Safe Heterogeneous Applications: Curing the Java Native Interface *

Gang Tan[⋆†] Andrew Appel[⋆] Srimat Chakradhar[†]
Anand Raghunathan[†] Srivaths Ravi[†] Daniel Wang[⋆]

[⋆] Department of Computer Science, Princeton University
[†] NEC Laboratories America

Abstract

The Java Native Interface (JNI) allows type-safe Java code to interact with unsafe C code. When a type-safe language interacts with an unsafe language in the same address space, the application becomes unsafe. We identify the loopholes specific to using JNI that would permit C code to bypass the type safety of the JVM. We have designed a solution based on an extension of CCured [9] that makes calling native methods in C as type-safe as pure Java code.

We have implemented a significant part of our solution and measured its effect on performance. Porting a native C library (Zlib) into our system requires only minimal changes to the C source code. The performance of this library is faster than a pure Java reimplementation of the library but slower than the original unsafe C version. During our experiments on Zlib, our system identified one type unsafety in the interface code between Zlib and Java. This insecurity can be exploited to crash, or gain extra privileges in a large number of commercially deployed JVMs.

1 Introduction

Large software systems often contain components developed using different programming languages. For software components to interoperate, there must be a standard interface between them. Heavyweight RPC based systems like COM [10], SOAP [12], and CORBA [4] allow components to be placed in different address spaces, which provides safe interoperation but with significant overheads per call. The per-call costs limit the way software components can be structured, and require a mechanism for address space isolation. We are interested in a more lightweight approach to heterogeneous component interoperation, using a foreign function interface (FFI) rather than RPC based approaches.

Any FFI becomes an integral part of a programming language. It enables the language to reuse legacy components written in another language. It gives the language access to features that may not fit into the programming model of the language. For example, Java provides a standard Java Native Interface (JNI) [8]. JNI is a native programming interface that allows Java code running inside a Java Virtual Machine to interoperate with components that are written in C, C++, or assembly. JNI was designed to be a standard interface for all JVM implementations, hence JNI-compatible code should interoperate with many different JVM implementations.

*This research was funded in part by ARDA grant NBCFC030106. This information does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

Unsafe and insecure interoperation. An FFI usually addresses discrepancies between the representation of primitive values, memory management, calling conventions, and so on. However, being able to call components written in a different language is only part of the story. For example, when a component written in a safe language directly interacts with a component written in an unsafe language, the whole application becomes unsafe. Even rich systems like Microsoft's .NET CLR [5] have this problem. The .NET CLR distinguishes between "managed" and "unmanaged" code. linking unmanaged code with managed code nullifies the safety guarantees of the managed code.

Java language and APIs provide type safety and certain security. In practice, all safety and security guarantees must be qualified by the native code in the implementations of JVMs. For example, Sun's JDK 1.4.2 contains over 600,000 lines of native C code. Any error in the native code base could lead to a type-safety or security violation.

Existing approaches to safe interoperation. Previous work shows that native machine code can be automatically translated to Java byte code [1]. This provides safety with no extra-programming effort, but incurs a large performance penalty of 3x to 10x. Another approach is to reimplement C libraries in pure Java. This requires substantial programming effort, and often results in a noticeable performance penalty. Sometimes it is not even possible because C code needs to access certain OS services which are not available in Java.

Toward safe and secure interoperation. Ideally, we would like the linking of C code to Java be as safe as the linking of Java code to Java. To achieve this goal, we have examined how C code, interacting via the JNI, may exploit loopholes to violate Java's type safety.

The most obvious problem is that C code is inherently unsafe and may read and write addresses in the JVM. Fortunately, there are systems such as CCured [9] and Cyclone [7] which provide safety guarantees for legacy C code. We have adopted the approach that requires the least programmer intervention, which is CCured by Necula et al. CCured performs source-to-source translation to insert the smallest number of run-time checks to make C code memory safe.

However, just providing internal safety for the C code is not sufficient to guarantee safe interoperation between Java

and C. The JNI, if not used properly, exposes several loopholes, which we will enumerate in Section 2. One example is that C code can read private members of a Java object through JNI. Hence, to have safe native methods for Java, more work needs to be done beyond ensuring type safety of the C code alone.

Our solution to ensure the safety of JNI is composed of three parts: we extend the type system of CCured to better capture the implicit invariants of the JNI interface; we insert certain dynamic checks before JNI API calls; and finally, we formulate a scheme to achieve safe memory management in JNI. We will describe our approach in detail in Section 3. In Section 5, we will formalize a small type system so that we can prove a safety claim for an interesting subset of C and JNI.

We have implemented a prototype system that incorporates some of the proposed techniques, through a combination of static analysis and source code modification to implement dynamic checks. Although it does not close all the loopholes, it serves as a reasonable platform to evaluate the performance impact of the dynamic checking that our techniques would introduce. We have conducted a preliminary experiment using the Zlib library that is distributed with Sun's JDK. Our experiments indicate that the performance overheads of the proposed techniques are reasonable. For Zlib, the overhead is 57% with respect to the original unsafe implementation, but 11% faster than an expert Java reimplementation of Zlib.

2 JNI and its loopholes

JNI is Java's mechanism for interfacing to native code. It enables native code to have essentially the same functionality as Java code. Through JNI, native code can inspect, modify and create Java objects, invoke methods, catch and throw exceptions, and so on. Figure 1 shows a simple but complete example of using JNI: Java passes C an array of integers and C returns back the sum of the array. The Java code needs to declare a method to be a foreign method using the keyword "`native`". Then Java can call the native method just as it would call other Java methods.

The C code accepts a reference to the Java array as an argument. Then, C can manipulate the array object through JNI API functions, such as calling `GetArrayLength` to get the length of the array and `GetIntArrayElements` to get a pointer to the array elements. So the common idiom is that Java code passes Java object references to C code, which calls JNI API functions to manipulate Java objects.

When a JVM passes control to a native method, the JVM will also pass an interface pointer to the native method (argument `env` in the example). This interface pointer points to a location that contains a pointer to a function table. Every JNI API function (such as `GetArrayLength`) is at a predefined offset in the table. Through the interface pointer, the native method can invoke JNI API functions.

Mapping of types. There are two kinds of types in Java: primitive types such as `int`, `float`, and `char`, and reference types such as objects and classes. JNI treats primitive types and reference types differently. The mapping of primitive types is direct. For example, the type `int` is mapped to C type `jint` (defined as 32-bit integer in jni.h). On the other hand, objects of reference types are passed to native methods as *opaque references*, which are C pointers to internal data structures in the JVM. The exact layout of the internal data structures, however, is hidden from the programmer. All opaque references have type "`_jobject *`" in C, such as the type of argument arr in Figure 1. The C code treats all Java objects as being members of one type.

2.1 Loopholes in JNI

The JNI interface exposes loopholes that may cause unsafe interoperation between Java and C. We enumerate them in this section. We have tested all these loopholes using real code, and most of them frequently cause a JVM crash. However, it is conceivable that, in some cases, these loopholes may be exploited to achieve malicious effects such as leakage of private data and execution of malicious code.

Out-of-bound array access. Sometimes, Java needs to pass an array of data to a native method. For efficiency reason, JNI functions such as `GetIntArrayElements` [1] and `GetStringUTFChars` allow the native method to directly address the Java heap. It is easy to imagine a native method accidentally reading or writing past the bounds of the actual array.

Direct access through Java references. Opaque references in C code are supposed to be manipulated only by JNI API functions. However, there is no mechanism to prevent C code from performing direct reads/writes via these references.

Interface pointers. An interface pointer passed from JVM points to a function table of JNI API functions. We must prevent C code from overwriting entries in the function table in the interface pointer. Otherwise, C code can replace a JNI API function with its own version and bypass any check in the function.

Data privacy. JNI does not enforce class, field, and method access control that can be expressed in Java language through the use of modifiers such as `private`. Therefore, C code can read a private field of an object. As stated in the specification [8, sect. 10.9], this was a conscious design decision, since native methods in type unsafe languages can access and modify any memory location in the heap anyway. However, once type safety of native methods is assured, data privacy also needs to be addressed in order to ensure type safety of the overall heterogeneous application.

Manual memory management. JNI's scheme of managing memory is similar to C's `malloc`/`free`. For example, when the native method is done with the integer array buffer returned by `GetIntArrayElements`, it is supposed to call `ReleaseIntArrayElements` to inform the JVM that the buffer is no longer needed by the native method. This kind of manual memory management has well-known problems such as memory leaks, dangling pointers, and multiple releases.

[1] Sometimes the JVM makes a copy of the Java array. In either case, the function returns a direct pointer into the JVM heap.

```java
class IntArray {
  //declare a native method
  private native int sumArray(int arr[]);

  public static void main(String args[]) {
    IntArray p = new IntArray();
    int arr[] = new int [10];
    for (int i = 0; i < 10; i++) arr[i] = i;
    //call the native method
    int sum = p.sumArray(arr);
    System.out.println("sum = " + sum);
  }

  static {
    //load the DLL library that implements
    //the native method
    System.loadLibrary("IntArray");
  }
}
```

```c
#include <jni.h>
#include "IntArray.h"

JNIEXPORT jint JNICALL
Java_IntArray_sumArray
(JNIEnv *env, _jobject *self, _jobject *arr)
//env is an interface pointer through which
//a JNI API function can be called.
//self is the reference to the calling object.
//arr is the reference to the array.
{
  jsize len = (*env)->GetArrayLength(env, arr);
  int i, sum = 0;
  jint *body =
    (*env)->GetIntArrayElements(env, arr, 0);
  for (i=0; i<len; i++) {
    sum += body[i];
  }
  (*env)->ReleaseIntArrayElements(env,arr,body,0);
  return sum;
}
```

Figure 1: A JNI example: Java passes C an array of integers and C returns back the sum of the array. On the left is the Java code; on the right is the C code.

Arguments of wrong classes. Since native code treats all references to Java objects having one single type (`_jobject *`), an object of class `A` may be wrongly passed to a JNI API function that actually requires class `B`. The JNI specification states that the behavior of the JVM is unspecified in such a case. In reality, it will usually cause a JVM crash.

Exception handling. A native method can call an ordinary Java method. When the Java method returns, the native method should call certain JNI functions to check, handle and clear pending exceptions. Calling arbitrary JNI functions with a pending exception may lead to unexpected results.

Security. Java's security model confines the capabilities of untrusted Java code. JVM will consult a security manager before it performs potentially dangerous operations such as writing to a file. Once a native method is called, however, the JVM can no longer verify, catch, or prevent the program from violating the security of the environment in which the JVM is running. In this work, we only address type safety, not security.

3 Achieving type safety in JNI: Overall approach

Our approach to achieving type safety of heterogeneous applications that use JNI consists of three steps:

- We propose a pointer type system to model JNI specific pointers in native C code, such as JNI opaque references and read-only interface pointers. We use the new type system to statically enforce JNI related invariants, which avoids direct accesses to opaque references and overwriting function entries in an interface pointer. In this work, we have augmented CCured's type system for our purpose. To prevent the out-of-bound array accesses, we model Java array pointers as CCured sequence pointers so that CCured can dynamically check out-of-bound access violation. We present details of our extension to CCured in Section 4.
- Some of the loopholes can be fixed by inserting dynamic checks before a JNI API is called, such as checking if a field is private before the field is read. We describe those dynamic checks we insert into the JNI interface in Section 3.1.
- Finally, in Section 3.2, we propose a solution to solve the manual memory management problem in JNI.

3.1 Insert dynamic checks to JNI APIs

Data privacy checking. We insert runtime checks to enforce access-control rules of Java fields/methods, such as checking that a native method is not accessing a private field. This is possible to check dynamically since all Java objects keep a runtime representation of permissions.

JNI factors out the cost of locating a field by using a two-step process: first get the field ID; then use the ID to access the value of the field. Our checks need to be done only in the first step. Thus, the performance overhead for this check is not significant.

Class checking. As we have discussed, if C passes Java an object of a wrong class, the JVM's behavior is unspecified. One example is when the C code in Figure 1 calls `GetArrayLength` function with a non-array object.

Since Java keeps all class information at runtime, we dynamically check that every object passed back from C is an instance of the correct class. This is a simple but effective solution when performance is not critical.

An alternative and more efficient way would be to make our system be aware of Java class signatures and statically track classes of objects in C. This strategy would reduce the number of dynamic checks performed.

Pending exception checking. We insert code to check if an exception is pending before calling a JNI API function.

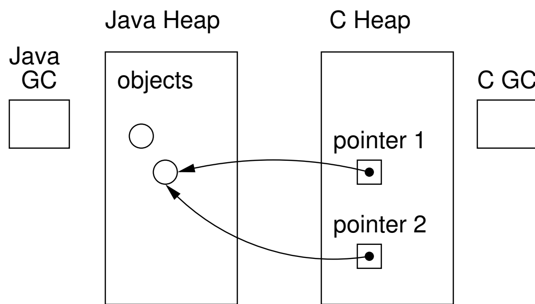

Figure 2: Memory management in JNI

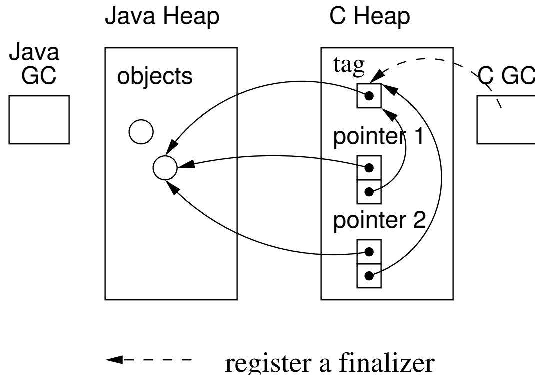

Figure 3: Safe memory management

3.2 Safe memory management

CCured ignores explicit deallocation in C and uses the Boehm conservative garbage collector [2] to reclaim storage. Assuming there is a C garbage collector in place, we present our solution to get safe memory management in JNI.

In Figure 2, we show how JNI manage memory. Suppose that C initiates a `GetIntArrayElements` and `ReleaseIntArrayElements` sequence, we list the steps of what happens:

1. C calls `GetIntArrayElements` and gets a pointer to the buffer used by the array (pointer 1 in Figure 2).
2. In `GetIntArrayElements`, the JVM also pins the buffer so that Java's GC will not garbage collect it. The ownership of the buffer has been transferred to C.
3. When C is done with the buffer, it calls `ReleaseIntArrayElements` on "pointer 1".
4. JVM unpins the buffer and now the buffer is back to Java.

The above scheme is fine except when C makes a copy of "pointer 1" (get "pointer 2" in Figure 2). Then, after step 4, "pointer 2" becomes dangling when Java's GC decides to garbage collect the buffer.

Our solution (Figure 3) can be stated in two steps. In the first step, we create a validity tag for the buffer, and change the representation of pointers to be a structure that also has metadata pointing to the validity tag. With this change, we have created the case that there are pointers pointing to the buffer if and only if there are pointers pointing to the validity tag.

In the second step, we register a function that calls `ReleaseIntArrayElements` as the finalizer for the validity tag in the Boehm garbage collector, and also makes user's call to `ReleaseIntArrayElements` be a nop. This way, when the C program has no live pointers to the buffer, GC's finalizer will call `ReleaseIntArrayElements` to release the buffer.

Our solution has a space cost for each pointer pointing to the buffer. Pointer dereferencing has no time cost, however, because the tag pointer need not be dereferenced during normal use.

4 A type system for type-safe JNI

CCured [9] is a tool that adds memory safety to C. It analyzes the C program to identify places where memory safety might be violated and does a source-to-source translation to insert runtime checks to ensure safety.

During the analysis phase, CCured classifies pointers according to their usage. Pointers in C programs that are used without pointer arithmetic or type casts are classified as SAFE pointers, and they are either null or valid references. Pointers that are used with pointer arithmetic but not type casts are classified as SEQ ("sequence") pointers. CCured enhances sequence pointers to carry bounds of the array so that all dereferences can be dynamically checked to be within bounds. Pointers that involve bad type casts (such as from integer to pointer) are classified as WILD pointers. CCured enhances WILD pointers to carry information to distinguish a pointer from an integer, and dynamically prevents the dereferencing of arbitrary integers.

We extend CCured by adding two new pointer kinds to model pointers passed from Java to C.

Bounds for Java arrays. Functions such as `GetIntArrayElements` return a pointer to an array. We model an array pointer as a CCured SEQ pointer, since C needs to do arithmetic on this pointer to walk through the array.

The only complexity is that a SEQ pointer needs to carry bounds information to verify that any use of the pointer is within bounds. Therefore, we need to set up the bounds when the `GetIntArrayElements` function is called. In this case, we can easily get the bounds by calling the `GetArrayLength` function.

Java handle pointers. As we have stated, all references to Java objects in C code should not be directly accessed; they are opaque to C code. To enforce this abstraction, we classify such pointers as HNDL ("handle") pointers — pointers that can be neither read nor written. Handle pointers are passed as arguments to JNI API functions.

CCured allows casts between certain kinds of pointers. One case is to cast a SEQ pointer to a SAFE pointer, since a SEQ pointers carry more privileges than SAFE pointers. However, casts to HNDL pointers are not allowed since otherwise C could forge a Java reference through a SAFE pointer, for example. We maintain the invariant that the only way to get a handle pointer is by calling a JNI API function.

Pointer Kind	Description	Capability
t *HNDL	Java handle pointers	as arguments to JNI API functions
t *RO	read-only pointers	read
t *SAFE	safe pointers	read/write
t *SEQ	sequence pointers	pointer arithmetic; read/write
t *WILD	wild pointers	type casts; pointer arithmetic; read/write

Table 1: Pointer kinds and their capability

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \mathsf{int} \mid \mathsf{_jobject} \\
 & & \mid & \tau \ast \mathsf{SAFE} \mid \tau \ast \mathsf{SEQ} \mid \tau \ast \mathsf{HNDL} \\
\textit{Expr's} & e & ::= & x \mid n \mid e_1 + e_2 \\
 & & \mid & !e \mid (\tau)e \\
\textit{Stmt's} & s & ::= & e \mid s_1; s_2 \mid e_1 := e_2 \\
 & & \mid & \mathrm{GetArrayLength}(e) \\
 & & \mid & \mathrm{GetIntArrayElements}(e) \\
 & & \mid & \mathrm{GetObjectArrayElement}(e, e) \\
 & & \mid & \mathrm{SetObjectArrayElement}(e, e, e) \\
 & & & \\
\textit{Values} & v & ::= & n \mid \textit{Hndl}(?) \mid \textit{Safe}(n) \mid \textit{Seq}(n, b, e)
\end{array}
$$

Figure 4: Language Syntax

Read-only pointers. Read-only pointers are pointers that can be read, but not written. We model a Java interface pointer as a read-only pointer to prevent C code from replacing a function entry in the table pointed by the interface pointer.

Our read-only pointers are related to the C `const` qualifier. For example, C type "`const` int *" is the same as our "int *RO". We do not use the `const` qualifier since CCured's convention for a pointer kind is to associate attributes with pointer types, instead of the underlying types.

In Table 1, we list all pointer kinds used in our type system, including the ones in CCured.

5 Formalization and Soundness Proof

To have a formal claim of our safety guarantee, we have extended CCured's formalization [9] to include the handle pointer kind and a representative subset of JNI API functions. Based on this formalization, We have proved a safety theorem: well-typed C programs will not access Java's memory. We do not model read-only pointers, although this is straightforward. We also do not model memory management.

Figure 4 gives the syntax we use for C. As in CCured's formalization, our syntax is a great simplification of real C syntax for the purpose of presenting key ideas. For example, control-flow statements are ignored since our approach is control-flow insensitive.

In the type category, we have type $\mathsf{_jobject}$ for the type of all Java objects. In addition to safe pointers ($\tau \ast \mathsf{SAFE}$) and sequence pointers ($\tau \ast \mathsf{SEQ}$), we also have handle pointers. ($\tau \ast \mathsf{HNDL}$).

We distinguish between expressions, which have no side effects, and statements, which may have side effects. For expressions, we have x for variables; n for integer literals; $e_1 + e_2$ for pointer arithmetic; $!e$ for the result of reading from the memory location pointed by e (like $\ast e$ in C); we also have $(\tau)e$ for type casts.

For statements, we have $s_1; s_2$ for sequential statements; $e_1 := e_2$ for assignments (like $\ast e_1 = e_2$ in C), and a subset of JNI API functions. These functions are described in Table 2. Note that JNI treats primitive-type arrays and object arrays differently: for integer arrays, function `GetIntArrayElements` returns a direct pointer to the array; for object arrays, `GetObjectArrayElement` and `SetObjectArrayElement` are the getter and setter of the array.

Finally, we define values. Values of handle pointer types are of the form $\textit{Hndl}(?)$. Since handle pointers are opaque to C programs, its exact values do not matter; so we use $\textit{Hndl}(?)$ to represent all Java handle pointers. Values of safe pointer types are of the form $\textit{Safe}(n)$, where n is the pointer value. Values of sequence pointer types are of the form $\textit{Seq}(n, b, e)$, where b and e are metadata and are the beginning and the end of the array, respectively.

5.1 Operational semantics

In Figure 5, we present a big-step operational semantics for our language. The operational semantics are expressed by means of two judgments:

expression evaluation: $\Sigma, M \vdash_{\mathsf{e}} e \Downarrow v$
statement evaluation: $\Sigma, M \vdash_{\mathsf{s}} e \Downarrow v, M'$

In these judgments, Σ is a mapping from variables to values and memory M is a mapping from addresses to values. Statements may have side effects, so its evaluation judgment has a new memory M' in addition to the value.

One important aspect of our semantics is that M is the memory that can be accessed by C and does not include the Java memory. So if a C program tries to read a location outside of M (possibly in Java memory), then the abstract machine will get stuck.

Some rules in Figure 5 are the same as CCured. The pointer-arithmetic rule ARITH requires the pointer to be a SEQ pointer. The read and write rules (SAFERD and SAFEWR) require the pointer to be at least a SAFE pointer; these two rules also come with null-pointer checks (boxed premises). Certain casts are allowed (rule C1-C4) such as from SEQ pointers to SAFE pointers.

In our modeling of JNI API functions, we use two auxiliary functions: arrlen returns the length of an array and $\mathsf{startloc}$ returns the starting location of where an array is stored. The rule for `GetArrayLength` returns the array length directly. In the rule for `GetIntArrayElements`, since this function returns an array pointer, we need to set up its metadata (bounds of the array) appropriately. Our semantics for `GetIntArrayElements` also expand the C memory by including the array buffer so that program can access the region. We assume that every time `GetIntArrayElements` is called, a new memory region is returned; this function behaves like a memory allocation function in our semantics. Function `SetObjectArrayElement` have side effects on Java

Function	Description
`GetArrayLength`(arr)	Return the length of the array
`GetIntArrayElements`(arr)	Return the body of the integer array
`GetObjectArrayElement`(arr,n)	Return the nth element in the object array
`SetObjectArrayElement`(arr,n,obj)	Set the nth element of array arr to obj

Table 2: Explanation of some JNI API functions

Expressions:

$$\frac{\Sigma(x) = v}{\Sigma, M \vdash_{\mathsf{e}} x \Downarrow v}\ \text{VAR} \qquad \frac{}{\Sigma, M \vdash_{\mathsf{e}} n \Downarrow n}\ \text{INT}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Seq}(n, b, e) \quad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow n_2}{\Sigma, M \vdash_{\mathsf{e}} e_1 + e_2 \Downarrow \mathit{Seq}(n + n_2, b, e)}\ \text{ARITH} \qquad \frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{Safe}(n) \quad \boxed{n \neq 0} \quad M(n) = v}{\Sigma, M \vdash_{\mathsf{e}} !e \Downarrow v}\ \text{SAFERD}$$

Casts:

$$\frac{}{\Sigma, M \vdash_{\mathsf{e}} (\tau\ *\mathsf{SEQ})0 \Downarrow \mathit{Seq}(0, 0, 0)}\ \text{C1} \qquad \frac{}{\Sigma, M \vdash_{\mathsf{e}} (\tau\ *\mathsf{SAFE})0 \Downarrow \mathit{Safe}(0)}\ \text{C2}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{Seq}(n, b, e) \quad \boxed{b \leq n < e}}{\Sigma, M \vdash_{\mathsf{e}} (\tau\ *\mathsf{SAFE})e \Downarrow \mathit{Safe}(n)}\ \text{C3} \qquad \frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{Safe}(n)}{\Sigma, M \vdash_{\mathsf{e}} (\mathsf{int})e \Downarrow n}\ \text{C4}$$

Statements:

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow v}{\Sigma, M \vdash_{\mathsf{s}} e \Downarrow v, M}\ \text{EXP} \qquad \frac{\Sigma, M \vdash_{\mathsf{s}} s_1 \Downarrow v_1, M' \quad \Sigma, M' \vdash_{\mathsf{s}} s_2 \Downarrow v_2, M''}{\Sigma, M \vdash_{\mathsf{s}} s_1; s_2 \Downarrow v_2, M''}\ \text{SEQ}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Safe}(n) \quad \boxed{n \neq 0} \quad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow v_2}{\Sigma, M \vdash_{\mathsf{s}} e_1 := e_2 \Downarrow 0, M[n \mapsto v_2]}\ \text{SAFEWR}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{Hndl}(?)}{\Sigma, M \vdash_{\mathsf{s}} \text{GetArrayLength}(e) \Downarrow \mathsf{arrlen}(\mathit{Hndl}(?)), M}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{Hndl}(?) \quad n = \mathsf{startloc}(\mathit{Hndl}(?)) \quad \text{end} = n + \mathsf{arrlen}(\mathit{Hndl}(?)) \quad \text{dom}(M_1) = [n, \text{end}) \quad [n, \text{end}) \cap \text{dom}(M) = \emptyset}{\Sigma, M \vdash_{\mathsf{s}} \text{GetIntArrayElements}(e) \Downarrow \mathit{Seq}(n, n, \text{end}), M \cup M_1}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Hndl}(?) \quad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow n}{\Sigma, M \vdash_{\mathsf{s}} \text{GetObjectArrayElement}(e_1, e_2) \Downarrow \mathit{Hndl}(?), M}$$

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Hndl}(?) \quad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow n \quad \Sigma, M \vdash_{\mathsf{e}} e_3 \Downarrow \mathit{Hndl}(?)}{\Sigma, M \vdash_{\mathsf{s}} \text{SetObjectArrayElement}(e_1, e_2, e_3) \Downarrow 0, M}$$

Figure 5: Operational semantics. The boxed premises are runtime checks.

memory, but not on C memory; this is why memory M does not change in the rule for `SetObjectArrayElement`.

5.2 Type system

Our type system is expressed by the following judgments:

expression typing: $\Gamma \vdash_{\mathsf{e}} e : \tau$
statement typing: $\Gamma \vdash_{\mathsf{s}} e : \tau$
convertibility: $\tau' \leq \tau$

In these judgments, Γ is a mapping from variables to types.

Figure 6 presents our type system. Rules for SAFE and SEQ pointers are completely the same as those in CCured.

There are several points worth mentioning about our type system. First, since we want to maintain the invariant that the only way to get a handle pointer is by calling a JNI API function, so we do not have casts for handle pointers such as from safe pointers to handle pointers or from literal 0 to handle pointers. Second, any place that expects a reference to a Java object is given a type "_jobject *HNDL", such as the argument type of `GetArrayLength`. Last, the return type of `GetIntArrayElements` is modeled as CCured's SEQ pointer, or an array pointer.

5.3 Type safety

In this section we discuss a formal safety guarantee we obtain for a C program when it interacts with Java: the C program will only access its own memory M, but not Java memory. Note that our modeling deliberately models only C memory. This enables us to avoid modeling Java memory and Java states altogether. [2]

To get the safety theorem, we first introduce a store typing Λ. A store typing maps locations to types and captures the invariant of a C memory. Then, we define for type τ a set of valid values $[\![\tau]\!]_\Lambda$ belong to that type. This set depends on

[2] To formalize memory management in JNI, we believe the modeling of Java states and Java memory is needed.

Expressions:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathsf{e}} x : \tau} \qquad \frac{}{\Gamma \vdash_{\mathsf{e}} n : \mathsf{int}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : \tau\ *\mathsf{SEQ} \quad \Gamma \vdash_{\mathsf{e}} e_2 : \mathsf{int}}{\Gamma \vdash_{\mathsf{e}} e_1 + e_2 : \tau\ *\mathsf{SEQ}} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : \tau\ *\mathsf{SAFE}}{\Gamma \vdash_{\mathsf{e}} !e : \tau}$$

Casts:

$$\frac{k = \mathsf{SEQ} \text{ or } \mathsf{SAFE}}{\Gamma \vdash_{\mathsf{e}} (\tau\ *\ k)0 : (\tau\ *\ k)} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash_{\mathsf{e}} (\tau)e : \tau}$$

$$\frac{}{\tau\ *\mathsf{SEQ} \leq \tau\ *\mathsf{SAFE}} \qquad \frac{}{\tau\ *\mathsf{SAFE} \leq \mathsf{int}}$$

Statements:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \tau}{\Gamma \vdash_{\mathsf{s}} e : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{s}} s_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{s}} s_2 : \tau_2}{\Gamma \vdash_{\mathsf{s}} s_1; s_2 : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : \tau\ *\mathsf{SAFE} \quad \Gamma \vdash_{\mathsf{e}} e_2 : \tau}{\Gamma \vdash_{\mathsf{s}} e_1 := e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \mathsf{_jobject}\ *\mathsf{HNDL}}{\Gamma \vdash_{\mathsf{e}} \mathrm{GetArrayLength}(e) : \mathsf{int}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \mathsf{_jobject}\ *\mathsf{HNDL}}{\Gamma \vdash_{\mathsf{s}} \mathrm{GetIntArrayElements}(e) : \mathsf{int}\ *\mathsf{SEQ}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : \mathsf{_jobject}\ *\mathsf{HNDL} \quad \Gamma \vdash_{\mathsf{e}} e_2 : \mathsf{int}}{\Gamma \vdash_{\mathsf{s}} \mathrm{GetObjectArrayElement}(e_1, e_2) : \mathsf{_jobject}\ *\mathsf{HNDL}}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : \mathsf{_jobject}\ *\mathsf{HNDL} \quad \Gamma \vdash_{\mathsf{e}} e_2 : \mathsf{int} \quad \Gamma \vdash_{\mathsf{e}} e_3 : \mathsf{_jobject}\ *\mathsf{HNDL}}{\Gamma \vdash_{\mathsf{s}} \mathrm{SetObjectArrayElement}(e_1, e_2, e_3) : \mathsf{int}}$$

Figure 6: Typing rules

the store typing Λ in general, but the case for "$\tau\ *\mathsf{HNDL}$" does not, since Java objects are outside of C memory.

$$\begin{array}{lcl} [\![\mathsf{int}]\!]_\Lambda & = & \{ n \mid n \in \mathsf{N} \} \\ [\![\tau\ *\mathsf{HNDL}]\!]_\Lambda & = & \{ \mathit{Hndl}(?) \} \\ [\![\tau\ *\mathsf{SAFE}]\!]_\Lambda & = & \\ \quad \{ \mathit{Safe}(n) \mid n \in \mathrm{dom}(\Lambda)\ \wedge\ \Lambda(n) = \tau \} \cup \{ \mathit{Safe}(0) \} & & \\ [\![\tau\ *\mathsf{SEQ}]\!]_\Lambda & = & \\ \quad \{ \mathit{Seq}(n, b, e) \mid [b, e) \subseteq \mathrm{dom}(\Lambda)\ \wedge\ \forall b \leq i < e.\ \Lambda(i) = \tau \} & & \end{array}$$

We extend this relation element-wise to type environment $\Sigma \in [\![\Gamma]\!]_\Lambda$:

$$\begin{array}{lcl} \Sigma \in [\![\Gamma]\!]_\Lambda & = & \mathrm{dom}(\Sigma) = \mathrm{dom}(\Gamma)\ \wedge \\ & & \forall x \in \mathrm{dom}(\Sigma).\ \Sigma(x) \in [\![\Gamma(x)]\!]_\Lambda \end{array}$$

Store typing is the invariant that is respected by memory M throughout the computation. We formalize this invariant by:

$$\begin{array}{lcl} \mathit{WF}_\Lambda(M) & = & \mathrm{dom}(M) = \mathrm{dom}(\Lambda)\ \wedge \\ & & \forall x \in \mathrm{dom}(M).\ M(x) \in [\![\Lambda(x)]\!]_\Lambda \end{array}$$

Our abstract machine will stop either because memory safety is violated (access to an address outside of M) or because one of the runtime checks fails (boxed premises in Figure 5). We actually consider the second case to be safe. To distinguish these two cases, we introduce a new value *failsafe*. When a runtime check fails, the expression evaluates to a *failsafe* value: $\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{failsafe}$. Similarly for statements, we have $\Sigma, M \vdash_{\mathsf{s}} e \Downarrow \mathit{failsafe}, M'$. We also add evaluation rules that initiate the *failsafe* result when one of the runtime checks fails and the rules that propagate the *failsafe* result from the subexpressions to the enclosing expressions.

Now, we give type safety theorems for both expressions and statements.

Theorem 1 (Type Safety for expressions)
If $\Gamma \vdash_{\mathsf{e}} e : \tau$, and the initial state Σ and M respects some store typing Λ, i.e., $\mathit{WF}_\Lambda(M)$ and $\Sigma \in [\![\Gamma]\!]_\Lambda$, then

1) either $\Sigma, M \vdash_{\mathsf{e}} e \Downarrow \mathit{failsafe}$
2) or $\exists v.\ \ (\Sigma, M \vdash_{\mathsf{e}} e \Downarrow v)\ \ \wedge\ \ (v \in [\![\tau]\!]_\Lambda)$

Theorem 2 (Type Safety for statements)
If $\Gamma \vdash_{\mathsf{s}} s : \tau$, and the initial state Σ and M respects some store typing Λ, i.e., $\mathit{WF}_\Lambda(M)$ and $\Sigma \in [\![\Gamma]\!]_\Lambda$, then

1) either $\Sigma, M \vdash_{\mathsf{s}} s \Downarrow \mathit{failsafe}, M'$
2) or $\exists v, M', \Lambda'.\ \ (\Sigma, M \vdash_{\mathsf{s}} s \Downarrow v, M')\ \ \wedge\ \ (\Lambda \subseteq \Lambda')$
$\wedge\ \ (\mathit{WF}_{\Lambda'}(M'))\ \ \wedge\ \ (v \in [\![\tau]\!]_{\Lambda'})$

The proofs of these two theorems are by induction over typing derivations. They give the result that well-typed programs will not get stuck and thus will not violate memory safety. The other result from the theorem is the abstraction for handle pointer types: throughout the computation, all values of type "$\tau\ *\mathsf{HNDL}$" are of the form *Hndl*(?) and thus are pointers to Java objects.

6 Prototype System and Experiments

We have developed a prototype system on top of CCured. Our system extends CCured's type system with handle pointer kind and read-only pointer kind. Using the new system, we have hand annotated all the JNI API function prototypes in the JNI header file (jni.h). For example, the old prototype for JNI' `NewIntArray` is

```
_jobject * NewIntArray (JNIEnv * env, jint len);
```

The new prototype for JNI' `NewIntArray` is:

```
_jobject *HNDL __jni_NewIntArray
  (JNIEnv *RO env, jint len);
```

There are two differences between the old prototype and the new one. The first is that we annotate the argument and return types of `NewIntArray` with pointer kinds. The second change is that our system replaces every JNI API function call with a corresponding API wrapper call. The wrapper for `NewIntArray` is `__jni_NewIntArray`. The wrapper function does additional checks in addition to calling the corresponding JNI API function.

Then we enhanced CCured's type checker to reject C programs that perform reads/writes through handle pointers and writes through read-only pointers. We also changed CCured to allow certain casts such as a safe pointer to a read-only pointer and a read-only pointer to a handle pointer. We do not perform automatic pointer inference

```
//metadata: begining and end of the array
struct meta_seq {
   void *_b ;
   void *_e ;
};

struct seq_jint {
   jint *  _p ;
   struct meta_seq _ms ;
};

struct seq_jint __jni_GetIntArrayElements
(JNIEnv * env , jobject array , jboolean * isCopy)
{
  seq_jint f;
  f._p =
    (*env)->GetIntArrayElements(env,array,isCopy);
  f._ms._b = f._p;
  f._ms._e =
    f._p + (*env)->GetArrayLength(env,array);
  return f;
}
```

Figure 7: Wrapper function for `GetIntArrayElements`

of handle and read-only kinds, since those two kinds are created only to enforce correct usage of the JNI interface.

Since we need to set up the array bounds for arrays returned by functions such as `GetIntArrayElements`, its wrapper function (`__jni_GetIntArrayElements`, in Figure 7) returns metadata for the array in addition to the array pointer. The metadata are the beginning and end of the array. Using these metadata, any use of the array can be dynamically checked to make sure no out-of-bound array access.

Wrapper functions are also good places to insert dynamic checks. For example, to achieve data privacy, we dynamically check that the member is not private in the wrappers for functions such as `GetFieldID`, which gets the field ID of a field in a Java class. We have not implemented class checking and pending exception checking in our prototype system, although adding them should be straightforward. We have also not implemented our scheme for safe memory management and this requires some work.

6.1 Experiments

We have tested our system on many small examples that exploit JNI loopholes. Our system is able to catch unsafeties inside those examples either statically or dynamically.

To fully evaluate our system's impact on performance, we carried out one experiment on the Zlib compression library (nearly 9000 lines of C code) distributed with JDK 1.4.2. Zlib is a general-purpose data compression C library that is meant to be called C programs. On top of the Zlib library, JDK provides an extra 262 lines of interface code that link Java code with Zlib through JNI. JDK also provides Java classes (java.util.zip) that can be used by programmers to perform compression and decompression; these classes contains native methods that are implemented by those interface code.

Since our system is built on top of CCured, the performance overhead includes the cost for CCured to ensure the internal safety of C code, and the cost for our system to ensure safe interoperation. Thus our first experiment is to test the overhead added by CCured to the Zlib library. The result is shown in the top table of Table 3: the first column shows the running time in the case of a C main function calling the Zlib library; the second column is the case after applying the CCured tool. The result show that CCured adds 46% overhead to Zlib.

C main + Zlib	C main + CCured Zlib
1.38s	2.02s (1.46x)

JNI + Zlib	Safe JNI + CCured Zlib
1.80s	2.82s (1.57x)

Pure Java
3.13s

Table 3: Performance measurement on Zlib (seconds). All experiments are to gzip a 13MB file on a Linux cycle server with 4 CPUs and 1GB memory; buffer size is 16KB; results reported are the average of five tests.

	total lines	lines changed
Zlib library	8933	155
Interface code	262	84

Table 4: Lines changed in the curing process for Zlib.

In the second experiment, through JNI, Java passes a buffer of data to the Zlib library to perform compression. The result is shown in the second table of Table 3. Compared to the unsafe case (the first column), our system (including CCured) adds a total of 57% overhead.

Finally, we also tested the performance of a pure Java implementation of the Zlib library (jzlib-1.0.5). The result show that it is 11% slower than the case of using our system.

Although we have only done one set of experiments, the result showed our tool adds an acceptable performance overhead to achieve safety. Also, our system has a slight performance advantage over a pure Java implementation, let alone the fact that people use JNI because they do not want to spend time and energy to port everything into Java. One thing worth mentioning is that we have not done any optimization to our system, such as inlining JNI API wrapper function calls.

One limitation of our approach is that programmers occasionally need to modify the source code. One case is to add type annotations so that CCured has a better understanding of the code to insert less dynamic checks. Another case is to change C code so that it uses JNI properly to pass our static type checker. We report the number of lines changed during the curing process of Zlib (Table 4). We need to change 155 lines out of 8933 lines to let CCured to cure C code; this is a small portion. For the interface code that uses JNI to communicate between Java and Zlib, we changed 84 lines out of 262 lines. The reason for this change is that we identified a vulnerability in the original implementation; we describe the bug and our change next.

6.2 Uncovering a vulnerability in JDK

In the `java.util.zip` of JDK 1.4.2, the class `Deflater` has native methods which serve as wrappers to call functions in the underlying Zlib C library to do compression. The Zlib

```
/* Bug.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 to execute Bug.java in a secure
   environment */
grant {
  permission java.lang.RuntimePermission
                          "accessDeclaredMembers";
  permission java.lang.reflect.ReflectPermission
                          "suppressAccessChecks";
};
```

Figure 8: An exploitable bug in the JVM.

C library maintains a structure (`z_stream`) to store information related to a compression data stream. Java objects of class `Deflater` need to store a pointer to a `z_stream` structure, so that when the object calls Zlib the second time, all the state information is still available. However, the problem is that `z_stream` is a C structure, and it is difficult for Java to define a pointer to a C structure.

JDK avoids this problem by storing the pointer into a private field as a Java long. Other native methods cast this Java long field back into a `z_stream` pointer and use it to call functions in Zlib.

If we assume that the native methods are only called by the JVM, the definition of `Deflater` never changes the long field, and Java's data privacy guarantees are respected, we would conclude that the cast is in fact safe. *Our system, on the other hand, thinks it is a bad cast and rejects the C code.* Initial we assumed our system was being too conservative because, under reasonable assumptions, it seems like the code should be safe. So, it seems that for this code be accepted by our system, we must also analyze Java code to avoid being too conservative. However, it turns out that one of our "reasonable assumptions" is wrong. The code is actually unsafe.

Java reflection considered harmful. Java provides a reflection API to aid in the debugging and dynamic loading of unknown Java code at runtime. It turns out that a Java program given the appropriate permissions can at runtime bypass the data privacy constraints of any object. Figure 8 demonstrates the bug as well as the minimum set of security permissions need to exploit the bug when the Java security manager is active. The code sets a private long field in `Deflater` to an arbitrary illegal value. If this were a normal Java class, doing so would perhaps break the `Deflater` class but not violate type safety of the JVM. However, since this private long happens to be a C pointer that is passed to a native method, the results are devastating. This is a pervasive problem for many JVM implementations. In fact, the relatively simple Java program in Figure 8 crashes the latest versions of Sun's Java VM on three platforms, as well as the latest JVM for MacOS X and IBM's VM. This same problem also appears in the Kaffe JVM.

Fortunately, the default security policy when running untrusted Java code does not allow our exploit to work. However, when given the right security privileges, code using our attack can gain access to all privileges. The ability for untrusted code to escalate the set of security privileges given to it is clearly a violation of the intended security policy provided by the Java security model.

Since it seems many programmers of the JNI code did not take into consideration the possibility of the Java reflection API violating data privacy guarantees, code that seems safe is in fact not. So, although our system at first seemed too conservative, it in fact helped us discover a real source of unsoundness in many commercially deployed JVMs.

Our fix. Our change is to introduce an indirection table of `z_stream` pointers, very much in the spirit of an OS file descriptor table. We store table IDs, not pointers, into objects of `Deflater`. Native methods using `z_stream` pointers do a table lookup by a table ID and verify that the ID returned is in fact a valid ID. A more systematic fix will likely require changes to either the JNI or Java Reflection API standards.

7 Related Work

NestedVM [1] is another approach for JVMs to link with unsafe native code. It translates MIPS machine code (compiled from source native code) into Java code which implements a virtual machine on top of JVM. NestedVM achieves safety and security by putting native code into a separate virtual machine and allowing only controlled interaction with JVMs. This approach is similar to COM and CORBA model in the sense that they all achieve safety by separation. However, they all suffer efficiency problem. NestedVM incurs 200% to 900% overheads, compared to our 57%.

Janet [3] is a Java language extension. It provides a more clear interface than JNI for programmers to write a combination of Java and C code in the same file. Janet's translator translates the source file into separate Java and C files that conform to the JNI interface. By having an easy-to-use interface for programmers to access Java features from C, Janet makes JNI programming less error-prone, but it does not guarantee a safe interoperation. For example, C code can still do out-of-bound array access.

Java 2 SDK supports a "-Xcheck:jni" option that optionally turns on additional checks for JNI API functions. IBM's JVM [6] does a more extensive checking. Some of these checks are similar to what we do. The problem is that each JVM does its own set of checks, which are usually not documented. Also, we have shown that only checking across the interface is not enough to achieve type-safe JNI, such as when enforcing array bounds.

Our paper addresses the interoperation problem between safe Java and unsafe C, Trifonoc and Shao [11] addresses the problem of interoperation between two safe languages when they have different systems of computation effects such as exceptions.

8 Conclusion

We have shown how different programming languages can safely interoperate. In particular, we study the case of interoperation between safe Java and unsafe C through JNI. As a conclusion, we summarize two main contributions in this work:

- We have identified a series of loopholes in the standard Java Native Interface implemented in all JVMs. Any solution has to address these issues.
- We have designed a solution that makes calling native methods in C as type-safe as pure Java code.
- We have proved, implemented and tested a prototype system that includes a significant part of our design. This simple system has already helped us to identify one vulnerability in JDK.

Acknowledgment

We thank Matthew Harren and George Necula for providing us a CCured version of the Zlib library and for helping us to set up our experiments.

References

[1] Brian Alliet and Adam Megacz. Complete translation of unsafe native code to safe bytecode. In *ACM 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME'04)*, 2004.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

[3] M. Bubak, D. Kurzyniec, and P. Luszczek. Creating java to native code interfaces with janet extension. In *Proceedings of the First Worldwide SGI Users' Conference*, pages 283–294, 2000.

[4] Object Mangagement Group(OMG). Common object request broker architecture: Core specification, version 3.0.3. `http://www.omg.org/docs/formal/04-03-01.pdf`, 2004.

[5] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

[6] IBM. IBM developer kit and runtime environment, Java 2 technology edition, version 1.4.2, diagnostic guide., 2004.

[7] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

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

[9] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[10] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[11] Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 128–146, 1999.

[12] W3C. SOAP version 1.2 sepcification. `http://www.w3.org/TR/soap12-part1/`, 2003.