Bringing Java's Wild Native World under Control

MENGTAO SUN, GANG TAN, JOSEPH SIEFERS, and BIN ZENG, Lehigh University
GREG MORRISETT, Harvard University

For performance and for incorporating legacy libraries, many Java applications contain native-code components written in unsafe languages such as C and C++. Native-code components interoperate with Java components through the Java Native Interface (JNI). As native code is not regulated by Java's security model, it poses serious security threats to the managed Java world. We introduce a security framework that extends Java's security model and brings native code under control. Leveraging software-based fault isolation, the framework puts native code in a separate sandbox and allows the interaction between the native world and the Java world only through a carefully designed pathway. Two different implementations were built. In one implementation, the security framework is integrated into a Java Virtual Machine (JVM). In the second implementation, the framework is built outside of the JVM and takes advantage of JVM-independent interfaces. The second implementation provides JVM portability, at the expense of some performance degradation. Evaluation of our framework demonstrates that it incurs modest runtime overhead while significantly enhancing the security of Java applications.

Categories and Subject Descriptors: D.4.6 [**Software**]: Operating Systems—*Security and protection*; D.2.12 [**Software**]: Software Engineering—*Interoperability*

General Terms: Security

Additional Key Words and Phrases: Java native interface, Java virtual machine, software-based fault isolation

ACM Reference Format:
Sun, M., Tan, G., Siefers, J., Zeng, B., and Morrisett, G. 2013. Bringing Java's wild native world under control. ACM Trans. Info. Syst. Sec. 16, 3, Article 9 (November 2013), 28 pages.
DOI: http://dx.doi.org/10.1145/2535505

1. INTRODUCTION

Large software systems are typically *multilingual*, in which components are written in various programming languages. Oftentimes, high-level, type-safe programming languages interact with code written in low-level languages through *Foreign Function Interfaces* (FFIs). Among them are the Java Native Interface (JNI [Liang 1999]), the Python/C interface [Python/C FFI 2009], the OCaml/C interface [Leroy 2008], and many others. Multilingual software systems are convenient in practice. Developers can take advantage of the strengths of various languages and reuse a large amount of legacy code without reinventing the wheel. Moreover, performance-critical or computation-intensive components are usually implemented in low-level languages such as C or C++. As a result, a large number of Java applications contain some native code.

This research is supported by NSF grants CCF-0915157, CCF-0915030, CCF-1149211, and a research award from Google.
Authors' addresses: M. Sun (corresponding author), G. Tan, J. Siefers, and B. Zeng, Computer Science and Engineering Department, Lehigh University, Bethlehem, PA; email: mengtao.sun@lehigh.edu; G. Morrisett, School of Engineering and Applied Sciences, Harvard University, Cambridge, MA.

DOI: http://dx.doi.org/10.1145/2535505

While incorporating native-code components in applications written in type-safe languages brings convenience and performance, it is also a security black hole. Extensive use of native libraries in Java applications, as a “snake in the grass”, is notoriously unsafe. The fundamental reason why native code threatens Java security is that native libraries in a Java application reside in the same address space as the Java Virtual Machine (JVM), but are not subject to the control of the Java security model. Any native code included in a Java application has to be completely trusted. Hence, native libraries with vulnerabilities may result in privacy violations or a complete compromise of the JVM.

Given the threat of native code to the JVM, what are the possible approaches of limiting its damage? Before presenting our solution, we next discuss a few possibilities.

What about hardware memory protection? Can’t we avoid the security problem by putting native modules into separate processes? This provides some protection, but the main drawback is the high overhead due to context switches and interprocess communications. This is the reason why native code is in the same address space as the host language in all foreign function interfaces.

What about manual rewriting? One could manually rewrite native modules in Java or a safe C dialect such as Cyclone [Jim et al. 2002], but it is impractical to rewrite a large amount of C/C++ code manually.

What about automatic translation to Java? Tools such as c2j[1] can automatically translate subsets of C/C++ languages to Java, but do not support features such as pointer arithmetic, casting between unrelated types, unions, etc.

What about bug finders? Tools from companies such as Fortify and Coverity are useful for finding security-relevant bugs, but they usually target certain kinds of bugs, while ignoring others; neither do they enforce a security policy.

What about proof-carrying code? If native code is in the form of proof-carrying code [Necula 1997] or typed assembly languages [Morrisett et al. 1998], then its safety can be verified before execution. This approach would be ideal, but generating safety proofs showing that a native module respects the security of the JVM is nontrivial.

Our objective is to propose, implement, and evaluate a practical security framework that regulates native code in Java programs. For it to be practical, it must enforce a meaningful security policy, require little programmer effort, and bring only modest overhead. Our proposed framework builds on Software-based Fault Isolation (SFI), a low-overhead technique for isolating native code. Starting from SFI, the framework offers a security layer between the JVM and native code. For convenience, we will call this layer the NS (Native Sandboxing) layer hereafter. Figure 1 shows its high-level architecture. As shown in the diagram, SFI sandboxes are constructed for untrusted native libraries. It further restricts untrusted native code so that it interacts with the outside world through only the JNI interface and OS system calls, both of which are regulated by the NS layer to prohibit violation of a given security policy.

Our framework can be used to improve Java applications’ security in a number of scenarios. Figure 1 presents two scenarios. First, it can be used to sandbox core native libraries that support Java’s system classes in the standard Java Class Library (JCL). Oracle’s JDK 1.6 has over 800,000 lines of C/C++ code in its native libraries. These native libraries are critical to Java’s security. However, a previous study [Tan and Croft 2008] found 126 software bugs in a small subset of those native libraries (38,000 lines), of which 59 are security critical. Through our sandboxing framework, those

[1]http://tech.novosoft-us.com/.

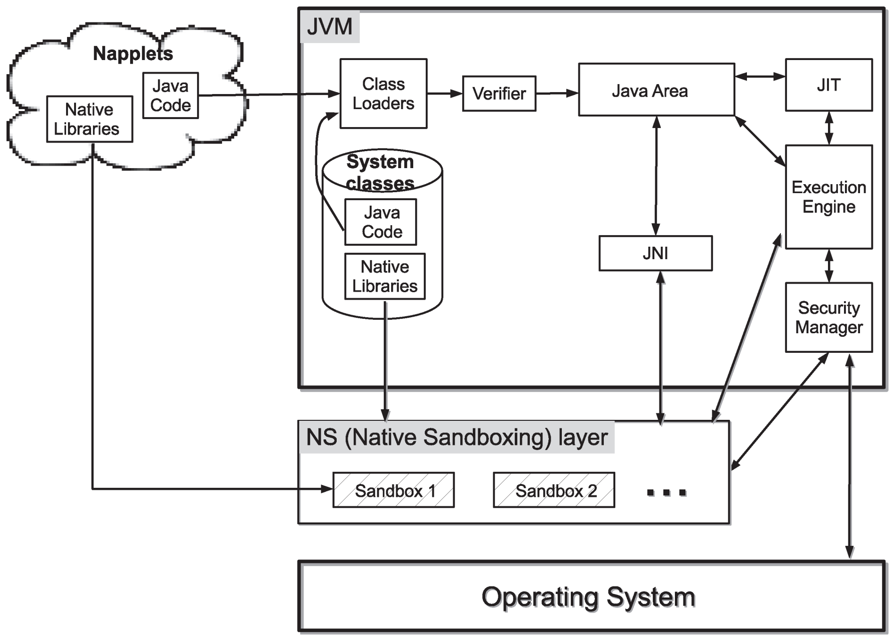

Fig. 1. High-level architecture of the sandboxing framework.

native libraries can be moved outside of the Trusted Computing Base (TCB). We will discuss our experience of porting several core native libraries into our framework in Section 7.

Our framework also enables the technology of *napplets* (native applets). Because of the inherent insecurity of native code, the default policy for Java applets is to disallow native code. Without constraints, the permission to allow native code in a Java applet is equivalent to granting all permissions to the applet. Our approach allows a Java applet to safely include native libraries with a meaningful policy enforced on those libraries.

The relationship of this article to previous conference publications [Siefers et al. 2010; Sun and Tan 2012] is as follows. Two contributions of this article are new: the support for multiple sandboxes (Section 5.3), and the support for Java napplets (Section 6). In previous designs, all native libraries are put into one single sandbox and as a result interference may exist between native libraries. Our new design instead constructs separate sandboxes for separate native libraries and limits each sandbox with a minimum set of permissions. Second, to support napplets, we design a new technique to enable online sandboxing of native libraries in a Java package. This requires online generation of stub functions for native libraries. Moreover, more experiments are added for this article. Previously, we could run only a subset of SPECjvm 2008 benchmark programs. With the sandboxing of `libawt`, we report the performance numbers for the entire SPECjvm 2008 suite.

2. BACKGROUND: JAVA NATIVE INTERFACE

The Java Native Interface (JNI [Liang 1999]) is Java's Foreign Function Interface. It is a programming framework that allows Java programs to communicate with native code. Through the JNI, Java applications can reuse legacy code written in C and C++ and can incorporate native components for platform-specific or performance-critical

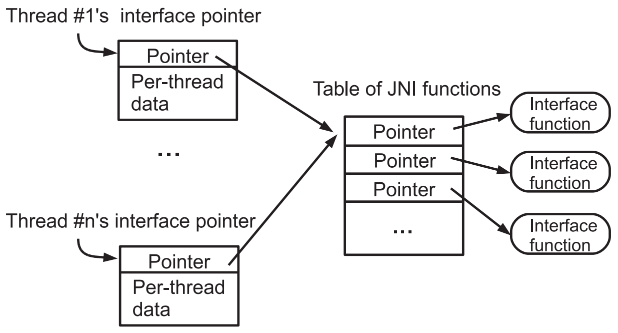

Fig. 2. JNIEnv interface pointers (from Liang [1999]).

features. A native method is declared in a Java class with the `native` keyword. As an example, the following code snippet is extracted from the package `java.util.zip` in Oracle's OpenJDK. It declares a native `updateBytes` method. Once declared, the native method can be invoked in the same way as other Java methods. In the example, the `update` Java method calls `updateBytes`.

```
public class CRC32 {
  private int crc;
  ...
  public void update
    (byte[] b, int off, int len)
      { ...; crc = updateBytes(crc, b, off, len);}

  private native static int updateBytes
    (int crc, byte[] b, int off, int len);

  static {System.loadLibrary(‘‘zip’’); ...;}
}
```

Native methods are usually written in unsafe languages such as C, C++, or even assembly languages. The implementation of `updateBytes` earlier invokes the popular Zlib C library to compute the CRC-32 checksum of a data stream. Between Java and the Zlib C library is a small amount of glue code, which interacts with Java using *JNI functions*. By issuing JNI functions, native code can retrieve information from the JVM or change its runtime state. A partial list of features provided by JNI functions is as follows.

—Perform read and write access to fields. For instance, `GetIntField` returns the value of an integer field of a Java object.
—Invoke Java methods. For instance, `CallIntMethod` invokes a method of a Java object and the method returns an integer.
—Perform various operations on Java classes, interfaces, objects, strings, and arrays. For instance, `FindClass` returns a reference to a class.
—Throw and handle Java exceptions. For instance, the `Throw` function throws a Java exception and `ExceptionClear` clears the pending Java exception.
—Perform synchronization between threads. `MonitorEnter` grabs the lock of a Java object and `MonitorExit` releases the lock of an object.

JNI functions are invoked through a *JNIEnv interface pointer*, which is visualized in Figure 2. Each thread has its own JNIEnv interface pointer, which points to a

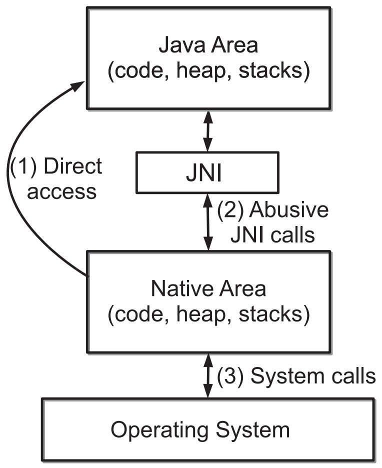

Fig. 3. Triple threats from native code.

struct that includes thread-local data and a pointer to an interface function table. The function table includes function pointers to JNI interface functions. A JNI function's entry in the table is at a predefined offset. As a simple example, suppose `env` is the pointer to the current JNIEnv interface pointer. Then native code can write "`(*env)->GetIntField(...)`" to retrieve the value of a field in an object.

Unfortunately, the design of JNI only provides very limited safety and security. For example, a JNI function may have undefined effects on the JVM when invoked with illegal arguments. The JNI specification discusses many programming pitfalls that JNI programmers should avoid, but there is no enforcement to prevent those pitfalls in JNI implementations. We will discuss some of these pitfalls and introduce a security layer that allows the JVM to obtain a level of security on native code.

3. THREAT MODEL, DEFENSE, AND PROTECTION STRENGTH

In this section, we present the threat model, an overview of defenses employed by our NS layer, as well as security protections it provides. In general, the NS layer can enforce polices despite the attacks described in the threat model.

3.1. Threat Model

As native code resides in the same address space as the JVM, vulnerabilities in native code such as buffer overruns and format string attacks may jeopardize Java programs. We focus on the most hazardous ways through which buggy or malicious native code may harm a Java program, as shown in Figure 3.

(1) Native code has access to the entire address space. Malicious native code can potentially read/write any Java object, resulting in confidentiality violation or a complete takeover of the JVM.
(2) Abusive JNI function invocations can also violate Java's confidentiality or integrity. The JNI does not enforce security; it is the programmers' responsibility to follow a set of recommended rules. The JNI specification explicitly lists 15 traps and pitfalls that have occurred in real projects [Liang 1999]. For example, native code can update the field of a Java object with values of unexpected types—a type-confusion attack.
(3) Native code is outside the control of Java's security manager and is free to issue system calls to manipulate files or send/receive network packets. Once a Java

program is allowed to load a native library, it is essentially granted full privileges because the native library is unrestricted by the security manager.

3.2. Defenses

Our NS layer defends against the triple threats listed before. First of all, it utilizes Software-based Fault Isolation (SFI [Wahbe et al. 1993]) to isolate untrusted native code into a separate, logical address space. In particular, native code is constrained within an SFI sandbox so that no direct memory access and control transfers outside of the sandbox are allowed. The NS layer builds on top of Google's Native Client (NaCl [Yee et al. 2009; Sehr et al. 2010; Ansel et al. 2011]), a cutting-edge SFI implementation currently deployed in Chrome Web browser and Chrome OS.

Second, the NS layer serves as a reference monitor that regulates JNI function calls. A JNI function call is allowed only if certain conditions are satisfied to preserve the security of the JVM. This is implemented by reserving a set of JNI trampolines in a nonwritable region of a sandbox. Those trampolines monitor attempts from native code to call JNI functions; all invocations are redirected to trusted wrappers outside of the sandbox where they go through security checks.

Last, the framework connects to Java's security manager to mediate system calls initiated by native code. Similar to JNI calls, native system calls are redirected to their corresponding system-call trampolines, which then invoke trusted wrappers outside of the sandbox to perform security checks. Specifically, system call wrappers invoke the `checkPermission` method in the `java.lang.SecurityManager` class to decide on the system call's safety based on a predefined security policy. This design extends Java's security infrastructure to the native world, enabling a uniform security policy configuration for both Java and native code.

3.3. Protection Strength

Our framework grants JVM administrators the ability to configure the security of native code. We next discuss what kinds of security policies it enforces to defend against the attacks described in the threat model.

First of all, the NS layer protects the integrity of the JVM. Native code should not change the stacks in the JVM and should perform only type-preserving modifications to the Java heap. Enforcing this integrity policy ensures that Java code is type safe even after native code's execution, therefore preventing type-confusion attacks. The integrity policy can be formalized in a way similar to previous work [Tan and Morrisett 2007]. Roughly, we can define a notion of well-typed Java states, which consists of a well-typed Java method-call stack and a well-typed Java heap. A well-typed Java heap is a collection of Java objects whose types are consistent with their runtime type tags. Then, we say that native code respects the integrity of the JVM if it transforms a well-typed Java state to another well-typed Java state.

Second, NS protects the confidentiality of the JVM. That is, native code accesses only objects reachable by the references provided by the JVM (as arguments of native methods) and the access-control modifiers (e.g., `private`) of fields and methods are respected. Enforcing this confidentiality policy makes sure that no confidential information is leaked to native code explicitly.

Third, NS constrains native code's access to system calls so that it conforms to a given policy. This rules out the possibility of issuing system calls by native code without being authorized.

At a high level, the NS layer enforces JVM integrity and confidentiality policies through SFI and by embedding safety checks into JNI-call wrappers. SFI makes sure native code can interact with the JVM only through a set of well-defined JNI functions. JNI checks further rule out JNI function calls that violate the JVM's security. Moreover, NS controls access to system calls by consulting Java's security manager in system-call wrappers.

It is worth mentioning that this article focuses on protecting Java from native code. Native code resides in its own logical address space and can still have buffer overruns within its address space. But our framework prevents attackers from exploiting such kinds of vulnerabilities to take over the JVM.

4. NATIVE CODE SANDBOXING

The NS layer regulates the behavior of native code by placing it into a separate "address space" through SFI and also monitoring its interaction with the rest of the system. This section presents the general design of how native-code sandboxing is achieved. We first discuss background information of Native Client (NaCl), on which we built NS. We then describe the high-level operations provided by the NS layer, but leave implementation details to the next section. We then discuss how JNI and OS system calls are supported.

4.1. Background: NaCl

Our framework uses SFI to isolate untrusted native code into a logical address space within the JVM address space. Without this low-level isolation as the basis, high-level security policies cannot be meaningfully enforced on native code. We utilize an industrial-strength SFI tool, NaCl [Yee et al. 2009; Sehr et al. 2010; Ansel et al. 2011], to constrain native code into a separate address space called the *NaCl address space*. The implementation of NaCl applies many ideas of previous SFI systems [Wahbe et al. 1993; Small 1997; Erlingsson and Schneider 1999; McCamant and Morrisett 2006; Ford and Cox 2008]. It is portable across multiple architectures and operating systems. Its runtime overhead is small (around 5–10%) thanks to clever uses of hardware features. We next highlight a few technical aspects of NaCl that are relevant to the following discussion.

Sandboxing unsafe instructions. The NaCl address space is separated into a nonwritable code region and a nonexecutable data region. NaCl disallows direct memory reads/writes outside of the data region and control transfers outside of the code region. This is achieved by inserting instructions to mask addresses before memory and control-transfer instructions. On x86-32, NaCl takes advantage of hardware segmentation to sandbox memory operations with extra masking instructions. NaCl also enforces every jump target to be jump aligned, following Pittsfield [McCamant and Morrisett 2006]. This restriction ensures that NaCl binaries can be reliably disassembled for static verification and that inlined masks cannot be bypassed by attackers.

Trampolines and springboards. Trampolines are the only ways through which control flow can be transferred out of the SFI sandbox. The NaCl runtime sets up system-call trampolines during the loading of NaCl binaries and relies on page protection to make the trampoline code immutable. Springboards allow crossings in the opposite direction, from trusted code to untrusted code.

NaCl toolchain. NaCl provides a modified GNU toolchain for generating SFI-compliant binaries and also provides profiling and debugging support. System calls in generated binary code are redirected to their corresponding trampolines, which are at fixed addresses and immutable. For security and portability, NaCl does not provide OS system calls directly but instead provides a restricted subset of system calls that are either emulated or implemented as wrappers on top of OS system calls.

A separate verifier. NaCl ships with a small verifier for validating the safety of NaCl binaries. The verifier disassembles NaCl binaries and uses static analysis to rule out unsafe binaries. The separation between the NaCl toolchain and the verifier reduces the size of the TCB; only the verifier needs to be trusted, and it is much smaller than the toolchain.

Support for dynamic linking/loading. To incorporate an SFI implementation into a JVM to sandbox native code, it must support dynamic loading and linking of native libraries. The reason is that a native library is dynamically loaded into the JVM after its associated Java class is loaded by a class loader. Before a native method is invoked for the first time, the JVM dynamically resolves the symbol that represents the native method to an address in a loaded library.

The original NaCl implementation supported only binaries with statically linked libraries. The first implementation of our framework had to generalize NaCl to support dynamic loading/linking and to support safety verification of dynamically loaded libraries. Interested readers can refer to our previous conference publication [Siefers et al. 2010] to see how dynamically linked libraries are supported through the use of the Non-eXecutable (NX) bit. Since our implementation, NaCl has also added support for dynamically linked libraries [Ansel et al. 2011].

4.2. Integrating Native-Code Sandboxing into a JVM

Let us first discuss how a regular JVM (without a framework for sandboxing native code) interacts with native libraries. A native library is usually loaded with the invocation of `System.loadLibrary(libname)` in a Java application. When a library-loading request is received by the JVM, it searches for the library in the library search paths. If successful, the library is loaded into the JVM's address space through the help of a dynamic loader. In Linux, this is realized by calling `dlopen` of `ld.so`—Linux's dynamic linker/loader. After a library is loaded, the Java application may call a native method implemented inside the library. To handle a native method call, the JVM first performs symbol resolution to find the address of the native method in the library, achieved by calling `dlsym` in `ld.so`. It then copies method arguments to the native call stack and transfers the control to the address of the native method. After the native method finishes, the JVM copies the return result to the Java stack and transfers the control back to the caller.

Our NS layer interposes between a JVM and native libraries. Whenever the JVM needs to interact with a native library, the interaction is intercepted by the NS layer and appropriate actions are performed. We next discuss the major operations that the NS layer performs, without delving into implementation details. The next section will present two implementations that adopt different strategies for realizing the NS layers. They intercept events and implement the following operations in different ways. As another note, in the following discussion we will assume only one sandbox is constructed for all native libraries; Section 5.3 will explain the case of multiple sandboxes.

(a) *Sandbox initialization.* When the JVM starts, NS reserves a memory region within the JVM address space for holding the SFI sandbox. The sandbox occupies a contiguous region of memory. Its initialization consists of several steps. First, a code and a data region need to be set up. The code region is configured to be readable and executable through memory page protection; and the data region is readable and writable. Second, trusted trampolines are installed at fixed addresses of the immutable code region. These trampolines include JNI trampolines and system-call trampolines (as well as a special trampoline called `OutOfJail`). They allow the flow of control to be transferred out of the sandbox in a controlled fashion (see Section 4.3 for more discussion). Finally, the initialization loads the dynamic loader into the sandbox (`ld.so` in Linux). The dynamic loader resides in the sandbox so that the NS layer can invoke its service routines to load a native library into the sandbox dynamically and to perform symbol resolution. The dynamic loader is not in the TCB because it is in the sandbox and the only way out of the sandbox is through known safe exits (i.e., trampolines).

(b) *Native-library loading.* When the NS layer intercepts a native-library loading event from the JVM, it instructs the dynamic loader in the sandbox to load the native library.

(c) *Symbol resolution.* Dynamic symbol resolution within sandboxed native libraries follows a similar pattern as native-library loading. When the JVM needs to look up the address of a native method in a library, the NS intercepts the event, and instructs the dynamic loader in the sandbox to perform a symbol resolution. The result of the symbol resolution is the address of the native method in the sandbox.

(d) *Calling a native method and returning*. After a native-method-call event in the JVM is intercepted, arguments are copied from the Java stack to the native stack by NS and the control is transferred to the address of the native method in the sandbox. The address is the result of symbol resolution in a previous step.

After the native method finishes execution, NS copies out the return value (if there is one) and continues the execution of the calling JVM thread. It cannot trust native code for remembering the *return information*, including the return address and the register state of the JVM. Instead, trusted code outside of the sandbox saves the state before the control is transferred into the sandbox and restores the state after the native method finishes.

Through the JNI, native code can call Java methods. This can result in a complicated "ping-pong" behavior between Java and native methods. For example, suppose a Java method m_j calls a native method m_c. The method m_c may call a second Java method m'_j. The method m'_j in turn may call a second native method m'_c, and so on. The resulting call stack is a sequence of Java-method and native-method frames.[2] To cope with the ping-pong behavior, NS associates the return information with a native-method frame so that when a native-method frame is popped from the stack the return information for that frame is used to continue the execution of the JVM.

(e) *Support for Java multithreading.* In the case of multiple Java threads, each Java thread begins life outside the sandbox and may pass freely in and out each time it makes a native call. Therefore, it is possible that multiple Java threads execute inside the sandbox concurrently. To support concurrent execution, the NS layer associates a data structure with each Java thread. The data structure stores per-thread information, including the location of the native stack in the sandbox for the thread. Each Java thread needs a native stack in the sandbox. However, not all Java threads use native methods. NS avoids performance penalties during Java thread creation by delaying the allocation of a native stack until the first time a thread attempts to enter the sandbox.

(f) *JNI and system-call checking.* The NS layer inserts safety checks at the boundary of the JNI and the system-call interface. We present how these checks are performed in the next two sections.

4.3. Strengthening JNI for Security

Abusive JNI calls may result in security violations or program crashes. For security, the JNI interface has to be strengthened. We explained before that native code invokes JNI functions through a `JNIEnv` interface pointer (see Figure 2), which is a double pointer to a function table. The function table contains a list of function pointers to JNI

[2]Note a native-method frame is still a frame inside the Java stack; it is for recording the invocation of a native method inside the JVM and it is different from a frame in the native stack, which is inside the sandbox.

functions provided by the JVM. Therefore, the C syntax "`(*env)->f(...)`" invokes a JNI function `f` through the interface pointer `env`.

Native code in the sandbox cannot access the real interface pointer because the function table contains pointers to functions outside the sandbox and the native sandbox is constructed so that the only way out of the sandbox is through a set of predefined trampolines. Our solution is to introduce a set of JNI trampolines and a proxy interface pointer inside the sandbox. The proxy interface pointer points to a function table containing pointers to those JNI trampolines. A JNI trampoline is introduced for every JNI function. The NS layer passes the proxy interface pointer as a fake JNI interface pointer to native code. As an example, when native code invokes "`(*env)->f(...)`", the control is transferred to the JNI trampoline for `f`, which jumps outside of the sandbox and invokes a trusted wrapper. The wrapper calls the real JNI function through the real interface pointer. In this design, native code is unchanged and still uses the same syntax for invoking a JNI function. Furthermore, the wrapper provides a natural place for inserting safety checks that prevent abusive JNI calls.

One more complication arises when dealing with multiple threads. The real interface pointer is a per-thread pointer, but for efficiency the NS layer's fake interface pointer is shared by all threads in the sandbox. This does not pose a problem because the thread-local data in the interface structure is used only by the JVM and not by native code. In order to support the correct behavior, a JNI wrapper looks up the real interface pointer for a particular thread from the per-thread structure recorded in the NS layer.

Safety checks are inserted in the JNI wrappers before real JNI functions are invoked. These checks are necessary to maintain the integrity and confidentiality of the JVM. The implementation of the safety checks follows Jinn [Lee et al. 2010], a tool for detecting bugs and safety violations in the JNI code. We next list the most important safety checks performed by JNI wrappers.

—*Exception checking*. A Java exception may be pending when native code is running. With a pending exception, native code is allowed to call only a small set of JNI functions (about 20 are allowed). Calling other JNI functions results in JVM crashes or other undefined behaviors. Therefore, checks must be there to ensure no exception is pending before calling those JNI functions.
—*Type checking*. JNI functions expect arguments of certain classes. Therefore, type checking must be performed to avoid calling JNI functions with arguments of wrong classes.
—*Access control*. When accessing fields/methods of a Java class, checks must be there to ensure access-control modifiers such as `private` are respected.
—*Nonnull checking*. Certain JNI functions cannot accept null arguments; so nonnull checking must be performed for those arguments.

Let us use a simple example to illustrate the safety checks. Native code can invoke a Java method through a JNI function. When native code invokes a Java method, the JNI wrapper checks if an exception is pending, checks if the method is allowed to be invoked by access-control rules, and checks if each argument is of the expected type. The last check requires a dynamic lookup of the Java method's type signature based on its method ID.

Another issue is about direct pointers to the Java heap. Java often passes references to an array of data to native code. The JNI allows efficient access to primitive arrays (i.e., arrays with primitive types such as `int`) and strings through direct pointers to the Java heap. This kind of direct access is enabled by a set of `Get/Release` JNI functions. For instance, given a reference to a Java integer array, `GetIntArrayElements` returns the address of the first element of the array (or a copy, decided by the JVM). Native code is then able to perform pointer arithmetic to access array elements in the usual way. After native code is finished with the array, `ReleaseIntArrayElements` releases the pointer.

Direct access to the JVM heap is dangerous and must be prevented. To accommodate the `Get/Release` JNI functions, the NS layer performs a copy-in and copy-out operation between the JVM heap and the sandbox's heap. In particular, when `GetIntArrayElements` is invoked, its wrapper allocates a buffer in the sandbox, copies the elements of the array into the buffer, and returns the buffer's address to native code. When `ReleaseIntArrayElements` is invoked, its wrapper copies the buffer's contents back into the original Java array. This technique redirects pointers referencing the JVM heap to the sandbox area. It incurs the extra runtime overhead of copying the referenced data in and out of the sandbox. However, there is no need for dynamic bounds checking for pointer access in the sandbox. Hence, NS compares favorably to SafeJNI [Tan et al. 2006], in which every array access comes with a dynamic bounds check.

One optimization can be used to reduce the copy-in and copy-out overhead. The implementation of `GetIntArrayElements` inside the JVM may already need to make a copy in the JVM heap. In that case, we can change the JVM so that it makes a copy directly in the sandbox, avoiding a second copying in its wrapper. If the JVM's garbage collector does not support pinning and is allowed to move objects in the Java heap, then a copy operation is inevitable because direct pointers to the array become invalid after the GC moves the array. As it turns out, OpenJDK 1.7 always makes a copy for `GetIntArrayElements`.

4.4. Managing Native System Calls

The basic idea of how the NS layer regulates native system calls is to consult Java's security manager. The security manager decides whether to deny a system call by referring to a security policy. The benefit of this design is that a single security mechanism regulates both Java and native code security and as a result it is sufficient to have a single security policy for an entire application of mixed Java and native code. This design also enables our system to reuse much of the infrastructure provided by Java security, including its policy specification framework and enforcement mechanism.

When enabled, the security manager of a JVM manages what system resources and privileges a program has access to by enforcing a predefined policy. Our framework extends the coverage of Java's security manager to the native world, regulating the system-access permissions of native code in the same way. For instance, a policy can grant a Java application the permission to read files, but prevent it from writing files. Our system can enforce such policies in the native libraries of a Java application. The enforcement is carried out in a similar way as how native JNI calls are checked.

—All system calls in native libraries are redirected to the system-call trampolines, which in turn invoke trusted system-call wrappers outside of the sandbox.
—The system-call wrappers invoke the `checkPermission` method of Java's security manager after constructing necessary permission objects. For the previous example policy, the `checkPermission` method will grant the access for a file read, but will throw a `SecurityException` for a file write.

Before proceeding, we make a few clarifications. First, the preceding design does not impede Java's stack inspection. In the presence of native method calls, the method-call stack interleaves Java-method and native-method frames. When the JVM performs stack inspection, it can find the right protection domain even for a native-method frame. Since a native-method frame is associated with a native method in a Java class, the JVM can find the protection domain based on the class.

Second, our framework disallows spawning native threads. The reason is twofold. First, creating native threads is strongly discouraged in the JNI [Liang 1999] because the native thread model may not match the Java thread model and the mismatch may cause synchronization problems between Java and native code. Second, creating a native thread might enable the new thread to have more privileges than the original native thread (unless something similar to *protection-domain inheritance* in Java is supported [Gong 2002]). For these two reasons, native code should call back to Java to create new Java threads, which is allowed.

5. IMPLEMENTATION

We describe two prototype implementations of the NS layer: Robusta and Arabica.[3] We start with Robusta, an implementation that modifies the internals of OpenJDK 1.7.0. We then introduce Arabica, which is a JVM-portable implementation of native-code sandboxing. Arabica achieves JVM portability by taking advantage of standard JVM interfaces, at the expense of some performance degradation.

5.1. Robusta: Directly Modifying a JVM

The first implementation of the NS layer is called Robusta [Siefers et al. 2010]. It modifies OpenJDK 1.7, which is a mature JVM implementation. Robusta is compiled separately into a shared library that OpenJDK loads during runtime. Various places in OpenJDK are changed to add hooks that intercept relevant events and transfer the control to routines defined in the Robusta library. Let us use the support for native-library loading (item b in Section 4.2) as an example. First, the part of code in OpenJDK whose functionality is to load a native library is located and removed. Second, a hook is added to the place so that, when OpenJDK needs to load a native library, the control is transferred to Robusta, which loads the library into the sandbox.

The implementation of Robusta also modifies the execution of those bytecode instructions that invoke a native method (e.g., `invokespecial` with a native method ID) so that they invoke native code in the sandbox. OpenJDK provides two implementations for the execution engine, a default ASM (assembly-code) template version and a (slower) C++ version. In order to fully evaluate Robusta, Robusta is integrated with the ASM template version.

Robusta's implementation includes seven OpenJDK hooks, and about 150 lines of C code for a utility module that is loaded into the sandbox during initialization. The utility module, named `dlWrappers`, provides a gateway for Robusta to access services housed within a native sandbox. Recall that the sandbox contains a copy of the dynamic loader/linker (`ld.so`). The utility module provides a set of service routines for convenient interaction with `ld.so`. For instance, the routine `dlopen_wrapper` is a wrapper for `dlopen`. When Robusta needs to load a native library, it invokes `dlopen_wrapper` in the sandbox. The wrapper then calls the actual `dlopen` method provided by `ld.so`, and propagates the resulting handle back to OpenJDK through calling a special trampoline called `OutOfJail`. The `OutOfJail` trampoline is set up during sandbox initialization and is invoked when native code finishes execution.

Robusta reuses the NaCl toolchain for generating SFI-compliant native libraries. It further reuses NaCl's verifier to validate safety of a native library before loading it into the sandbox.

[3]Both Coffee Robusta and Coffee Arabica are species of coffee. Robusta has a powerful flavor while Arabica has better quality and taste.

5.2. Arabica: JVM-Portable Native-Code Sandboxing

Robusta demonstrates the feasibility of sandboxing native libraries by modifying the internals of a JVM. However, its 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 is implemented in OpenJDK 1.7, but OpenJDK has upgraded from version 1.7 to 1.8.[4] From the perspective of end-users, 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 almost all JVM implementations support two standard interfaces: the JNI and the Java Virtual Machine Tool Interface (JVMTI [Oracle 2010]). 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, our second implementation of the NS layer, called Arabica [Sun and Tan 2012], achieves JVM-portable sandboxing through a two-layer approach: a layer of trusted stub libraries plus a JVMTI-based agent, as depicted in Figure 4. The first layer is a layer of trusted stub libraries. The stub libraries serve as intermediaries between the JVM and real native libraries. Their 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 multithreading, and JNI safety checking.

We next present Arabica 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, 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. In general, the JVMTI agent library in Arabica performs sandbox initialization (item a in Section 4.2), provides support for multithreading (item e), and performs JNI safety checking (item f).

For sandbox initialization, Arabica constructs a preconfigured number of NaCl sandboxes in function `Agent_OnLoad`, which is part of the JVMTI agent. This function is

[4]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 is implemented on version 1.7.

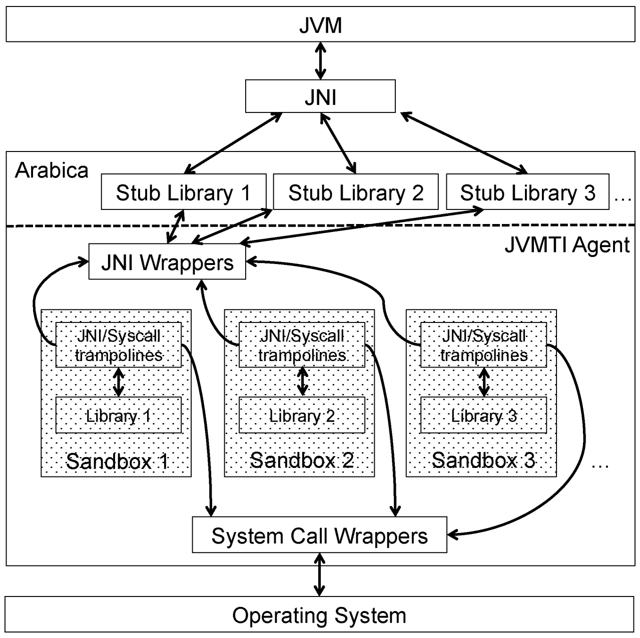

Fig. 4. The two-layer design of Arabica.

automatically invoked by the JVM when the agent is loaded during the start of the JVM. The initialization of a sandbox follows what we described before. Roughly, a memory region is reserved for the sandbox address space; a code and a data region are set up; trampolines including system-call trampolines and JNI trampolines are installed; the dynamic loader/linker is installed in the sandbox.

To support Java multithreading, 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 structure is constructed to store information such as the per-thread JNI environment pointer [Liang 1999]. The structure is freed in the callback function for the thread-end event.

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 (item b in Section 4.2), symbol resolution (item c), and native-method calling and returning (item d).

We next illustrate the basic process using the native-method implementation of `updateBytes` 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 `updateBytes`.[5]

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

3  jint Java_java_util_zip_CRC32_updateBytes
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_CRC32_updateBytes");

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

The following steps describe what happens when the JVM loads `libzip.so` and invokes the method `updateBytes`.

(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 `updateBytes`, it finds the address of the stub function for `updateBytes` in the stub library.
(3) When the JVM invokes the `updateBytes` native method, the control transfers to the stub function. The stub function first uses `loadLib` to load the real library into the sandbox if it has not been loaded (lines 5 and 6). The `loadLib` routine invokes the `dlopen` routine of the dynamic loader in the sandbox to load a native library.
(4) Following a similar pattern, the stub function then uses `loadSym` to find the address of `updateBytes` in the real library (lines 7–9).
(5) As the last step, the stub function uses a function `call_in` to perform a function call to the real `updateBytes` (line 10). The `call_in` service routine first copies parameters from the Java stack to the native stack in the sandbox and performs a function call to the real `updateBytes`.

For each sandboxed native library, a stub library needs to be generated. The stub library can be manually written, but Arabica automates the process of stub-library generation using a stub-library generator.

Arabica is implemented in Linux. Arabica's JVMTI agent is written in around 14,000 lines of C code. It has around 4,200 lines of code for implementing its core functionality, around 6,200 lines of code for JNI wrappers, and 4,100 lines of code inherited from Jinn for performing JNI safety checks. When a JVM starts, the agent is loaded into the JVM by specifying the “-agentlib” or the “-agentpath” option.

5.3. Support for Multiple Sandboxes

The implementations of Robusta and Arabica put all native libraries into one single sandbox. This is sufficient for protecting JVM integrity and confidentiality from untrusted native code. Even with only one sandbox, native code for one Java package cannot gain more permissions by exploiting other Java packages' native code that has

[5]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.

a larger permission set because of 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 libraries. But it cannot affect Java's security manager because the security manager stays outside of the sandbox and Java's stack inspection is based on a stack outside of the sandbox.

On the other hand, there are situations where constructing one sandbox for all native libraries is undesirable. For instance, native libraries that are downloaded from different Web sites may interfere with one another through the shared data region. Furthermore, having only one sandbox would also not allow us to put native code that implements Java security into the sandbox, as we will discuss in Section 7.

Therefore, we augmented the implementation of Arabica to support multiple sandboxes, which allows isolation between native libraries of varying trust levels. The new NS layer allocates a fixed number of sandboxes during JVM start-up time (alternatively, sandboxes could be allocated and initialized on demand). It maintains a centralized table for recording information of sandboxes, including whether a sandbox has been occupied by a native library, the sandbox state, and also a pointer to a data structure that maintains per-Java thread information to support multithreading inside the sandbox. When the JVM loads a native library, the library is loaded into a vacant sandbox. Symbol resolution and native method invocation are also adjusted to consult the central table first. For symbol resolution, the NS layer first finds which library the symbol belongs to, identifies the sandbox where the library has been loaded, and then uses the dynamic loader in the sandbox to resolve the symbol. In the case of multiple sandboxes, each sandbox has its own copy of the dynamic loader.

6. ENABLING JAVA NAPPLETS

One strong motivation of sandboxing native libraries is to enable the technology of Java napplets (native applets), which allows a remotely downloaded applet to have a mix of Java and native code in a safe way. In a conventional JVM, the default policy for Java applets is to disallow native code. Alternatively, an applet that includes native libraries can be signed by a trusted third party; a user of the applet is then prompted to give explicit permissions to the applet based on who signed the applet. The downside of this approach, of course, is to delegate security to a trusted third party. By contrast, napplets provide a safe way of mixing Java and native code without a trusted third party. JVM administrators can define a security policy for native libraries for each napplet. Native libraries in a napplet are then sandboxed to respect Java's security. For example, suppose a napplet is downloaded from a remote untrusted host and contains a fast mathematics native library as well as some Java code for the user interface. Through native-code sandboxing, the mathematics library is restricted to have no access to the local file system or network.

The original implementation of Arabica [Sun and Tan 2012] did not support napplets because of its stub-library generator. Recall that one layer in Arabica is its stub libraries, responsible for work such as native-library loading. The stub libraries cannot be part of a napplet because malicious stub libraries could be included and stub libraries are unconstrained as they are loaded outside the sandbox. Arabica included a stub-library generator, which parses Java source files to search for functions that implement native methods and generates stub functions based on native methods' names, parameter types, and return types. This stub-library generator can be used in an offline fashion to generate stub libraries when Java source code is available. However, in an environment where napplets are downloaded and executed, what is really needed is a stub-library generator that can generate stub libraries based on Java class files (i.e., bytecode) in an online fashion.

We have therefore implemented a new version of the stub-library generator. The new stub generator no longer requires the source code of native libraries to be available. Instead, the stub generator relies on Java class files to search for native methods. For each native method, the stub generator records the method signature and generates the stub method accordingly. This is possible because a Java class file contains signatures of all methods declared in the class, including those of native methods. The stub generator generates stub functions solely based on function names, parameter types, and return types; this information can be easily extracted from method signatures. The implementation of the stub generator is based on `javap`, a class file disassembler shipped with the JDK. `javap` disassembles a class file and extracts the package, methods, and fields of the classes. The new stub generator enables sandboxing of native libraries in a napplet package on-the-fly: stub libraries can be generated online based on Java class files in the napplet.

The new stub generator has been integrated into Arabica. The implementation extends the JAR File Specification by Oracle [1999] to allow SFI-compliant native libraries to reside under a fixed directory `lib/` in a Java napplet. In addition, we add to Arabica a special *harness* to execute Java napplets with native code sandboxed. The major steps of running a Java napplet under the new Arabica are as follows.

(1) The harness takes as input a napplet JAR file and obtains the entry point of the application.
(2) The harness extracts the napplet package and locates all Java class files.
(3) For each class file, the harness invokes `javap` to disassemble it and utilizes the stub generator to parse the output of `javap`, record the essential information, and generate stub methods.
(4) Finally, the harness builds the layer of stub libraries, compiles it, and starts the Arabica JVM with the Java program in the napplet as input.

The implementation can be easily adapted to the scenario where a Java napplet runs in a Web browser. The idea may also be beneficial to other scenarios, such as when isolating untrusted third-party native libraries in an Android application.

7. SANDBOXING STANDARD NATIVE LIBRARIES

The most common usage of native libraries in Java is actually to support classes in the standard Java Class Library (JCL). Oracle's JDK 1.6 has over 800,000 lines of C/C++ code in its native libraries. Sandboxing these standard native libraries in JCL provides multiple benefits. First, it improves Java security. Without constraints, those native libraries are in the TCB. A security vulnerability in the libraries may enable attackers to take over the JVM. By sandboxing those libraries and constraining their capability, the size of the TCB is substantially reduced and Java'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 [Blackburn et al. 2006]. One difficulty in evaluating JNI-based systems is that there are no standard benchmark suites that target the JNI. As a result, the experimental evaluation of Robusta was performed on a set of handpicked, medium-sized JNI programs. With the sandboxing of standard native libraries, a more sound evaluation strategy can be adopted. Even though SPECjvm and DaCapo contain pure-Java applications, we can just sandbox JCL's native libraries and run standard Java benchmark suites for evaluation since all Java applications make heavy use of those native libraries.

Given the benefits, one natural approach is just to put all code in JCL's native libraries into a sandbox. However, this approach has two major drawbacks.

—*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.[6] 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 has 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.ClassLoader`, and `java.security.AccessController`. Hence, putting native code that implements Java security in the same sandbox as other untrusted native libraries might jeopardize security: one vulnerability in untrusted native code might allow attackers to disable Java's security manager.

Therefore, we 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 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 among 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

[6]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 in which Java code invokes a native method.

Table I. Standard JCL Libraries and their Descriptions

LIBRARY	DESCRIPTION
`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`.
`libawt`	The library that supports Java's Abstract Window Toolkit (AWT).
`libjava`	The core library that supports `java.lang`, `java.io` and part of `java.util`; it is loaded every time the JVM starts. It also includes the `fdlibm`, the "Freely Distributable Math Library", which supports `java.lang.StrictMath`.

Table II. JCL Library Sizes (lines of source code)

Library	Total size	Sandboxed portion	Trusted portion
libzip	8725	8070	1325
libnet	6339	6245	192
libnio	3566	3566	0
libawt	6948	6775	173
libjava	11011	7919	3459

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 evaluating the performance of native-library sandboxing, we investigated what JCL libraries are used by benchmark programs in SPECjvm 2008. Table I lists these libraries.[7] Table II 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 Arabica's implementation, the trusted portion is put into the stub library. That is, the new stub library contains both the stub functions and the trusted portion.

Adding permissions for the JCL libraries. As described in Section 4.4, the NS layer forwards system calls issued by native libraries to let Java's security manager decide whether system calls are allowed according to a security policy. 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 [Wallach and Felten 1998]. A native-method call results in a native frame in Java's stack. 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.

[7]In OpenJDK, native libraries are under directories `jdk/src/share/native`, `jdk/src/solaris/native`, and `jdk/src/windows/native`. There are about 350,000 lines of C/C++ code in the first two directories. The majority of the code is under their sun subdirectories, which is not part of the Java standard packages and not used by SPECjvm programs. The five libraries in Table I account for about 70% of the code not under the sun subdirectories.

8. EVALUATION

We evaluated the sandboxing framework to test its functionality and efficiency. We will discuss mostly the evaluation of Arabica, the portable sandboxing framework. Robusta will also be discussed in places where comparison between Arabica and Robusta is meaningful. We note that the performance numbers in this section are slightly different from the numbers in previous conference publications because the experiments were rerun with the support of multiple sandboxes and also because we cleaned up the implementation.

The evaluation was carried out in multiple steps. We first verified the functionality of the framework using a set of microbenchmarks. Second, we evaluated the performance overheads of the sandboxing framework with a set of handpicked JNI programs. Third, we evaluated the overall performance of the framework with SPECjvm 2008. Finally, we converted a set of JNI programs to Java napplets and evaluated their performance on Arabica. All tests were performed on a system with Ubuntu 8.10 and an Intel Core2 Quad CPU at 2.66 GHz. All tests were run ten times and we took the arithmetic mean of the 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 the basic functionality of the sandboxing framework. The microbenchmarks included 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 included programs for testing the effectiveness of the framework for preventing errors such as unsafe JNI calls. All microbenchmarks performed correctly on both OpenJDK and IBM J9.

A set of handpicked JNI programs. The runtime overhead of the sandboxing framework can roughly be put into several categories.

—First, there is the SFI cost. For NaCl, this is the cost of masking indirect jump instructions and the cost of making the program properly aligned at 32-byte blocks. NaCl has been reported to incur 5% of overhead on x86-32 for SPEC2000 programs [Yee et al. 2009].
—The second class of runtime overhead happens during *context switches*. With native code sandboxed, the execution context may switch between the JVM and the sandbox in a number of situations: when the JVM invokes a native method, the context is switched into a sandbox; when native code finishes execution, the context is switched outside the sandbox; when native code invokes a JNI call or a system call, the context is switched outside of the sandbox to invoke trusted wrappers and is then switched back into the sandbox. Each context switch comes with the cost of saving and restoring states, and other costs depending on the kinds of context switches (e.g., the cost of safety checking in JNI calls and the cost of invoking the security manager in system calls).
—The implementation of Arabica additionally comes with the overhead of using the JVMTI interface for portable sandboxing. JVMTI traces events that happen inside the JVM and therefore comes with extra overhead.

The runtime overhead of both Robusta and Arabica depends greatly on how many context switches a program makes. If a program stays in the sandbox for a long time without performing a context switch (for example, in computationally intensive programs), then the runtime overhead should be small. On the other hand, if a program makes frequent context switches between Java and native code, then there

Table III. Performance Overheads of Robusta/Arabica on a Set of JNI Programs

Program	Context switches (per millisecond)	Robusta increase	Arabica increase (OpenJDK 1.7)	Arabica increase (IBM J9 1.7.0)
zip (1KB)	18.50	9.6%	25.7%	24.3%
zip (2KB)	9.93	7.5%	12.9%	12.6%
zip (4KB)	5.00	5.2%	5.9%	7.1%
zip (8KB)	2.34	2.4%	3.0%	3.7%
zip (16KB)	0.95	1.4%	0.9%	2.3%
libharu	68.85	48.2%	59.2%	58.9%
libjpeg	0.002	3.8%	5.7%	14.9%
StrictMath	269.57	729.5%	1320.7%	1335.9%

should be a significant runtime overhead. Therefore, an interesting question is to *explore the relationship between the runtime overhead and the frequency of context switches*. An answer helps to understand what kinds of applications should be put under the control of the sandboxing framework.

We compiled a set of medium-sized JNI programs, explained as follows.

—Java classes in `java.util.zip` invoke the popular Zlib C library for performing general-purpose data compression/decompression. We extracted from OpenJDK the Java classes in `java.util.zip`, the Zlib 1.2.3 library, and the JNI glue code that links Zlib with Java.
—Java classes in `java.lang.StrictMath` invoke native methods implemented in `fdlibm`, the C "Freely Distributable Math Library". The library implements basic mathematical functions such as sine, cosine, and tangent.
—`libharu` is an open-source C library for PDF creation. As it does not ship with JNI bindings, we created our own.
—We created JNI bindings to interface with the `libjpeg` library, which provides JPEG compression.

The experiments were set up as follows.

—*zip*. Experiments were set up to compress files with varying buffer sizes. The zip program compresses a file by dividing the file into data segments of small sizes. Its Java side passes 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 number of context switches differs significantly with different buffer sizes. We tested the zip program with buffer sizes 1KB, 2KB, 4KB, 8KB, and 16KB.
—*StrictMath*. Experiments were set up to invoke library functions in the `fdlibm` math library repeatedly.
—*libharu*. Experiments were set up to generate a 100-page PDF document from a sample text file.
—*libjpeg*. Experiments were set up to convert a 5Mb BMP image into the JPEG format.

Table III presents the experimental results for both Robusta and Arabica. In the case of Arabica, the results are separately presented for OpenJDK and IBM J9. As we can see from the table, the runtime overhead correlates strongly with the context-switch intensity. In the `zip` benchmark, as the context-switch intensity decreases from 18.50 (with the buffer size 1KB) to 0.95 (with the buffer size 16KB), the performance overhead also decreases from 9.64% to 1.40% in Robusta and from 25.68% to 0.87% in Arabica under OpenJDK. `StrictMath` is an extreme case. It stays in the sandbox for only a very short amount of time for calculating the result of a single mathematical function before

Table IV. Performance of Arabica on SPECjvm2008

Benchmark	Context switches (per millisecond)	Arabica increase
compiler	4.38	0.3%
compress	0.10	14.1%
crypto	1.15	2.4%
derby	0.03	2.0%
mpegaudio	19.36	7.0%
scimark	0.02	1.1%
serial	3.10	3.5%
xml	55.60	61.6%
sunflow	61.91	30.1%

switching out. Consequently, it has high context-switch intensity and therefore high performance overhead.

The experiments also show that Arabica has a higher overhead than Robusta. This is not surprising as Arabica uses JVMTI for portable sandboxing. We believe the extra overhead is a reasonable price to pay for portability.

In general, the results show that native-code sandboxing is best used for applications that do not have intensive levels of context switching. For those applications where it is possible to control context-switch intensity (such as `zip`), we suggest increasing the amount of time that the applications stay in the sandbox before switching out.

SPECjvm 2008. As presented in Section 7, we manually separated the core JCL libraries used in SPECjvm 2008. This enabled a full evaluation of the performance overhead of native-library sandboxing since SPECjvm 2008 contains Java benchmarks whose workload resembles realistic Java applications.

Table IV presents the performance overheads of SPECjvm2008 benchmarks. 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). Note that the previous conference publication did not report the result for `sunflow` because `sunflow` used the `libawt` library, which was not sandboxed before.

On average, Arabica caused moderate overhead on most benchmarks (less than 15% except for `xml` and `sunflow`). The `xml` and `sunflow` benchmarks made frequent invocations of the `StrictMath` library and incurred significant performance penalty.

Java napplets. We manually created a set of command-line Java napplets by packaging the handpicked JNI programs used in table III. All programs ran correctly under Arabica with similar performance overhead as those presented in the table. These experiments showed the feasibility of the idea of Java napplets.

In general, the experiments demonstrate that the idea of native-library sandboxing is practical and performs favorably when used for applications with low context-switching frequency. Moreover, the sandboxing framework can be made portable across JVMs with a reasonable overhead.

9. RELATED WORK

We divide related work into two categories: safety and security of language interoperation, and techniques for isolating untrusted code in a trusted environment.

Safety and security of language interoperation. Most Foreign Function Interfaces (FFIs) are designed for the purpose of interoperating with native code, but do not consider how the interoperation might impact the host language's safety and security. The JNI does not mandate any checking of native methods. On the other hand, native methods are notoriously unsafe and are a rich source of software errors. Recent studies have reported hundreds of interface bugs in JNI programs [Furr and Foster 2006; Tan and Croft 2008; Kondoh and Onodera 2008].

A number of systems have been designed to improve and find misuses of the JNI interface. They are classified into three categories: (1) Jeannie [Hirzel and Grimm 2007] is a new interface language that allows programmers to mix Java with C code and a Jeannie program is then compiled into JNI code by the Jeannie compiler. (2) Several recent systems employ static analysis to identify specific classes of errors in JNI code [Furr and Foster 2006; Tan and Morrisett 2007; Kondoh and Onodera 2008; Li and Tan 2009]. (3) Jinn [Lee et al. 2010] generates dynamic checks at the language boundary to find interface errors. These systems have improved the JNI's overall safety by reducing errors in the JNI code, but they are not designed to enforce a security policy as our framework does.

SafeJNI [Tan et al. 2006] is in spirit closest to our NS layer in that they both protect Java from untrusted native code. However, SafeJNI is based on CCured [Necula et al. 2002], which performs source-code rewriting for security. Our system is based on native-code sandboxing and is more efficient.

Another related system is a JVM heap-protection mechanism designed by Chiba [2006]. It modifies an OS kernel to provide a notion of per-thread protection domains. A domain is a memory region that a thread can access and the domain boundary is enforced through hardware page protection. In addition, it modifies the JVM so that, when a thread in the JVM is about to execute a native method, the thread's protection domain is switched to a memory region that is disjoint from Java's heap. As a result, if the native method writes to memory in the Java heap, a hardware trap is generated. The main goal of Chiba's system is for debugging (reporting when invalid memory references happen in native code). In contrast, our system's goal is security and therefore it also regulates JNI calls and connects to Java's security manager. Further, our system does not require the modification of the OS kernel.

The problem of sandboxing the JVM's native libraries is analogous to the problem of sandboxing unmanaged code in a .NET implementation. Klinkoff et al. implemented a sandboxing mechanism for protecting managed code and the .NET runtime from unmanaged code [Klinkoff et al. 2007]. Although in a different context, their system addresses similar security problems as our framework does. Their system puts unmanaged code into a separate process and seems to suffer from high performance overhead due to interprocess communication. In addition, system calls in unmanaged code are intercepted and regulated by a kernel add-on module. By contrast, our framework is more efficient thanks to SFI and it is a pure user-space implementation. In another related system, Ansel et al. [2011] sandboxed the language runtime of Mono, an open-source .NET implementation. Their emphasis is on enforcing SFI for dynamically generated code and self-modifying code.

One of our implementations of the NS layer, 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 [Yee et al. 2009] does not function in browsers other than Chrome; but if all browsers support some common interface such as the Pepper Plugin API[8], 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 [Cappos et al. 2010] 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 microkernel approach in the operating system domain.

Isolating untrusted code in a trusted environment. A long-standing line of research in computer security is to isolate untrusted code in a trusted environment. Our NS framework adopts the approach of SFI, for its low-level overhead, simplicity, and verifiability. But untrusted code isolation can be achieved in a variety of ways. One common approach is through *language-based isolation*, which is fine-grained, portable, and flexible. For example, isolation techniques using pure static types (e.g., Morrisett et al. [1999]) have no runtime overhead, but require nontrivial support from developers and compilers. As another example, languages such as E [Miller 2006] and Joe-E [Mettler et al. 2010; Krishnamurthy et al. 2010] enforce language-level isolation through an object-capability model. Their downsides are that a single language model has to be adopted and it is unclear how to apply it at the native-code level. Another natural approach for untrusted-code isolation is an *OS-level solution*. In fact, operating systems have long used hardware-based protection to isolate one process from another. For instance, Nooks [Swift et al. 2004] isolates device drivers from kernel code, and Xax [Douceur et al. 2008] isolates Web applications from browsers. There are also a number of systems aiming at addressing the insufficiency of commodity-OS isolation primitives by implementing new OSes or augmenting OS kernels [Efstathopoulos et al. 2005; Zeldovich et al. 2006; Krohn et al. 2007; Bittau et al. 2008; Watson et al. 2010]. 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. A recent system called Dune [Belay et al. 2012] also provides a mechanism for sandboxing untrusted code within the same process. It includes a Linux kernel patch that takes advantage of virtualization hardware so that a user process can safely access privileged hardware features including ring protection and page tables. These features enable the building of a user-level sandbox with fast context switches. Our system relies on SFI, which is more portable because it does not require a kernel patch.

Yet another approach for isolating untrusted code is through *virtual machines* (e.g., Cox et al. [2006]). But it is even more heavyweight in terms of time, space, and communication costs than OS abstractions. Finally, *hardware-level protection domains* within a single address space have also been explored [Witchel et al. 2005; Neumann and Watson 2010]. This approach is efficient, but is incompatible with commodity hardware on which most user applications are running.

In terms of mediating system calls, our framework is related to a number of previous efforts such as systrace [Provos 2003] and many others (e.g., Goldberg et al. [1996], Ioannidis et al. [2002], and Garfinkel et al. [2004]). A Linux kernel patch called seccomp-bpf [Drewry 2012] allows the filtering of system calls following the idea of the packet filtering mechanism in the network layer. By contrast, our framework delegates the job to Java's security manager. We believe it is a general strategy of how system calls in native code can be handled in language virtual machines.

[8]http://code.google.com/p/ppapi/wiki/Concepts.

10. FUTURE WORK

Experiments show that our sandboxing framework incurs small overhead for most programs, but there is still room for improvements. The extreme case of the `StrictMath` library shows that perhaps a two-layered approach of static verification and sandboxing is beneficial. The idea is to rewrite the performance-critical portion of a native library in a safe dialect of JNI that is statically verified to be safe, and put the rest of the library into an SFI sandbox. The safe dialect can include a safe dialect of C (e.g., Cyclone [Jim et al. 2002]) as well as those JNI interface functions augmented with static typing for safety. A quick inspection of the `StrictMath` library suggests that the whole library can be rewritten in a safe dialect of JNI and can therefore stay outside the sandbox. The challenge is how to design the safe dialect so that it requires small effort for rewriting or even enables automatic translation to the safe dialect.

One interesting direction is to explore techniques for stronger security policies within our native-code sandbox. Our sandbox already prevents code-injection attacks in native libraries because all code is statically verified before execution and no memory region is both writable and executable (these invariants are also maintained during dynamic loading). On the other hand, it does not prevent exploits of vulnerabilities using code snippets already in the code region (e.g., return-to-libc attacks or return-oriented programming [Shacham 2007]). Control-Flow Integrity (CFI, [Abadi et al. 2005]) can prevent a large number of attacks that are based on illegal control transfers. Given an untrusted module, CFI predetermines its control-flow graph. The control-flow graph serves as a specification of the legal control flow allowed in the module and CFI inserts runtime checks to enforce the specification.

Our framework relies on NaCl's compiler toolchain to produce SFI-compatible binaries from source code of native libraries. Recent advances make substantial progress toward an implementation through pure binary rewriting [Wartell et al. 2012]. With sacrifice of some performance, this advance can possibly enable the rewriting of native libraries even when source code is unavailable. It remains to be seen whether the pure binary rewriting approach scales (e.g., the SFI implementation by Wartell et al. [2012] does not instrument return instructions, meaning it is unsafe in concurrent code).

The goal of our framework is to apply Java's security model to native libraries. On the other hand, an implementation of the JVM may contain bugs that enable attackers to bypass Java's security model, as witnessed by recent high-profile Java zero-day exploits [MITRE 2012, 2013]. Those exploits are possible partly because in some cases invocations of Java's reflection API allow the security manager to be bypassed.[9] It would be interesting to explore whether attackers could escalate privileges in similar ways by exploiting capabilities in native libraries included in critical JDK classes and, if so, what the defense would be.

We also plan to migrate our framework to other platforms. For instance, Android contains a specialized JVM called Dalvik Virtual Machine (DVM). It provides support for the JNI and uses standard Linux security features including user-IDs to isolate Android applications. Previous empirical studies have shown that Android applications may come with third-party native libraries. Although the permission mechanism of Android is built into the OS kernel and covers both Java and native code, buggy or malicious native code can still damage Java code's integrity and confidentiality. Furthermore, an Android application may be overprivileged, giving native libraries more permission than they require. For example, an application may require a permission to access the GPS location information for its Java part, but its native libraries may not

[9]Specifically, certain security manager checks are bypassed depending on the immediate caller's class loader.

need that permission. Our framework can be extended to cover the Android platform to constrain permissions of native libraries within a single process address space.

We believe our framework can be extended to other languages to enable safe language interoperation. The mechanisms we have explored for sandboxing native libraries in the JNI are clearly applicable to other foreign function interfaces, including the Python/C interface and the OCaml/C interface.

11. CONCLUSIONS

Native code in type-safe languages is always a security dark corner and introduces security risks. We have presented a complete security framework to constrain the behavior of native code in SFI sandboxes. We have also proposed two implementation strategies that demonstrate the approach's efficiency and portability. We believe our technique and experience in sandboxing native libraries will be beneficial in other contexts, including sandboxing native libraries in languages such as Python and OCaml, sandboxing plugins in Web browsers, and sandboxing native libraries in Android applications.

ACKNOWLEDGMENTS

We thank Mark Seaborn for explaining the NaCl implementation. We thank Martin Hirzel for suggesting the JVMTI approach for native-code sandboxing. We also thank the anonymous reviewers and the associate editor for their informative reviews.

REFERENCES

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 340–353.

Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D., Sehr, D., Biffle, C., and Yee, B. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'11)*. 355–366.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazieres, D., and Kozyrakis, C. 2012. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 335–348.

Bittau, A., Marchenko, P., Handley, M., and Karp, B. 2008. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 309–322.

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., and Wiedermann, B. 2006. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. 169–190.

Cappos, J., Dadgar, A., Rasley, J., Samuel, J., Beschastnikh, I., Barsan, C., Krishnamurthy, A., and Anderson, T. E. 2010. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 212–223.

Chiba, Y. 2006. Heap protection for java virtual machines. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*. 103–112.

Cox, R. S., Gribble, S. D., Levy, H. M., and Hansen, J. G. 2006. A safety-oriented platform for web applications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'06)*. 350–364.

Douceur, J. R., Elson, J., Howell, J., and Lorch, J. R. 2008. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. 339–354.

Drewry, W. 2012. Dynamic seccomp policies (using BPF filters). http://lwn.net/Articles/475019/.

Efstathopoulos, P., Krohn, M., Vandebogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, M. F., and Morris, R. 2005. Labels and event processes in the asbestos operating system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'05)*. 17–30.

Erlingsson, U. and Schneider, F. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW'99)*. ACM Press, New York, 87–95.

FORD, B. AND COX, R. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference*. 293–306.

FURR, M. AND FOSTER, J. 2006. Polymorphic type inference for the jni. In *Proceedings of the 15th European Symposium on Programming (ESOP'06)*. 309–324.

GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. 2004. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'04)*.

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium*.

GONG, L. 2002. *Java 2 Platform Security Architecture*. Sun Microsystems.

HIRZEL, M. AND GRIMM, R. 2007. Jeannie: Granting java native interface developers their wishes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 19–38.

IOANNIDIS, S., BELLOVIN, S. M., AND SMITH, J. M. 2002. Sub-operating systems: a new approach to application security. In *Proceedings of the ACM SIGOPS European Workshop*. 108–115.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track USENIX Annual Technical Conference*. USENIX Association, 275–288.

KLINKOFF, P., KIRDA, E., KRUEGEL, C., AND VIGNA, G. 2007. Extending .net security to unmanaged code. *Int. J. Inf. Secur. 6,* 6, 417–428.

KONDOH, G. AND ONODERA, T. 2008. Finding bugs in java native Interface programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM Press, New York, 109–118.

KRISHNAMURTHY, A., METTLER, A., AND WAGNER, D. 2010. Fine-grained privilege separation for web applications. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. 551–560.

KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. 2007. Information flow control for standard os abstractions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. 321–334.

LEE, B., HIRZEL, M., GRIMM, R., WIEDERMANN, B., AND MCKINLEY, K. S. 2010. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'10)*. 36–49.

LEROY, X. 2008. *The Objective Caml system*. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

LI, S. AND TAN, G. 2009. Finding bugs in exceptional situations of jni programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 442–452.

LIANG, S. 1999. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co.

MCCAMANT, S. AND MORRISETT, G. 2006. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Usenix Security Symposium*.

METTLER, A., WAGNER, D., AND CLOSE, T. 2010. Joe-E: A security-oriented subset of java. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'10)*.

MILLER, M. 2006. Robust composition: Towards a unified approach to access control and concurrency control. Ph.D. thesis, Johns Hopkins University, Baltimore, MD.

MITRE. 2012. CVE-2012-4681. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681.

MITRE. 2013. CVE-2013-0422. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0422.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From System F to typed assembly language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, New York, 85–97.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst. 21,* 3, 527–568.

NECULA, G. 1997. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, New York, 106–119.

NECULA, G., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*. 128–139.

NEUMANN, P. AND WATSON, R. 2010. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Proceedings of the 4th Layered Assurance Workshop*.

ORACLE. 1999. JAR file specification. http://docs.oracle.com/javase/1.4.2/docs/guide/jar/jar.html.

ORACLE. 2010. JVM tool interface, version 1.0. http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html.

PROVOS, N. 2003. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*. 257–272.

PYTHON/C FFI. 2009. Python/C api reference manual. http://docs.python.org/c-api/index.html.

SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. 2010. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th Usenix Security Symposium*. 1–12.

SHACHAM, H. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. 552–561.

SIEFERS, J., TAN, G., AND MORRISETT, G. 2010. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 201–211.

SMALL, C. 1997. A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS'97)*. 174–184.

SUN, M. AND TAN, G. 2012. JVM-portable sandboxing of java's native libraries. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS'12)*. 842–858.

SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. 2004. Recovering device drivers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. 1–16.

TAN, G., APPEL, A., CHAKRADHAR, S., RAGHUNATHAN, A., RAVI, S., AND WANG, D. 2006. Safe java native interface. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE'06)*. 97–106.

TAN, G. AND CROFT, J. 2008. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Usenix Security Symposium*. 365–377.

TAN, G. AND MORRISETT, G. 2007. ILEA: Inter-language analysis across java and C. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 39–56.

WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. 1993. Efficient software-based fault isolation. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'93)*. ACM Press, New York, 203–216.

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'98)*. 52–63.

WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. 2012. Securing untrusted code via compileragnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. 299–308.

WATSON, R., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. 2010. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th Usenix Security Symposium*. 29–46.

WITCHEL, E., RHEE, J., AND ASANOVIC, K. 2005. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'05)*. 31–44.

YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*. 79–93

ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIERES, D. 2006. Making information flow explicit in histar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 263–278.

Received February 2013; accepted June 2013