Science of Computer Programming 89 (2014) 273–297

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Exception analysis in the Java Native Interface

Siliang Li *, Gang Tan

P.C. Rossin College of Engineering & Applied Science, Computer Science and Engineering, Packard Laboratory, 19 Memorial Drive West, Lehigh University, Bethlehem, PA 18015, United States

HIGHLIGHTS

- Exception analysis and bug finding in the Java Native Interface (JNI).
- Extends Java's exception checking rules on native methods.
- A novel static analysis framework.
- Bug finding with high accuracy and low performance overhead.
- A user-friendly Eclipse plug-in tool to check JNI code.

ARTICLE INFO

Article history:
Received 3 May 2012
Received in revised form 30 January 2014
Accepted 30 January 2014
Available online 19 February 2014

Keywords:
Foreign Function Interface
Java Native Interface
Exception checking
Static analysis

ABSTRACT

A Foreign Function Interface (FFI) allows one host programming language to interoperate with another foreign language. It enables efficient software development by permitting developers to assemble components in different languages. One typical FFI is the Java Native Interface (JNI), through which Java programs can invoke native-code components developed in C, C++, or assembly code. Although FFIs bring convenience to software development, interface code developed in FFIs is often error prone because of the lack of safety and security enforcement. This paper introduces a static-analysis framework, TurboJet, which finds exception-related bugs in JNI applications. It finds bugs of inconsistent exception declarations and bugs of mishandling JNI exceptions. TurboJet is carefully engineered to achieve both high efficiency and accuracy. We have applied TurboJet on a set of benchmark programs and identified many errors. We have also implemented a practical Eclipse plug-in based on TurboJet that can be used by JNI programmers to find errors in their code.

1. Introduction

Today's software development rarely uses just one language or framework but often uses a mix of several languages: programmers design and implement some functionality in one language, and complete the building of the software by snapping together legacy systems or library code written in other languages. This is considered a more efficient practice because it allows programmers to focus on developing new features without "re-inventing the wheel". More importantly, the practice makes it possible to deliver a software product quickly to the competitive market place.

A Foreign Function Interface (FFI) is a mechanism that permits software written in one *host* programming language to interoperate with another *foreign* language in the form of invoking functions across language boundaries. In this paper, we focus on the Java Native Interface (JNI), which allows Java programs to interface with low-level C/C++/assembly code (i.e.,

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.01.018
0167-6423/

Java code

```
class ZipFile {
  // Declare a native method;
  // no exception declared
  private static native long open
    (String name, int mode,
    long lastModified);

  public ZipFile (...) {
    // Calling the method
    // may crash the program
    ...; open(...); ...
  }

  static {System.loadLibrary("ZipFile");}
}
```

C code

```
void Java_ZipFile_open (JNIEnv *env, \ldots) {
   ...
   // An exception is thrown
   ThrowIOException(env);
   ...
}
```

Fig. 1. A simple JNI example. This example also demonstrates how a native method can violate its exception declaration. `ThrowIOException` is a utility function that throws a Java `IOException`.

native code). A native method is declared in a Java class by adding the `native` modifier. For example, the `ZipFile` class in Fig. 1 declares a native method named `open`. The actual implementation of the `open` method is implemented in C. Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. We define a *JNI application* as a set of Java class files together with the native code that implements the native methods declared in the class files.

While the use of FFIs can bring convenience and efficiency in software development, it also has its downsides. Beyond the obvious differences in code syntax and semantics, different programming languages can have different type systems, exception-handling mechanisms, memory-management schemes, multithreading programming models, and so on. Because of these differences, programming with FFIs requires extreme care, making it an error-prone process. Misuse of FFIs can introduce issues that are difficult to debug, and make the software less safe and less reliable.

Specifically in this paper, we study the differences of exception-handling mechanisms between Java and native code. Java has two features related to exceptions that help improve program reliability.

- *Compile-time exception checking.* A Java compiler enforces that a checked exception must be declared in a method's (or constructor's) `throws` clause if it is thrown and not caught by the method (or constructor). While the usefulness of checked exceptions for large programs is not universally agreed upon by language designers, proper use of checked exceptions improve program robustness by enabling the compiler to identify unhandled exceptional situations during compile time.
- *Runtime exception handling.* When an exception is pending in Java code, the Java Virtual Machine (JVM) automatically transfers the control to the nearest enclosing try-catch block that matches the exception type.

Native methods can also throw, handle, and clear Java exceptions through a set of interface functions provided by the JNI. Consequently, an exception may be pending when the control is in a native method. For the rest of this paper, we will use the term *JNI exceptions* for those exceptions that are pending on the native side, while using the term *Java exceptions* for those pending on the Java side. JNI exceptions are treated differently from Java exceptions.

- A Java compiler does not perform compile-time exception checking on native methods, in contrast to how exception checking is performed on Java methods.
- The JVM does not provide runtime exception handling for JNI exceptions. An exception pending on the native side does not immediately disrupt the native-code execution, and only after the native code finishes execution will the JVM mechanism for exceptions start to take over.

Because of these differences, it is easy for JNI programmers to make mistakes. First, since there is a lack of compile-time exception checking on native methods, the exceptions declared in a native-method type signature might differ from those exceptions that can actually happen during runtime. Second, since there is no support for runtime exception handling in