JVM-Portable Sandboxing of Java's Native Libraries

Mengtao Sun and Gang Tan

Lehigh University, Bethlehem, PA 18015, USA
mes310,gat208@lehigh.edu

Abstract. Although Java provides strong support for safety and security, native libraries used in a Java application can open security holes. Previous work, Robusta, puts native libraries in a sandbox to protect the integrity and security of Java. However, Robusta's implementation modifies the internals of OpenJDK, a particular implementation of a Java Virtual Machine (JVM). As such, it is not portable to other JVM implementations. This paper shows how to make the idea of sandboxing native libraries JVM-portable. We present a two-layer approach for sandboxing without modifying the internals of a JVM. We also discuss our experience of sandboxing Java's core native libraries. Experiments show that our approach of JVM-portable sandboxing incurs modest performance overhead on SPECjvm 2008 benchmark programs.

1 Introduction

The Java Native Interface (JNI) [1] is Java's foreign function interface. Through the JNI, Java code can invoke native libraries developed in low-level languages such as C, C++, or even assembly languages. The JNI allows Java programmers to reuse legacy code modules without porting them to Java. Furthermore, performance-critical portions of an application can be developed in C/C++ and invoked through the JNI.

However, native libraries in Java applications, as the "snake in the grass", is notoriously unsafe [2]. Native libraries in a Java application reside in the same address space as a Java Virtual Machine (JVM), but are outside the control of Java's security model. Java provides strong safety and security support, but once a Java application incorporates native libraries, there is no assurance about the safety and security of the whole application. Native libraries with programming bugs or malicious native libraries may cause an unexpected crash of the JVM, leak of confidential information, or even a complete takeover of the JVM by attackers.

To counter the threats of native libraries, our idea is to put them in a sandbox and allow only controlled access from the code in the sandbox to JVM services. Following this idea, we implemented Robusta [3], a security layer incorporated into a JVM for sandboxing native libraries in Java applications. It adopts software-based fault isolation (SFI [4]) to isolate untrusted native libraries from the rest of the JVM. Furthermore, native libraries' access to the outside

world is restricted to the JNI interface and OS system calls, both of which are modulated by Robusta's reference monitor to ensure security. Robusta's design will be briefly described in Sec. 2.

Robusta demonstrated the feasibility of sandboxing native libraries inside the implementation of a JVM. At the same time, a few questions were left unanswered when it was used to evaluate the practicality of sandboxing native libraries.

- **JVM portability.** Robusta was implemented inside OpenJDK 1.7. Various places in OpenJDK were modified. The JVM-specific implementation makes it hard to evaluate whether the idea of native-library sandboxing can function well in a different JVM implementation, such as IBM's J9 or Kaffe JVM. In fact, since IBM J9 is not open sourced, it is not possible for an outsider to modify its internals. Even for an open-source JVM, a JVM-specific sandboxing framework such as Robusta has to be upgraded whenever the JVM makes an upgrade. Indeed, Robusta was implemented in OpenJDK 1.7, but OpenJDK has upgraded from version 1.7 to 1.8.[1]
- **Java's core libraries.** The most common usage of native libraries is actually to support classes in the standard Java Class Library (JCL). Sun's JDK 1.6 has over 800,000 lines of C/C++ code in its native libraries. Robusta did not attempt to sandbox those core Java libraries. It was not clear how difficult it would be to sandbox those core libraries and not clear what the performance slowdown would be for Java applications after the core libraries are sandboxed.
- **Evaluation on standard benchmarks.** Robusta's experimental evaluation was performed on a set of handpicked, medium-sized JNI programs. While the experiments were acceptable as preliminary evidence of demonstrating Robusta's practicality, it would be much more convincing if it were evaluated on some standard benchmark programs. However, the challenge is that there are no standard benchmark suites that target the evaluation of JNI applications. Java benchmark suites such as SPECjvm and DaCapo [5] are themselves pure Java programs and cannot directly be used to evaluate Robusta.

In this paper we describe Arabica[2], a newly designed JNI sandboxing framework to further evaluate the practicality of JNI native-code sandboxing. Arabica has a JVM-independent design that requires no changes to a JVM's internals. It relies on a combination of the standard Java Virtual Machine Tool Interface (JVMTI) and a layer of stub libraries. Using Arabica, we have sandboxed several Java's core native libraries. This improves Java's security by reducing the size of trusted native libraries. More importantly, sandboxing core Java libraries enables us to evaluate the performance overhead of native-library sandboxing

[1] Google engineers are interested in integrating Robusta into Google App Engine. During a discussion, they explicitly mentioned that App Engine uses OpenJDK 1.8, but Robusta was implemented on version 1.7.

[2] Coffee Arabica is a species of coffee with better taste and quality than coffee Robusta.

by running standard Java benchmark suites, because all Java applications make heavy use of the core libraries.

The rest of the paper is organized as follows. We first introduce necessary background about the JNI and Robusta in Sec. 2. In Sec. 3, Arabica's JVM-independent design and implementation are presented. We then discuss our experience of sandboxing OpenJDK's standard native libraries in Sec. 4. Evaluation of Arabica is presented in Sec. 5. We discuss related work in Sec. 6, future work in Sec. 7, and conclude in Sec. 8.

2 Background: JNI and Robusta

The Java Native Interface (JNI) allows Java code to invoke native methods. A native method is declared in a Java class by adding the `native` modifier. The following code snippet of the `Inflater` class is extracted from the package `java.util.zip` in Sun's JDK. It declares a native `inflateBytes` method. Once declared, native methods are invoked in the same way as how ordinary Java methods are invoked. In the example, the `inflate` Java method invokes `inflateBytes`.

```
public class Inflater {
  ...

  public synchronized int inflate(byte[] b, int off, int len)
    { ...; return inflateBytes(b, off, len);}

  private native int inflateBytes (byte[] b, int off, int len);

  static {System.loadLibrary(``zip''); ...;}
}
```

A native method is implemented in a language such as C, C++, or assembly languages. The JDK implementation of `inflateBytes` above invokes the popular Zlib C library for the inflation (decompression) operation. There is also a small amount of native glue code between Java and the Zlib C library. The glue code uses JNI functions to interact with Java directly. Through these JNI functions, native code can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on.

Threats posed by native libraries. We list the most vicious kinds of attacks that can be launched by exploiting vulnerabilities in a native library:

(1) Unconstrained native libraries have access to the entire address space. As native libraries reside in the same address space as a JVM, bugs in native libraries can enable attackers to read and write the JVM's memory.
(2) Abusive JNI calls can cause integrity or confidentiality violations. The JNI interface was not designed with security in mind and does not mandate any security checks. Native code can steal confidential information from the

Java side, for example, by reading a private field of a Java object through the JNI API function `GetObjectField`. Native code can also violate Java's type safety. For instance, it can invoke `SetObjectField` to modify a field of an object to a value whose type is incompatible with the field's declared type, resulting in so-called type-confusion attacks [6].

(3) Native code may invoke an OS system call to read from a system file or send data to the network. This may violate the security policy that JVM imposes on a Java application.

As examples, vulnerabilities have been discovered in the C/C++ modules of Sun's JDK [7–9].

Robusta. We next summarize Robusta's design and implementation; details can be found in the Robusta paper [3]. First, Robusta adopts software-based fault isolation (SFI [4]) to isolate untrusted native libraries from the rest of a JVM. Native libraries are constrained within a sandbox so that direct memory access and control transfers outside of the sandbox are disallowed. The implementation of Robusta extends Google's Native Client (NaCl [10]), a state-of-the-art SFI implementation. Since native libraries are loaded dynamically by the JVM, Robusta extended NaCl with support for dynamic linking and loading. Second, Robusta interposes between Java and native libraries, inserting security checks into the JNI to prevent abusive JNI calls. Finally, Robusta connects to Java's security manager to mediate native libraries' system calls. An OS system call issued by a native library is rerouted to Java's security manager to decide on the system call's safety based on a predefined security policy. This design enables Robusta to place native libraries under the same runtime security restrictions as Java code and reuse much of Java's policy-driven security infrastructure.

The implementation of Robusta modified various places in OpenJDK 1.7. We next summarize these changes. We will use the phrase *Robusta OpenJDK* to refer to the OpenJDK after Robusta's changes.

(a) *Sandbox initialization.* When Robusta OpenJDK starts running, it constructs an SFI sandbox and loads the dynamic linker/loader (`ld.so` in Linux) into the sandbox. The dynamic linker/loader is put into the sandbox to support dynamic loading of native libraries.

(b) *Loading native libraries and symbol resolution.* When Robusta OpenJDK needs to load a native library, it invokes the `dlopen` routine of `ld.so` for loading the library into the sandbox. When Robusta OpenJDK needs to look up a symbol in a native library, it invokes the `dlsym` routine for resolving the symbol's address in the sandbox.

(c) *Calling a native method and returning.* When Java code invokes a native method, Robusta OpenJDK transfers the control to the address of the native method in the sandbox (after copying method arguments into the sandbox). The address is the result of symbol resolution through an invocation of `dlsym`. After the method finishes, Robusta OpenJDK transfers the control back to the Java code and copies out the return value (if there is one).

(d) *Support for Java multi-threading.* Multiple Java threads may be running inside the JVM. Robusta OpenJDK maintains a per-thread data structure to support Java multi-threading.
(e) *JNI safety checking.* Robusta OpenJDK inserts safety checks at the boundary of the JNI. For instance, if native code invokes `SetObjectField` to update an object's field, Robusta checks that the new value is of the expected type of the field.

3 Arabica: JVM-Portable Sandboxing of Native Libraries

Robusta's JVM-specific implementation limits its applicability. A much better design should provide sandboxing of native libraries as a service to a JVM. This requires the sandboxing functionality be implemented outside of the JVM and be compatible with a variety of JVM implementations.

What made us believe that JVM-portable sandboxing is achievable is because that almost all implementations of JVMs support two standard interfaces: the JNI and the Java Virtual Machine Tool Interface (JVMTI [11]). We have discussed the JNI, the standard interface between Java and native code. The second interface, JVMTI, is the standard JVM interface that allows an external tool to inspect the internal JVM state and control the running of applications in a JVM.

Therefore, our initial idea to achieve JVM portability was to design and implement a JVMTI-based tool. The idea almost worked until we discovered that JVMTI is not fine grained enough to meet all our demands. Sandboxing native libraries requires a greater control over a JVM than what the JVMTI interface allows. In retrospect, this is not surprising as JVMTI was mainly designed to support debuggers and profilers, not security tools.

In the end, Arabica achieves JVM-portable sandboxing through a two-layer approach: a JVMTI-based agent plus a layer of trusted stub libraries. Fig. 1 shows the architecture of Arabica. The first layer is a layer of trusted stub libraries. The stub libraries serve as intermediaries between the JVM and the real native libraries. Its functionalities include native-library loading, symbol resolution, and native-method calling and returning. The second layer is a JVMTI agent library. Its functionalities include sandbox initialization, support for Java multi-threading, and JNI safety checking.

The detailed design of Arabica is presented next in three steps: (1) a brief overview of JVMTI; (2) Arabica's JVMTI agent; (3) Arabica's stub-library layer.

JVMTI overview. JVMTI provides a programming interface that allows Java programmers to write tools to inspect and control the execution of a JVM. Such a tool, called a *JVMTI agent*, is loaded during initialization of a JVM. A JVMTI agent monitors and controls a JVM by calling JVMTI interface functions. JVMTI supports an *event-driven model.* An agent can register a callback function that is invoked when a certain type of events happens inside a JVM. For instance, a callback function can be registered for the thread-start event,

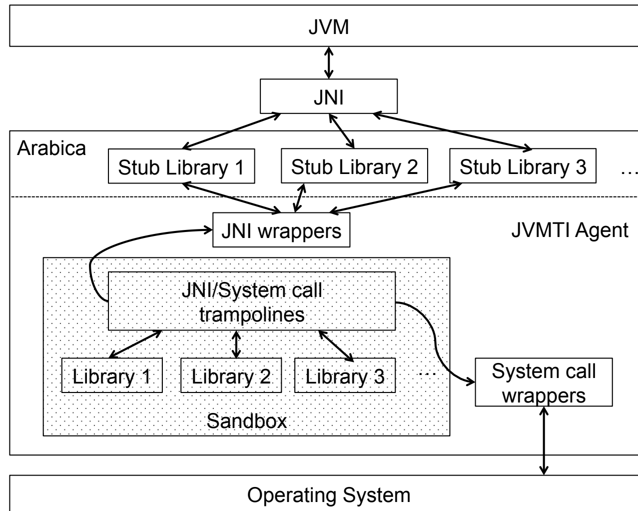

Fig. 1. Architecture of Arabica

which occurs when the JVM creates a new Java thread. The callback function can perform appropriate actions to support the implementation of a JVM tool.

Arabica's JVMTI agent. At a high level, Arabica's JVMTI agent implements those functionalities in items (a), (d), and (e), which were presented when we discussed Robusta's implementation at the end of Sec. 2.

For sandbox initialization, Arabica constructs an SFI sandbox in function `Agent_OnLoad`, which is part of the JVMTI agent. This function is automatically invoked by the JVM when the agent is loaded during the start of the JVM.

To support Java multi-threading, Arabica registers a callback function for the JVMTI thread-start event and also a callback function for the thread-end event. The callback functions are invoked whenever the JVM creates a new thread and terminates a thread, respectively. In the callback function for the thread-start event, a per-thread data structure is constructed to store information such as the per-thread JNI environment pointer [1]. The data structure is freed in the callback function for the thread-end event.

Finally, JNI safety checks are implemented in the following way. First, Arabica registers a callback function for the native-method-bind event, which occurs when the JVM binds a native method to the address of a native-library function that implements the native method. This callback function initializes the data structures that are necessary for performing JNI safety checks. Second, Arabica adds JNI interface function wrappers, which intercept JNI calls made by native code and perform necessary safety checks before invoking the real JNI functions in the JVM. The implementation of performing JNI safety checks follows the implementation of Jinn [12], a tool for detecting bugs and safety violations in the JNI code.

Arabica's layer of stub libraries. It turns out that a pure JVMTI-based approach is insufficient to achieve JVM-portable sandboxing of native libraries. The main reason is that JVMTI does not support a "native-library-loading" event. When a native library is loaded, a JVM loads the library into its memory using its own dynamic loader. Without the ability to intercept native-library-loading events, Arabica's JVMTI agent cannot change the process of library loading inside the JVM to allow it to load the native library into the sandbox via the dynamic loader installed inside the sandbox.

Arabica's solution is to introduce a level of indirection through a layer of trusted stub libraries. At a high level, the stub-library layer performs native-library loading, symbol resolution, and native-method calling and returning; that is, it performs functionalities (b) and (c) presented at the end of Sec. 2.

We next illustrate the basic process using the native-method implementation of `inflateBytes` in the library `libzip.so`. The first step is to rename `libzip.so` to `reallibzip.so`. The second step is to create a new `libzip.so`, a stub library for `libzip`. The stub library contains a stub function for each native-method implementation. The following code presents the implementation of the stub function for `inflateBytes`.

```
1  void * _handle = NULL;
2  void * _sym_addr = NULL;

3  jint Java_java_util_zip_Inflater_inflateBytes
4    (JNIEnv *env, jobject this, jarray b, jint off, jint len) {
5    if (_handle == NULL)
6      _handle = (void *) loadLib(env, ''reallibzip.so'');

7    if (_sym_addr == NULL)
8      _sym_addr = (void *)
9        loadSym(_handle, "Java_java_util_zip_Inflater_inflateBytes");

10    return call_in(_sym_addr, env, obj, b, off, len);
11  }
```

The following steps outline what happens when the JVM loads `libzip.so` and invokes the method `inflateBytes`:

(1) When the JVM loads `libzip.so`, it loads the stub version outside the sandbox, not the real one.
(2) When the JVM resolves the address for the native method `inflateBytes`, it finds the address of the stub function for `inflateBytes` in the stub library.
(3) When the JVM invokes the `inflateBytes` native method, the control transfers to the stub function. The stub function (a) uses `loadLib` to load the real library into the sandbox if it has not been loaded (lines 5 and 6); [3] (b) uses

[3] Note that the name of a native method in a library is mangled; additional information about package and class names are added to the method name.

loadSym to find the address of inflateBytes in the real library (lines 7–9); and (c) uses a function call_in to copy arguments and perform a function call to the real inflateBytes (line 10).

For each native library that needs to be sandboxed, a stub library needs to be generated. The stub library can be manually written, but we automated the process of stub-library generation using a stub-library generator. The generator first reads in a configuration file, which specifies the source files for native libraries, the output filename and other options. The stub generator then parses the source files to search for functions that implement native methods. For each native method, the stub generator records the function name, parameter list and return type, and generates the stub function accordingly.

Prototype implementation. Arabica has been implemented in Linux. Arabica's JVMTI agent is written in around 27,000 lines of C code. The majority of the code is for implementing the JNI function wrappers, which performs safety checks before invoking real JNI functions. When a JVM starts, the agent is loaded into the JVM by specifying the "-agentlib" or the "-agentpath" option. The stub-library generator is implemented in less than 1,000 lines of Java code. The current version of the stub generator works only with native libraries whose source code is available. The stub generator parses a source file, recognizes functions that implement native methods, and generates stub functions. A C function with the JNIEXPORT modifier is recognized by the generator as the implementation of a Java native method. In the future, we will change the stub generator so that it generates stub functions based on Java class files. The benefit of this approach is that it will enable sandboxing of native libraries in a Java package on the fly because stub libraries can be generated online based on the Java class files in the package—no off-line processing will be needed.

4 Sandboxing Standard Libraries in OpenJDK

Sandboxing the standard native libraries in the Java Class Library (JCL) provides multiple benefits. First, it improves a JVM's security. Without constraints, those native libraries are in the TCB. A security vulnerability in the libraries may enable attackers to take over the JVM. In an empirical security study [2], we examined the standard native libraries in Sun's JDK (version 1.6). In 38,000 lines of C code covered by the study, we identified 126 software bugs, of which 59 bugs are security critical. Since Sun's JDK 1.6 has over 800,000 lines of C/C++ code in its native libraries, we expect many more security-critical bugs are there. By sandboxing those libraries and constrain their capability, the size of the TCB is reduced and the JVM's security is improved.

The second benefit of sandboxing JCL's standard libraries is that it enables us to evaluate the performance of native-code sandboxing by running standard Java benchmark suites such as SPECjvm or DaCapo [5]. One difficulty in evaluating JNI-based systems is that there are no standard benchmark suites that target the JNI. As a result, the performance evaluation of Robusta was performed on

a set of handpicked JNI applications. With the sandboxing of standard native libraries, a more sound evaluation strategy can be adopted to evaluate Arabica. Since all Java applications make heavy use of JCL's native libraries, we can just sandbox those libraries and run standard Java benchmark suites for the evaluation.

Given the benefits of sandboxing standard native libraries, one natural approach is just to put all code in JCL's native libraries into the sandbox. However, this approach has two disadvantages.

- *Lack of portability.* In the ideal case, the JCL would be portable across JVM implementations. The reality, however, is that each JVM implementation uses its own version of native libraries. For instance, OpenJDK and IBM J9 come with their own JCL packages, which are incompatible with each other. One reason for the incompatibility is that many JCL native libraries may be used for purposes more than just implementing native methods declared in Java classes. Take `libzip` in OpenJDK as an example. Part of its code is invoked directly by OpenJDK's JVM, forming "native-to-native" communication.[4] For instance, the function `ZIP_Open` in `libzip` is directly invoked by OpenJDK during the JVM initialization stage. The second reason for incompatibility is that code in a native library may invoke JVM-specific intrinsics. For instance, native code that supports the `java.io` package in OpenJDK uses JVM intrinsics to manipulate files (e.g., `JVM_Open` for opening a file). Because of these reasons, if all code in a JCL native library were put in the sandbox, then the sandbox interface to Java had to go beyond the JNI interface to allow, for example, functions like `JVM_Open`. This approach would make the sandbox interface dependent on a specific JVM, while our goal is to keep the interface to be the portable JNI interface.
- *Security concerns.* Part of Java security is implemented through standard JCL packages such as `java.lang.SecurityManager`, `java.lang.ClassLoad -er`, and `java.security.AccessController`. Since Arabica has only one sandbox for all native code, putting native code that implements Java security in the same sandbox as other untrusted native libraries might jeopardize security: one vulnerability in untrusted native libraries might allow attackers to disable Java's security manager.[5]

Therefore, we decided to manually separate a native library in JCL into two portions: a portion that is put into the sandbox, and a trusted portion that is outside the sandbox. The sandboxed portion contains the code that implements native methods declared in a JCL class that is not part of Java's security infrastructure; it is JVM-independent and the only way it interacts with a JVM is through the standard JNI interface. The trusted portion contains the rest of

[4] We call this "native-to-native" communication because the JVM itself is implemented in native code. By contrast, "Java-to-native" communication includes the cases that Java code invokes a native method.

[5] One solution is to construct multiple sandboxes and put native code of different security levels into separate sandboxes, as discussed in the future-work section.

the code, including JVM-specific native code and native code that implements Java security. Take `libzip` in package `java.util.zip` as an example. A piece of native code is put into the sandboxed portion if the following conditions hold:

- There is a Java class in `java.util.zip` that is not part of Java security and the Java class declares a native method.
- The piece of native code is used to support the implementation of the native method.

Conceptually, the separation process takes out the JVM-independent, Java-security-independent portion from a JCL package. In our view, there is no fundamental reason why JCL packages cannot be reused in multiple JVMs. For instance, regardless of how a JVM is implemented, the standard package `java.util.zip` should include Java classes and a native library for compression/decompression; the library communicates with the Java side through the standard JNI interface. This would be a welcome design for JVM implementers as they do not need to reinvent those standard packages.

The manual separation process does come with a few complications. First, code occasionally needs to be duplicated in the two portions. For instance, if a function in a native library is both directly invoked by the JVM and used by another native function that implements a Java class's native method, then that function needs to be duplicated in both the sandboxed and the trusted portion of the library. Second, we may make mistakes during the manual separation process. For instance, code that should be sandboxed may be wrongly put into the trusted portion.

Separating JCL libraries. For the purpose of evaluating the performance of native-library sandboxing, we investigated what JCL libraries are used by benchmark programs in SPECjvm 2008. Table 1 lists these libraries except the AWT library. The AWT library contains over 100k lines of source code and we had difficulty of compiling it through NaCl's toolchain.[6] Fortunately, only one benchmark program, `sunflow`, in SPECjvm 2008 uses the AWT library and we excluded that program in our performance evaluation.

Table 2 shows the lines of source code of the JCL's libraries we have treated and the sizes after the manual separation process. Note that the total size is not the same as the sum of the sandboxed portion and the trusted portion because some code is duplicated during the separation process. Also note that a relatively large portion of `libjava` is not sandboxed because native code that implements Java security such as `java.lang.SecurityManager` is in that library. In our implementation, the trusted portion is put into the stub library, which was discussed in the previous section. That is, the new "stub library" contains both the stub functions and the trusted portion.

Adding permissions for the JCL libraries. Recall that Robusta reroutes system calls issued by native libraries to let Java's security manager decide

[6] Both Robusta and Arabica rely on NaCl's toolchain to create NaCl-compatible modules.

Library	Description
`libjava`	The core library that supports `java.lang`, `java.io` and part of `java.util`.
`libzip`	The library that supports `java.util.zip`; it includes the ZLib C library for compression/decompression.
`libnet`	The library that supports `java.net`.
`libnio`	The library that supports `java.nio`.

Table 1. Sandboxed JCL libraries and their descriptions.

Library	Total size	Sandboxed portion	Trusted portion
libnet	6339	6245	192
libnio	3566	3566	0
libzip	8725	8070	1325
libjava	11011	7919	3459

Table 2. JCL library sizes and the sizes of the two portions after manual separation (lines of source code)

whether system calls are allowed according to a security policy. Arabica inherited that mechanism from Robusta. For JCL packages, we changed the JVM's policy file to give a minimum set of permissions on a per-package basis. For instance, the package `java.util.zip` has permission `java.io.FilePermission` but no other permissions. As another example, the package `java.net` has permissions `java.net.NetPermission` and `java.net.SocketPermission`. Permissions assigned to a package apply to both Java and native code in the package and they are enforced by Java's stack inspection [13]. In particular, in the presence of native method calls, the JVM's method-call stack consists of a mixed Java and native frames. When the security manager performs stack inspection, it can find the right protection domain even for a native frame based on the class where the native method is declared. For instance, if the native code that supports `java.util.zip` attempted to access the network, the request would be rejected by the security manager since Java classes under `java.util.zip` do not have networking permissions.

Since Arabica has only one sandbox for all native libraries, one might worry that native code for one package might gain more permissions by exploiting other packages' native code that has a larger permission set. We do not believe this can happen for the following reason. The SFI sandbox for native libraries has separate code and data regions. The code region is immutable; as a result, one package's native code cannot modify other packages' native code. It can modify the data region, which is shared by all native code. But it cannot affect Java's security manager because the security manager stays outside of the sandbox and stack inspection is based on a stack outside of the sandbox. Note that when we

Program	Context switches (per millisecond)	Arabica increase (OpenJDK 1.7)	Arabica increase (IBM J9 1.7.0)	Robusta increase
zip (1KB)	18.50	23.90%	19.84%	9.64%
zip (2KB)	9.93	12.26%	10.41%	7.51%
zip (4KB)	5.00	6.36%	5.17%	5.22%
zip (8KB)	2.34	3.10%	2.75%	2.42%
zip (16KB)	0.95	1.31%	1.32%	1.40%
libharu	68.85	59.23%	58.86%	48.22%
libjpeg	0.002	8.43%	11.60%	3.82%
StrictMath	269.57	1588.81%	1647.83%	729.48%

Table 3. Performance overheads of Arabica on a set of JNI programs

sandboxed `libjava`, we did not put the native code of the security manager in the sandbox because otherwise it would create security-critical data in the data region modifiable by other untrusted native code.

5 Evaluation

We have evaluated Arabica's functionality, portability and performance using a set of microbenchmarks, a set of handpicked JNI programs, and SPECjvm 2008. All tests were performed on a system with Ubuntu 8.10 and an Intel Core2 Quad CPU at 2.66GHz. All experimental results we report is the average of ten runs. To evaluate Arabica's portability, experiments were conducted on two JVM implementations: OpenJDK 1.7.0 and IBM J9 1.7.0 R26.

Microbenchmarks for functionality testing. A set of small programs were used to test Arabica's basic functionality. The microbenchmarks include programs for testing basic JNI features, such as passing parameters of various types and sizes from Java to native code, calling back Java functions in native code, synchronization between Java and native code using `MonitorEnter` and `MonitorExit`, and others. The microbenchmarks also include programs for testing Arabica's effectiveness for preventing errors such as unsafe JNI calls. All microbenchmarks performed correctly on both OpenJDK and IBM J9.

A set of handpicked JNI programs. Our previous implementation, Robusta, was evaluated on a set of handpicked JNI programs. For comparison, we evaluated Arabica also on the same set of JNI programs. Table 3 presents these programs and the runtime increase of Arabica (on both OpenJDK and IBM J9) and Robusta.

In Robusta's experiments, we discovered that its runtime overhead is closely related to *context switch intensity*, which refers to how often an application

Benchmark	Context switches (per millisecond)	Arabica increase
compiler	4.38	7.49%
compress	0.10	11.89%
crypto	1.15	3.09%
derby	0.03	4.07%
mpegaudio	19.36	6.33%
scimark	0.02	9.49%
serial	3.10	4.86%
xml	55.60	112.12%

Table 4. Performance overheads of Arabica on SPECjvm2008

makes context switches between the JVM and the sandbox. Since each context switch comes with the cost of saving and restoring states and other costs such as JNI safety checks, the intuition is that the higher the context switch intensity, the higher the runtime overhead. This was confirmed by the experiment on the `zip` program. The Java side of the `zip` program compresses a file of a fixed size by dividing the file into data segments of small sizes and passing a data segment through a buffer to Zlib, which performs the compression and returns the result to the Java side. Then the Java side passes the next buffer of data to Zlib. Therefore, the bigger the buffer size, the smaller the number of context switches between Java and the sandbox, and therefore the smaller the performance overhead. The `zip` program was tested with buffer sizes 1KB, 2KB, 4KB, 8KB, and 16KB. As shown in the table, the runtime increase of Arabica on `zip` demonstrates the same trend as Robusta: as the buffer size increases, the performance overhead decreases. The `StrictMath` experiment is an extreme case. It repeatedly invokes native code for calculating mathematical functions such as `cos`. It stays in the sandbox for only a very short amount of time before switching out. Consequently, it has high context-switch intensity and thus high performance overhead.

Arabica has a higher overhead than Robusta. This is not surprising as Arabica uses JVMTI for portable sandboxing. JVMTI traces events that happen inside the JVM and therefore comes with extra overhead. We believe this is a reasonable price to pay for portability.

SPECjvm 2008. As presented in Section 4, we have manually separated the core JCL libraries used in SPECjvm 2008 (except for the AWT library). This enables a full evaluation of the performance overhead of native-library sandboxing since SPECjvm2008 contains Java benchmarks whose workload resembles realistic Java applications.

Table 4 presents the performance overheads of SPECjvm2008 benchmarks except for `sunflow`. The benchmark `sunflow` is a GUI program that uses the

Java AWT library; we have not yet sandboxed the AWT library, as noted before. All benchmarks were run ten times with sandboxed JCL libraries and another ten times with unsandboxed ones to calculate the average performance overhead. All benchmarks were run under the default configuration of SPECjvm 2008 (2-minute warming up time, one iteration with 4-minute iteration time).

On average, Arabica causes moderate overhead on most benchmarks (less than 15% except for `xml`). The `xml` benchmark makes a high number of invocations of the `StrictMath` library and incurs significant performance penalty.

In general, the experiments demonstrate that the idea of native-library sandboxing can be made portable across JVMs and with modest overhead, especially for programs with low context-switch intensity.

6 Related Work

We adopt SFI [4, 14–16, 10, 17] for sandboxing untrusted code in a trusted environment. Sandboxing is an intensively studied topic in computer security and can be achieved in many ways. One natural approach is an *OS-level solution*. A number of systems aimed to address the insufficiency of commodity-OS isolation primitives by implementing new OSes or augmenting OS kernels [18–22]. These systems map protection domains to OS processes. In comparison, SFI sandboxes untrusted code within the same address space and provides faster context switches between untrusted code and the trusted environment. *Virtual-machine based isolation* is both conceptually elegant and practically feasible (e.g., [23]). But it is even more heavyweight in terms of time, space, and communication costs than OS-level abstractions. Another approach is through *language-based isolation*, which is fine-grained, portable, and flexible. For example, languages such as E [24] and Joe-E [25, 26] enforce language-level isolation through an object-capability model. Their downsides are an overall loss of performance, and more importantly, a single language model has to be adopted. Isolation techniques using pure static types (e.g., [27]) have no runtime overhead, but require nontrivial support from developers and compilers. Finally, *hardware-level protection domains* within a single address space have also been explored [28, 29]. This approach is efficient, but is incompatible with commodity hardware on which most user applications run.

Arabica achieves portable sandboxing across JVMs. This is made possible by the availability of two standard interfaces supported by most JVM implementations: the JNI interface and the JVMTI interface. We believe it should be a desirable goal to design portable mechanisms for sandboxing untrusted code in other environments such as web browsers or other language runtimes (e.g., Python). For instance, Native Client [10] does not function in browsers other than Chrome; but if all browsers support some common interface such as the Pepper Plugin API [30], then browser-portable sandboxing should be obtainable with similar ideas we used to construct Arabica.

Sandboxing standard libraries like what we performed on Java's core libraries is always a good way of improving application security since all applications use

those libraries. A bug in a common library can impact a large number of applications. Previous work [31] demonstrated the security benefits of decomposing Python's runtime and libraries into a minimal, security-isolated kernel with a set of sandboxed modules for providing basic services such as networking and file I/O, similar to the micro-kernel approach in the operating system domain. Our sandboxing of Java's core libraries puts all native code in one sandbox, but a future direction is to construct one sandbox for each basic service the JVM supports.

7 Future Work

One immediate work we will perform is to support multiple sandboxes. Arabica constructs one sandbox for all native code because we are mainly concerned with protecting the JVM from native code. But the downside is that one native library may interfere with other libraries since they share the data region in the sandbox. Having only one sandbox is also the reason why we cannot put native code that implements Java security into the sandbox. We would like to construct one sandbox for each native library. This would isolate native code of varying trust levels.

We also plan to continue the process of sandboxing those JCL standard native libraries. Our experience for sandboxing the ones used by SPECjvm suggests that putting those native libraries into the sandbox causes only modest performance slowdown. The security benefits are substantial. In the long run, we would like to put most of the JCL's native libraries into the sandbox. The major challenge, as we have experienced, is to define a clear boundary and carve JVM-portable portions out of the libraries. The manual partitioning process we are using is slow and mistakes might be made. The approach of automatic partitioning based on static analysis seems promising.

8 Conclusions

Putting native libraries into a sandbox and constraining their capability is an effective strategy for preventing them from destroying Java's strong support for safety and security. Arabica demonstrates that this idea can be made JVM-portable with modest performance slowdown. Compared to Robusta, Arabica is much easier to be integrated into a Java environment as it does not modify a JVM's internals. We believe our techniques for making the sandboxing portable across environments and our experience of sandboxing core libraries will be helpful in other contexts, including sandboxing native libraries in languages such as Python and OCaml, and sandboxing plugins in web browsers.

Acknowledgments

We thank Martin Hirzel for suggesting the JVMTI approach for native-code sandboxing. This research is supported by US NSF grants CCF-0915157, CCF-

1149211, a research award from Google, and in part by National Natural Science Foundation of China grant 61170051.

References

1. Liang, S.: Java Native Interface: Programmer's Guide and Reference. Addison-Wesley Longman Publishing Co., Inc. (1999)
2. Tan, G., Croft, J.: An empirical security study of the native code in the JDK. In: 17th Usenix Security Symposium. (2008) 365–377
3. Siefers, J., Tan, G., Morrisett, G.: Robusta: Taming the native beast of the JVM. In: 17th ACM Conference on Computer and Communications Security (CCS). (2010) 201–211
4. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient software-based fault isolation. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP), New York, ACM Press (1993) 203–216
5. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2006) 169–190
6. McGraw, G., Felten, E.W.: Securing Java: Getting Down to Business with Mobile Code. John Wiley & Sons (1999)
7. Schoenefeld, M.: Denial-of-service holes in JDK 1.3.1 and 1.4.1_01. Retrieved Apr 26th, 2008, from `http://www.illegalaccess.org/java/ZipBugs.php` (2003)
8. US-CERT: Vulnerability note VU#939609: Sun Java JRE vulnerable to arbitrary code execution via an unspecified error (January 2007) Credit goes to Chris Evans.
9. US-CERT: Vulnerability note VU#138545: Java Runtime Environment image parsing code buffer overflow vulnerability (June 2007) Credit goes to Chris Evans.
10. Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: IEEE Symposium on Security and Privacy (S&P). (May 2009)
11. Oracle: JVM tool interface, version 1.0. `http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html` (2010)
12. Lee, B., Hirzel, M., Grimm, R., Wiedermann, B., McKinley, K.S.: Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In: ACM Conference on Programming Language Design and Implementation (PLDI). (2010) 36–49
13. Wallach, D.S., Felten, E.W.: Understanding java stack inspection. In: IEEE Symposium on Security and Privacy. (1998) 52–63
14. Erlingsson, Ú., Schneider, F.: SASI enforcement of security policies: A retrospective. In: Proceedings of the New Security Paradigms Workshop (NSPW), ACM Press (1999) 87–95
15. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: 15th Usenix Security Symposium. (2006)
16. Ford, B., Cox, R.: Vx32: Lightweight user-level sandboxing on the x86. In: USENIX Annual Technical Conference. (2008) 293–306
17. Castro, M., Costa, M., Martin, J.P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., Black, R.: Fast byte-granularity software fault isolation. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP). (2009) 45–58

18. Efstathopoulos, P., Krohn, M., Vandebogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, M.F., Morris, R.: Labels and event processes in the Asbestos operating system. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP). (2005) 17–30
19. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). (2006) 263–278
20. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP). (2007) 321–334
21. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: splitting applications into reduced-privilege compartments. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. (2008) 309–322
22. Watson, R., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: Practical capabilities for UNIX. In: 19th Usenix Security Symposium. (2010) 29–46
23. Cox, R.S., Gribble, S.D., Levy, H.M., Hansen, J.G.: A safety-oriented platform for web applications. In: IEEE Symposium on Security and Privacy (S&P). (2006) 350–364
24. Miller, M.: Robust composition: towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University, Baltimore, MD (2006)
25. Mettler, A., Wagner, D., Close, T.: Joe-E: A security-oriented subset of Java. In: Network and Distributed Systems Symposium (NDSS). (2010)
26. Krishnamurthy, A., Mettler, A., Wagner, D.: Fine-grained privilege separation for web applications. In: Proceedings of the 19th International Conference on World Wide Web (WWW '10). (2010) 551–560
27. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems **21**(3) (May 1999) 527–568
28. Witchel, E., Rhee, J., Asanović, K.: Mondrix: memory isolation for linux using Mondriaan memory protection. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP). (2005) 31–44
29. Neumann, P., Watson, R.: Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In: Fourth Layered Assurance Workshop. (2010)
30. : PPAPI. http://code.google.com/p/ppapi/wiki/Concepts
31. Cappos, J., Dadgar, A., Rasley, J., Samuel, J., Beschastnikh, I., Barsan, C., Krishnamurthy, A., Anderson, T.E.: Retaining sandbox containment despite bugs in privileged memory-safe code. In: 17th ACM Conference on Computer and Communications Security (CCS). (2010) 212–223