Detecting Memory Bugs in JNI Programs

JAEMIN HONG, KAIST

1 BACKGROUND

The Java Native Interface (JNI) allows Java programs to call functions written in C or C++. Functions written in C or C++ are usually called native methods in JNI programming. The following is a code snippet from an example JNI program (some unimportant parts are omitted):

```java
class Hello {
    native void hello();
    public static void main(String[] args) { (new Hello()).hello(); }
}
```

Hello.java

```c
JNIEXPORT void JNICALL Java_Hello_hello(JNIEnv *env) {
    printf("Hello world!\n");
}
```

Hello.c

The `hello` method of the `Hello` class is defined as a native method. When the method is invoked, the Java Virtual Machine (JVM) calls the `Java_Hello_hello` function, which is defined in the `Hello.c` file. Therefore, the program prints `"Hello world\n"` to the standard output.

Programmers use the JNI to write compute-intensive parts of their programs or to reuse code written in C or C++ in Java programs without implementing the whole logic again. Lots of programs using the JNI exist in the real world. For example, the `java.util.zip` package of the OpenJDK [2020] uses the JNI for high performance.

One typical pattern to use the JNI is the peer pattern, which was introduced by Liang [1999]. In this pattern, a pointer to a dynamically allocated memory block should be stored in a Java object. For instance, consider the following code (some unimportant parts are omitted):

```java
class Peer {
    native long malloc(int size);
    native void free(long ptr);
    long ptr;
    Peer(int size) { ptr = malloc(size); }
    void destroy() { free(ptr); }
}
```

Peer.java

```c
JNIEXPORT jlong JNICALL Java_Peer_malloc(JNIEnv *env, jint size) {
    return (jlong) malloc(size);
}
JNIEXPORT void JNICALL Java_Peer_free(JNIEnv *env, jlong ptr) {
    free((void *) ptr);
}
```

Peer.c

Since pointers are not Java values, they are represented as 64-bit integers in Java objects. Note that `jlong` and `jint` are types respectively denoting `long` and `int` of Java in native code. The pattern is useful to make Java wrapper classes of native structs and classes.

2 MOTIVATION

The peer pattern can be the source of a memory bug in practice. If a pointer is used after being freed, then the JVM crashes immediately due to access to a dangling pointer. On the other hand, if a Java object carrying a pointer is garbage-collected before the pointer is freed, then the memory allocated by the native side will never be reclaimed. It is a memory leak.

Although these kinds of memory bugs are common in C or C++ programming, I believe that they are more harmful in JNI programming than C or C++ programming. The reasons follow:

- Java programmers are unfamiliar with memory bugs since the JVM manages memory in Java programs. Besides, they used to debug their programs with logs provided by the JVM, but the JVM shows less useful messages when it crashes while executing native code.
- In many cases, programmers use existing native libraries for their programs. They implement only a small portion of the native code, which is required for interoperation. Thus, when they find memory bugs, they need to inspect the native code that they did not write.
- Since memory allocation and deallocation happen in the native side, programmers cannot easily determine whether invoking a certain Java method will make memory blocks be allocated or deallocated.

Programmers have suffered from memory bugs while using the JNI. For instance, libGDX [2020] is a Java game development framework using the JNI. Although the framework itself has been tested a lot, game developers have trouble with using the framework correctly. Qureshi [2017] claimed that he had experienced crashes in the issue tracker of the libGDX GitHub repository. However, the reason of the crashes was found in his code: `dispose` was called twice, and it caused double-free, which made the JVM crash. Several similar issues are in the issue tracker.

In this project, I will propose a novel static analyzer to detect memory bugs that can be found in the peer pattern. There has been multiple approaches to detect memory bugs in JNI programs [Hong et al. 2017; Kondoh and Onodera 2008; Lee et al. 2010; Tan et al. 2006]. However, most of them aimed different sorts of memory bugs in the JNI or used dynamic analyses. To the best of my knowledge, there is no static analyzer detecting memory bugs due to pointers stored in Java objects.

3 MY APPROACH

The most challenging part of the project is that the analyzer needs to deal with programs written in more than one language. It must analyze code written in Java and C (or C++). The analyzer will analyze native code first and then Java code. More precisely, the first part of the analyzer will extract the summaries of functions in native code. The summaries need to contain information about functions that allocate or deallocate memory. The second part of the analyzer will analyze Java code with the summaries. Due to the summaries, the analyzer can determine whether a particular Java method will allocate or deallocate memory. In this way, the analyzer will be able to detect memory bugs, including dangling pointers and memory leaks, in JNI programs.

4 EXPECTED RESULTS

I expect that the analyzer will be able to detect memory bugs in small JNI programs. However, to reduce false alarms, the anazlyer needs to use proper sensitivites. At the same time, to find memory bugs in large-scale JNI programs, which are typical cases in the real world, the analyzer needs to use proper advanced techniques, for example sparse analyses.

REFERENCES

Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongseok Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology* 82 (2017), 80–95.

Goh Kondoh and Tamiya Onodera. 2008. Finding bugs in Java native interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 109–118.

Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S McKinley. 2010. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 36–49.

Sheng Liang. 1999. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional.

libGDX. 2020. libgdx. https://github.com/libgdx/libgdx.

OpenJDK. 2020. jdk. https://github.com/openjdk/jdk/tree/fc842d2b4bb054f543990c715fcbebc3ccf052de/src/java.base/share/classes/java/util/zip.

Zaid Rehman Qureshi. 2017. fatal error when switching the screen with Box2d bodies. https://github.com/libgdx/libgdx/issues/4770. Accessed: 2020-05-20.

Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Vol. 97. Citeseer, 106.