliminated, *e.g.*, if Incinerator did not defer the nullification of a stale reference.

b) Incompatibility with the JVM specification: The Java language specification [11] states that if an exception is raised while holding a monitor, the Java compiler has to generate an exception handler that releases the monitor. This introduces the possibility of an infinite loop when using Incinerator, because Incinerator can nullify the reference to the object containing the monitor, causing the monitor release operation itself to raise a `NullPointerException`, which again triggers the execution of the same exception handler. To avoid the possibility of an infinite loop, we have modified the Just-In-Time compiler so that the code generated for a lock release simply leaves the block silently when the monitor is null, rather than raising a `NullPointerException` exception.

Except when Incinerator nullifies the references to the monitor, our workaround does not change the behavior of programs compiled from Java source code. Indeed, the Java compiler ensures that the reference given to a synchronized block is never touched between an acquire and a release instruction.[2] As a consequence, the argument of the release instruction can never be `null` as, if it were, the preceding acquire instruction would have thrown the `NullPointerException`. However, our workaround is incompatible with the Java specification and could change the behavior of programs written directly in Java byte code or generated in an ad-hoc fashion.

IV. Using Incinerator in OSGi

This section presents a use case of Incinerator in the context of OSGi [1]. An application in OSGi is called a *bundle*. It is represented by a Java object, which contains an identifier, called the *bundle ID*, OSGi-specific metadata, including a symbolic name, a vendor and a version, and a Java class loader, which is used to load the classes of the bundle.

[2]The reference used by the synchronized block is cached in a local variable that is not visible to the programmer.

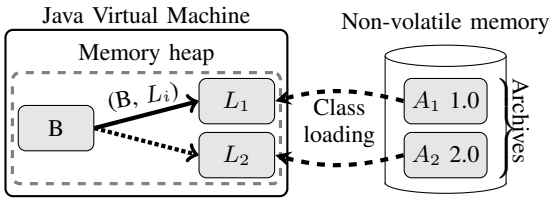

Fig. 3: OSGi bundle update. Each archive A_i is loaded using a separate class loader L_i. Updating bundle B from version 1.0 to version 2.0 implies creating class loader L_2, then loading A_2 using L_2, and then associating B with L_2 instead of L_1.

When a bundle is *uninstalled*, OSGi broadcasts an event to other bundles asking them to release their references to the objects that instantiate the classes of the outdated class loader. Then, OSGi removes all references to the bundle from the framework. The class loader is subsequently considered to be outdated. A bundle that continues, after having handled the event, to reference or use objects that instantiate classes of the outdated class loader is considered as invalid by the OSGi specification [1].

When a bundle is *updated*, OSGi broadcasts an event and removes the reference to the class loader from the bundle, making the class loader outdated. Then, OSGi creates a new class loader to load the new classes of the bundle, and stores this class loader in the bundle. The update process is illustrated in Figure 3.

We have modified the Knopflerfish OSGi framework version 3.5.0 in order to use Incinerator. This required modifying the functions that uninstall and update a bundle to (i) mark a class loader as outdated at the end of its update or an uninstall, and (ii) trigger a collection that ensures that all the stale references are eliminated. In total, 6 lines of code are modified in the framework.

V. Evaluation

In this section, we evaluate Incinerator in terms of both the increase in robustness and the performance penalty. We first evaluate Incinerator itself and then evaluate Incinerator in the context of OSGi.

All of our evaluations are performed on two computers, both of which run Debian 6.0 with a Linux 2.6.32-i386 kernel: (i) a low-end computer with a 927 Mhz Intel Pentium III processor, 248 megabytes of RAM and 4 gigabytes of swap space, which has performance comparable to that of a typical system in a smart-home environment, (ii) a high-end computer having two 2.66 Ghz Intel Xeon X5650 6 core processors, 12 gigabytes of RAM and no swap space. In the former case, the swap space is necessary, because J3 requires at least 1 gigabyte of address space to run the DaCapo benchmark suite.

J3 has been measured to be between 1.2 and 3 times slower than JikesRVM [3]. We use J3 in our evaluation because it is easy to extend with new functionalities.

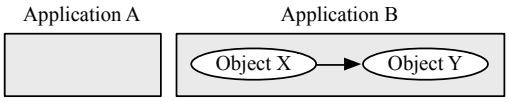

Fig. 4: Application configuration used in micro-benchmarks scenarios.

A. Robustness and Performance of Incinerator

We evaluate the robustness and performance of Incinerator from a number of perspectives. First, we present a test suite of seven micro-benchmarks that we have designed to cover the possible sources of stale references. This test suite covers different scenarios of memory leaks and is used to verify that Incinerator is able to identify and eliminate them. Then, we show the impact of repeated memory leaks caused by stale references. Finally, we study the performance overhead of Incinerator by running the DaCapo 2006 benchmark, which is representative of standard Java applications.

1) Stale reference micro-benchmarks: We have developed seven microbenchmarks covering the seven scenarios in which stale references can arise. To describe these scenarios, we use the application configuration shown in Figure 4. This configuration involves two applications, A and B, and two objects, X and Y, created by B. Stale references will appear in the faulty application A, which retains a stale reference to X or Y when B is uninstalled. The scenarios of these micro-benchmarks are classified by three criteria: *scope*, *synchronization*, and *finalization*, as follows:

a) Scope: Scope refers to the location of the reference, i.e., in a local variable, in a global variable, in an object field, or in a thread-local variable. Different types of locations are scanned in different ways and orders by the garbage collector. We have designed four scenarios to check that Incinerator finds stale references in all possible kinds of locations. A reference to X is retained by the application A, respectively, in a local variable (**Scenario 1**), in a global variable (**Scenario 2**), in an object field (**Scenario 3**), and in a thread-local variable (**Scenario 4**). In all scenarios, the reference is made stale by uninstalling B.

b) Synchronization: Synchronization refers to whether the referenced object's monitor is used to synchronize threads. As stated in Section III-B, if threads are blocked while waiting to obtain the monitor of a stale object, then Incinerator wakes up the blocked threads and only releases the memory of the object monitor when the last thread has woken up. **Scenario 5** illustrates this situation by having two threads created by A each synchronize on a local variable that references X. The two references to X are turned stale by uninstalling B.

c) Finalization: Incinerator tries to allow `finalize()` methods to run without introducing null pointer exceptions by not immediately nullifying stale references reachable by a finalizable object. To check the possible cases, we have defined two scenarios. In **Scenario 6**, the application A retains a reference to Y and Y is finalizable with a `finalize()` method that does not access memory. The object Y is made stale by uninstalling B. In **Scenario 7**, the application A retains a reference to X and X is finalizable with a `finalize()` method that uses the object Y. The objects X and Y are made stale by uninstalling B.

We have executed our scenarios using J3 and Incinerator. J3 suffers from memory leaks caused by stale references that go undetected in all scenarios. Incinerator detects and eliminates all the stale references. In particular, Incinerator handles correctly the case of the stale references used for synchronization (**Scenario 5**): the blocked thread is woken up by Incinerator when the reference to the stale object used for synchronization is nullified. Both threads receive a `NullPointerException`: the holder of the lock when it tries to re-acquire the lock and the blocked thread in the lock-acquiring method. Incinerator also handles correctly the two cases of stale references used by a finalizable object (**Scenario 6** and **Scenario 7**), successfully executing the `finalize()` method as expected. After the execution of the `finalize()` methods, the memory of the stale objects is reclaimed on the next garbage collection cycle.

2) Memory leaks: To investigate the memory leaks caused by stale references in quantitative terms, we repeat **Scenario 1** (Section V-A1) multiple times, to create a large number of stale references. In this experiment, the application A defines a set, in which it stores a reference to X. Each time we update B, A adds a reference to the new X provided by the new B in the same set. The application B is a small application with 150 lines of Java code distributed over 3 classes.

For the baseline, J3, each update of the application B makes one more reference stale and costs the JVM 892 kilobytes of leaked memory. This is due to the need to keep all the application class information, static objects and generated machine code. After only 230 updates of the application B, the amount of leaked memory reaches 200 megabytes and J3 starts raising `OutOfMemoryException` exceptions on our low-end test machine. Incinerator, however, continues to use the same amount of memory. These results show that even stale references into a small application may leak a significant amount of memory.

3) Performance benchmarks: In order to measure the performance impact of Incinerator on real Java applications, we ran the DaCapo 2006 benchmark suite [6] on J3 and on Incinerator. The DaCapo 2006 benchmark suite includes nine real Java applications[3] stressing many JVM subsystems.

This evaluation assesses the minimal impact of Incinerator, when there are no application updates, because the DaCapo 2006 applications do not cause class loaders to become outdated. In this case, the global flag (see Section III-D) is always cleared. As compared to the baseline, J3, adding Incinerator introduces overhead for each monitor acquisition in order to check whether the monitor is stale. Incinerator also replaces a direct call to the `scan` function by an indirect call, in order to switch from the original `scan` function to the Incinerator `scan` function when outdated class loaders exist. The cost of the indirection is paid for each scanned object.

We performed 20 runs of all DaCapo benchmark applications on J3 and on Incinerator, on the low-end and the high-end computers described at the beginning of this section. We do not

[3] http://www.dacapobench.org/benchmarks.html

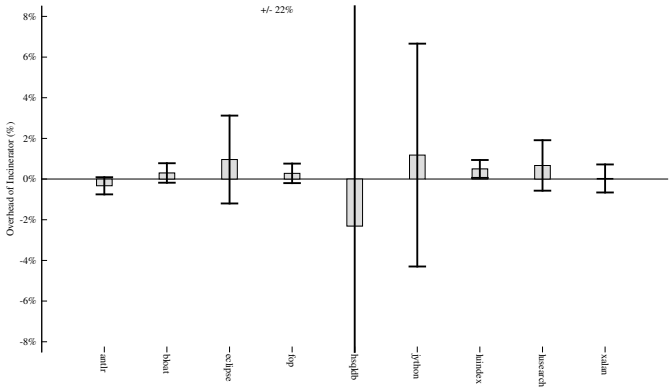

Fig. 5: Overhead of Incinerator as compared to J3 on the DaCapo 2006 benchmark applications on a low-end computer. *hsqldb* has a standard deviation of 22 percentage points, truncated here for clarity.

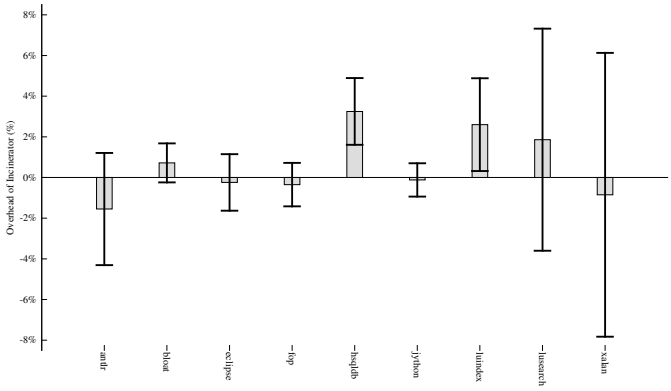

Fig. 6: Overhead of Incinerator as compared to J3 on the DaCapo 2006 benchmark applications on a high-end computer.

have results for two of the applications, pmd and chart, due to limitations of our low-end computer. On the low-end computer, Figure 5 shows that J3 performs better than Incinerator in 7 out of 9 applications, with a worst-case average slowdown of 3.3%. We have observed that Hsqldb can lead to swapping in this setting, which causes the high standard deviation in this case. On the high-end computer, Figure 6 shows that J3 performs better than Incinerator in 4 out of 9 applications, with a worst-case average slowdown of 1.2%. We have noted that J3 is 1.2 to 3 times slower than JikesRVM. We thus estimate that the worst-case average slowdown on JikesRVM would be around 10%.

As indicated by the standard deviations on Figure 5 and Figure 6, these comparisons are inverted in some runs, i.e., Incinerator sometimes performs better than J3. This is mostly due to disk and processor cache effects, and measurement bias [12].

Overall, our evaluation shows that Incinerator has only a marginal impact on the performance of J3. This result suggests that Incinerator could be used in a production environment.

B. Evaluation in the context of OSGi

We have also evaluated Incinerator in the context of OSGi. We first compare the behavior of Incinerator with Service Coroner [13], an existing tool that detects some stale references. Then, we show the potential impact of bundle conflicts in the context of a simple gas alarm application. Finally, we present a concrete case of a stale reference bug found in the Knopflerfish OSGi framework [5].

1) Comparison with Service Coroner: Service Coroner [13] is a state of the art framework to detect stale references in OSGi. It instruments the OSGi repository, which is used by a client bundle to acquire a first reference to an object provided by a server bundle. By instrumenting the repository, Service Coroner keeps track of the references given to the client bundles. Service Coroner detects stale references by analyzing the object references used by each bundle from a memory dump. If it finds a reference to an object that was registered in the repository for which the bundle is uninstalled or updated, it indicates that the reference is stale.

After having acquired a reference to a first object, a client bundle can acquire new references from the server bundle by invoking methods from this object. Service Coroner does not detect the exchange of these references because the communication between the client bundle and the server bundle does not involve the OSGi framework.

We have developed two scenarios illustrating the two cases, again based on the application configuration presented in Figure 4 (see Section V-A1). In **Scenario 8**, the bundle A retains a reference to X which is registered in the OSGi repository. The reference is turned stale by uninstalling B. In **Scenario 9**, X is also registered in the repository, but the bundle A retains a reference to Y obtained via X which is not registered. The reference is turned stale by uninstalling B.

We have tested Service Coroner using Knopflerfish 3.5.0 on top of Hotspot 6 with these scenarios. We have not evaluated Service Coroner with J3 because Service Coroner requires a full Java 6 environment. J3 relies on GNU Classpath [14], which only partially implements the Java 6 environment.

Service Coroner, detects the stale reference X in **Scenario 8**, thanks to the instrumentation of the OSGi repository. However, it does not eliminate the stale reference, which leads to a memory leak. Moreover, Service Coroner does not detect the stale reference Y in **Scenario 9**. Incinerator, on the other hand, detects and eliminates the stale references in both scenarios.

2) Bundle conflicts: To demonstrate the risk of inconsistencies caused by stale references, we have prototyped an alarm application in OSGi that is representative of a typical smart home system. Figure 7 shows an overview of the structure of this application. The application monitors a gas sensor device and fires a siren if it detects an abnormal level of gas. The application accesses physical devices via two driver bundles, `SirenDriver`, `GasSensorDriver`.

We perform the following experiment:

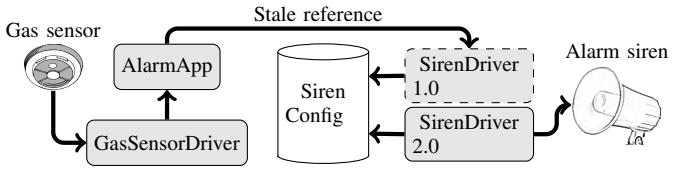

Fig. 7: Hardware and software configuration of the Alarm controller application.

1) Initially, the bundles `SirenDriver` 1.0 and `GasSensorDriver` are installed and started. Each bundle connects to its physical device (the alarm siren and the gas sensor, respectively) and exposes its features to smart-home applications. `SirenDriver` 1.0 saves the alarm siren configuration in a simple text file describing parameters and their values.
2) When the bundle `AlarmApp` is installed and started, it obtains references to the services provided by `SirenDriver` 1.0 and by `GasSensorDriver`.
3) We upgrade the bundle `SirenDriver` from version 1.0 to 2.0. As part of the upgrade, the siren configuration file is converted to an XML-based format, to simplify the addition of new configuration options. `SirenDriver` 1.0 is stopped and uninstalled, and is thus disconnected from the alarm siren. When `SirenDriver` 2.0 is started, it connects to the alarm siren and exposes its new features. After this upgrade, the OSGi framework broadcasts an event to all bundles indicating that the bundle `SirenDriver` was updated.
4) We deliberately introduced a bug in `AlarmApp` so that it does not modify the reference it holds to the service provided by SirenDriver 1.0 when it receives the broadcast update event. This reference becomes stale.

After the upgrade, we observed three problems while executing the alarm application.

First, the memory used by the JVM increased. We executed a garbage collection and observed, via the debug logs, that `SirenDriver` 1.0 was not collected, thus leaking memory.

Second, we observed that changing the settings of the siren overwrites the XML configuration file by a file in the old text format. To change the settings of the siren, the `AlarmApp` invokes the service provided by `SirenDriver` 1.0 via its stale reference to the bundle. By doing so, `SirenDriver` 1.0 overrides the XML configuration file that was previously migrated and saved by `SirenDriver` 2.0. This problem is an example of data corruption caused by stale references.

Third, after simulating a gas leak to the gas sensor, we observe alarm signals repeatedly shown by the `AlarmApp`, but the siren remains silent. Despite the fact that the `AlarmApp` knows about the gas alarm reported by the gas sensor via `GasSensorDriver`, calling `SirenDriver` 1.0 does not activate the physical siren because that version is disconnected from the device, and only `SirenDriver` 2.0 provides the service. This problem makes the siren device unusable, and represents a physical hazard to the home inhabitants.

This example is only meant for demonstration purposes, and such a bug would be easy to identify during the test phase because of the simplicity of the scenario. However, similar problems can occur in a real application that involves bundles provided by different tiers because such an application is more complex and thus harder to test exhaustively.

C. Stale references in Knopflerfish

Knopflerfish is an open source OSGi framework implementation that is commonly used in smart home gateways, because of its stability and because it only requires a Java 1.4 runtime, still commonly used in the embedded market. `HTTP-Server` is a key bundle delivered with Knopflerfish, that is used by other bundles that expose Web-based interfaces. Web-based interfaces are often used in the context of smart home applications to interact with the end user.

Using Incinerator, we have identified a bug in `HTTP-Server` version 3.1.2. `HTTP-Server` defines a group of threads to handle transactions. When `HTTP-Server` is uninstalled or updated, it does not destroy these threads as it should. As these threads reference objects allocated by the `HTTP-Server` bundle, the stale class loader stays reachable. `HTTP-Server` thus suffers from two kinds of leaks: leaked threads, which silently continue to run, and stale references from these threads.

Stale references in the leaked threads of `HTTP-Server` cause a loss of 6 megabytes of memory on each bundle update. Indeed, the `HTTP-Server` bundle contains 46 classes, which in all contain 8551 lines of Java code.

Incinerator successfully eliminates the stale references, thus avoiding the memory leaks they cause and increasing the availability of the JVM, even in the presence of stale reference bugs in running applications. As the leaked threads regularly access the stale references, they quickly receive a `NullPointerException`, which is not caught by `HTTP-server`, and thus the threads stop. Incinerator thus simultaneously solves the two leaks: the leaked thread is stopped and the memory of the stale bundle is reclaimed.

We sent a bug report and a patch to the Knopflerfish developer community.[4] The patch destroys the leaked threads when the bundle is uninstalled and updated, thus avoiding the leaked threads and consequently the stale references. The patch was approved and has been integrated in the framework since the release 4.0.0. This shows that even a well-recognized OSGi framework can suffer from the problem of stale references.

VI. Related work

We first discuss work that targets the detection and elimination of memory leaks in managed runtime environments, and then compare the management of stale references with the management of weak references already found in Java. We also present various techniques for code updating and show how they deal with stale references. Finally, we discuss work that targets stale references in the context of OSGi.

A. Memory leaks in managed runtime environments

The problem of memory leaks in managed runtime environments that use garbage collection is more general than

[4] http://sourceforge.net/p/gatespace/bugs/175/

the problem of stale references considered by Incinerator. Cork [15] and LeakBot [16] identify growing data structures to find potential memory leaks. They do not eliminate the leaks because they do not know whether the objects will be accessed in the future. With Incinerator, we suppose that the middleware knows the point in time after which a class loader should not be used. Thus, Incinerator reclaims the memory of the class loader and the other objects that are only reachable from it. Nguyen et al. [17] bound the number of objects simultaneously allocated at sites identified during a training phase. This approach is subject to false positives that may lead to crashes when it overestimates the bound. Incinerator does not have this problem. Bond [18] stores objects that are expected to be unused on disk, in order to preserve memory, while Incinerator reclaims their memory directly.

B. Weak references

The Java specification defines a *weak reference*[5] as a special reference that enables accessing an object without ensuring that the object will stay alive. If the garbage collector finds that an object is only accessible by weak references, then it nullifies all the remaining weak references and collects the referenced object.

Conceptually, our approach amounts to replacing an ordinary (i.e., strong) reference by a weak reference when the reference becomes stale. Nevertheless, the syntax for accessing ordinary references is different from that for accessing weak references. Converting ordinary references to weak references would require modifying and re-JITing the code, which would make the conversion impractical at run time.

C. Application update and memory leak

In environments that allow multiple applications to run simultaneously, the management of data isolation between applications ranges from full isolation to full sharing [19]. The Multitasking Virtual machine [20], [21], KaffeOS [22] and Singularity [23], for example, provide full isolation. They avoid stale references by preventing the use of direct inter-application references. Instead, applications exchange proxies to communicate. When an application is unloaded, only the proxy may leak, not the class loader and all loaded classes. However, proxies degrade performance, as compared to direct references, and complicate the communication between applications.

Systems that allow direct references, such as OSGi [1], trade security for performance. They provide weaker isolation between applications and the possibility that references become stale when bundles are updated or uninstalled. I-JVM [24] addresses the problem of isolation by using resource accounting techniques inspired by JRes [25]. I-JVM, however, does not address the problem of stale references.

Work on Dynamic Software Updating (DSU), such as JVolve [26], proposes to update an application by patching the code and the data structures directly in memory. DSU approaches do not suffer from memory leaks because unused fields are directly removed from objects during an update. DSU, however, does not address the management of multiple applications and their communication, as is done by middlewares such as OSGi.

[5] http://docs.oracle.com/javase/6/docs/api/java/lang/ref/package-summary.html#reachability

Application	Number of stale references detected
JOnAS 5.0.1 / Felix 1.0	7
Jitsi alpha3 / Felix 1.0	19
Sling 2.0 / Felix 1.0	3
Newton 1.2.3 / Equinox 3.3.0	58

Fig. 8: Stale references found by Service Coroner.

D. OSGi-specific tools

Service Coroner [13] is a profiling tool that detects stale references in OSGi by periodically dumping all of the memory and analyzing the reference usage graph. Performing such a memory dump incurs high CPU, memory and disk overheads, which limits the use of this tool to testing environments. Experiments made by the authors of Service Coroner have shown that stale references exist in several applications, as shown in Figure 8. Nevertheless, Service Coroner is not able to eliminate stale references because it analyses the memory dump offline.

As presented in Section V, Service Coroner only detects stale references that reference *services*, *i.e.*, objects that a bundle has registered in a repository. If a method of a service returns a reference to an object allocated by the bundle of the service, this reference is not visible to OSGi and thus is not visible to Service Coroner.

Another solution to avoid the memory leaks caused by stale references in OSGi is the *OSGi ME* specification [27], developed by IS2T and Orange. Like Service Coroner, OSGi ME also only focuses on stale references to services.

Another solution to eliminate stale references is to simply reboot the framework after each bundle update or uninstall. This solution is taken by Eclipse. Nevertheless, this solution is not suitable for a Smart Home environment, which is highly diverse and distributed, and where updates are frequent and asynchronous.

VII. Conclusion

This paper presents Incinerator, a garbage collector extension that detects and eliminates memory leaks caused by stale references, which may appear when applications are unloaded or updated in Java. Incinerator introduces an "outdated" flag to the class loader, which indicates whether the class loader is outdated, and uses this flag during a garbage collection cycle to identify stale references.

We have shown that implementing Incinerator requires modifying only around 150 lines of code in a Java virtual machine. The average CPU overhead induced by Incinerator in J3 is always less than 1.2% on the applications of the DaCapo benchmark suite on a high-end computer, and less than 3.3% on a low-end computer. These results tend to show that Incinerator should have a limited overhead.

We have also presented the use of Incinerator in the context of OSGi. Incinerator detects more kinds of stale

references than the existing OSGi stale reference detector, Service Coroner. Furthermore, while Service Coroner only detects stale references, Incinerator also eliminates them by setting them to `null`. This allows the garbage collector to reclaim the referenced stale objects. Indeed, we have found that stale references can cause significant memory leaks, such as the 6 megabyte memory leak on each update of the `HTTP-Server` bundle caused by the stale reference bug we discovered in Knopflerfish. Preventing memory leaks increases the *availability* of the JVM. Finally, Incinerator is mostly independent of a specific OSGi implementation and, indeed, only 6 lines need to be modified in the Knopflerfish OSGi framework in order to integrate Incinerator.

ACKNOWLEDGMENT

This research was supported by the ANR (France) project Infra-JVM (ANR- 11-INFR-008-01).

REFERENCES

[1] The OSGi Alliance, "OSGi service platform core specification, release 4, version 4.2," http://www.osgi.org/download/r4v42/r4.core.pdf, 2009.

[2] SWEX Group, "Home gateway initiative - requirements for software modularity on the home gateway version 1.0," SWEX Group, Tech. Rep., 2011.

[3] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot, "VMKit: a substrate for managed runtime environments," in *VEE'10*. ACM, 2010, pp. 51–62.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? High performance garbage collection in Java with MMTk," in *ICSE'04*. IEEE, 2004, pp. 137–146.

[5] Knopflerfish web page, http://www.knopflerfish.org/, 2012.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA'06*. ACM, 2006, pp. 169–190.

[7] T. Lindholm and F. Yellin, *The Java^{TM} virtual machine specification*, 2nd ed. Addison-Wesley, 1999.

[8] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*, 1st ed. Chapman & Hall/CRC, 2011.

[9] L. DeMichiel and B. Shannon, "Java^{TM} platform, enterprise edition (Java EE) specification, v7," https://java.net/downloads/javaee-spec/JavaEE_Platform_Spec_EDR.pdf, 2012.

[10] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin locks: featherweight synchronization for Java," in *PLDI'98*. ACM, 1998, pp. 258–268.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java^{TM} language specification*, 3rd ed. Addison-Wesley, 2005.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS'09*. ACM, 2009, pp. 265–276.

[13] K. Gama and D. Donsez, "Service Coroner: A diagnostic tool for locating OSGi stale references," in *SEAA 08*. IEEE, 2008, pp. 108–115.

[14] "The GNU classpath project," http://www.gnu.org/software/classpath, 2014.

[15] M. Jump and K. S. McKinley, "Cork: Dynamic memory leak detection for garbage-collected languages," in *POPL'07*. ACM, 2007, pp. 31–38.

[16] N. Mitchell and G. Sevitsky, "Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," in *ECOOP'03*. Springer-Verlag, 2003, pp. 351–377.

[17] H. H. Nguyen and M. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," in *ISMM'07*. ACM, 2007, pp. 15–30.

[18] M. D. Bond and K. S. McKinley, "Tolerating memory leaks," in *OOPSLA'08*. ACM, 2008, pp. 109–126.

[19] J. S. Rellermeyer, S.-W. Lee, and M. Kistler, "Cloud platforms and embedded computing: the operating systems of the future," in *DAC'13*. ACM, 2013, pp. 1–6.

[20] G. Czajkowski and L. Daynès, "Multitasking without comprimise: a virtual machine evolution," in *OOPSLA'01*. ACM, 2001, pp. 125–138.

[21] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynès, "Incommunicado: efficient communication for isolates," in *OOPSLA'02*. ACM, 2002, pp. 262–274.

[22] G. Back, W. C. Hsieh, and J. Lepreau, "Processes in KaffeOS: isolation, resource management, and sharing in Java," in *OSDI'00*. USENIX, 2000, pp. 23–23.

[23] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber, "Sealing OS processes to improve dependability and safety," in *EuroSys'07*. ACM, 2007, pp. 341–354.

[24] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot, "I-JVM: a Java virtual machine for component isolation in OSGi," in *DSN'09*. IEEE, 2009, pp. 544–553.

[25] G. Czajkowski and T. von Eicken, "JRes: a resource accounting interface for Java," in *OOPSLA'98*. ACM, 1998, pp. 21–35.

[26] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a VM-centric approach," in *PLDI'09*. ACM, 2009, pp. 1–12.

[27] A. Bottaro and F. Rivard, "OSGi ME - an OSGi profile for embedded devices." London, UK: OSGi Community Event, Sep. 2010.