Robusta: Taming the Native Beast of the JVM

Joseph Siefers
Department of Computer Science and Engineering
Lehigh University
joseph.siefers@gmail.com

Gang Tan
Department of Computer Science and Engineering
Lehigh University
gtan@cse.lehigh.edu

Greg Morrisett
School of Engineering and Applied Sciences
Harvard University
greg@eecs.harvard.edu

ABSTRACT

Java applications often need to incorporate native-code components for efficiency and for reusing legacy code. However, it is well known that the use of native code defeats Java's security model. We describe the design and implementation of Robusta, a complete framework that provides safety and security to native code in Java applications. Starting from software-based fault isolation (SFI), Robusta isolates native code into a sandbox where dynamic linking/loading of libraries is supported and unsafe system modification and confidentiality violations are prevented. It also mediates native system calls according to a security policy by connecting to Java's security manager. Our prototype implementation of Robusta is based on Native Client and OpenJDK. Experiments in this prototype demonstrate Robusta is effective and efficient, with modest runtime overhead on a set of JNI benchmark programs. Robusta can be used to sandbox native libraries used in Java's system classes to prevent attackers from exploiting bugs in the libraries. It can also enable trustworthy execution of mobile Java programs with native libraries. The design of Robusta should also be applicable when other type-safe languages (e.g., C#, Python) want to ensure safe interoperation with native libraries.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; D.2.12 [**Software**]: Software Engineering—*Interoperability*

General Terms

Security

1. INTRODUCTION

It is rare that a large software system is written in one single programming language. Java programmers often find it necessary to incorporate native-code components into their applications. Frequently, they reuse a widely adopted software library written in a different programming language for its maturity and for saving development time. Programmers may also implement performance-critical components in low-level languages such as C or C++. For these reasons, almost all Java applications include some native code. Even Sun's JDK 1.6 contains over 800,000 lines of C/C++ code (compared to around two million lines of Java code).

On the other hand, the dangers posed by native code are well understood. Most of the issues result from the fact that native code is not subject to the control of the Java security model. Any native code included in a Java application has to be completely trusted. Granting the permission to allow a Java program to load native code is equivalent to granting all permissions to the program. As a result, Java programmers face the difficult dilemma of deciding whether or not to include native code; a decision between security and other practical considerations (e.g., efficiency). Often, security is simply put aside.

The thrust of this project is to design a framework that allows JVM administrators to constrain native code with different trust levels, similar to how the security of Java code is configured. We were greatly encouraged by the recent successes of software-based fault isolation (SFI) on modern CISC architectures [18] and in internet browsers [34]. SFI provides a foundation for enforcing security on native code, while sacrificing only modest efficiency.

Starting from SFI, we built Robusta[1], which is a security layer incorporated into the JVM for regulating native code. Fig. 1 presents a high-level diagram showing the relationship between the JVM and Robusta. As shown in the diagram, Robusta provides a sandbox for untrusted native code; all interactions between the native code and the rest of the system are mediated by Robusta. Only interactions that obey a given policy will be allowed.

Fig. 1 also presents two scenarios showing how Robusta can improve Java's security. First, Robusta can be used to sandbox native libraries used in Java's system classes. We mentioned that there is already a large amount of trusted native C/C++ code that comprises a significant portion of the Java Development Kit (JDK). Our previous study [31] found 126 software bugs in a subset of the total native code (38,000 lines), of which 59 were security critical. We believe most of the native libraries used in the JDK can be managed by Robusta, eliminating many undiscovered potential exploits. For instance, the system classes under `java.util.zip` in-

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

[1]Coffee Robusta is a species of coffee. Its powerful flavor gives it perceived "strength" and "finish".

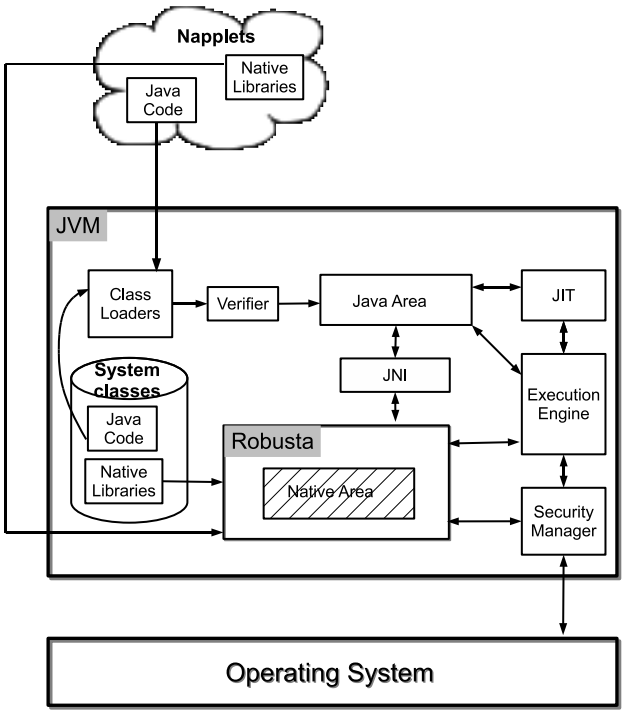


Figure 1: JVM and Robusta.

voke the popular Zlib C library for performing compression/decompression. As we will show, the Zlib C library can be sandboxed by Robusta, removing it from the TCB.

Robusta can also allow mobile Java programs (Java applets) to utilize native libraries. Such an advance allows the distribution of Java and native code together, which we term a *napplet*[2]. In the vanilla JVM, the default policy for Java applets is to disallow native code entirely, due to the inherent security risks. Instead, Robusta allows Java administrators to define a security policy for the native libraries within each napplet. For example, suppose a napplet is downloaded from a remote (untrusted) host and contains a fast mathematics library as well as some Java code for the GUI. Robusta can sandbox the napplet's native code to prevent access to the local file system or network.

The main contribution of this paper is the design and implementation of Robusta, a framework that enables the isolation and security configuration of native code within the JVM. We propose solutions for many technical challenges in this context, including dynamic linking/loading, sandboxing of JNI functions, and the accommodation of multithreading. We also propose a novel architecture for the security configuration of native code, which reuses much of Java's security infrastructure. Our prototype implementation in Sun's OpenJDK demonstrates the approach is practical and efficient.

The rest of the paper is organized as follows. We start in Sec. 2 with a brief introduction to the background of Java-native interoperation. We then present a high-level overview of Robusta in Sec. 3. We describe the technical challenges in isolating native code in the JVM and present our solutions in Sec. 4. In Sec. 5, we describe how Robusta regulates native system calls. We discuss our prototype implementation and evaluation results in Sec. 6, providing a guideline for how to make the tradeoff between efficiency and security in Robusta. Finally, we discuss related work, future work and conclude.

[2]napplet—a Native Applet

2. BACKGROUND: JAVA-NATIVE INTEROPERATION

The Java Native Interface (JNI) [17] is Java's mechanism for interfacing with native code. A native method is declared in a Java class by adding the `native` modifier. The following code snippet of the `Deflater` class is extracted from Sun's JDK. It declares a native `deflateBytes` method. Once declared, native methods are invoked in the same way as ordinary Java methods. In the example, the `deflate` Java method invokes `deflateBytes`.

```
public class Deflater {
  ...
  public synchronized int deflate
    (byte[] b, int off, int len)
      { ...; return deflateBytes(b, off, len);}

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

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

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

3. ROBUSTA OVERVIEW

In this section, we discuss Robusta's threat model, its defenses, and the security policies that it enforces. Robusta can enforce polices despite the attacks described in the threat model. Details of Robusta's defense mechanisms are left to Sec. 4 and 5.

3.1 Threat model

Native code resides in the same address space as Java code but is not constrained by the Java security model. It poses the same kinds of threats to a Java environment as any untrusted code does. We focus on the most vicious kinds of attacks: system modification and privacy invasion. Fig. 2 presents three ways in which these attacks can happen. Firstly, unconstrained native code can directly read-/write any memory location in its address space, resulting in possible destruction of the JVM state or leak of confidential information.

Secondly, abusive JNI calls can also cause integrity or confidentiality violations. The JNI interface itself does not mandate any safety checks and native code could deliberately or unintentionally misuse the JNI interface, resulting in unsafe modification of the JVM state or access to private

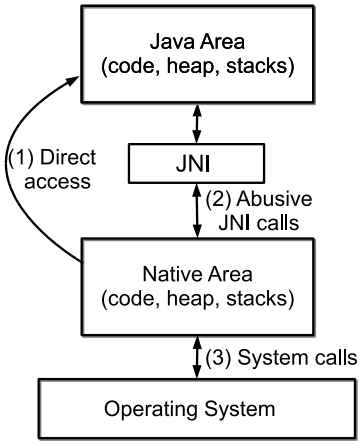


Figure 2: Triple threats from native code.

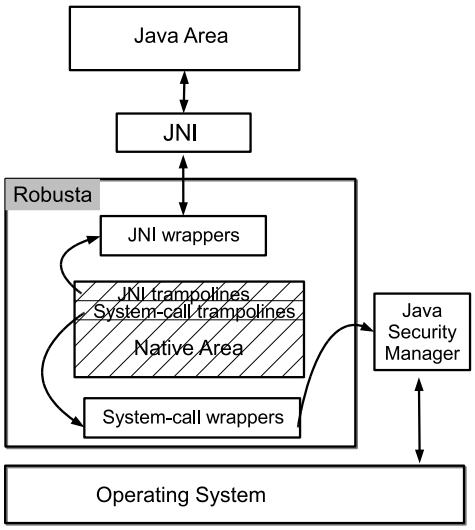


Figure 3: Defending against the triple threats in Robusta.

fields. For instance, native code can invoke `setObjectField` to modify a field of an object to a value whose type is incompatible with the field's declared type, resulting in so-called *type-confusion* attacks [19]. As another example, through `getObjectField`, native code can read any field of an object even if it is a private field.

Finally, native code may issue OS system calls to cause unsafe side effects such as overwriting security-sensitive files or sending network packets.

3.2 Defenses in Robusta

Fig. 3 depicts how Robusta defends against the triple threats discussed in the previous section. As a first step, Robusta adopts SFI [33] to isolate untrusted native code from the rest of the JVM. Native code is constrained within a sandbox so that memory access and illegal control transfers outside of the sandbox are disallowed. Robusta's implementation builds upon Google's Native Client (NaCl [34, 26]), a state-of-the-art SFI tool. Robusta extends NaCl to support secure dynamic linking/loading of native libraries, which is necessary in the JVM context as Java classes are loaded dynamically. Robusta acts as an agent for the JVM to fulfill tasks such as initialization of the sandbox, loading libraries, and resolving symbols.

To prevent abusive JNI calls, Robusta sandboxes JNI calls in the following way. First, all JNI calls are redirected to JNI *trampolines* by a technique that provides "fake" interface pointers to untrusted native code (details in Sec. 4). These trampolines reside in an unmodifiable region at the beginning of the sandbox. Trampolines are the only ways that native code can escape the sandbox. The JNI trampolines then invoke trusted JNI wrappers outside of the sandbox to perform safety checks, preventing unsafe modification or privacy invasion. Robusta also addresses several other issues in the JNI interface, including direct pointers to Java primitive arrays and calling Java methods from native code.

Finally, Robusta connects to Java's security manager to mediate native system calls. A system call issued by native code is redirected to its corresponding system-call trampoline, which in turn invokes a trusted system-call wrapper. The wrapper invokes the `checkPermission` method of Java's security manager to decide on the system call's safety based on a pre-defined security policy. This design enables Robusta to place native code under the same runtime security restrictions as Java code and reuses much of Java's policy-driven security infrastructure.

3.3 Protection strength

The permission for loading native code implies `AllPermission` in the original Java security model. By contrast, Robusta gives JVM administrators the ability to configure the security of native code. We next discuss what kinds of security policies Robusta enforces despite attacks described in the threat model.

Our discussion will be based on a lightweight formal notation. We define a JVM state to be a pair, (s, h), where s is a stack of method calls and h a heap. The heap is a map from locations to Java objects. Native code has its own world, including its heap, stack, I/O behavior, and so on. Therefore, when considering both the JVM and native code, a complete state is (s, h, w), where w is a native world.

JVM integrity. The integrity policy states that when native code makes a computation step from (s, h, w) to (s', h', w'), the new Java state (s, h) should be consistent with the old one (s', h'). We define the consistency to be respect to Java's type system, with the help of the following two notions.

A *well-typed Java heap* is one where each heap object is well typed according to its runtime type tag. *A type-preserving heap extension* from h to h', written as $h \sqsubseteq h'$, means that both h and h' are well typed and objects in h and h' at the same location have the same type. Note that h' may have more objects than h and the definition does not require the fields of objects at the same location to have the same value. These two notions can be formalized precisely based on a Java model [32].

Definition 1. *Suppose one execution step in native code brings a state (s, h, w) to (s', h', w'). Then the integrity of the JVM is respected if $s = s'$ and $h \sqsubseteq h'$.*

That is, native code should not change the stack of the JVM. Native code may perform type-preserving modifications to the Java heap (including type-preserving mutation

and allocation), and may make arbitrary changes to its own world. Enforcing this integrity policy ensures that Java code is type safe even after a native step, therefore preventing type-confusion attacks.

JVM confidentiality. During runtime, the JVM provides native code with a set of object references. Confidentiality is respected if native code accesses only objects reachable by those references and if the access-control modifiers (e.g., `private`) of fields and methods are respected. This is an *intentional* definition of confidentiality. It could also be formalized *extensionally* following a relational approach. Roughly, in a extensional formalization, confidentiality is respected if native code behaves the same when given two Java heaps whose public data are the same, but whose private data may differ. This is similar to how information-flow policies can be formalized in a relational setting. The formalization is beyond the scope of this paper.

Access control of system calls. Native code interacts with the native system via system calls. A policy on system calls defines which system calls native code is allowed to access. For example, one policy can allow file accesses, but deny network accesses. Given a policy P, when native code makes a change to its own world from w to w', that change should respect P.

Robusta enforces JVM integrity and confidentiality policies through SFI and by embedding safety checks into JNI-call wrappers. Robusta controls access to system calls by consulting Java's security manager in system-call wrappers.

4. ISOLATING NATIVE CODE

Robusta isolates native code into its own "address space" so that it cannot damage the rest of the system. We discuss how the isolation is achieved in four steps: (1) NaCl, the basis of isolation; (2) secure dynamic linking/loading; (3) various technical issues when incorporating NaCl into the JVM context; and (4) how JNI calls are sandboxed.

4.1 The starting point: NaCl

SFI is the basis of Robusta and isolates untrusted native code into a "user address space" within the JVM address space. We will call the "user address space" the *NaCl address space*. Without this low-level isolation, high-level security policies cannot be meaningfully enforced on native code.

Robusta utilizes an industrial-strength SFI tool, Native Client (NaCl [34, 26]). NaCl builds upon the ideas of previous SFI systems [33, 28, 4, 18, 5]. It is portable across multiple architectures and operating systems. Its runtime overhead is small due to clever uses of hardware features (e.g., segment protection on x86-32). It reports an average of 5% overhead on SPEC2000 for memory and jump protection (on x86-32). We next highlight a few technical aspects of NaCl that are the most relevant for the discussion of Robusta.

Sandboxing unsafe instructions. The NaCl address space is separated into a non-writable code region and a non-executable data region. NaCl disallows memory reads/writes outside of the data region and control transfers outside of the code region. This is achieved by segment protection or by inserting instructions for masking addresses before memory and control-transfer instructions. NaCl also enforces every jump target to be jump aligned, following Pittsfield [18]. The restrictions ensure NaCl binaries can be reliably disassembled for static verification and ensure that inserted dynamic checks cannot be bypassed.

Trampolines and springboards. Trampolines are the only ways through which untrusted code can escape the sandbox. The NaCl runtime sets up system-call trampolines during the loading of NaCl binaries and relies on OS 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 NaCl-compliant binaries and also provides profiling and debugging support. System calls in generated binary code are redirected to their corresponding trampolines (at fixed addresses and are immutable).

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 instructions. 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.

4.2 Secure dynamic linking/loading

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 native library.

NaCl supports only statically linked libraries. The NaCl address space layout does not support dynamic linking/loading because its code and data regions have no room for loading extra libraries and also because its security configuration requires all code to be known before execution.

Robusta generalizes NaCl to support dynamic linking/loading in two steps. First, it adopts a new address space layout that accommodates library code and data. Second, it extends NaCl's verification scheme to ensure the safety of dynamically loaded code.

Address space layout. The prototype implementation of Robusta adopted Seaborn's suggestion [25] for a new layout of the NaCl address space on Linux/x86-32. The layout is shown in Fig. 4. This is a Harvard-style architecture, where code and data are in separate regions. The x86 segment registers `cs` and `ds` point to the beginning of the code and the data region, respectively. Therefore, an address is interpreted differently depending on whether it is used as a code or a data address. To avoid confusion, the layout does not allow code and data pages to be mapped at the same address. This is why, for example, (D1) in the code region and (C1) in the data region are left unmapped. One downside of this memory layout is that it wastes some memory.

The address-space layout is compatible with PIC (position-independent code). PIC is used by dynamically loaded libraries in Linux and stipulates that the relative offset between code and data segments of a library should be constant. To see why the layout accommodates PIC, assume the beginning address of library 1's code is `cs:x`, and the beginning address of library 1's data is `ds:y`. As long as the offset between `x` and `y` is kept unchanged, PIC is supported. That is, PIC code is oblivious to how hardware segments are configured.

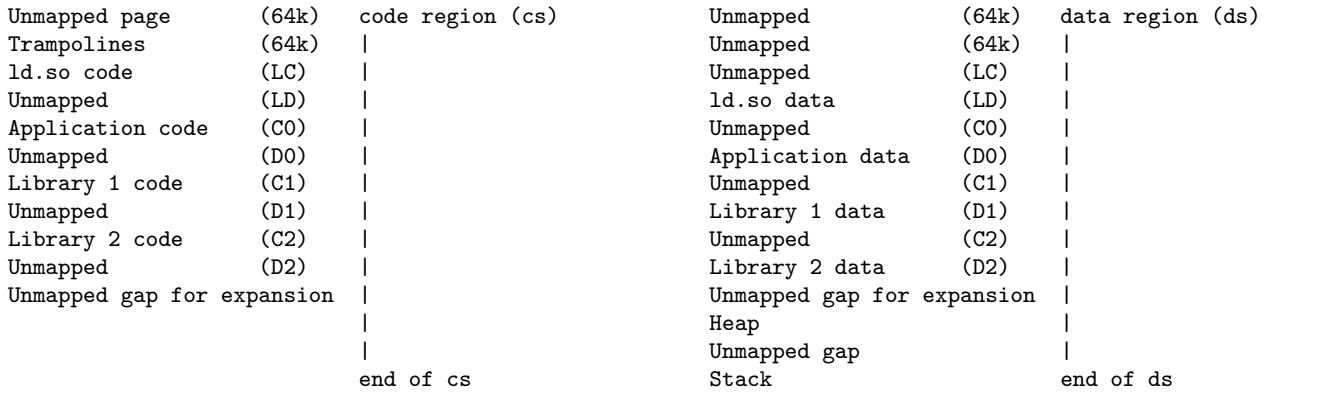


Figure 4: A layout of the NaCl address space for dynamic linking/loading.

Finally, note that the dynamic linker/loader (`ld.so` in Linux) is loaded into the address space to allow for possible `dlopen` and `dlsym` requests from a NaCl application.

Secure dynamic loading. There are three main security concerns for dynamically loaded code:

- Dynamic loading inserts into the code region new untrusted code, which must be verified before it can be safely executed.
- Library images may be changed by an adversary after verification. Precautionary measures must be taken to ensure that the code to be executed is the same as the one that was verified.
- Loading code into memory is not an atomic operation. When one thread is loading a dynamic library, other malicious threads could take advantage of a partially loaded library and carry out an attack.

To address these concerns, Robusta adopts a novel solution based on the Non-eXecutable (NX) bit of page protection. The NX-bit support is available in mainstream processors including Intel and AMD processors. Most operating systems now support the NX-bit (see [2] for a survey).

Access permissions for memory pages include readable, writable, and executable. Unmapped pages in the NaCl address space are memory protected to be unreadable, unwritable, and unexecutable. This prevents any access to unmapped pages.

In Robusta, the process of loading a dynamic library through `dlopen` is as follows:

(1) The `dlopen` request for a library is sent to `ld.so`, the dynamic linker/loader. `ld.so` invokes the system call `mmap` to map a segment of the library's image into the NaCl address space.

(2) A `mmap` system call is redirected through the trampoline mechanism of NaCl to a trusted `mmap` wrapper. The wrapper checks to see if the requested region has been occupied. If so, it reports failure to `ld.so`, which will request another memory region to perform the memory mapping.

(3) If the requested region, called R hereafter, is not occupied and if the segment to be mapped in the library is a code segment, the wrapper will first make R writable (but it remains unreadable and unexecutable) and will copy the segment into R. Since R is not executable, other threads cannot execute partially copied code.

(4) The wrapper marks R as readable, but unwritable and unexecutable. It then invokes NaCl's verifier to check the safety of the newly copied code. R is made readable so that the verifier can read the code for verification. Moreover, future changes to the library's image do not invalidate the verification result since the verification is performed on the code that was copied in.

(5) Finally, if the verification succeeds, R is marked as readable and executable, but unwritable. That is, R is not executable until this stage, which prevents any code from being executed before it is verified.

4.3 Incorporating NaCl into the JVM

Robusta is constructed as an intermediate layer between the JVM and NaCl. Hooks are added to the JVM so that whenever the JVM needs to interact with native code, execution transfers to the Robusta layer, which fulfills the required operation. We next discuss how Robusta functions.

JVM initialization. When the JVM starts, Robusta constructs a NaCl sandbox. It reserves a memory region for the NaCl address space and sets up a code and a data region. Page protection is configured so that the code region is readable and executable, and the data region is readable and writable.

Trusted trampolines are installed in the code region. These trampolines include system-call trampolines and JNI trampolines, as well as a special trampoline called `OutOfJail`, which is invoked when native code finishes execution.

Finally, the initialization of the sandbox loads the dynamic linker/loader (`ld.so`) into the NaCl address space. It also loads a utility module, named `dlWrappers`, which provides a gateway for Robusta to access services housed within the sandbox (see the following discussion). The dynamic linker/loader and the utility module are not in the TCB because they are in the sandbox and the only way out of the sandbox is through known safe exits (i.e., trampolines).

Only one sandbox is constructed for all native code because we are mainly concerned with protecting the JVM from native code. However, there are situations where isolating one native library from another is desired (e.g., when

napplets are downloaded from multiple websites that have differing trust levels). We believe it is straightforward to extend Robusta with support for multiple sandboxes.

Loading a native library and symbol resolution. When the JVM decides to load a native sandboxed library, Robusta checks if the security policy allows the operation (see Sec. 5). If so, it invokes `dlopen_wrapper` in the utility module. The wrapper then calls the actual `dlopen` method implemented by the dynamic linker/loader, and propagates the resulting handle back to the JVM through calling `OutOfJail`.[3] Dynamic symbol resolution within sandboxed native libraries follows a similar pattern. A service routine called `dlsym_wrapper` invokes `dlsym` in the dynamic linker/loader to resolve the symbol's address, and propagates the resulting address back to the JVM through `OutOfJail`.

Calling in and returning. When the JVM invokes a native method (e.g., as a result of executing an `invokespecial` bytecode instruction), Robusta copies parameters from the Java stack to the native stack in the sandbox and invokes `method_dispatch` in the utility module with the address of the native method as a parameter. `method_dispatch` then invokes the native method and collects the return value before calling `OutOfJail`.

After a context switch outside of the sandbox through `OutOfJail`, Robusta needs to continue the execution of the JVM. 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 uses `setjmp` for saving the state and `longjmp` for restoring the state.

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 collection of interlaced Java and native frames. To cope with the ping-pong behavior, Robusta associates the return information with a native frame so that when a native frame is popped from the stack the return information for that frame is used to continue the execution of the JVM.

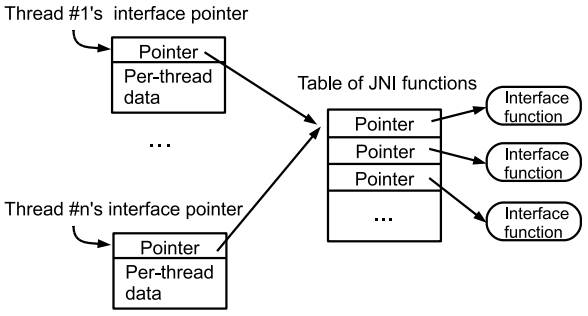
Multiple Java threads. In the case of multiple Java threads, each Java thread begins life outside the sandbox and may pass freely in and out each time it makes a native call. Therefore, it is possible that multiple Java threads may be inside the sandbox concurrently. To this end, Robusta associates a `natp` structure with a Java thread.[4] Since each thread needs its own stack area while it is in the sandbox, the `natp` structure remembers the location of that stack. Other per-thread information is also in the `natp` structure.

Lazy allocation of stacks. Each Java thread needs a native stack in the sandbox. However, not all Java threads use native methods. Robusta 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. To create the stack, Robusta calls the `allocate_stack` wrapper within the sandbox. It in turn calls `malloc` to reserve a block of memory for the stack.

4.4 Sandboxing JNI calls

As we have discussed, abusive JNI calls may cause integrity and confidentiality violations. We next explain how Robusta prevents abusive JNI calls.

The JNI interface pointer. Native methods access JNI functions through an interface pointer. The interface pointer points to a location that references a function table, as the following diagram (Fig 11.3 [17]) displays:



For example, the C syntax "`(*env)->f(...)`" invokes a JNI function `f` through the interface pointer `env`. Robusta cannot pass the real interface pointer to native code because all of the functions in the table are outside the sandbox (the only way to exit the sandbox is through trampolines). Robusta's solution is to duplicate the interface pointer structure in the sandbox except that the table of JNI functions contains pointers to JNI trampolines. Robusta then passes the duplicate structure's location as a *fake* JNI interface pointer to native code. 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 interface function through the real interface pointer. In this design, native code 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, as we will discuss.

One more complication arises when dealing with multiple threads. The real interface pointer is a per-thread pointer, but for efficiency Robusta'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 only needs to look up the real interface pointer for a particular thread from the `natp` structure.

JNI checking. Robusta inserts safety checks in its JNI wrappers before the real JNI functions are invoked. These checks are necessary to maintain the integrity and confidentiality of the JVM. The inserted checks are similar to what previous JNI checking systems perform, including SafeJNI [30] and Jinn [15]. For instance, when native code invokes a Java method, Robusta checks that the type of each parameter matches the Java method's signature, preventing type confusion attacks. Additionally, it verifies that exactly the right number of parameters are provided. Robusta classifies the checks into two groups: integrity and confidential-

[3]The wrapper is not strictly necessary, but provides a convenient way of stringing together calls to `dlopen` and `OutOfJail`.

[4]The `natp` structure is used by NaCl to store per-thread information; Robusta piggybacks on it to store extra information.

ity. For example, during a field-write JNI call, an integrity check would ensure the new value is of the same type as the target field. Similarly, a confidentiality check would ensure private fields are not accessible.

Direct pointers to the Java heap. Java often needs to pass 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, Robusta 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 and thus compares favorably to SafeJNI [30], where 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 a JVM may already need to make a copy in the JVM heap. In that case, we can change the implementation 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 1.7.0 always makes a copy for GetIntArrayElements.

5. MANAGING NATIVE CODE SECURITY

The basic idea behind Robusta's regulation of 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, uniform security policy for an entire application of mixed Java and native code. This design also enables Robusta to reuse much of the infrastructure provided by Java security, including its policy-specification framework and enforcement mechanism; only a minimum amount of extra code needs to be added to make the idea work. We next discuss the major points in our design.

A security policy can grant the native libraries of an application two kinds of permissions: mode permissions and system-access permissions. *Mode permissions* specify whether a native library can be loaded into the JVM and whether it should be sandboxed. A Java application that has been granted the "loadLibrary.*libraryName*" runtime permission is allowed to load the library with *libraryName* in the unconstrained mode. If the application has been granted the "loadSNL.*libraryName*" permission, then the application can load the library in the sandboxed mode (SNL stands for Sandboxed Native Library). If there is no mode permission granted for a Java application, then by default it is not allowed to load a native library. The mode policy is enforced during library-loading time in Robusta.

We note that although the policy specification for an application allows a native library to be loaded in the sandboxed mode and another library in the unconstrained mode, such a policy essentially grants all permissions to the application because the unconstrained library can access any resource freely. As in Java security, extreme care must be taken to allow an application to load native libraries in the unconstrained mode. On the other hand, mixing sandboxed and unconstrained libraries might still be beneficial from the point of view of isolating faults in the sandboxed libraries.

System-access permissions specify what system accesses an application is allowed to perform. Robusta's system-access permissions are the same as those provided by Java security. For instance, a policy can grant a Java application the permission to read files, but restrict it from writing files. Robusta can enforce such policies in the native libraries of a Java application. The enforcement is carried out in the following way:

- All system calls in the native library are redirected to the system-call trampolines in the sandbox. The system-call trampolines then invoke trusted system-call wrappers outside of the sandbox.
- The system-call wrappers invoke the `checkPermission` method of Java's security manager after constructing the 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.

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

Second, Robusta disallows spawning native threads. The reason is twofold. First, creating native threads is strongly discouraged in the JNI [17] 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 [9]). For these two reasons, native code should call back to Java to create new Java threads, which is allowed in Robusta.

6. IMPLEMENTATION AND EVALUATION

In this section, we discuss the prototype implementation and experimental evaluation of Robusta.

6.1 Prototype implementation

Our proof-of-concept implementation is based on OpenJDK 1.7.0. Robusta is compiled separately from the OpenJDK into a shared library that the OpenJDK loads during runtime. Various hooks are added to the OpenJDK to invoke routines defined in the Robusta library. For instance, a hook is added so that, when the OpenJDK needs to load a native library, the control is transferred to Robusta for loading the library into the sandbox (if the library should be sandboxed according to the policy). This design of minimizing changes to the OpenJDK has the advantage of reducing the development time of Robusta because OpenJDK's re-compilation delay is significant (at least 10 minutes).

We modified 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. The JVM provides two implementations for the execution engine, a default ASM template version and a (slower) C++ version. In order to fully evaluate Robusta, we integrated Robusta with the ASM template version.

Robusta's implementation is small. It is comprised of about 2,000 lines of C code, 7 JVM and 8 NaCl hooks, and about 150 lines of C code for the utility module that is loaded into the sandbox during initialization. Robusta made few changes to the NaCl toolchain. One small modification by Seaborn [25] made the linker generate PLT (Procedure Linkage Table) entries for dynamically linked code. The same verifier in NaCl is used in Robusta for validating code safety.

Robusta's implementation is restricted to Linux and x86-32. It has not yet dealt with portability, both from the OS and ISA points of view. However, since NaCl has been ported to multiple ISAs and OSes, we believe most of Robusta will be portable. The memory layout for dynamic loading in Robusta targets the ELF format on Linux and needs adjustment for Windows's PE format.

6.2 Experimental evaluation

To evaluate Robusta, we conducted experiments to test its functionality and performance overhead. Experiments were carried out on a Linux Ubuntu 8.1 box with Intel Core2 Duo CPU at 2.26GHz. When evaluating performance overhead, all experiments were averaged over 10 runs.

Functionality testing. We created a set of microbenchmarks for testing the functionality and testing the security of Robusta. These microbenchmarks include programs for passing parameters of various types and sizes from Java to native code, programs for getting and setting fields of Java objects, programs for accessing Java arrays, programs for allocating Java objects and arrays, and programs for making system calls. We also included programs for testing Robusta's effectiveness for preventing abusive JNI calls and preventing unsafe system calls. These microbenchmarks were fully evaluated before we conducted performance evaluation on larger Java programs.

Runtime overhead. The runtime overhead of Robusta can roughly be put into two classes. 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. The second class of runtime overhead happens during *context switches*. In Robusta, 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 the sandbox; when native code finishes execution, the context is switched outside of 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 runtime overhead 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; the overhead would be similar to the overhead of NaCl, which has been reported to incur 5% of overhead on SPEC2000. On the other hand, if a program makes frequent context switches between Java and native code, then there should be a significant runtime overhead. Therefore, an interesting question is to *explore the relationship between the runtime overhead and how frequent context switches happen*. An answer helps to understand what kinds of applications should be put under the control of Robusta.

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.
- `libec` is a C library for elliptic curve cryptography. OpenJDK provides JNI bindings for interfacing with the library.
- 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 PDF creation C library. 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.

Table 1 shows the code sizes of the benchmark programs. For each program, the table lists its lines of Java code and its lines of C code. The C code is divided into the category of glue (JNI) code and the category of library C code.

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

Program	Java LOC	Glue/Library C LOC
zip	3,351	2,295/11,319
libec	2,689	416/19,049
libjpeg	31	96/20,346
libharu	188	114/120,959
StrictMath	1,128	153/8,505

Table 1: A set of benchmark programs and their code sizes (reported by SLOCCount).

Program	Robusta increase	Context switches (per millisecond)
zip (1KB)	9.64%	18.50
zip (2KB)	7.51%	9.93
zip (4KB)	5.22%	5.00
zip (8KB)	2.42%	2.34
zip (16KB)	1.40%	0.95
libec (112)	3.41%	5.80
libec (160)	2.82%	1.08
libec (224)	6.20%	0.46
libec (256)	2.55%	0.30
libec (384)	-0.92%	0.06
libec (521)	-0.24%	0.05
libec (571)	5.03%	0.03
libjpeg	3.82%	0.002
libharu	48.22%	68.85
StrictMath	729.48%	269.57

Table 2: Runtime overheads of Robusta for a set of JNI programs.

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.

- **libec.** Experiments were set up to generate pairs of public and private keys of varying sizes from random seeds. We experimented with key sizes 112, 160, 224, 256, 384, 521 and 571.
- **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 sample text.
- **libjpeg.** Experiments were set up to convert a 5Mb bmp image into JPEG format.

Table 2 presents the experimental results for the benchmark programs. The numbers in the column of "Robusta increase" were obtained in the following way. We first compiled the program through the GNU toolchain and ran it in the vanilla OpenJDK. Let x_1 denote the total runtime. The program was then fed to the NaCl toolchain to produce NaCl-compliant binaries and was run in Robusta. Let y_1 be the total runtime in Robusta. With these measurements, $(y_1 - x_1)/x_1$ is the runtime overhead of Robusta over OpenJDK and is the number in the "Robusta increase" column of the result table.

Table 2 also shows the measurements for the *context switch intensity*, or the number of context switches per millisecond. 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%. A similar story applies to the `libec` benchmark. `StrictMath` is an extreme case. It makes around 270 context switches per millisecond and its performance overhead is significant. The high context switch intensity is due to the fact that each native method call in this benchmark stays in the sandbox very shortly. Another extreme case is `libjpeg`, which has a very low number of context switches per millisecond and thus Robusta incurs little overhead over SFI.

The result table also shows some performance improvements. We believe it is due to NaCl because in those cases the context-switch intensity is low. NaCl reported performance improvements for some SPEC2000 benchmark programs because of positive interactions between alignment and processor microarchitectures.

In general, the results show that Robusta 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.

Robusta's runtime overhead compares favorably to SafeJNI [30], which reports 63% performance overhead for the zip benchmark with the 16KB buffer size. The reason is that SafeJNI performs array bounds checking for every access to the buffer passed from Java to native code, while it is unnecessary in Robusta thanks to SFI.[5] Robusta's runtime overhead also compares favorably to reimplementation of native libraries in Java code. SafeJNI reported that the runtime overhead of a pure Java implementation of the Zlib library (jzlib-1.0.5) is 74%.

7. RELATED WORK

There has been a rich tradition of computer-security research aiming to isolate untrusted code from a trusted environment. Operating systems have long used hardware-based protection to isolate one process from another. Nooks [29] isolates device drivers from kernel code, and Xax [3] isolates web applications from browsers. Robusta follows a software-based approach [33, 28, 4, 5, 18, 34, 26]. Robusta shows SFI can serve as a basis for efficiently isolating native libraries in a type-safe language (Java), even in the case that the two sides communicate through a tight interface (the JNI). In terms of mediating system calls, Robusta is related to a number of previous efforts such as systrace [24] and many others (e.g., [8, 11, 7]). The difference of Robusta is that it delegates the job to Java's security manager. This is a general strategy of how system calls in native code can be handled in language virtual machines.

[5]We assume a "ensure, but don't check" strategy [18]. When an out-of-bound access happens, the access is re-routed to an address within the sandbox. Bounds checking would be required if it is necessary to abort execution.

Klinkoff *et al.* designed a sandboxing mechanism for protecting managed code and the .NET runtime from unmanaged code [13]. Although in a different context, their system addresses similar security problems as Robusta does. Their system puts unmanaged code into a separate process and seems to suffer from high performance overhead due to inter-process communication. In addition, system calls in unmanaged code are intercepted and regulated by a kernel add-on. By contrast, Robusta is a more-efficient security mechanism thanks to SFI and it is a pure user-space module.

An alternative approach to achieving safety in Java-native interoperation requires native code to be compiled from a type-safe low-level language, such as Cyclone [12]. The compiler can produce binaries in the form of Proof-Carrying Code [22] or Typed Assembly Languages [21], whose safety can be independently verified at the code consumer's site. This approach can avoid some of the runtime overhead in Robusta and is profitable for performance-critical native code. On the flip side, rewriting legacy code in type-safe languages requires a large amount of effort and does not address the issue of safe interoperation between Java and native code.

The JNI does not mandate any checking of native methods. Native methods are notoriously unsafe and is a rich source of software errors. Recent studies have reported hundreds of interface bugs in JNI programs [6, 31, 14]. A number of systems are designed to improve and find misuses of the JNI interface. They can be classified into three categories: (1) Jeannie [10] 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 [6, 32, 14, 16]. (3) Jinn [15] 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. SafeJNI [30] is in spirit closest to Robusta in that they both protect Java from untrusted native code. However, SafeJNI is based on CCured [23], which performs source-code rewriting for security, while Robusta is based on SFI.

The Joe-E system [20] by Mettler *et al.* designs a Java subset that aims at the development of secure software systems. To reason about the security capabilities possessed by a Joe-E class, its verifier rejects many potentially unsafe Java features. This includes the use of native methods, which could escalate the capabilities of Joe-E classes. We believe the features provided by Robusta can enable systems such as Joe-E to extend their security reach to native code. It would be interesting to combine Joe-E and Robusta.

8. FUTURE WORK

We plan to explore what portions of native libraries in Java's system classes can be put under the control of Robusta. There are 800,000 lines of C/C++ code in Sun's JDK 1.6. We expect Robusta should be able to sandbox most of JDK's system libraries, as we have demonstrated for `zip` and `libec`. However, it is possible that not all native libraries for system classes are suitable for Robusta because of restrictions related to functionality or performance. Some system native libraries may need direct access to the JVM state. For instance, a security manager accesses JVM's method-call stack directly. Some system classes' native libraries may cross the boundary between the Java and native worlds so often that putting them into a sandbox would have a significant performance penalty for the JVM; in these cases, perhaps a combination of Robusta and static verification is beneficial.

We are developing a napplet mechanism for Robusta, which will allow the distribution of both a Java applet and its required native libraries in a single package (in the spirit of a standard JAR file). Robusta's sandbox will then prevent any abusive native code in the napplet from harming the host system.

We plan to explore techniques for stronger security policies within Robusta. Robusta 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 [27]). Control-Flow Integrity (CFI, [1]) can foil 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. We are in the process of incorporating CFI into Robusta.

9. CONCLUSIONS

Native code has always been the dark corner of Java security. Although powerful, native code in Java applications poses serious security threats. We have discussed the design and implementation of Robusta, which protects the JVM from native code while incurring modest runtime overhead. Robusta is especially suitable for applications without intense context switching between Java and native code. It provides a migration path for moving the 800,000 lines of native code outside of the JDK, and for enabling mobile Java programs with native libraries.

Acknowledgments

We thank Mark Seaborn for explaining his scheme of a new address space layout for supporting dynamic linking/loading in NaCl. We thank anonymous referees of CCS '10 for detailed comments on an earlier version of this paper. This research is supported in part by NSF grant CCF-0915157, CCF-0915030, and a research grant from Google.

10. REFERENCES

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM conference on Computer and communications security (CCS)*, pages 340–353, 2005.

[2] A. Daniele, B. Patrizio, and D. B. Luca. NX bit: A hardware-enforced *BOF* protection. `http://patrizioboschi.it/work/nx-bit/NX-bit.pdf`.

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

[4] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *NSPW '99:*

Proceedings of the 1999 workshop on New security paradigms, pages 87–95, 2000.

[5] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[6] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.

[7] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium*, 1996.

[9] L. Gong. *Java 2 Platform Security Architecture*. Sun Microsystems, 1997-2002.

[10] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.

[11] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *ACM SIGOPS European Workshop*, pages 108–115, 2002.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

[13] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET security to unmanaged code. *Internation Journal of Information Security*, 6(6):417–428, 2007.

[14] G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

[15] B. Lee, M. Hirzel, R. Grimm, B. Wiedermann, and K. S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 36–49, 2010.

[16] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *16th ACM conference on Computer and communications security (CCS)*, pages 442–452, 2009.

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

[18] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium*, 2006.

[19] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[20] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium (NDSS)*, 2010.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 85–97, New York, 1998. ACM Press.

[22] G. C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, New York, 1997. ACM Press.

[23] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.

[24] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium*, pages 257–272, 2003.

[25] M. Seaborn. Segment layout. Sent to the Native Client mailing list, Dec 2008.

[26] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th Usenix Security Symposium*, 2010. to appear.

[27] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM conference on Computer and communications security (CCS)*, pages 552–561, 2007.

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

[29] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2004.

[30] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[31] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *17th Usenix Security Symposium*, pages 365–377, 2008.

[32] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 39–56, 2007.

[33] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 203–216, New York, 1993. ACM Press.

[34] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)*, pages 79–93, May 2009.