Safe Java Native Interface

Gang Tan*, Andrew W. Appel#, Srimat Chakradhar◊, Anand Raghunathan◊, Srivaths Ravi◊, and Daniel Wang#

*Boston College
#Princeton University
◊NEC Labs America
Heterogeneous Programming Paradigm

- Reuse legacy code
- Mix-and-match benefits of different languages
  - E.g., C is faster and more flexible than Java
  - E.g., Java based GUIs are easier to develop
Foreign Function Interfaces (FFIs)

- Most modern languages have FFIs
  - Java, ML, OCaml, Haskell, ...
- FFIs address issues such as
  - Representation of data
  - Calling conventions
  - Memory management
  - ...

‡ Most modern languages have FFIs
‡ FFIs address issues such as
What about Safety and Security?

Same address space

Java code

C code

Application

Strongly typed
Memory safe

Weakly typed
Memory unsafe

The whole application becomes memory unsafe!
Approaches for Safe Interoperation:

- Component models: COM, DCOM, SOAP, CORBA
  - Different address spaces
  - Communication via RPCs
  - But, high performance overhead and inflexible

- Rewrite every component in a safe language
  - Substantial programming effort
  - Hard/impossible sometimes

We focus on FFI-based approaches.
Our Focus: Java Native Interface (JNI)

- Java can call C-implemented methods
- C code can
  - Access, update and create Java objects
  - Call a Java method
  - Catch and raise exceptions
  - ...

Diagram:

Java code → JNI → C code
Our Goal

- Make calling native C code in Java as **safe** as calling Java code in Java

**Benefits:**
- Reuse legacy C code in Java safely and conveniently
- Improve the security of Java platform
  - JDK 1.4.2 contains over 600,000 lines of C code under the cover of JNI
- More lightweight and flexible comparing to RPC-based approaches
Two Subproblems

- Provide internal safety for C Code.
  - CCured [Necula, Condit, et al.]
    - Ensure memory-safety by source-to-source transformation
    - Insert runtime checks
    - Heavily optimized
  - Cyclone [Jim, Morrisett, et al.]

- Safe interoperation between C and Java
  - Ensure that C uses JNI in a principled way
Outline

- Motivation
- JNI and its loopholes
- SafeJNI system
- Preliminary experiments
- Future work
An Example Of Using JNI

```java
class IntArray {
    ...
    native int sumArray(int arr[]);
    ...
}
```

```c
jint Java_IntArray_sumArray
(JNIEnv *env, jobject *obj, jobject *arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    jint *body =
        (*env)->GetIntArrayElements(env, arr, 0);
    int i, sum = 0;
    for (i=0; i<len; i++) sum+=body[i];
    (*env)->ReleaseIntArrayElements(env, arr, body, 0);
    return sum;
}
```
Using JNI in C Code

Get a pointer into the Java heap

Pass in a pointer to the int array

```c
jint Java_IntArray_sumArray
(JNIEnv *env, jobject *obj, /* object */ jobject *arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    jint *body =
        (*env)->GetIntArrayElements(env, arr, 0);
    int i, sum = 0;
    for (i=0; i<len; i++) sum+=body[i];
    (*env)->ReleaseIntArrayElements(env, arr, body, 0);
    return sum;
}
```

- Well-behaved C code manipulates Java objects through JNI APIs
Loophole: Out-of-Bounds Accesses

```c
jint Java_IntArray_sumArray
(JNIEnv *env, jobject *obj, jobject *arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    jint *body =
        (*env)->GetIntArrayElements(env, arr, 0);
    ...
    body[100]=9831;
    ...
}
```

Out-of-bound write; destroys JVM’s state
Loophole: Arguments of Wrong Classes

JNI completely ignores the Java class hierarchy
- All Java classes are mapped to jobject * in C

```c
jint Java_IntArray_sumArray
(JNIEnv *env, jobject *obj, jobject *arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    ...
}
```

C can pass objects of wrong classes to Java
Loophole: Calling Wrong Methods

```c
jint Java_IntArray_sumArray  
(JNIEnv *env, jobject *obj, jobject *arr)  
{  
jsize len = (*env)->GetArrayLength(env, arr);  
jint *body =  
    (*env)->GetIntArrayElements(env, arr, 0);  
...  
}
```

Nothing prevents C from calling `GetFloatArrayElements`
Loophole: Manual Memory Management

```c
jint Java_IntArray_sumArray
(JNILong *env, jobject *obj, jobject *arr)
{
    jsize len = (*env)->GetArrayLength(env, arr);
    jint *body =
        (*env)->GetIntArrayElements(env, arr, 0);
    ...
    (*env)->ReleaseIntArrayElements(env, arr, body, 0);
    ...
}
```

Dangling pointers; memory leak; release twice
Safety/Security Vulnerabilities in JNI

- Bypassing JNI: direct read/write through Java pointers
- Out-of-bounds array access
- Passing objects of wrong classes to Java
- No access control
- Manual memory management
- Calling wrong methods
- Exception handling
- Out of the Java sandbox security model

At best: causes a JVM crash
At worst: security violation
Outline

- Motivation
- JNI and its loopholes
- SafeJNI system
- Preliminary experiments
- Future work
Safe Java Native Interface (SafeJNI)

- **Goal:**
  - Make calling native C code in Java as **safe** as calling Java code in Java

- **Features:**
  - A pointer kind system
  - Safe mem. management
  - Various dynamic checks
Restricting Capabilities of Pointers

- Opaqueness of Java object pointers
  - Can pass them as arguments to JNI APIs
  - No pointer arith./cast/read/write

- Java primitive array pointers
  - Allow pointer arith., but must be within bounds
  - Carry bounds information at runtime
A Pointer Kind System

- Classify pointers with different capabilities
- An extension of CCured’s pointer kinds

<table>
<thead>
<tr>
<th>Pointer Kind</th>
<th>Description</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t *HNDL</code></td>
<td>Java handle Pointers</td>
<td>Pass to JNI APIs; equality testing</td>
</tr>
<tr>
<td><code>t *RO</code></td>
<td>Read-only pointers</td>
<td>read</td>
</tr>
<tr>
<td><code>t *SAFE</code></td>
<td>Safe pointer</td>
<td>read/write</td>
</tr>
<tr>
<td><code>t *SEQ</code></td>
<td>Sequence pointers</td>
<td>Above + pointer arithmetic</td>
</tr>
<tr>
<td><code>t *WILD</code></td>
<td>Wild pointers</td>
<td>Above + casts</td>
</tr>
</tbody>
</table>
After step 4, “Pointer 1” is dangling if GC recycles the buffer.

1. C calls `GetIntArrayElements` and gets “pointer 1”
2. In `GetIntArrayElements`, JVM pins the buffer so that GC will not move it
3. When it’s done, C calls `ReleaseIntArrayElements`
4. JVM unpins the buffer
Our Solution for Mem. Management

- Create a validity tag
- Change the representation of a pointer to a struct
- In `GetIntArrayElements`, init the tag to 1
- In `ReleaseIntArrayElements`, change the tag to 0
- Before dereferencing, check that the tag is 1
Various Dynamic Checks

- Runtime type checking
  - E.g., when GetIntArrayElements is called, check the second arg. is an int-array object
  - When a Java method is called, check the number and classes of arguments

- Access control
  - Check during “get field ID”

- Exception checking

- Non-null checking

Java maintains all information at runtime
SafeJNI System: On Top of CCured

- C code
- Annotated jni.h
- Kind Inference Engine
  - Annotated C code
  - Type Checker
  - NO
  - Yes

- Insert Checks
  - Safe C code
  - gcc
    - Library code
    - link
    - Bytecode

- Java code
- Java Compiler
Outline

- Motivation
- JNI and its loopholes
- SafeJNI system
- Preliminary experiments
- Future work
Microbenchmarks

![Bar chart showing the SafeJNI/JNI Ratio for different operations: field access, callback, array, string, and comp. The values are 1.49, 1.29, 2.19, 2.09, and 1.14 respectively.]
Zlib Experiment

- Zlib compression library
  - 9,000 lines of C code + 262 lines of glue code
  - The basis for java.util.zip

<table>
<thead>
<tr>
<th></th>
<th>SafeJni Ratio</th>
<th>CCured Ratio</th>
<th>JZlib* Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlib</td>
<td>1.63</td>
<td>1.46</td>
<td>1.74</td>
</tr>
</tbody>
</table>

* JZlib is a 100% pure Java reimplementation of Zlib
A Safety Loophole in java.util.zip

- Zlib maintains a z_stream struct
  - For keeping state info
- The Deflater object needs to store a pointer to this C struct
  - However, it’s difficult to define a pointer to a C struct in Java!

```java
class Deflater {
    private long strm;
    ...
}
```

- Then C casts it back to a pointer
A Safety Loophole in java.util.zip

With reflection support, we can change the private long.

```java
import java.lang.reflect.*;
import java.util.zip.Deflater;
public class Bug {
    public static void main(String args[]) {
        Deflater deflate = new Deflater();
        byte[] buf = new byte[0];
        Class deflate_class = deflate.getClass();
        try {
            Field strm = deflate_class.getDeclaredField("strm");
            strm.setAccessible(true);
            strm.setLong(deflate, 1L);
        } catch (Throwable e) {
            e.printStackTrace();
        }
        deflate.deflate(buf);
    }
    */ Policy file needed in a secure environment */
    grant {
        permission java.lang.RuntimePermission "accessDeclaredMembers";
        permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    };
}
```

Crashed Sun’s JVM and IBM’s VM
Related Work

- OCaml’s FFI [Furr and Foster]
  - Track OCaml types in C to prevent misuse
- NestedVM [Alliet and Megacz]
  - Put native code into a separate VM
  - Slowdown ratio: 200% to 900%
- Janet [Bubak et al.]
  - A cleaner interface
- “-Xcheck:jni”
  - Incomplete and undocumented
Future Work

- Reduce the amount of dynamic checks
  - Keep track of Java types in C code
  - Use static analysis/theorem proving
- .Net: interaction between managed and unmanaged code
SafeJNI: Conclusions

- Reuse legacy C code **safely** and conveniently
- More lightweight and flexible comparing to RPC-based approaches
The End