Safe Java Native Interface [*]

Gang Tan[†‡§] Andrew W. Appel[‡] Srimat Chakradhar[§]
Anand Raghunathan[§] Srivaths Ravi[§] Daniel Wang[‡]

[†] Department of Computer Science, Boston College
[‡] Department of Computer Science, Princeton University
[§] NEC Laboratories America

Abstract

Type safety is a promising approach to enhancing software security. Programs written in type-safe programming languages such as Java are type-safe by construction. However, in practice, many complex applications are heterogeneous, i.e.*, they contain components written in different languages. 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, in general, the overall application becomes unsafe.*

In this work, we propose a framework called Safe Java Native Interface (SafeJNI) that ensures type safety of heterogeneous programs that contain Java and C components. We identify the loopholes of using JNI that would permit C code to bypass the type safety of Java. The proposed SafeJNI system fixes these loopholes and guarantees type safety when native C methods are called. The overall approach consists of (i) retro-fitting the native C methods to make them safe, and (ii) developing an enhanced system that captures additional invariants that must be satisfied to guarantee safe interoperation. The SafeJNI framework is implemented through a combination of static and dynamic checks on the C code.

We have measured our system's effectiveness and performance on a set of benchmarks. During our experiments on the Zlib open source compression library, our system identified one vulnerability in the glue code between Zlib and Java. This vulnerability could be exploited to crash a large number of commercially deployed Java Virtual Machines (JVMs). The performance impact of SafeJNI on Zlib, while considerable, is less than reimplementing the C code in Java.

1 Introduction

Large software systems often contain components developed using different programming languages. This may occur due to a variety of factors, including ease of development and maintenance, reuse of legacy code, and efficiency. For example, a browser for an embedded device such as a cell phone may have a user interface written in Java, and modules written in C that implement plugins, media codecs, and so on.

For software components in different languages to interoperate, there must be a standard interface between them. One way to achieve this is through a *foreign function interface* (FFI). Most modern programming languages provide a foreign function interface [14, 3, 13, 6, 7]. For example, the Java Native Interface (JNI) [14] enables Java code running inside a Java Virtual Machine (JVM) to interoperate with components that are written in C.

Unsafe and insecure interoperation. An FFI usually addresses discrepancies between the two languages on issues such as the representation of data, memory management, calling conventions. However, what most interfaces fail to address is the discrepancy between the *safety guarantees* of different languages. When a component written in a safe language directly interacts with a component written in an unsafe language, the whole application becomes unsafe. This is certainly the case when Java code interoperates with C code through the JNI interface. Even rich systems like Microsoft's .NET CLR [10] 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.

Approaches to safe interoperation. Systems like COM [16], SOAP [18], and CORBA [9] are based on Remote Procedure Calls (RPC); they achieve safe interoperation by

[*] To appear in *IEEE International Symposium on Secure Software Engineering*, March 2006. This research was supported in part by ARDA award NBCHC030106. This information does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

[‡] Daniel Wang is currently at Microsoft Corporation.

placing components in different address spaces, thus isolating the unsafe components from the safe components. However, they still do not guarantee the safety of the individual components. Moreover, these approaches come with a significant overhead, in the form of context switches and inter-component communication. These costs limit the practical applicability of these approaches in many scenarios. We are interested in FFI-based approaches since they are more lightweight.

Another approach is to manually reimplement every component in safe languages. For instance, the JCraft website [1] provides reimplementations in pure Java of many programs originally written in C, including the X Window server, Zlib compression libraries, *etc*. This approach requires substantial programming effort, and also negatively impacts execution speed.

Towards safe interoperation through FFIs. Our goal is to have an FFI-based approach that achieves safe interoperation between different languages. In particular, we target the JNI interface, since Java and C are two popular languages. Ideally, we would like the calling of C code in Java to be as safe as the calling of Java code in Java. To achieve this, we have examined how C code, through JNI, may exploit loopholes to violate Java's type safety.

The most obvious problem is that C code is inherently unsafe and may read/write any memory address. Fortunately, there are systems such as CCured [15] and Cyclone [12] that provide safety guarantees for legacy C code through a combination of static and dynamic checks.

However, just providing internal safety for the C code is not sufficient to guarantee safe interoperation between Java and C. The JNI interface, if not used properly, has many loopholes. A simple example is that C code can pass objects of the wrong classes to Java and thus violate Java's type safety. In summary, ensuring the safety of the individual components separately is not sufficient to guarantee the safety of the entire program.

The contribution of this paper is the SafeJNI system, which can ensure type safety when Java programs interact with C programs through JNI. We decompose the overall problem into two parts: ensuring that the C code is safe in itself, and ensuring that the C code does not violate the safe interoperation rules of JNI. Our current implementation of SafeJNI leverages CCured [15] to provide internal safety for the C code. In addition, SafeJNI has the following components: we extend the type system of CCured to enforce invariants associated with JNI-specific pointers; we wrap dynamic checks around JNI API calls and insert checks into the C code; and finally, we implement a scheme to achieve safe memory management in JNI. We have also formalized a subset of our system and proved that our system is sufficient to guarantee safety.

Compared to RPC-based systems, our system is more lightweight, thanks to the lack of context switches. It is also more flexible since Java and C code are placed in the same address space. Compared to manual reimplementation of C components, our system provides a way to reuse legacy C libraries conveniently. Native libraries such as the Zlib compression/decompression library and image processing libraries can be safely invoked by Java with reasonable performance.

Furthermore, SafeJNI provides a way to improve the safety of the Java platform itself. Any implementation of a Java platform contains a significant amount of native C code in addition to Java code, for functionality and convenience. The Java code and C code interact through the JNI interface. For example, Sun's JDK 1.4.2 contains over 600,000 lines of native C code. Any error in the native code (*e.g.*, a buffer overflow) could lead to a violation of type-safety or security. Our system can render the C native code safe, and guarantee that it won't bypass Java's type system.

The rest of this paper is organized as follows. We offer a brief introduction to JNI in Section 2 and then enumerate the type-safety loopholes that we have identified in it. We describe our system, SafeJNI, in Section 3. In Section 4, we describe the experimental evaluation of the SafeJNI framework and discuss a vulnerability that we have discovered in JDK during our experiments. We conclude with related work and future work.

2 JNI and its loopholes

JNI is Java's mechanism for interfacing with 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 example of using JNI: Java code passes an array of integers to C code, which computes and returns the sum of the array. The Java code declares the method, `sumArray`, to be a native method using the keyword "`native`". Then, Java code calls 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 manipulates the array object through JNI API functions, *e.g.*, by calling `GetArrayLength` to get the length of the array and `GetIntArrayElements` to get a pointer to the array elements. So the common idiom of using JNI is that Java code passes Java-object references to C code; C code then 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

```
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");
  }
}
```

```
#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 object on which the method is invoked.
  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 code passes C code an array of integers and C code returns the sum of the array. On the left is the Java code; on the right is the C code.

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 Java type `int` is mapped to the C type `jint` (defined as 32-bit integers 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. 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. In some cases, these loopholes may be exploited to achieve malicious effects such as leakage of private data and execution of malicious code.

Direct access through Java references. Opaque references in C code are supposed to be manipulated only by JNI API functions. However, C code can perform direct reads/writes via these references, and potentially retrieve or corrupt the internal state of a JVM.

Interface pointers. An interface pointer 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 could replace a JNI API function with its own version and bypass any check in the function.

Out-of-bounds array access. Java often needs to pass an array of data to a native method. For the sake of efficiency, JNI functions such as `GetIntArrayElements` and `GetStringUTFChars` return pointers to the Java Heap and thus allow the native method to directly read/write the Java heap. It is easy for a native method to accidentally read or write past the bounds of the array.

Violating access control rules. JNI does not enforce access control on classes, fields, and methods that is expressed in the Java language through the use of modifiers such as `private`. Therefore, C code can read a private field of a Java object. As stated in the JNI manual [14, sect. 10.9], this was a conscious design decision, since native methods can access and modify any memory location in the heap anyway.

Manual memory management. JNI's scheme of managing memory is similar to `malloc`/`free` in C. 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. This kind of manual memory management has well known problems such as dangling pointers, multiple releases, and memory leaks.

Arguments of wrong classes. Since native code treats all references to Java objects as having one single type (`_jobject *`), an object of class `A` may be wrongly passed to a JNI API function that actually requires an object of class `B`. The JNI manual states that the behavior of the JVM is unspecified in this case. This usually results in a JVM crash in practice, but could be exploited to achieve more serious consequences.

Calling wrong methods. JNI provides different methods for accessing arrays of different types. For an integer array, C code should call `GetIntArrayElements`. For an array of `long` type, C code should call `GetLongArrayElements`. This is the case for many other operations, including accessing fields and invoking Java methods. C code may use a method that is for a wrong type, resulting in improper memory accesses.

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. With a pending exception, calling arbitrary JNI functions may lead to unexpected results.

Bypassing the security manager. Java's security model confines the capabilities of untrusted Java code. The 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 prevent the program from violating the security of the environment where the JVM is running.

3 Achieving safety in JNI

In this section, we describe our system, SafeJNI, which achieves type-safe interoperation between Java and C through JNI. Our system catches unsafe loopholes, and consists of three components. The first component is a static type system to ensure opaqueness of Java pointers, and to ensure that interface pointers are read-only. Second, SafeJNI inserts various runtime checks to ensure properties such as that objects to Java are of the right classes. Finally, SafeJNI implements a scheme to achieve safe memory management. Next, we describe each of these components in detail. Note that our current system addresses only type safety; we leave other security concerns in JNI as a subject for future work.

3.1 A static, JNI-specific pointer type system

We propose a 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, to avoid direct accesses to opaque references, and to prevent C code from overwriting function entries in an interface pointer.

Our pointer type system is an augmentation of CCured's type system. CCured [15] is a system that ensures memory safety in C. It analyzes C programs to identify places where memory safety might be violated and performs a source-to-source translation to insert runtime checks to ensure safety.

During the analysis phase, CCured classifies pointers into several categories according to their usage. Pointers in C programs that are used without pointer arithmetic and 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 information for the array that they point to, so that all dereferences can be dynamically checked to be within bounds. Pointers that involve type casts that cannot be understood statically (*e.g.*, casts from integers to pointers) 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.

Some of the pointer kinds in CCured are also used in SafeJNI to fix loopholes. For example, functions such as `GetIntArrayElements` return a pointer to an array. We model the array pointer as a CCured SEQ pointer, since C needs to perform 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, SafeJNI sets up the bounds when the `GetIntArrayElements` function is called; this is done by calling the `GetArrayLength` function.

The pointer kinds in CCured are not sufficient to ensure complete safety in JNI. For example, CCured cannot enforce the property of pointer opaqueness. Therefore, we extend CCured by adding two new pointer kinds to model pointers passed from Java to C.

Java handle pointers. As we have stated, references to Java objects from 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 around 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 SEQ pointers carry more privileges than SAFE pointers. However, SafeJNI does not allow casts to HNDL pointers since otherwise C could forge a Java reference through other kinds of pointers. We maintain the invariant that the only way to get a handle pointer is by calling a JNI API function.

Read-only pointers. We can read from the memory location through a read-only pointer but cannot write to the location. We model a Java interface pointer as a read-only pointer to prevent C code from replacing a function entry in

Pointer Kind	Description	Capability
t *HNDL	**Java handle pointers**	arguments to JNI APIs; equality testing
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 capabilities. Kinds in bold face are new in SafeJNI.

the table pointed to by the interface pointer. Our read-only pointers are related to the C `const` qualifier. For example, the 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 pointers, not with the type that the pointer points to.

In Table 1, we list all pointer kinds used in SafeJNI, including the ones in CCured. The SafeJNI system implements a type checker, which statically enforces usage of pointers according to their capability.

3.2 Insertion of dynamic checks

In addition to the static pointer type system, SafeJNI also inserts dynamic checks to address other loopholes in JNI. Some of these checks are performed immediately before and after a JNI API function is invoked, and therefore are placed into a wrapper for the API function. Other checks such as array bounds checking happen at the place where pointers are dereferenced. When these checks fail, the system will stop the program from running and print out a warning message. We next describe the kinds of dynamic checks the system inserts.

Runtime type checking. SafeJNI checks that objects passed to Java from C are of the right classes. For example, when `GetIntArrayElements` is invoked, the second argument is checked to be an integer array object. Similarly, when a Java method is invoked by C, the system checks that the number of arguments and the types of the arguments match the prescribed types of the method. This kind of checking is possible since Java keeps all class information at runtime.

Access control. SafeJNI inserts 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. Note that the access-control checks are not expensive since JNI uses a two-step process to access a field (or call a method): first get the field ID; then use the ID to access the value of the field. Access-control checks are necessary only in the first step; only one "get field ID" is necessary for arbitrarily many field accesses.

Since a native method is conceptually part of the class that declares the method, SafeJNI allows writing to private fields that belong to the self object (passed in as an argument to the native method).

Bounds checking. SafeJNI inserts bounds checks before each dereference of pointers to Java arrays.

Exception checking. The JNI manual [14, sect. 11.8.2] specifies a list of JNI API functions that can be called safely when there is a pending exception. For other functions, SafeJNI inserts checks to make sure there is no pending exceptions.

3.3 Safe memory management

We first show how JNI manages memory using an example. 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; see "pointer 1" in Figure 2.

2. In `GetIntArrayElements`, the JVM also pins the buffer so that Java's Garbage Collector (GC) will not garbage-collect it.[1] The ownership of the buffer has been transferred to C.

3. When the buffer is no longer needed, C calls `ReleaseIntArrayElements` on "pointer 1".

4. JVM un-pins the buffer and now the buffer is back to Java.

The above scheme is unsafe when C code continues to use "pointer 1". After step 4, "pointer 1" becomes dangling when Java's GC decides to garbage collect the buffer. To make matters more complicated, if C code makes a copy of "pointer 1" to get "pointer 2" in Figure 2, we have to restrict the use of "pointer 2" as well to be safe.

Next, we present two schemes to achieve safe memory management in JNI. The first scheme is simpler and is adopted in our current implementation, but it has an overhead for each pointer dereference. The scheme is depicted in Figure 3. It creates a validity tag in C for the buffer, and changes the representation of pointers to be a structure

[1] Instead of pinning the buffer, JVM may decide to make a copy of the buffer and return the pointer to the copy. In either case, the function returns a direct pointer into the JVM heap.

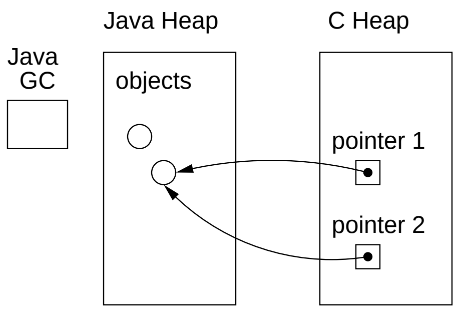

Figure 2. Memory management in JNI

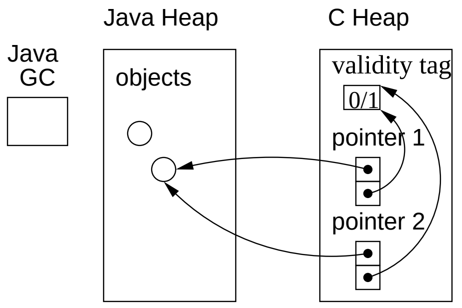

Figure 3. Safe memory management: Scheme I

that also has metadata pointing to the validity tag. When `GetIntArrayElements` is called, the validity tag is set to one. When `ReleaseIntArrayElements` is called, we first check that the tag is one to prevent multiple releases, and then set the tag to be zero. Furthermore, before each dereference of pointers that point to the buffer, we insert checks to ensure the tag is one, and thus guarantee the pointer will not be dereferenced after the release. This scheme safely manages memory, but each dereference comes with a cost.

Next, we present a scheme that avoids the per-dereference cost. The scheme is depicted in Figure 4. It assumes there is a C garbage collector in place; CCured already uses the Boehm conservative garbage collector [4] to reclaim storage. The scheme also uses a validity tag, but the tag itself is also a pointer pointing to the buffer. Same as the first scheme, it also changes the representation of pointers to be a structure that has metadata pointing to the validity tag. With these changes, there are pointers pointing to the buffer if and only if there are pointers pointing to the validity tag. Furthermore, we make a user's call to `ReleaseIntArrayElements` be a nop, and register a finalizer for the validity tag in the Boehm garbage collector. The finalizer will call `ReleaseIntArrayElements` to

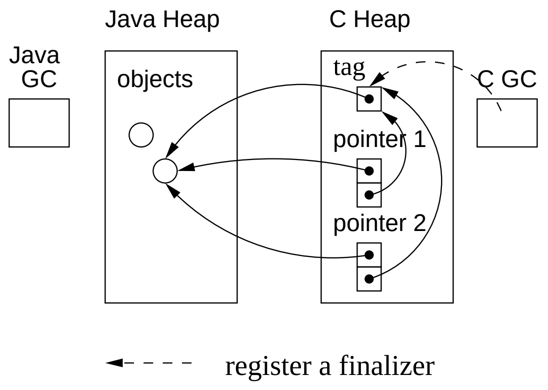

Figure 4. Safe memory management: Scheme II

release the buffer, when the C program has no live pointers to the tag. In this scheme, pointer dereferences have no extra cost, because the tag is not checked. On the other hand, this approach has the disadvantage of delaying collection of Java objects.

4 Experimental Results

To measure the effectiveness and overhead of our system, we have performed preliminary experiments. These experiments are conducted on programs that range from a set of small benchmarks to a real-world application. In general, these experiments show that our system is able to catch many loopholes, with reasonable slowdown. During our experiments, our system allowed us to identify a vulnerability in the implementation of Sun's JDK 1.4.2.

4.1 Microbenchmarks

We selected a set of microbenchmark programs from the JNI manual [14] to evaluate SafeJNI's effectiveness in catching loopholes in JNI, and to measure the overhead associated with common JNI operations.

To evaluate the effectiveness of SafeJNI, we deliberately injected unsafe code into otherwise safe programs to test if our system could catch the safety violations. The experiments showed that our system can catch all the loopholes[2] mentioned in Section 2.1. We describe some examples below.

The first example is the `Java_IntArray_sumArray` procedure shown in Figure 1. We inserted the following statement after the `ReleaseIntArrayElements` statement: "`sum+=*body;`". The inserted statement is unsafe

[2]Except that it is still possible to bypass Java's security manager.

because it dereferences a pointer after the pointer has been released. The pointer may be a dangling pointer if Java's GC has garbage-collected the array buffer. Since our system dynamically checks if every dereference of pointers like `body` is valid (or, has not been released), we got the following error message for the illegal dereference:

```
Failure JSEQ at IntArray.c:30:
Java_IntArray_sumArray():
The Java pointer has been released!
```

The message warns us that the pointer has been released, and the program is stopped at the point where the dereference occurred.

Next, we discuss an example in which the SafeJNI system statically catches safety violations. As we have discussed, all Java object pointers are opaque references. However, C code could try to perform direct read/write via these references. Our system statically prevents this direct access from happening. In the example in Figure 1, we inserted code to cast the handle pointer `arr` to a pointer through which the code performed direct access. Then our system complained

```
IntArray.c:29: Error: typecheck:
unfamiliar HNDL -> WILD cast
```

The message complained that, during type checking, it encountered a cast from a handle pointer to a wild pointer, through which direct access may happen.

These experiments demonstrate that SafeJNI is effective in catching loopholes in JNI, but it is at the expense of some overhead; in many cases, SafeJNI inserts runtime checks to ensure safety. To quantify the overhead, we have compiled a set of programs from the JNI manual. We have made changes so that each program will iterate for many times. The program we used are listed below:

- fieldaccess: Repeatedly reads fields from a Java object.
- callbacks: Repeatedly calls back a Java method.
- array: Repeatedly computes the sum of an integer array.
- string: Repeatedly accesses characters in a string.
- comp: Computes the sum of an integer array and then repeatedly adds random numbers to the sum; this is to simulate computation-intensive programs.

Table 2 presents the results of using SafeJNI on this set of microbenchmarks. Each result reported is the average of five runs. As the table shows, our system adds between 14% to 119% to the execution time of these programs. In general, programs with heavy pointer dereferences, including the array and the string programs, have the greatest slowdown. The reason is that, for each pointer dereference, our system performs bounds checking and also pointer validity checking (i.e., makes sure that the pointer has not been released). On the other side of the spectrum, computation-intensive programs have the smallest slowdown, since most of the running time is spent on computation on the C side.

Table 2 also presents the kinds of runtime checks involved in each test. For example, the fieldaccess test involves runtime type checking (check that the C type of the field matches the actual Java type and check that the right `Get<Type>Field` is called), access control (check that the field is not private), and exception checking (check that there is no pending exceptions).

4.2 The Zlib library

To fully evaluate our system's impact on performance, we have carried out an experiment on a real-word application—the Zlib compression library distributed with JDK 1.4.2. Zlib is a general-purpose data compression library, with nearly 9000 lines of C code. On top of Zlib, JDK provides an extra 262 lines of glue code that calls JNI API functions to link Zlib with Java. These glue code corresponds to the native methods in the JDK classes in `java.util.zip`. The classes in `java.util.zip` are then used by Java programmers to perform compression/decompression.

The following table presents the result of the performance test on Zlib. In the experiment, Java passes a buffer of data through JNI to the Zlib library to perform compression. We set the buffer size to be 16KB. The experiment compresses a 13MB file on a Linux server (DELL PowerEdge 2650; 2.2 GHz; 4 CPUs; 1GB memory). The result reported is the average of five runs.

	SafeJNI Ratio	CCured Ratio	Pure Java Ratio
Zlib	**1.63**	1.46	1.74

The result in the table shows that our SafeJNI system adds 63% to the running time of Zlib during our tests. We believe that the majority of the cost is incurred by the pointer bounds checking, and pointer validity checking. On the other hand, the slowdown of Zlib is not as great as the array and string tests, because Zlib also spends a lot of time on pure computation.

We also measured the performance cost if only CCured is used. CCured alone incurs 46% slowdown. Of course, CCured only guarantees the internal safety of C code, not the safe interoperation between Java and C. Finally, we also tested the performance of a pure Java implementation of the Zlib library (jzlib-1.0.5) [1]. The performance slowdown is

Name	**Ratio**	Lines of C/Java Code	Runtime checks				
			T	A	B	V	E
fieldaccess	**1.49**	28/20	√	√			√
callback	**1.29**	21/16	√	√			√
array	**2.19**	29/16	√		√	√	√
string	**2.09**	30/13	√		√	√	√
comp	**1.14**	32/16	√		√	√	√

Table 2. SafeJNI performance on a set of small programs. The results are presented as ratios, where 1.50 means that the program takes 50% longer to run when instrumented through SafeJNI. The "Runtime checks" column shows the kinds of runtime checks each program involves. The letter "T" stands for runtime Type checking, "A" for Access control, "B" for Bounds checking, "V" for pointer Validity checking, and "E" for exception checking.

74%. Our system is faster than the pure Java implementation in the Zlib example, not to mention the time and effort incurred in porting everything into Java.

One inconvenience of our system is that programmers occasionally need to modify the source code to have good performance. Our system could cure unmodified C programs, but this would usually incur a large overhead on execution time. The main reason is that, without help from the programmers, CCured is typically unable to prove the safety of many bad casts in a large program; CCured will make all pointers involved in such casts as WILD pointers. WILD pointers in CCured come with space and time overhead. Therefore, getting a good performance requires some annotation and modification to the original program so that CCured is able to eliminate those WILD pointers. We report the number of lines changed during the curing process of Zlib as follows:

	total lines	lines changed
Zlib library	8933	155
JNI glue code	262	84

We changed 155 lines out of 8933 lines to let CCured to cure Zlib; this is a small portion. For the glue code between Java and Zlib, we changed 84 lines out of 262 lines. The reason for this rather big change is that we identified a vulnerability in the glue code and had to fix it; we describe the vulnerability and our fix next.

4.3 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 invoke functions in the underlying Zlib C library. The Zlib C library maintains a structure (`z_stream`) to store information related to a compression data stream. A Java object of class `Deflater` needs to store a pointer to the `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 in a private field as a Java long. Other native methods cast this Java long field back into the `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 (a cast from an integer to a pointer) and rejects the C code*. Initially we assumed our system was being too conservative because, under reasonable assumptions, it seemed like the code should be safe. However, it turns out that one of the "reasonable assumptions" is wrong. The code is actually unsafe with Java reflection APIs.

Java reflection considered harmful. Java provides reflection APIs to aid in the debugging and dynamic loading of unknown Java code at runtime. With reflection APIs, a Java program can at runtime arbitrarily change the private long field that stores the pointer to the `z_stream` structure. Figure 5 demonstrates this exploit as well as the minimum set of security permissions needed when the Java security manager is active. The code sets the 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 5 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 Java reflections and thus

```
/* 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 5. An exploitable vulnerability in the JVM.

not allow our exploit to work. However, when given the right security privileges, we believe that our exploit can enable us to 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.

Our fix. We changed the glue code to introduce an indirection table of `z_stream` pointers, very similar to an OS file descriptor table. We store table IDs, not pointers, into objects of class `Deflater`. Native methods using `z_stream` pointers perform a table lookup by a table ID; if the table ID is not in the table, the code will warn and stop.

5 Formal Proof of Safety

We have described our system for safe interoperation through JNI. It is natural to wonder how we can be sure that the techniques presented are sufficient to ensure type safety. As a first step towards a formal proof, we have formalized a subset of our SafeJNI system, which includes many pointer kinds, a representative subset of JNI API functions, and the runtime checks that the system inserts. The formalization is described in the appendix. Based on this formalization, we have proved the following safety theorem: C programs that pass SafeJNI's type system and do not violate any of the inserted dynamic checks will not violate memory safety or access Java's memory arbitrarily.

6 Related Work

A recent concurrent work by Furr and Foster [8] has a similar goal as ours: to prevent foreign function calls from introducing loopholes into otherwise safe languages. Their system targets OCaml's foreign function interface instead of JNI. Their work tracks OCaml types in C to statically prevent C code from misusing OCaml types. Since OCaml does not carry type information during runtime as Java does, statically tracking types is a reasonable way to proceed. Since all type information is available in Java at runtime, in some cases we choose to insert dynamic checks. However, it is conceivable to use the techniques of Furr and Foster to eliminate some of the dynamic checks. Another point is that the work by Furr and Foster does not guarantee the internal safety of C. Consequently C code can read/write any memory location, by casting an integer to a pointer, and thus render the whole system unsafe.

NestedVM [2] is another approach for JVMs to link with unsafe native code. It translates MIPS machine code (compiled from source native code) into Java code that 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 the efficiency problem. NestedVM incurs 200% to 900% overhead, which is much higher than our system.

Janet [5] is a Java language extension. It provides a clearer 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 safe interoperation. For example, C code can still perform out-of-bounds array access.

The Java 2 SDK supports a "-Xcheck:jni" option that optionally turns on additional checks for JNI API functions. IBM's JVM [11] performs a more extensive checking. Some of these checks are similar to what we do. The problem is that each JVM implementation performs its own set of checks, which are usually not documented. Also, we have shown that checking across the interface is not enough to achieve safety of JNI.

Our work addresses the interoperation problem between a type-safe language (Java) and an unsafe language (C); Trifonov and Shao [17] address the problem of interoperation between two safe languages when they have different systems of computation effects such as exceptions.

7 Future work

There are several aspects where the SafeJNI system could be improved. The first feasible improvement is to reduce the number of dynamic checks by static analysis. Our system enables Java to use C code safely without porting all the C code to Java, but it comes with a performance slowdown. We believe that simple static analysis techniques can reduce a large number of dynamic checks. For example, many pointer validity checks for pointer dereferences can be eliminated if static analysis can guarantee that the dereferences happen within a pair of *get* and *release* methods. Another possibility is to track the Java types in C using the techniques of Furr and Foster [8], so that some of the runtime type checks are unnecessary.

Our system cannot prevent C code from bypassing JVM's security manager, because C code can invoke system calls directly to perform insecure operations, such as writing to files. A solution to this would be to replace C code's system calls with calls to secure versions that consult JVM's security manager first.

Finally, while our system targets the JNI interface, some of the proposed techniques should be applicable to other scenarios as well. One interesting future work is to investigate the interaction between managed and unmanaged code in .NET CLR.

Acknowledgment

We would like to thank Matthew Harren and George Necula for helping us set up the Zlib experiment.

References

[1] JCraft. `http://www.jcraft.com/`.

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

[3] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electr. Notes Theor. Comput. Sci.*, 59(1), 2001.

[4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

[5] 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.

[6] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional programming*, pages 114–125, 1999.

[7] K. Fisher, R. Pucella, and J. H. Reppy. A framework for interoperability. *Electr. Notes Theor. Comput. Sci.*, 59(1), 2001.

[8] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 62–72, 2005.

[9] O. M. Group(OMG). Common Object Request Broker Architecture: Core Specification, Version 3.0.3. `http://www.omg.org/docs/formal/04-03-01.pdf`, 2004.

[10] J. Hamilton. Language integration in the Common Language Runtime. *SIGPLAN Notices*, 38(2):19–28, 2003.

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

[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] X. Leroy. The Objective Caml system, release 3.08. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, Aug. 2004.

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

[15] G. C. Necula, S. McPeak, and W. 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.

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

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

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

A Formalization and Soundness Proof

To have a formal claim of our safety guarantee, we have extended CCured's formalization [15] to include the handle pointer kind and a representative subset of JNI API functions. Based on this formalization, We have proved a safety theorem: C programs that pass SafeJNI's type system and do not violate any of the inserted dynamic checks will not violate memory safety or access Java's memory arbitrarily.

On the other hand, the formalization models only a subset of our system. It does not model read-only pointers, although this should be straightforward. It does not model wild pointers; the modeling is the same as the one in CCured. We do not model our memory management scheme, whose modeling we believe requires the modeling of Java states and Java memory. We also do not model many checks the SafeJNI system inserts, including runtime type checking, access control, and exception checking.

Figure 6 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 _jobject for the type of all Java objects. In addition to safe pointers ($\tau * \mathsf{SAFE}$) and sequence pointers ($\tau * \mathsf{SEQ}$), we also have handle pointers. ($\tau * \mathsf{HNDL}$).

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 $*e$ 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 $*e_1 = e_2$ in C), and a subset of JNI API functions. These functions are described in Table 3. 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 *Hndl*$(?)$. Since handle pointers are opaque to C programs, their exact values do not matter; so we use *Hndl*$(?)$ to represent all Java handle pointers. Values of safe pointer types are of the form *Safe*(n), where n is the pointer value. Values of sequence pointer types are of the form *Seq*(n, b, e), where b and e are metadata and are the beginning and the end of the array, respectively.

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 3. Explanation of some JNI API functions

A.1 Operational semantics

In Figure 7, 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: $\Sigma, M \vdash_{\mathsf{s}} s \Downarrow v, M'$

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 7 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

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

Figure 6. Language Syntax

Expressions:

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

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

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:

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

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

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

$$\frac{\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Hndl}(?) \quad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow n}{\Sigma, M \vdash_{\mathsf{s}} \mathrm{GetObjectArrayElement}(e_1, e_2) \Downarrow \mathit{Hndl}(?), M} \qquad \frac{\begin{array}{c}\Sigma, M \vdash_{\mathsf{e}} e_1 \Downarrow \mathit{Hndl}(?) \qquad \Sigma, M \vdash_{\mathsf{e}} e_2 \Downarrow n \\ \Sigma, M \vdash_{\mathsf{e}} e_3 \Downarrow \mathit{Hndl}(?)\end{array}}{\Sigma, M \vdash_{\mathsf{s}} \mathrm{SetObjectArrayElement}(e_1, e_2, e_3) \Downarrow 0, M}$$

Figure 7. Operational semantics. The boxed premises are runtime checks.

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

A.2 Type system

Our type system is expressed by the following judgments:

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

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

Figure 8 presents our type system. Rules for SAFE and SEQ pointers are the same as those in CCured. There are several points worth mentioning. 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 argu-

Expressions:

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

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

Casts:

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

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

Statements:

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{e}} e_1 : \mathsf{\_jobject} \ast \mathsf{HNDL} \qquad \Gamma \vdash_{\mathsf{e}} e_2 : \mathsf{int} \\ \Gamma \vdash_{\mathsf{e}} e_3 : \mathsf{\_jobject} \ast \mathsf{HNDL}\end{array}}{\Gamma \vdash_{\mathsf{s}} \mathrm{SetObjectArrayElement}(e_1, e_2, e_3) : \mathsf{int}}$$

Figure 8. Typing rules

ment type of `GetArrayLength`. Last, the return type of `GetIntArrayElements` is modeled as CCured's SEQ pointer, or an array pointer.

A.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. [3]

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

$$\begin{array}{lcl}
[\![\mathsf{int}]\!]_\Lambda & = & \{\, n \mid n \in \mathsf{N} \,\} \\
[\![\tau \ast \mathsf{HNDL}]\!]_\Lambda & = & \{\mathit{Hndl}(?)\} \\
[\![\tau \ast \mathsf{SAFE}]\!]_\Lambda & = & \\
\multicolumn{3}{l}{\quad \{\, \mathit{Safe}(n) \mid n \in \mathrm{dom}(\Lambda) \ \wedge \ \Lambda(n) = \tau \,\} \cup \{\mathit{Safe}(0)\}} \\
[\![\tau \ast \mathsf{SEQ}]\!]_\Lambda & = & \\
\multicolumn{3}{l}{\quad \{\, \mathit{Seq}(n, b, e) \mid [b, e) \subseteq \mathrm{dom}(\Lambda) \ \wedge \ \forall b \leq i < e.\ \Lambda(i) = \tau \,\}}
\end{array}$$

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

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

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 "$\tau\ *$ HNDL" are of the form *Hndl*$(?)$ and thus are pointers to Java objects.