An Empirical Security Study of the Native Code in the JDK [*]

Gang Tan and Jason Croft
Boston College
gtan@cs.bc.edu, croftj@bc.edu

Abstract

It is well known that the use of native methods in Java defeats Java's guarantees of safety and security, which is why the default policy of Java applets, for example, does not allow loading non-local native code. However, there is already a large amount of trusted native C/C++ code that comprises a significant portion of the Java Development Kit (JDK). We have carried out an empirical security study on a portion of the native code in Sun's JDK 1.6. By applying static analysis tools and manual inspection, we have identified in this security-critical code previously undiscovered bugs. Based on our study, we describe a taxonomy to classify bugs. Our taxonomy provides guidance to construction of automated and accurate bug-finding tools. We also suggest systematic remedies that can mediate the threats posed by the native code.

1 Introduction

Since its birth in the mid 90s, Java has grown to be one of the most popular computing platforms. Recognizing Java's importance, security researchers have scrutinized Java's security from its early days (c.f., [9, 29, 33, 26]). Various vulnerabilities in the Java security model have been identified and fixed; formal models of various aspects of Java security have been proposed (e.g., [41, 14]), sometimes with machine-checked theorems and proofs [23].

In this paper we examine a less-scrutinized aspect of Java security: the native methods used by Java classes. It is well known that once a Java application uses native C/C++ methods through the Java Native Interface (JNI), any security guarantees provided by Java might be invalidated by the native methods. Figure 1 shows a contrived example. The Java class "Vulnerable" contains a native method, which is realized by a C function. The C function is susceptible to a buffer overflow as it performs an unbounded string copy to a 512-byte buffer. Consequently, an attacker can craft malicious inputs to the public Java byteCopy() method, and overtake the JVM.

Java code

```
class Vulnerable {
  //Declare a native method
  private native void bcopy(byte[] arr);
  public void byteCopy(byte[] arr) {
    //Call the native method
    bcopy(arr);
  }
  static {
    System.loadLibrary("Vulnerable");
  }
}
```

C code

```
#include <jni.h>
#include "Vulnerable.h"
JNIEXPORT void JNICALL Java_Vulnerable_bcopy
  (JNIEnv *env, jobject obj, jobject arr)
{
  char buffer[512];
  jbyte *carr;
  carr = (*env)->GetByteArrayElements
                  (env,arr,0);
  //Unbounded string copy to a local buffer
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements
            (env,arr,carr,0);
}
```

Figure 1: Vulnerable JNI Code.

Due to the fundamental insecurity of native C/C++ code, the default policy of Java applets, for example, does not allow loading non-local native code. Nonetheless, there is already a large amount of trusted native code that comprises a significant portion of the Java Development Kit (JDK). For instance, the classes under java.util.zip in Sun's JDK are just wrappers that invoke

[*]To appear in the 17th USENIX Security Symposium, San Jose, CA, 2008.

the popular Zlib C library. In JDK 1.6, there are over 800,000 lines of C/C++ code. Over the time, the size of C/C++ code has been on the increase: JDK 1.4.2 has 500,000 lines; JDK 1.5 has 700,000 lines; and JDK 1.6 has 800,000 lines. Any vulnerability in this trusted native code can compromise the security of the JVM. Several vulnerabilities have already been discovered in this code [34, 38, 37].

Since the native code in the JDK is critical to Java security, examining and ensuring its security is of great practical value. As a first step toward this goal, we have carried out an empirical security study of this large and security-critical code. Our research makes the following contributions:

- This is the first systematic security study of the native code in Sun's JDK, a security-critical and ubiquitous piece of software. A few sporadic bug reports exist, but none have scrutinized this aspect of Java security.
- We discovered previously unknown security-critical bugs (59 in total). By removing them, the overall Java security will be strengthened. Furthermore, we describe a taxonomy of bugs based on our study (Section 3). New bug patterns that arise in the context of the JNI are discussed and analyzed. Our taxonomy provides guidance to construction of scalable and accurate bug-finding tools.
- We will propose remedies (Section 4) to mediate the threats posed by the native code, with various trade-offs among security, performance, and effort. We also discuss limitations of current approaches and point out future directions.

2 Overview of the JDK's native code and our approach to characterizing bug patterns

The JNI is Java's mechanism for interfacing with native C/C++ code. Programmers use the `native` modifier to declare native methods in Java classes (e.g. the `bcopy` method in Figure 1 is declared as a native method). Once declared, native methods can be invoked in Java in the same way as how ordinary Java methods are invoked. Programmers then provide in C or C++ implementation of the declared native methods. The implementation can use various API functions provided by the JNI interface to cooperate with the Java side. Through the API functions, native methods can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on.

In the source directories `share/native`, `solaris/native`, and `windows/native` of Sun's JDK 1.6 (v6u2), there are over 800,000 lines of C/C++ code (counted using `wc`). The native code in these directories implements the native methods declared in the JDK classes. The native code in the directory `share/native` is shared across platforms, while the code in `solaris/native` and `windows/native` is platform dependent. The majority of the native code in the JDK is in the C language; around 700,000 lines are in C, while the rest are in C++. In our following discussion, we will mostly refer to the C code in the JDK. All of our discussion, unless specially noted, applies to the C++ code as well.

The 800k lines of native code can be conceptually divided into two parts: *library code* and *interface code*. The library code is the C code that belongs to a common C library. For example, the code under `share/native/java/util/zip/zlib-1.1.3` is from Zlib 1.1.3. The interface code implements Java native methods, and glues Java with C libraries through the JNI. For example, the C code in `native/java/util/zip/Deflater.c` implements the native methods in the `java.util.zip.Deflater` class, and glues Java with the Zlib C library.

Our approach to characterizing bug patterns. Given the large amount of trusted native code in the JDK, bugs are likely to exist. Our ultimate goal is to build highly automatic tools that can identify bugs in the JDK's native code. However, as no general methodology exists to identify all bugs accurately in a program, we believe that the important first step is to collect empirical evidence, and characterize relevant bug patterns. Only after this due diligence, we can select the right techniques to take advantage of the domain knowledge of the JDK and the JNI, and construct effective bug-finding tools.

In the first step, we intend to cover as many bug patterns as we can. We decided to scan the source code using off-the-shelf static analysis tools and also simple tools (scripts and scanners) built by us. Although these tools are inaccurate, their scanning results are fairly complete and thus enable us to compile enough evidence to conclude the characteristics of bug patterns. Next, we discuss the tools used in our study:

- To scan the common bug patterns inside C code, such as buffer overflows, integer overflows, and race conditions, we used a combination of Splint [11], Cigital's ITS4 [39], and Flawfinder [42]. We chose a combination of these tools, rather than a single one, because their strengths complement one another. For example, Splint performs full parsing and can flag many incompatible type casts. ITS4 and Flawfinder can flag time-of-check-to-time-of-use (TOCTTOU) flaws, among others.

- Some bug patterns in the JDK's native code are particular to the Java Native Interface (JNI) and we cannot use existing tools to scan for errors in these patterns. We have built simple tools, including grep-based scripts and scanners implemented in CIL [31], to search for bugs in these patterns.

- For the list of warnings produced by the static analysis tools, we manually inspected the source code to identify true bugs. To help the manual inspection, we used the GNU GLOBAL Source Code Tag System [17] to build a database of tags in the JDK source code, and used htags to generate HTML files for the source code. This made source-code navigation much easier. For example, with one click, we can find all places where a particular function is invoked.

Although the foregoing approach is sufficient for characterizing bug patterns, it is clear the tools will not be scalable to cover all 800,000 lines of native code in the JDK. In Section 4, we will discuss techniques that make a significant progress toward providing safety to the JDK's native code.

Target directories. Limited by our time to perform manual inspection, we focused our study on the code under the directories `share/native/java` and `solaris/native/java`. We will call these directories the target directories in the following text. The target directories include approximately 38,000 lines of C code, which implement the native methods in the `java.*` classes.

3 Taxonomy of bugs in the JDK's native code

We now present a collection of bug patterns in the JDK's native code. Some of these patterns are well known, such as buffer overflows, but we will discuss them in the context of the JDK. Some bug patterns are due to the mismatch between Java's programming model and C's, and thus are unique in the context.

Table 1 shows a summary of the results of our security study. For each bug pattern, the table shows the number of bugs we identified. We include a bug in the table when two conditions hold. First, there must be a *programming error* in the native code. For example, the C code in Figure 1 has a programming error, which performs an unbounded string copy. The second condition is that an attacker must be able to trigger the programming error. For the example in Figure 1, the attacker can trigger the error of unbounded string copy by passing malicious data to the `Vulnerable` class.

Table 1 also classifies whether a bug pattern is *security critical*. We define that a bug pattern is security critical if, by exploiting bugs in the pattern, an attacker can take over the JVM, gain authorized privileges, or crash the JVM (a denial-of-service attack). A security-critical bug is a vulnerability.

Finally, Table 1 shows the static analysis tools we used to identify the bugs in a bug pattern, and the section that describes detailed findings on the bug pattern. In each section, we will show representative examples, but refer readers to the appendix of our technical report [**?**] for a full list of the bugs we identified. We will also suggest ad-hoc fixes for some bug patterns, but defer discussions of more systematic remedies to the next section.

Not included in the table are the false positive rates of the static analysis tools; they will be presented when we discuss static analysis as a remedy in the next section.

3.1 Unexpected control flows due to mishandling Exceptions

The JNI interface provides API functions such as `Throw` and `ThrowNew` for raising Java exceptions. By throwing an exception, a native method can notify the JVM of errors. However, there is a mismatch between Java's exception-handling mechanism and the JNI's. In Java, when an exception occurs, the JVM automatically transfers the control to the nearest enclosing try/catch statement that matches the exception type. In contrast, an exception raised through the JNI does not immediately disrupt the native method execution, and only after the native method finishes execution will the JVM mechanism for exceptions start to take over. Therefore, JNI programmers must explicitly implement the control flow after an exception has occurred, by either immediately returning to Java or checking and clearing the exception explicitly using JNI API functions such as `ExceptionOccurred` and `ExceptionClear`.

Because Java and the JNI handle exceptions differently, it is easy for JNI programmers to make mistakes. Figure 2 presents a contrived example that shows how mishandling of exceptions may lead to vulnerabilities. At first sight, the `strcpy` from the incoming Java array to a local buffer is safe: there is a bounds check before the copy, and when the check fails, an exception is thrown. However, since the exception does not disrupt the control flow, the `strcpy` will always be executed and may result in an unbounded string copy. This example shows that mishandling exceptions creates unexpected control-flow paths where dangerous operations might happen.

The fix for the example in Figure 2 is simple—just put a return statement after the throwing-exception statement. However, it becomes complicated when function

Bug patterns		Errors	Security Critical	Static Tools Used	Section
Unexpected control flows due to mishandling exceptions		11	Y	grep-based scripts	3.1
C pointers as Java integers		38	N	Our scanner (implemented in CIL)	3.2
Race conditions in file accesses		3	Y	ITS4, Flawfinder	3.3
Buffer overflows		5*	Y	Splint, ITS4, Flawfinder	3.4
Mem. management flaws	C mem.	1	N	Splint	3.5
	Java mem.	28	N	grep-based scripts	
Insufficient error checking	JNI APIs	35	Y	grep-based scripts	3.6
	misc.	5	Y	Splint	
Total		126	59		

*One buffer-overflow flaw is not in the target directory.

Table 1: A summary of the bugs we identified in the target directories.

```
void Java_Vulnerable_bcopy (JNIEnv *env, jobject obj, jbyteArray jarr) {
  char buffer[512];

  if ((*env)->GetArrayLength(env, jarr) > 512) {
    JNU_ThrowArrayIndexOutOfBoundsException(env, 0);
  }

  //Get a pointer to the Java array, then copy the Java array to a local buffer
  jbyte *carr = (*env)->GetByteArrayElements(env, jarr, NULL);
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements(env,arr,carr,0);
}
```

Figure 2: An example of mishandling JNI exceptions

calls are involved. Imagine a C function, say `f`, invokes another C function, say `g`, and the function `g` throws an exception when an error occurs. The `f` function has to explicitly deal with two cases of calling `g`: the successful case, and the exceptional case. Mishandling it may result in the same error as the one in Figure 2. It becomes much more complicated when the C function `f` invokes a Java method. The JVM mechanism for exceptions will not take effect until the C function returns, even for the exceptions raised in the Java method.

We developed a grep-based script to search for all places where an exception is explicitly thrown. Of the 337 hits in the target directories, we found 11 places where the control flows for exceptions are implemented incorrectly. A representative example from `solaris/native/java/lang/UNIXProcess_md.c` is shown in Figure 3. The macro `NEW` invokes the function `xmalloc`, which in turn invokes `malloc` to allocate a specified amount of memory. If the `malloc` function returns null, the `NEW` throws a `JNU_ThrowOutOfMemoryError` exception. However, the exception does not disrupt the control flow, and as a result the `pathv` variable in `splitPath` gets null. The subsequent "`pathv[count] = NULL`" will crash the JVM.

We classify this bug pattern as being security critical because dangerous operations in unexpected control-flow paths may enable an attacker to crash or take over the JVM.

3.2 C pointers as Java integers

Programs that use the JNI often need to pass C pointers through Java. Due to differences between Java's type system and C's, it is difficult (and sometimes impossible) for Java to assign types to C pointer values. The commonly used pattern in JNI programming is to cast C pointers to Java integers, and pass the resulting integers.

```
static void* xmalloc(JNIEnv *env, size_t size) {
  void *p = malloc(size);
  if (p == NULL) JNU_ThrowOutOfMemoryError(env, NULL);
  return p;
}

#define NEW(type, n) ((type *) xmalloc(env, (n) * sizeof(type)))

static const char * const * splitPath(JNIEnv *env, const char *path) {
  ...
  pathv = NEW(char*, count+1);
  pathv[count] = NULL;
  ...
}
```

Figure 3: An excerpt from solaris/native/java/lang/UNIXProcess_md.c. Even when `xmalloc` returns NULL, "pathv[count] = NULL" will be executed.

The pattern is used, for example, in the class `java.util.zip.Deflater`. The `Deflater` class supports compression using the Zlib C library. The Zlib library maintains a C structure (`z_stream`) for storing the state information of a compression data stream. A `Deflater` object holds a pointer to the `z_stream` structure, so that when the object calls Zlib the second time, the state information can be recovered through the pointer. As it is impossible for Java to declare the pointer as having the C type "`z_stream *`", the C code casts it into an integer before passing it to Java:

```
typedef struct z_stream_s {...} z_stream;

jlong Java_java_util_zip_Deflater_init
        ( ... ) {
  z_stream *strm =
    calloc(1, sizeof(z_stream));
  ... //initialize strm
  return (jlong) strm; //cast it to an integer
}
```

Whenever Java needs to access the compression stream, it passes to C the integer. C code then casts the integer back to a `z_stream` pointer, through which the state information of the stream can be retrieved or updated.

From Java's perspective, integers that represent C pointers are just ordinary Java integers. The pattern of treating C pointers as Java integers is unsafe if an attacker can inject to the C side arbitrary integer values that will be interpreted as pointers. Greenfieldboyce and Foster [18] examined the Gimp Toolkit (GTK) and discovered seven places where the injection of arbitrary integers is possible. For example, the native method `setFocus` in the GTK (shown below) has an integer parameter that represents a window pointer. Since the method is declared as a public method, an attacker can invoke it with an arbitrary integer value, which may corrupt memory and result in JVM crashes.

```
class GUILib {
  public native static void
    setFocus (int windowPtr);
  ...
}
```

We have built a custom scanner that searches for dangerous type casts from integers to pointers. The scanner is implemented in the CIL framework [31] as a CIL feature. We found 38 native methods that accept Java integers as arguments and then cast the integers to pointers. Compared to the GTK, the JDK's protection of these integers is safer. First, the native methods are all declared as private methods. An attacker cannot invoke them arbitrarily. Second, the Java integers that represent C pointers are stored in private fields.

If we assume Java's access control rules on private fields and methods are strictly enforced, then the JDK's protection on the integers is sufficient. However, with the Java reflection API, a Java program can at runtime change the private fields that store the C pointers, or invoke private methods.

If an attacker can use the Java reflection API, then he can read and write arbitrary memory locations by exploiting the pattern of C pointers as Java integers. For example, the `getAdler` native method (shown below) in the `java.util.zip.Deflater` class accepts a Java long, casts it to a pointer to the `z_stream` struct, and returns the `adler` field in the struct. If an attacker invokes it with the number that equals a target memory address minus the offset of the `adler` field, then he can read the value at the target address.

```
jint Java_java_util_zip_Deflater_getAdler
        (..., jlong strm) {
    return ((z_stream *)strm)->adler;
}
```

In a similar vein, the attacker can write to any memory location with his data through the `setDictionary` method in the `Deflater` class; the `setDictionary` method updates a `z_stream` structure with user-supplied data.

Although the default security policy when running untrusted Java code does not allow the Java reflection, we believe that passing C pointers as Java integers is dangerous, for the following reason. For a program in pure Java, an attacker can violate the access-control policy of the Java program (e.g. reading private fields) using the Java reflection, but the program remains type safe, which implies no reading/writing arbitrary memory locations. However, with the Java reflection *and* passing C pointers as Java integers through the JNI, an attacker could violate type safety by reading/writing arbitrary memory locations (shown by previous examples). We believe the privilege escalation from using the Java reflection to reading/writing arbitrary memory locations is a violation of the Java security model.

Proposed fixes. We recommend a fix based on an indirection table of pointers, similar to the OS file-descriptor table. The C side uses the indirection table to store pointers and passes table IDs, not pointers, to Java. When C gets the table IDs back from Java, it checks the validity of the IDs before carrying out dangerous operations. If bogus IDs were passed to C, the validity-checking step would catch it.

3.3 Race conditions in file accesses

Time-of-check-to-time-of-use (TOCTTOU) bugs refer to race conditions in which *"a program checks for a particular characteristic of an object, and then takes some action that assumes the characteristic still holds when in fact it does not"* [3]. Bishop and Dilger [3] identified a category of TOCTTOU bugs in file accesses. Such flaws occur, for example, when a program checks the access privilege of a file through a file path name and then use the file through the same file path name. Between the check and the use, if an attacker can change the file associated with the file path name, then the program may be fooled to access privileged files that the attacker cannot access otherwise.

We used ITS4 and Flawfinder to scan for file-access race conditions in the JDK. We identified three places in the target directories where file-access race conditions might occur. An example in `solaris/native/java/io/UnixFileSystem_md.c` is the race window between `stat` (line 144) and `chmod` (line 236). If the file in question were in a directory writable by the attacker, then during the race window he can link to that file any target file. The `chmod` at line 236 will then change the protection mode of the target file.

Besides the three race conditions we identified, we also discovered that the implementation of all the native methods in the class `java.io.UnixFileSystem` is based on path names, instead of file descriptors. For example, the `checkAccess` method checks whether the file or directory denoted by a given path name may be accessed; the `setPermission` method set on or off the access permission of the file or directory denoted by a given path name. The class `java.io.File`, a client of `java.io.UnixFileSystem`, uses `checkAccess` in methods such as `canRead` to check access permissions of a file path name stored in a field of `java.io.File`. It also uses `setPermission` in methods such as `setReadable` to set access permissions of the file path name. As a result, a Java program that uses `java.io.File` may have race conditions, if it first invokes `canRead`, and then invokes `setReadable`.

File-access race conditions are most relevant in a multi-user system, which is not a typical environment of using Java. Nevertheless, Java has been and will be used in a diverse set of scenarios (e.g., Java programs are run as root in the Java Authorization Toolkit [1]). Fixing the TOCTTOU flaws is usually straightforward. For example, the race window created by `stat` followed by `chmod` can be fixed by first opening the file to get its file descriptor, and then using `fstat` and `fchmod` on the file descriptor.

3.4 Buffer overflows

By automatically inserting array bounds checks, Java provides built-in protection against buffer overflows. If a program is developed in pure Java, we are rest assured that no buffer will be overflowed. However, since the implementation of the JDK contains C/C++ code, it is possible for an attacker to pass Java applications unexpected values, which flow to the C code in the JDK and trigger a buffer overflow.

Buffer overflows occur when a C program does not perform sufficient bounds checking. Native methods that use the JNI often need to check integers from Java for negative values. Since Java supports only signed integer types, the JNI maps all Java integer types to signed integer types in C. To use these signed integers safely for indices, sizes, and loop counters that should never have negative values, explicit checks are necessary. Missing checks for negative values may crash the JVM, as past bug reports have shown [20, 38].

We employed ITS4, Splint, and Flawfinder to scan the C files under the target directories for buffer-overflow bugs. ITS4 and Flawfinder scan for and report dangerous operations such as `strcpy`, `memcpy`, and `fscanf`. Splint reports many type-incompatibility warnings. For example, it issues a warning when a signed integer is used as an unsigned integer, which is helpful to identify missing checks for negative values. With the help of the static analysis tools, we discovered seven places where there are insufficient bounds checks. Two of them are in C functions that are not used by Java, and do not pose a security threat to the JVM. The rest pose real threats to the JVM: one bug is due to a missing width specifier in the format-string argument of `fscanf`; three bugs are due to possible integer overflows that may subsequently lead to buffer overflows; one bug is due to insufficient bounds checking of a public native method.[1]

3.5 Bugs related to dynamic memory management

The C code in the JDK needs to manage two memory regions, the C memory region and the Java memory region. It may mismanage both memory regions.

Dynamic memory management in C. Unlike Java, the C language provides programmers the power of manually managing memory through functions such as `malloc` and `free`. This power, which seems indispensable in system programming, has always been a constant source of programming defects, and consequently security vulnerabilities. Due to manual memory management, the C code in the JDK may suffer from a range of flaws, including dereferencing dangling pointers, multiple frees, and memory leaks. These defects may make the JVM unstable and vulnerable.

We employed Splint to identify defects related to memory management in the target directories. We manually inspected a large number of warnings and found only one case of memory leaks.

Managing Java memory through the JNI. Through the JNI, native methods can manage the Java memory. Certain JNI APIs manage Java memory in a style similar to `malloc` and `free`. For instance, to access a Java integer array, a native method first invokes `GetIntArrayElements` to have a pointer to the integer array. When the method finishes with the array, it is supposed to invoke `ReleaseIntArrayElements` to release the pointer. These JNI API functions enable the C method to communicate with Java's Garbage Collector (GC). `GetIntArrayElements` informs the GC of the creation of a C pointer to the Java array; the GC should not garbage collect or move the array. `ReleaseIntArrayElements` informs the GC that the C pointer is no longer needed.

This style of manual memory management is error prone and has similar problems to the ones of `malloc`/`free`. For example, using the C pointer after `ReleaseIntArrayElements` is similar to using a dangling pointer, since the Java GC may have already moved or garbage collected the array. Failure to invoke `ReleaseIntArrayElements` will make the GC retain the array indefinitely. Other pairs of functions that are similar to `Get/ReleaseIntArrayElements` include `Get/ReleaseStringUTFChars`, `New/DeleteGlobalRef`, and `Push/PopLocalFrame`.

We developed grep-based scripts to pattern match places where relevant JNI API functions such as `ReleaseIntArrayElements` are used. In the target directories, we discovered one place where `ReleaseStringUTFChars` is not invoked (in one control-flow path) to release a Java String reference. There are also 27 places where JNI global references are not released.[2] Although these bugs are not security critical, they result in memory leaks and are worth fixing.

3.6 Insufficient error checking

One of the most common mistakes when writing C code is missing checks for error cases. Since the C language does not have an exception mechanism, programmers are required to perform explicit checks after many function calls that may return special values for reporting errors. For instance, the standard `malloc` function returns a null value if the required space cannot be allocated. The correct usage of the `malloc` function should first check the return value for nonnull before using it. We encountered two places where the C code in the target directories forgets the check for the `malloc` function.

In addition, many JNI API functions use null values to report errors. For example, the `GetFieldID` function returns null when the operation fails.[3] The following code crashes Sun's JVM, when `fid` gets null.

```
//Get the field ID
fid=(*env)->
      GetFieldID(env, cls, "x", "I");
//Get the int field
int i=(*env)->GetIntField(env, obj, fid);
```

The above code should first check `fid` to be nonnull, before invoking `GetIntField`.

We developed grep-based scripts to scan for JNI API functions whose return values should be checked. We inspected suspicious JNI API calls to check whether their return values are checked before used. In total, we found

JNI API FUNCTIONS	# OF VIOLATIONS
GetFieldID/GetStaticFieldID	5
GetMethodID/GetStaticMethodID	3
GetStringUTFChars	4
FindClass	11
New⟨Type⟩Array	1
NewGlobalRef	11
Total	35

Table 2: Insufficient error checking. For each JNI API, the table lists the number of cases in the target directories where there is no checking of the return value of the API before using the value.

35 violations. Table 2 summarizes the results in the target directories. Note that the table does not include those JNI API functions for which we did not find violations. We consider insufficient error checking to be security critical because they may result in JVM crashes.

3.7 Other flaws resulting from misusing the JNI

For completeness, we next mention other bug patterns in the native code of the JDK. For these patterns, we either have not found any bugs in the target directories, or have not successfully applied static analysis tools.

Type misuses. The JNI maps Java types to C/C++ types and performs necessary conversions when data go through the interface. Java primitive types and Java reference types are mapped differently. Java primitive types are mapped directly. For example, the Java type `int` is mapped to the native type `jint` (declared as 32-bit integers in jni.h). Java objects of reference types are mapped to *opaque references*, which are pointers to internal data structures in the JVM. The exact layout of the internal data structures is hidden from programmers. In C, all opaque references have the type `jobject`. Native C/C++ code manipulates these references through JNI API functions.

Since native C code treats all references to Java objects as having one single type[4], C compilers cannot distinguish references to objects of different Java classes. As a result, an object of Java class `A` may be wrongly passed to a JNI API function that actually requires an object of Java class `B`. Type checking in C compilers cannot catch this kind of mistakes, which usually results in JVM crashes. More serious is the case that a native method invokes a Java method with objects of wrong classes. A type confusion like this could have serious consequences, as past research on Java security has shown [33, 6].

Another case of type misuses in the JNI is that programmers may invoke wrong JNI API functions. For example, programmers may use wrong JNI array APIs, as the JNI provides different APIs for accessing arrays of different types. There are `GetByteArrayElements`, `GetIntArrayElements`, and others. Calling wrong JNI API functions may result in improper memory accesses or JVM crashes.

JSaffire [16] by Furr and Foster is a tool that can check type misuses in the JNI code. We did not incorporate JSaffire into our step of characterizing bug patterns for two reasons. First, this category of bugs has been well characterized in previous work [16, 35]. Second, we suspect type-misuse bugs in the JDK's native code would be rare. Type-misuse bugs usually result in immediate program crashes and are easy to trigger with a small amount of test code. As the JDK has been extensively "tested" by its users, we believe that most of the type-misuse bugs have been fixed. This is partly confirmed by our experiment. We constructed scripts to search for the most common cases of type-misuse bugs, such as passing wrong classes to JNI API functions and confusing jclass with jobject [25, ch 10.3]; we did not find any such kinds of bugs in the target directories.

Deadlocks. The JNI includes pairs of functions `Get/ReleaseStringCritical` and `Get/ReleasePrimitiveArrayCritical`, which introduce a critical region. Inside the region, the C code cannot issue blocking calls or allocate new Java objects. Otherwise, the JVM may deadlock. We inspected all such critical regions in the target directories and did not find any risk of deadlock.

Violating the Java security model. The JNI does not enforce access controls on classes, fields, and methods that are expressed in the Java language through the use of modifiers such as `private`. Therefore, a native method can read private fields of any Java object. Furthermore, a native method can violate the Java sandbox security model, by performing dangerous operations that would otherwise be blocked by the JVM. We have not checked the JDK's native code for these kinds of violations.

4 Remedies, limitations, and directions

The native code inside the JDK is critical to Java security. As we and others have demonstrated, after more than a decade, there are still flaws remaining in this critical code. Once identified, these flaws are generally not hard to fix. However, the perpetual mode of patching is less than satisfying. Next we discuss more systematic approaches, their limitations, and future directions.

4.1 Static analysis

Static analysis is useful for isolating and eliminating security bugs, as demonstrated by the number of bugs we identified with the help of static analysis tools. On the other hand, there are a few limitations of the current generation of static analysis tools that prevent us from using them to cover all 800k lines of native code in the JDK.

Limitations of static analysis tools. The tools we used issued a large number of warnings that are false positives. For each of the three off-the-shelf tools, the following table lists the number of warnings it issued, the number of true errors, and its false-positive rate.

Off-the-Shelf Tools	Warnings	Errors	FP rates
ITS4 -c1	241	6	97.5%
Flawfinder	297	5	98.3%
Splint[5]	3532	7	99.8%

Our own scripts and scanners perform slightly better, but the false-positive rates are still high; see Table 3.

Due to the large number of false positives, we had to manually sift through many cases—the principal reason why we examined only a portion of the native code in the JDK. In addition to false positives, static analysis tools may have false negatives. For example, of the four buffer-overflow bugs identified in the target directories, ITS4 and Flawfinder missed one and Splint missed two.

Another limitation of the static analysis tools is that they analyze C code alone, without considering how the Java side interacts with the C side. This is a severe limitation because the interface code between Java classes and C libraries is where most bugs arise. In fact, all the bugs we identified are in the interface code. This is not only because the two libraries in the target directories (namely, Zlib and fdlibm) have been used in many other applications besides the JDK and are mature, but because programmers tend to make wrong assumptions of the Java and C sides when writing interface code.

When analyzing the interface code, considering both sides of Java and C can significantly increase analysis precision and reduce false positives and negatives. To illustrate, we use the `java.util.zip.Deflater` class as an example. The public `deflate` method shown in Figure 4 accepts a buffer, an offset, and a length from users, and then invokes the native method `deflateBytes`. To be safe, the `deflate` method checks the bounds of the offset and the length parameters before invoking the native method `deflateBytes`.

For the example in Figure 4, a static analysis that analyzes only C code has to make either an optimistic or a pessimistic assumption about whether the Java side has performed the bounds checking. If the analysis makes the optimistic assumption, it would produce false negatives if the Java side had forgotten to check the bounds. If it makes the pessimistic assumption, it would have to flag any access to the `b` buffer through the offset and the length as a possible error. For example, the `SetByteArrayRegion` operation in `deflateBytes` would be flagged as a possible out-of-bounds array write, even though that is impossible given the Java context. Bug finders usually make pessimistic assumptions for the purpose of minimizing false negatives. For instance, Splint flags "`malloc(len)`" in `deflateBytes` and complains about an incompatible type cast from the signed integer `len` to an unsigned integer expected by `malloc`—it does not know that the Java side invokes `deflateBytes` only with positive lengths.

The necessity of inter-language analysis is also sharply enforced by our experience of manual inspection. For many warnings, we inspected both their C and Java contexts to decide if they are true errors. To give a rough idea of how many warnings cannot be eliminated as false positives without taking the Java context into account, we examined the 139 incompatible-type-cast warnings that Splint issued for the C code under `java.util.zip` and found that in 22 cases the Java context must be inspected.

Future directions of improving static analysis tools. Some of the limitations we mentioned are particular to the tools we used, and are not fundamental to static analysis. The off-the-shelf tools used in this study are known for having high false-positive rates. ITS4, Flawfinder, and our own tools are based on simple syntactic pattern matching; Splint performs certain type-based analyses, but is still a coarse-grained tool. We believe false-positives rates can be significantly reduced through advanced static techniques such as software model checking (e.g., MOPS [4], CMC [30], SLAM [2]), and Bandera [7]), type qualifiers [13, 18], theorem proving techniques (e.g., ESC/Java [12]), and others.

To better analyze interface code, we advocate *inter-language analysis* across Java and C. Most existing tools are limited *a priori* to code written in a single language. Few inter-language analyses across Java and C exist. JSaffire [16] is an exception, but can only check for type misuses of data from Java to C. Our previous work, ILEA [36], enables general inter-language analysis across Java and C. The basic approach of ILEA is to perform a partial compilation from C code to a specification based on Java so that an existing Java analysis can understand the behavior of the C code through the Java specification. ILEA extends Java with a set of simple, yet powerful approximation primitives, which enable auto-

BUG PATTERNS	OUR TOOLS	WARNINGS	ERRORS	FP RATES
Unexpected control flows due to mishandling exceptions	grep-based scripts	337	11	96.7%
C pointers as Java integers	scanner built in CIL	46	38	17.4%
Mem. management flaws (Java Mem.)	grep-based scripts	43	28	34.9%
Insufficient error checking (JNI APIs)	grep-based scripts	230	35	84.8%

Table 3: False-positive rates of our tools.

java.util.zip.Deflater

```
public class Deflater {
  public synchronized int deflate(byte[] b, int off, int len) {
    ...
     if (off < 0 || len < 0 || off > b.length - len) {
       throw new ArrayIndexOutOfBoundsException();
     }
    return deflateBytes(b, off, len);
  }

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

C implementation of deflateBytes()

```
jint Java_java_util_zip_Deflater_deflateBytes
       (JNIEnv *env, jobject this, jarray b, jint off, jint len) {
  ...
  out_buf = (jbyte *) malloc(len);
  ...
  (*env)->SetByteArrayRegion(env, b, off, len - strm->avail_out, out_buf);
  ...
}
```

Figure 4: An example illustrating the necessity of inter-language analysis

matic extraction of partial Java specifications of C code. Through ILEA, any existing analysis on Java in principle can be extended to also cover C code. In practice, however, ILEA is restricted by its compilation precision, and also by the effectiveness of the Java analysis.

We plan to combine advanced static analysis techniques with the ideas in ILEA to build high-precision, inter-language tools that hunt for bugs in the JDK's native code. We are particularly interested in taint analysis and software model checking. Static taint analysis (e.g., [27]) can track attacker-controllable data that flow from Java to C. Software model checking can check for violations of many patterns we have discussed as they can be formalized as state machines. We plan to investigate C model checkers such as MOPS [4] and CMC [30] and extend them to perform inter-language checking using the ideas in ILEA.

Finally, we believe it is important to formalize the soundness proofs of static analysis tools. Formal study helps understand the assumptions, clarify guarantees, and reduce false negatives. In the context of the JNI, formal study is complicated by the lack of formal semantics of the C language. It is perhaps helpful to focus instead on a well-defined subset of C such as Cminor [24].

4.2 Dynamic Mechanisms

Static analysis analyzes programs to find implementation errors before the programs are run. An alternative is to use dynamic mechanisms to prevent or isolate errors during runtime. Dynamic mechanisms can take advantage of richer runtime information to check certain properties easily, although sacrificing some performance.

Our previous work, SafeJNI [35], is a mostly dynamic mechanism for ensuring the safety of JNI-based programs such as the JDK. It first leverages CCured [32] to provide internal memory safety to the C code. CCured analyzes C programs to identify places where memory safety might be violated and then inserts runtime checks to ensure safety. SafeJNI also inserts runtime checks at the boundary between Java and C to make sure that the C code accesses the Java state safely and cooperates with Java's garbage collector. SafeJNI incurs a performance overhead of 14–119% on a set of microbenchmark programs, and incurs 63% on Zlib.

Table 4 summarizes how SafeJNI protects Java from bugs in the native code in terms of the various bug patterns discussed before. SafeJNI protects Java from most kinds of bugs in the native code. Its main limitation is that it does not protect against concurrency-related bugs (race conditions and deadlocks); we believe concurrency-related bugs should be best addressed through advanced static analysis techniques.

Future directions. We believe that SafeJNI is a promising direction to prevent errors in the native code. We plan to reduce its overhead in two ways. First, static analysis techniques can reduce a large number of dynamic checks. For example, many runtime type checking can be eliminated if we can statically track the classes of Java objects in C, similar to what JSaffire does [15].

Second, we plan to explore other more efficient ways of providing internal safety to C code than CCured. Our experiment showed that CCured accounted for most of the performance overhead in SafeJNI (46% out of 63% in Zlib). The relatively large performance slowdown is because CCured guarantees every C buffer is well protected. For instance, given the code below

```
int *p = (int *) malloc (1024);
*(p+i) = 3;
```

CCured in general will insert the runtime check "0 <= i < 1024" before "*(p+i) = 3".

If the safety policy is to protect the JVM state from being accidentally destroyed by C code, then Software Fault Isolation (SFI [40, 28]) of the C code is sufficient. Whenever the JVM starts to execute a native method, it can first allocate a trunk of memory, say 16MB, and hand the memory region to the native method. A SFI-based scheme can then guarantee that any access of the C memory will not escape the memory region, and thus will not destroy the JVM state.

Schemes based on SFI can isolate errors within native components, but does not prevent exploits of vulnerabilities inside the components. XFI [10], on the other hand, can prevent exploits of a large number of vulnerabilities by enforcing properties such as control-flow integrity. In addition, it works on assembly code and is not restricted to a source programming language.

4.3 Reimplementation in safer languages

It can be argued that the C language is intrinsically unsafe and should not be used in the JDK. In the long run, we believe the C code in the JDK should be reimplemented in safer languages. The obvious choice is Java. This is a feasible approach, as there exist implementations in pure Java of many programs originally written in C, such as the Zlib library [21]. GNU Classpath, an open-source replacement of Sun's JDK, takes this approach seriously; one of their long-term goals is to become JNI independent by implementing everything in Java [5]. On the flip side, rewriting the existing 800 kloc of C/C++ code in Java will require a substantial investment, and will likely have a negative impact on execution speed.

Another idea is to use a safe C variant to port the C code. Cyclone [22] is a reasonable choice. Since the syntax and semantics of Cyclone are close to C, porting C code to Cyclone should take less time than, say, a complete rewrite in Java. However, as Cyclone has a strong type system and uses region-based memory management, converting to type-checkable Cyclone code will not be a trivial effort. Furthermore, this approach alone can guarantee only the internal safety of C code. The C code can still misuse the JNI interface.

Since the JNI interface is extraordinarily verbose and error prone, one approach to reducing flaws is to use a better interface between Java and C. A notable example is Jeannie [19], which allows programmers to write mixed Java and C code in a single file. The Jeannie compiler then translates mixed Java/C code into code that uses the JNI. Although in Jeannie it is still possible to write unsafe code, Jeannie helps programmers reduce errors. For example, in Jeannie programmers can raise Java exceptions directly, thus avoiding the control-flow problem when raising JNI exceptions (Section 3.1).

5 Conclusion

The large amount of native code in the JDK is a time bomb in Java security. Our study has examined a range of bug patterns in the JDK's native code, from well-known buffer overflows to new patterns such as unexpected control flow paths due to mishandling JNI exceptions. Given the importance of Java, it is imperative to develop better, inter-language static and dynamic mechanisms to mediate the threats posed by the native code.

Through our study, we hope to send the message that the native code should be kept at a minimum in the JDK. On the contrary, the native code in Sun's JDK has been on the increase. The native code is outside of the Java security model and defeats Java's main goals: safety, security, and platform independence. In the long run, most

Bug patterns		How SafeJNI works against the bugs?
Unexpected control flows due to mishandling exceptions		Through SafeJNI's dynamic checks on pending exceptions.
Race conditions in file accesses		N/A
Buffer overflows		Through CCured and SafeJNI's static pointer kind system.
Mem. management flaws	C mem.	Through CCured.
	Java mem.	Through SafeJNI's memory management scheme.
Insufficient error checking	JNI APIs	Through SafeJNI's dynamic checks.
	misc.	Through CCured.
Type misuses		Through SafeJNI's dynamic checks.
Deadlocks		N/A
Violating the Java security model		Partly addressed through SafeJNI's dynamic checks on access-control rules on Java fields/methods.

Table 4: How SafeJNI protects the JVM from bugs?

of the native code should be ported to safer languages such as Java.

Acknowledgments

We would like to thank Andrew Appel and Edward Felten for their comments; and to anonymous reviewers for their constructive feedbacks on earlier versions of this paper; and to Xiaolan Zhang and Angelos Stavrou for shepherding the paper to its final form.

References

[1] Overview of authorization toolkit for Java. Retrieved Apr 26th, 2008, from `http://www.amug.org/~glguerin/sw/authkit/overview.html`.

[2] Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 203–213.

[3] Bishop, M., and Dilger, M. Checking for race conditions in file accesses. *Computing Systems 9*, 2 (1996), 131–152.

[4] Chen, H., and Wagner, D. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)* (2002), pp. 235–244.

[5] Classpath decisions. Retrieved Apr 26th, 2008, from `http://developer.classpath.org/mediation/ClasspathDecisionsPage`.

[6] Coglio, A., and Goldberg, A. Type safety in the JVM: some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience 13*, 13 (2001), 1153–1171.

[7] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)* (2000), pp. 439–448.

[8] CVE-2007-1657. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1657`.

[9] Dean, D., Felten, E. W., and Wallach, D. S. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy (S&P)* (1996), pp. 190–200.

[10] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *OSDI* (2006), pp. 75–88.

[11] Evans, D., and Larochelle, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (2002), 42–51.

[12] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 234–245.

[13] Foster, J. S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2002), ACM Press, pp. 1–12.

[14] Freund, S. N., and Mitchell, J. C. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning 30*, 3-4 (2003), 271–321.

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

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

[17] GNU GLOBAL source code tag system. `http://www.gnu.org/software/global/`.

[18] Greenfieldboyce, D., and Foster, J. S. Type qualifier inference for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007). 321-336.

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

[20] Java bug report 4153825. `http://bugs.sun.com/view_bug.do?bug_id=4153825`.

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

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

[23] Klein, G., and Nipkow, T. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. on Programming Languages and Systems 28*, 4 (2006), 619–695.

[24] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages (POPL)* (2006), pp. 42–54.

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

[26] Liang, S., and Bracha, G. Dynamics class loading in the Java virtual machine. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 1998), ACM, pp. 36–44.

[27] Livshits, V. B., and Lam, M. S. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium* (Aug. 2005), pp. 271–286.

[28] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (2006).

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

[30] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A pragmatic approach to model checking real code. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[31] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)* (2002), pp. 213–228.

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

[33] Saraswat, V. Java is not type safe, 1997.

[34] Schoenefeld, M. Denial-of-service holes in JDK 1.3.1 and 1.4.1_01. Retrieved Apr 26th, 2008, from `http://www.illegalaccess.org/java/ZipBugs.php`, 2003.

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

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

[37] US-CERT. Vulnerability note VU#138545: Java Runtime Environment image parsing code buffer overflow vulnerability, June 2007. Credit goes to Chris Evans.

[38] US-CERT. Vulnerability note VU#939609: Sun Java JRE vulnerable to arbitrary code execution via an unspecified error, Jan. 2007. Credit goes to Chris Evans.

[39] Viega, J., Bloch, J. T., Kohno, Y., and McGraw, G. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference (ACSAC)* (2000).

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

[41] Wallach, D. S., and Felten, E. W. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy (S&P)* (1998), pp. 52–63.

[42] Wheeler, D. A. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

Notes

[1]This bug is not in the target directory and was found in a casual inspection.

[2]Global references are never released in the code we examined, although the JNI manual explicitly mentioned the necessity of freeing global references [25, ch5.2.3].

[3]It fails if the specified field cannot be found, or if the class initializer fails, or if the system runs out of memory [25].

[4]In C++, certain Java built-in classes have corresponding C++ classes in the JNI (predefined in jni.h). References to objects of other Java classes, including all user-defined classes, are still mapped to `jobject`.

[5]With the options "+posixlib -paramuse -redef -noeffect -varuse -exportlocal -incondefs -booltype jboolean -booltrue JNI_TRUE -boolfalse JNI_FALSE -predboolint -compdef".