2009

Quarantine: Java Heap Protection in the Presence of Native Code

Du Li
University of Nebraska-Lincoln, dli@cse.unl.edu

Witawas Srisa-an
University of Nebraska-Lincoln, witawas@unl.edu

Quarantine: Java Heap Protection in the Presence of Native Code

Du Li and Witawas Srisa-an

Department of Computer Science and Engineering
University of Nebraska-Lincoln
256 Avery Hall
Lincoln, NE 68588-0115
dli@cse.unl.edu and witty@cse.unl.edu

Abstract

By using Java Native Interface (JNI), programmers can integrate Java programs with legacy systems or third-party libraries written in other languages (e.g., C, C++, and Pascal). However, the use of JNI may violate Java type safety feature because these native programs are not type-safe. As a result, such integration can cause memory errors that can be difficult to isolate.

In this paper, we propose *Quarantine*, a runtime system that preserves memory safety of Java objects in spite of integration with native code. The goal of Quarantine is ensuring that no native threads can directly access objects in the Java heap. We provide a formal proof that our technique can achieve this goal. We then implement a prototype of Quarantine in the OpenJDK 1.7 running in interpreter mode. To evaluate the feasibility of our prototype, we conduct experiments to measure the runtime overhead of Quarantine. Because our current implementation is unoptimized, the overhead can be as high as 42%. We then discuss ways to reduce this overhead and perform a case study of using Quarantine to avoid heap corruption due to out-of-bound writes.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Memory management (garbage collection)

General Terms Experimentation, Languages, Performance

1. Introduction

Foreign Function Interface (FFI) provides a facility for higher-level languages to interface with lower-level languages [20, 8, 13, 6]. As an example, *Java Native Interface* [12] provides a standardized way for Java programs to interface with native libraries written in languages such as C, C++, FORTRAN, Pascal, and COBOL. Such interfaces are needed for various reasons including obtaining system services, integrating with legacy libraries, and achieving higher system performance.

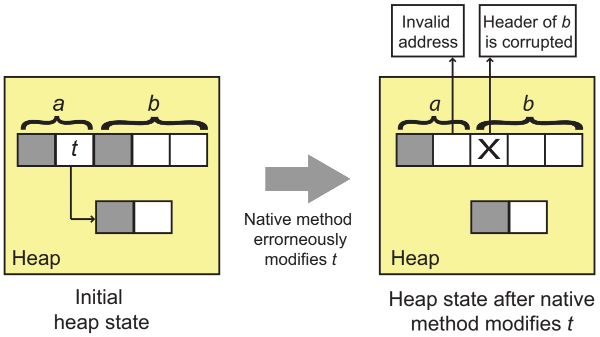
Invalid address
Header of *b* is corrupted
a
b
t
Heap
Initial heap state
Native method erroneously modifies *t*
a
b
Heap
Heap state after native method modifies *t*

Figure 1. Native method corrupting Java heap

While the use of JNI can significantly reduce the redevelopment time of existing native or legacy libraries, its usage also incurs significant penalties that include additional code complexities, less powerful exception mechanisms, and increased memory errors, which is the main focus of our work. By design, Java language is type-safe. However, when a Java program interacts with software components written in unsafe languages, that Java program is no longer type-safe. As a result, data accesses by these unsafe components can corrupt the heap, possibly causing the Java Virtual Machine (JVM) to fail.

Previous work by Chiba [4, 5] has shown that once Java programs suffer from this type of errors, they can be very difficult to debug. This is because the sources of such errors lie in native libraries. Moreover, the corrupted data may not manifest itself for a long time, making isolating the component that causes the error challenging. We provide an illustration of this type of errors in Figure 1.

In this example, field *t* in *object a* holds a 32-bit reference to a *ThreadLocal* object. There is a native method *modifyObj* that is written in C. The reference to *object a* is exported to

Department of Computer Science & Engineering, University of Nebraska-Lincoln,
Technical Report, TR-UNL-CSE-2009-0018

this native method. The native method then manipulates field *t* and incorrectly assigns a 64-bit value to it. In doing so, *t* no longer contains a valid heap reference. Moreover, the header of its neighboring object, *object b*, is also corrupted. As a result, whenever the program tries to access *object b* or field *t* in *object a*, it would crash. This simple example clearly shows a type-safety violation instigated by a native thread. Past studies and reports have shown that this type of memory errors occurs quite frequently in large servers that use heterogeneous components [4, 5, 18, 20].

As shown in the example, *object a* is unsafe as soon as it is manipulated by the native method. We refer to this type of objects as *unsafe* objects because they have interactions with native methods and in turn, act as gateways for native methods to possibly corrupt other Java objects in the heap (i.e., *object b* in the previous example). In a typical server application, the number of created objects can be several hundred million; however, the number of these unsafe objects is a fraction of the total objects. Such a disparity in the numbers of unsafe objects and total objects can make identifying unsafe objects difficult because there are only a few of them to be discovered.

On the other hand, the ability to identify these unsafe objects provides an opportunity for better isolation. For example, these unsafe objects can be hosted in a separate memory region that only contains this type of objects. (We refer to this region as a *quarantine* site.) Quarantining these unsafe objects provides at least three important benefits:

1. *Simplify debugging*. As in the example above, the JVM throws an exception when it tries to access field *t*. The provided address of the object that contains the invalid reference would indicate that it resides in the quarantine site. Thus, it is likely that the source of error is due to improper access by a native method.
2. *Limit contagiousness*. By isolating unsafe objects, improper accesses to them are less likely to corrupt the neighboring objects (such as in the case of out-of-bound writes), which can cause the virtual machine to fail due to invalid references or corrupted headers. Neighboring object corruptions make debugging very difficult because the corrupted objects are often not the ones that were initially accessed by native methods.
3. *Better heap protection*. A small number of unsafe objects when compared to the total number of objects means that the quarantine site should also be relatively small. Thus, existing heap protection techniques [1, 14] that have shown to work well in moderately sized heaps but might incur too much overhead for large heaps can be feasibly applied to make the Java program more tolerable to certain classes of memory errors such as out-of-bound writes and object header corruptions.

This work. We evaluate the feasibility of a proposed runtime technique called *Quarantine*, which enforces a simple runtime property that *no native methods can ever access objects in the nursery and mature spaces of Java heap*. Instead, any access to an object by a native method must be done in a quarantine site that is specifically created to hold objects that can possibly be corrupted. Our technique extends the JVM to identify any objects that have been exported to native methods. These objects and their transitive closures are then moved to the quarantine site before the native methods can access them. The proposed technique has been implemented in the OpenJDK 1.7 (running in the interpreter mode only) from Sun Microsystems. The initial result indicates that Quarantine can successfully enforce the desired property but at a cost of about 42% overhead when compared to the performance of OpenJDK's interpreter without Quarantine.

We then try to reduce the number of copied objects by exploiting an observation that native methods rarely access descendant objects that are referenced by fields inside an exported object. To exploit this observation, we modify Quarantine to copy objects in a reference graph based on a specified level. For example, when the level is set to 2, Quarantine only copies an exported object (i.e., the first level) and any objects that can be directly referenced from the exported object (i.e., the second level). When we set the level to 1, the overhead is reduced to 15%.

Lastly, we conduct a case study that applies an 8-byte padding to the end of each object in the quarantine site. This padding is used to neutralize out-of-bound writes so that they cannot corrupt the neighboring objects. We find that the space overhead to support padding is only 0.044% of the entire heap space, making it a feasible candidate for more complex heap protection mechanisms. For future work, we discuss optimization techniques that can further reduce the overhead of our proposed system.

The remainder of this paper is organized as follows. Section 2 introduces the proposed Quarantine. Section 3 discusses our prototyping effort. Section 4 elaborates on our methodology to evaluate Quarantine. Section 5 reports the result of our overhead investigation. Section 6 describes our case study to handle out-of-bound writes in Quarantine. Section 7 discusses related work. The last section concludes this paper.

2. Introducing Quarantine

The goal of Quarantine is to eliminate the presence of unsafe objects in the nursery and mature space that are managed by the JVM. We take this approach because native code can potentially corrupt objects it manipulates as well as the neighboring objects. Therefore, preventing native code from manipulating objects in the heap can make the heap more robust.

We design Quarantine as a runtime system so that it can provide heap protection without the need to analyze and recompile the native programs. We believe that our approach is appealing to large heterogeneous systems because often times, the source code of many legacy libraries may not be available or be too antiquated to recompile. With Quarantine, all the work to identify and segregate unsafe objects is done entirely during runtime by the JVM.

Unsafe objects. An object is considered unsafe when it is accessible by native code. There are two possible ways for this to happen. First, an object is created by the native code. This object is typically created in the JVM managed heap; and therefore, it intermingles with the rest of the Java objects. In most JVMs, a native method is executed as a thread that share the same virtual address space as Java threads. Thus, creating objects in the heap allows native threads and Java threads to communicate easily. On the other hand, allowing native threads to directly create objects in the Java heap makes the entire heap unsafe. In Quarantine, we handle this situation by simply creating such objects directly in the quarantine site. (From now on, we will refer to quarantine site as *JNI space*.)

Second, an object created by a Java thread is exposed to native threads through Java Native Interfaces. When this occurs, Quarantine intercepts the object's address and moves this object and its transitive closure to the JNI space. It then leaves a forwarding address in the data portion of each original object for the JVM to update existing stale references to point to the new locations. In Listing 1, object $newVal$ at line 5 is created by native thread. In Listing 2, object $value$ at line 4 is created by a Java thread and then passed as an argument to a native method. Furthermore, every object that is directly or indirectly referenced by $value$ is considered unsafe.

```
1 JNIEXPORT void JNICALL
2 Java_Example_modifyObj(JNIEnv *env,jobject obj1,
      jobject obj2) {
3   jclass objClass = (*env)->GetObjectClass(env, ”
        java/lang/Object”);
4   jmethodID cid = (*env)->GetMethodID(env,
        objClass, ”<init>”, ”()V”);
5   jobject newVal = (*env)->NewObject(env,
        objClass, cid);
6   jclass cls = (*env)->GetObjectClass(env, obj2);
7   jfieldID fid = (*env)->GetFieldID(env, cls, ”
        foo”, ”Ljava/lang/Object;”);
8   (*env)->SetObjectField(env, obj2, fid, newVal);
9 }
10 ...
```

Listing 1. Example.c

```
1 class Example{
2     public Object foo,
3     public void m(){
4     Example value = new Example();
5             modifyObj(value);
6     }
7   public native void modifyObj(Example o);
8       ...
9 }
```

Listing 2. Example.java

2.1 Formal Proof

This section presents a proof that quarantine can enforce the desired property that *no native threads can directly access any object in the nursery and the mature space of a Java heap*. To enforce this property, we assume that native threads can only get reference to a Java object through Java Native Interface. We make this assumption because this is the most common and safest way for Java methods and native methods to interact. Quarantine does not work in the case that JNI is not used to expose an object to a native method.

2.2 Basic Definitions

DEFINITION 2.1. *(access function). Let $\mathcal{T}$ be a thread representing a native method and $\mathcal{B}$ is the set of objects which are accessible by $\mathcal{T}$. The access function $\mathcal{F}_a$ is defined as: $\mathcal{F}_a(\mathcal{T}) = \mathcal{B}$.*

DEFINITION 2.2. *(export function). Let f be a function call to a native method (e.g., line 7 in Listing 2) and $\mathcal{C}$ is the set of objects that are exported by f as parameters or receiver objects to the native method. The export function $\mathcal{F}_e$ is defined as: $\mathcal{F}_e(f) = \mathcal{C}$.*

DEFINITION 2.3. *(reference function). Let o be an object and $\mathcal{D}$ is the set of Java objects that can be referenced from o. The reference function $\mathcal{F}_r$ is defined as: $\mathcal{F}_r(o) = \mathcal{D}$.*

DEFINITION 2.4. *(reference tree). Let f be a call to a native method, o be an object, and $\mathcal{F}_r^*(o) = \mathcal{F}_r(\mathcal{F}_r(...\mathcal{F}_r(o)))$. The reference tree is defined as:*

$$\Pi(f) = \bigcup_{\beta \in \mathcal{F}_e(f)} \mathcal{F}_r^*(\beta)$$

.

DEFINITION 2.5. *(moving function). Let o be an object in a Java heap, o' is the identical copy of o in the JNI space and all Java objects in $\mathcal{F}_r(o')$ are in the JNI space. The moving function is defined as: $\mathcal{M}_v(o) = o'$.*

2.3 Interface Formats

There are four possible formats for making native method calls from Java programs: (i) static method without instance parameters; (ii) static method with instance parameters; (iii) non-static method without instance parameters; and (iv) non-static method with instance parameters. These four formats are presented in Table 2.3 with the following representation: C: class, M: method, O: object, b_i: basic type parameters, and i_i: instance parameters.

In Quarantine, the two formats with instance parameters and one format with non-static method are represented below. Note that the format of static method without instance parameters remains unchanged.

static method without instance parameters	$C.M(b_1, b_2, ..., b_n)$
static method with instance parameters	$C.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n)$
non-static method without instance parameters	$O.M(b_1, b_2, ..., b_n)$
non-static method with instance parameters	$O.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n)$

Table 1. Four formats of Java Native Interfaces

$C.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n) \equiv$
$C.M(\mathcal{M}_v(i_1), \mathcal{M}_v(i_2), ..., \mathcal{M}_v(i_n), b_1, b_2, ..., b_n)$

$O.M(b_1, b_2, ..., b_n) \equiv \mathcal{M}_v(O).M(b_1, b_2, ..., b_n)$

$O.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n) \equiv$
$\mathcal{M}_v(O).M(\mathcal{M}_v(i_1), \mathcal{M}_v(i_2), ..., \mathcal{M}_v(i_n), b_1, b_2, ..., b_n)$

2.4 Proof

AXIOM 2.1. *Let Ω be the set of all the native methods, Θ be the set of all the native method calls in a Java program,*

$$\bigcup_{\mathcal{T} \in \Omega} \mathcal{F}_a(\mathcal{T}) = \Sigma_c + \bigcup_{\mathcal{F} \in \Theta} \Pi(\mathcal{F})$$

LEMMA 2.1. *In Quarantine, objects created by native methods, denoted as Σ_c, are in the JNI space.*

Proof: Lemma 2.1 is trivial because Quarantine satisfies any allocation requests from native methods in the JNI space.

LEMMA 2.2. *Let $\mathcal{F}$ be a native method call in a Java program, all Java objects in the set $\Pi(\mathcal{F})$ are in the JNI space.*

Proof: As mentioned earlier, Java Native Interfaces can be any of the four formats.

Case 1: $\mathcal{F}$ is static method without instance parameters.
$C.M(b_1, b_2, ..., b_n) \equiv C.M(b_1, b_2, ..., b_n)$
$\Rightarrow \Pi(\mathcal{F}) = \phi$, therefore no parameters need to be in the JNI space.

Case 2: $\mathcal{F}$ is static method with instance parameters.
$C.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n) \equiv$
$C.M(\mathcal{M}_v(i_1), \mathcal{M}_v(i_2), ..., \mathcal{M}_v(i_n), b_1, b_2, ..., b_n)$
$\Rightarrow \Pi(\mathcal{F}) = \mathcal{F}_r^*(\mathcal{M}_v(i_1)) + \mathcal{F}_r^*(\mathcal{M}_v(i_2)) + ... + \mathcal{F}_r^*(\mathcal{M}_v(i_n))$
For any $k \in [1..n], \mathcal{F}_r^*(\mathcal{M}_v(i_k))$ are in the JNI space according to the definition of moving function, so $\Pi(\mathcal{F})$ are in the JNI space.

Case 3: $\mathcal{F}$ is non-static method without instance parameters.
$O.M(b_1, b_2, ..., b_n) \equiv O.M(b_1, b_2, ..., b_n)$
$\Rightarrow \Pi(\mathcal{F}) = \mathcal{F}_r^*(\mathcal{M}_v(O))$
All Java objects (i.e., receivers) in $\mathcal{F}_r^*(\mathcal{M}_v(O))$ are in the JNI Space according to the definition of moving function, so $\Pi(\mathcal{F})$ are in the JNI space.

Case 4: $\mathcal{F}$ is non-static method with instance parameters.
$O.M(i_1, i_2, ..., i_n, b_1, b_2, ..., b_n) \equiv$
$\mathcal{O}.M(\mathcal{M}_v(i_1), \mathcal{M}_v(i_2), ..., \mathcal{M}_v(i_n), b_1, b_2, ..., b_n)$
$\Rightarrow \Pi(\mathcal{F}) = \mathcal{F}_r^*(\mathcal{M}_v(O)) + \mathcal{F}_r^*(\mathcal{M}_v(i_1)) + \mathcal{F}_r^*(\mathcal{M}_v(i_2)) + ... + \mathcal{F}_r^*(\mathcal{M}_v(i_n))$
All objects in $\mathcal{M}_v(O)$ and $\mathcal{F}_r^*(\mathcal{M}_v(i_k))(k \in [1..n])$ are in the JNI space according to the definition of moving function, so $\Pi(\mathcal{F})$ are in the JNI space.

All Java objects in set $\Pi(\mathcal{F})$ are in the JNI space in these four cases.

THEOREM 2.1. *Invariant*
Let be Ω is the set of all native methods in a program

$$M = \bigcup_{f \in \Omega} \mathcal{F}_a(f),$$

all objects in M are in the JNI space.

Proof: According to Lemma 2.1, Σ_c are all in the JNI space. According to Lemma 2.2, for any native method call $\mathcal{F}$, $\Pi(\mathcal{F})$ are also in the JNI space.

$\Rightarrow$ All Java objects in

$$\bigcup_{\mathcal{F} \in \Theta} \Pi(\mathcal{F})$$

are in the JNI space.

$\Rightarrow$ All Java objects in

$$\Sigma_c + \bigcup_{\mathcal{F} \in \Theta} \Pi(\mathcal{F})$$

are in the JNI space.

$\Rightarrow$ Accoding to Axiom 2.1,

$$M = \Sigma_c + \bigcup_{\mathcal{F} \in \Theta} \Pi(\mathcal{F})$$

Therefore, all objects in M are in the JNI space.

3. Prototyping Quarantine

We prototype Quarantine on the OpenJDK 1.7 b24 [19] with Linux Fedora 8 operating system. As a prototype system, our goal is to validate our design and evaluate whether Quarantine can be implemented to enforce the desired runtime property. Thus, we implement Quarantine in the HotSpot's C++ interpreter. We discuss the ramifications of this implementation choice in Section 4.1.

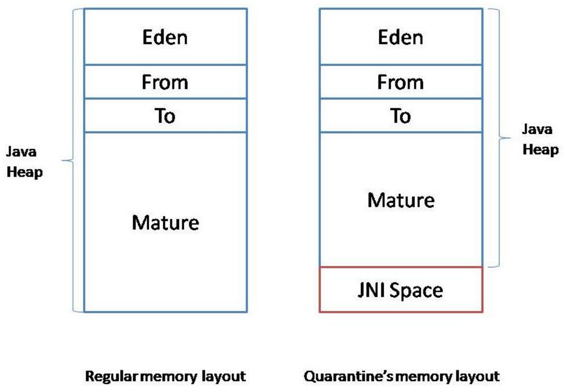

Figure 2. Heap layout

Currently, our prototype can successfully run a few benchmarks that include SPECjbb2005 and a subset of SPECjvm98. To activate Quarantine, a JVM option "UseJNIHeap" must be set. Recently, we have begun to port our implementation to the JIT compiler in HotSpot. We plan to report the result of our implementation and performance analysis in the subsequent version of this paper.

In the next few subsections, we discuss the implementation details to support Quarantine in HotSpot.

3.1 Heap Layout

To support Quarantine, we add *JNI space*, a new memory area to hold unsafe objects (see Figure 2). We can customize the size of this memory area using a command line option.

3.2 JNI Modifications

As stated earlier, there are two types of objects that are hosted in the JNI space: objects created by native methods and objects that are exposed to native methods. For objects created by native methods, Quarantine uses a modified memory allocator instead of the typical allocator to create these objects directly in the JNI space. This is accomplished by modifying the implementation of the JNI interface that creates objects in the Java heap. For other unsafe objects created by Java code but then exported to native code, Quarantine copies these objects and their transitive closures from Java heap to the JNI space as soon as the objects are exported but before they are accessed by native methods. We support object moving by modifying the interpretation of the following bytecodes: *invoke_interface*, *invoke_special*, *invoke_static*, and *invoke_virtual*. Quarantine checks the to-be-executed method whether it is native method or not. If it is a native method, Quarantine copies the object parameters and receiver objects, if any, to the JNI space according to the method signature. At the same time, our modified interpreter also leaves a forwarding address in the data portion of an original object. It also modifies its object's header to indicate that this object is now a forwarding object and not the actual object.

3.3 Supporting Forwarding References

As stated above, we need to distinguish actual objects from forwarding objects. To accomplish this task, Quarantine steals two unused bits from the object header as a status tag of the object. The meaning of the tag is defined in Table 2.

3.4 Read and Write Barriers

Once these unsafe objects have been moved, the program may still have references to the stale objects. These stale references must be updated to maintain correct execution states. To lazily update these stale references, we modify the write barrier mechanism in HotSpot to check the status tag in the object header. If a write attempt is made to a forwarding object, the reference is first updated with the forwarding address so that the reference is now made to the copied object in the JNI space, then the write is performed.

Unfortunately, HotSpot does not have read-barrier mechanism. Therefore, we have to implement our own version. Our read barrier works as follows: when a read attempt is made, our barrier checks whether the read attempt is to a regular object or a forwarding object. For an actual object, our read barrier does nothing. For a forwarding object, our read barrier takes the forwarding address and updates the reference to point to the new location and the read operation is performed. We then install our read barrier in all the bytecode interpretations that perform read accesses into the heap. Furthermore, we also install our read-barrier at the follow execution locations.

- **Popping objects from execution stack.** When a method is executed, all the parameters and receiver objects, if any, are popped from the execution stack. Quarantine checks the status of each object when it is popped from the stack and ready to be used by some methods.
- **Dereference Java object handles.** There are some handles in HotSpot that contain references to Java objects. These handles may contain stale references if these handles are constructed before we moved unsafe objects. It is necessary to check and update these handles when they are dereferenced.
- **Tracing phase of garbage collection.** The garbage collector needs to be cognizant of forwarding objects. If the collector encounters a forwarding object during tracing, it then forwards the reference to the new location.

Once all the stale references to an original object have been updated, the object is automatically garbage collected in the next collection cycle.

3.5 Modifying Garbage Collector in HotSpot

To collect dead objects in the JNI space, we modify the serialized generation collector in HotSpot [17, 10]. Specifically, we extend the full collector, which is a mark-compact scheme [10], to also perform collection of this new space.

Tag bits	Status of objects
00	regular Java object
01	object in JNI space
10	forwarding object
11	undefined

Table 2. Object tagging

There are currently two ways to trigger JNI space collection. First, the space is collected when the mature generation is full. Second, the space is collected when the JNI space is full.

In both instances, the entire heap collection is performed. As part of this process, the full collector works the entire Java heap while the JNI collector only works the JNI space. While this approach incurs high collection overhead, it is also much simpler because the entire heap is scanned for live objects. There is no need to maintain a list of intergenerational references [21], which is needed by a typical minor or nursery collection. During a collection of the JNI space, the collector simply slides live objects toward the lowest address of the JNI space. Once an object is hosted in the JNI space, it stays in the space until it dies.

3.6 Native Method Selection

HotSpot also invokes several native methods through JNI as part of its initialization and execution. In this work, we consider these methods as "trusted" methods since they are provided by Sun as part of HotSpot. These trusted methods are not considered by Quarantine to create unsafe objects, and therefore, objects accessible by these "trusted" native methods are not moved to the JNI space.

On the other hand, any native methods directly called by applications are considered "unsafe" by Quarantine. Any objects accessible by these methods reside in the JNI space.

4. Evaluation Methodology

We conduct a preliminary experiment to analyze the runtime overhead of Quarantine. In terms of benchmark selection, we are limited by two factors. First, there are currently no benchmarks that have been designed specifically to measure JNI performance. Thus, our focus turns toward benchmarks that have been commonly used in academic research. Second, Quarantine still has some issues running newer benchmarks. Currently, it can run a subset of SPECjvm98 and SPECjbb2005. We evaluate these benchmarks that can run on Quarantine. Small benchmarks tend to have very small percentages of native method invocations (fewer than five percents). The only large benchmark that can currently run on Quarantine is SPECjbb2005, which is a standardized benchmark from SPEC that has been designed to measure performance of the middle tier (business logic) in a three-tier client server architecture [16]. In a four-warehouse setting, SPECjbb2005 invokes 420 million methods. Out of these, about 40 million or 9.7% are native method invocations. It also allocates over 20 million objects.

4.1 Threats to Validity

Similar to most prototype systems, Quarantine is currently implemented to perform correctly but not efficiently. Our choice to implement Quarantine in the C++ interpreter of HotSpot makes the system performs very slowly due to high interpretation overhead. Typically, the performance of C++ interpreter is about 10 times slower than that of the fastest execution in HotSpot (combining interpretation and compilation). However, the use of C++ interpreter allows us to implement our prototype relatively quickly due to a much higher level of abstraction. Thus, the result of our performance evaluation that is reported in the next section has to be taken with a "grain of salt". Our performance comparisons are done using the runtime performance of the unmodified C++ interpreter and that of the modified C++ interpreter. This means that the result may not reflect the performance of Quarantine in a high-performance VM that utilizes dynamic compilation.

In addition, we have not fully optimized our implementation so the reported overhead is likely to be higher than the lowest possible value for this particular interpreter. Furthermore, our current implementation still contains errors that sometimes interfere with the HotSpot's finalizer when benchmarks such as DaCapo and SPECjvm2008 are used. We are currently working to overcome these small but annoying errors and expect to have Quarantine be fully compliant with the latest JDK and JNI specifications by the next revision of this paper.

4.2 Experimental Methodology

We execute SPECjbb2005 *five times and report the best, the worst, and the average scores*. The average scores are also used to provide graphical illustrations in the next section. For garbage collection performance, we obtain information by running the benchmark one time on the modified HotSpot and another time on the unmodified HotSpot with the corresponding instrumentation. For method invocation profile, we instrument the unmodified HotSpot to capture method invocation information.

4.3 Hardware Platform

We conduct our experiment on a PC with dual-core Athlon-64 running at 2.0 GHz. The system has 3GB of physical memory.

5. Results

In this section, we report the runtime overhead of Quarantine when running SPECjbb2005. We also report the effects of Quarantine on garbage collection performance. In our experiment, we set the workload to four warehouses. With this workload, the heap usage is 274MB. The nursery is set to

26MB. The mature space is set to 233MB and 210MB for the default HotSpot and Quarantine, respectively. The mature space for Quarantine is smaller because we allocate 23MB of the mature space for the JNI space. HotSpot also has a permanent space to store various permanent data structures. That space is set to 16MB.

5.1 Runtime Overhead

As stated earlier, we run the benchmark five times. Table 3 reports the lowest, average, and highest throughput performances of SPECjbb2005. The table shows that Quarantine degrades the throughput performance by 42% (based on the average throughput performances).

Default			Quarantine		
Lowest	Highest	Average	Lowest	Highest	Average
1157	1215	1190.80	671	711	689.4

Table 3. Comparing throughput performances between HotSpot with C++ interpreter and Quarantine when running SPECjbb2005

5.2 Effects on Garbage Collection

Because garbage collection in the JNI space also invokes full collection, we investigate the effects of Quarantine on garbage collection behavior. To fairly compare the behaviors of the two systems, we modify SPECjbb2005 to provide consistent allocation pressure. As suggested by Blackburn et al. [2], measuring throughput performance does not provide a fair environment to evaluate garbage collection performance. This is because system that is less efficient (such as Quarantine in this case) performs less work in a given amount of time. Less work often means fewer allocated objects than those in a more efficient system. This difference can result in different garbage collection behaviors.

To ensure fairness, we observe the garbage collection performance of our two systems given the same number of allocation requests. We first execute the default system and record the number of objects that have been allocated during its run. This number is then used by Quarantine to determine the amount of work that must be done. We then compare the garbage collection behavior once Quarantine reaches the same amount of work as the unmodified HotSpot.

Default		Quarantine	
Minor	Full	Minor	Full
75	2	76	2

Table 4. Comparing GC behaviors between HotSpot with C++ interpreter and Quarantine when running SPECjbb2005

As shown in the table, Quarantine has very little effect on garbage collection. It invokes one additional minor collection and the same number of full collection. The amount of memory that we allocate to the JNI space is plenty to handle the unsafe object and it does not get fill up during execution.

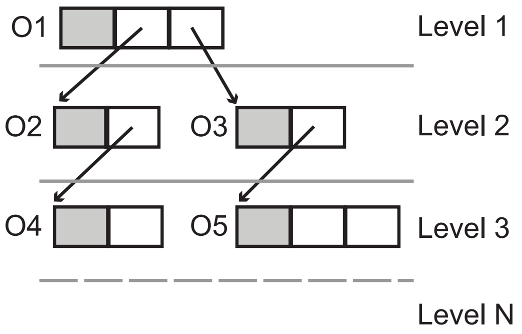

Figure 3. An illustration of exported object and its descendant

5.3 Optimization Opportunities

The two dominating costs of our implementation are read-/write barriers and copying. By porting our implementation to a dynamic compilation system, we should be able to significantly reduce the barrier costs. Past studies have shown that efficient implementations of barriers can be quite cheap [9, 3]. On the other hand, the cost of copying will likely remain high even when a dynamic compiler is utilized.

During our investigation, we observe that native code rarely accesses descendant objects (i.e., objects that are accessible via fields in an exported object). Intuitively, such an observation makes sense because native code does not have thorough knowledge of the structure of an object. Therefore, it tends to access the exported object and a few of its descendants. As a result, it may not be necessary to copy its transitive closure.

To exploit this observation, we modify Quarantine to only copy a specific level of descendant objects. As shown in Figure 3, when level is set to one, only the exported object, *O1*, is copied. When the level is set to two, *O1*, *O2*, and *O3* are copied to the JNI space. When the level is set to N, the transitive closure of *O1* is copied. Figure 4 illustrates the saving that can be obtained by exploiting this observation.

As seen in the Figure 4, when we set the level to one, the runtime overhead of our approach (based on the average of five run) is only 14.88% higher than that of HotSpot without Quarantine.

6. Case Study: Handling Out-of-Bound Writes

Over the past few years, there have been numerous techniques that have been introduced as a way to increase heap robustness [1, 14, 15]. As a simple case study, we investigate the space overhead to add padding to the end of each object in the JNI space. We use this padding to avoid instances of neighboring object corruptions.

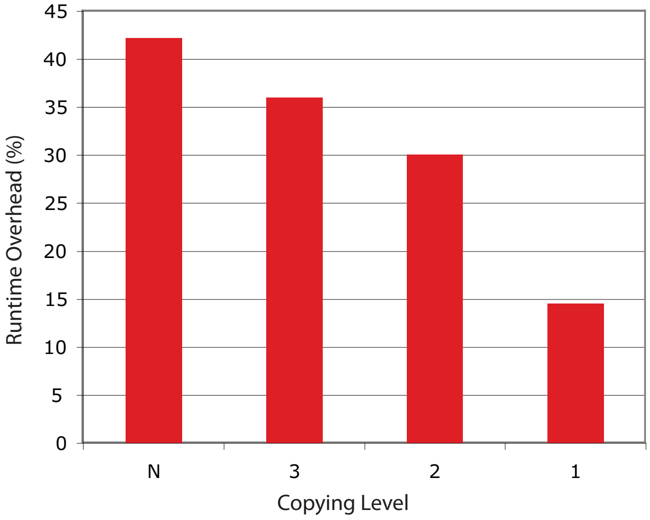

Figure 4. Reduction in runtime overhead

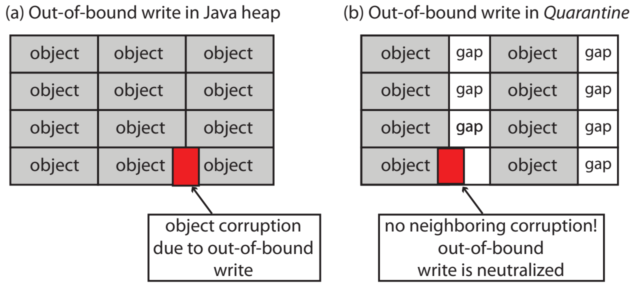
(a) Out-of-bound write in Java heap
object
object
object
object
object
object
object
object
object
object
object
object
object corruption due to out-of-bound write
(b) Out-of-bound write in *Quarantine*
object
gap
object
gap
object
gap
object
gap
object
gap
object
gap
object
object
gap
no neighboring corruption! out-of-bound write is neutralized

Figure 5. Comparing the effects of out-of-bound writes in Java heap and Quarantine

To do so, we create a fault injection system that acts as native methods that perform out-of-bound writes and possibly corrupt neighboring objects (see Figure 5(a)). Our injector can be configured to randomly pick a percentage of unsafe objects to corrupt.

In our experiment, we configure the injector to corrupt 5% of all unsafe objects. When we apply our fault injection to SPECjbb2005 running on the unmodified HotSpot, it crashes every time due to invalid references.

To neutralize the effects of memory corruption, we simply add 8 bytes to the end of every object in the JNI space (see Figure 5(b)). In doing so, our Java program can better tolerate this type of memory errors and run to completion. It is arguable that the same treatment can be applied to every object in the Java heap but the space overhead of such an approach would be very high. It is also possible to customize the padding size for each object. For example, we can set a the padding size, $S_{padding}$ using the following formula:

$$S_{padding} = \lceil \alpha * S_{object} + \beta \rceil (\alpha, \beta \geq 0),$$

where S_{object} is the size of the object.

Obviously, larger padding may provide better safety, but at a cost of large space overhead. By using an 8-byte padding, we discover that our space overhead is less than 128KB, which only accounts for 0.044% of the entire heap.

6.1 Discussion

By using Quarantine, it is feasible to add memory padding to unsafe objects. Since these unsafe objects are only a fraction of the total objects, the space overhead for padding is negligible. Such a low space overhead provides a golden opportunity to fine tune coefficients like α, β to achieve greater memory safety. For example, we can easily set α as 2 in Quarantine, the memory cost is still acceptable, but in regular JVM, it will double the usage of Java heap, which is likely to be infeasible in heavy load servers.

In addition, low space overhead also provides an opportunity to apply more complex techniques to protect the JNI space. As part of future work, we will experiments with other techniques (e.g., randomized heap [1], redundant heap [14]) and identify a few real-world programs that contain JNI related memory bugs to be used as experimental subjects.

7. Related Work

There have been several research efforts to analyze for faults in JNI applications [7, 20, 11]. In these techniques, the source code is required and the analysis is done on the native code. For example, work by Kondoh and Onodera [11] applies static analysis to find bugs such as exception handling errors, resource leaks, and invalid references in applications that utilize on JNI. Work by Tan et al. [20] proposes a *Safe-JNI* framework to ensure type-safety in heterogeneous applications using Java and C components. The work performs analysis to the C source code and modifies the code to ensure type-safety.

Work by Hirzel and Grimm [8] create a new language design for better integration of Java and C. The goal is to streamline code, enable static error detection, and simplify dynamic resource management, allowing JNI-based applications to be developed with less complexities and greater safety.

While these efforts can achieve better type safety and greater reliability for JNI applications, they are only applicable to applications with source code. They cannot work with Legacy libraries that may only be available in the binary form. Our technique, on the other hand, can work with binary libraries. However, our main focus is to identify unsafe objects and then segregate them. The goal of our work is not to identify bugs; instead our goal is to make debugging easier and make JNI based applications more tolerable to memory errors.

Work by Chiba [4, 5] attempts to ease the debugging process in JNI based applications by isolating memory errors due to invalid accesses by native code as they happen and then identify the sources of errors in the native code. His proposed system relies on page protection mechanism in the operating system to prevent native code from writing directly to the heap. When an illegal reference occurs, an page-fault exception is thrown and the address of the code that attempts to make the invalid reference is reported. Currently, his work can accomplish more than ours because it can identify the sources of errors. Our system should be able to provide a similar debugging feature by associating native methods to exported objects and their descendants when the VM is run in debug-mode.

8. Conclusions

In this paper, we present a prototype of Quarantine, a runtime system to identify *unsafe* objects in JNI-based applications and then segregate them into a quarantine site. The main goal of Quarantine is *to prevent native methods from directly accessing objects in the nursery and mature space in a Java heap*. In doing so, we can prevent native threads from corrupting Java objects that happen to be neighbors of unsafe objects. The basic mechanism of Quarantine is to satisfy allocation requests from native methods in a special memory area called JNI space. It also moves any object in the Java heap that are accessible by native methods.

As part of this work, we provide a proof that Quarantine can enforce its runtime goal. We also implement a prototype system in the C++ interpreter of OpenJDK 1.7 and test the prototype using SPECjbb2005, a server benchmark from SPEC. The initial result indicates that Quarantine requires about 42% of overhead to operate. However, by selectively copying only objects that are directly accessible by native threads but not their descendants, we can reduce the overhead to about 15%. We also conduct a case study to show that we can easily apply additional padding space to every object in JNI space to increase robustness without incurring substantial overhead (about 0.04%). In doing so, SPECjbb2005 can better tolerate memory errors due to out-of-bound writes.

For future work, we will refine our implementation to further reduce overhead. We are in the process of porting our implementation to the high performance dynamic compiler in HotSpot. We anticipate a significant reduction in the runtime overhead. In addition, we are also identifying more opportunities to further optimize our implementation.

References

[1] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168, New York, NY, USA, 2006. ACM.

[2] 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. Eliot, 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: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, Portland, Oregon, USA, 2006.

[3] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, Vancouver, BC, Canada, 2004.

[4] Y. Chiba. Heap protection for java virtual machines. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 103–112, Mannheim, Germany, 2006.

[5] Y. Chiba. Java heap protection for debugging native methods. *Science of Computer Programming*, 70(2-3):149–167, 2008.

[6] D. J. Dimmich and C. L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[7] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *SIGPLAN Not.*, 40(6):62–72, 2005.

[8] M. Hirzel and R. Grimm. Jeannie: granting java native interface developers their wishes. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 19–38, Montreal, Quebec, Canada, 2007. ACM.

[9] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA'93 Garbage Collection Workshop*, Washington D.C., October 1993.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

[11] G. Kondoh and T. Onodera. Finding bugs in java native interface programs. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

[12] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] Manuel Chakravarty. Haskell 98 Foreign Function Interface 1.0. On-Line Documentation, 1998. http://www.cse.unsw.edu.au/ chak/haskell/ffi/.

[14] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, 2008.

[15] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 21–21, San Francisco, CA, 2004. USENIX Association.

[16] Standard Performance Evaluation Corporation. SPECjbb2005. On-Line Documentation, 2005. http://www.spec.org/jbb2005.

[17] Sun. Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine. On-Line Documentation, Last Retrieved: June 2007. http://java.sun.com/docs/hotspot/gc1.4.2.

[18] Sun Microsystems. JVM Crash Log Analysis.

http://forums.sun.com/thread.jspa?threadID=5369763.

[19] Sun Microsystems. OpenJDK. http://openjdk.java.net/.

[20] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang. Safe java native interface. In *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[21] D. Ungar. The Design and Evaluation of a High Performance Smalltalk System. *ACM Distinguished Dissertations*, 1987.