JNI Program Analysis with Automatically Extracted C Semantic Summary

Sungho Lee
KAIST
South Korea
eshaj@kaist.ac.kr

ABSTRACT

From Oracle JVM to Android Runtime, most Java runtime environments officially support Java Native Interface (JNI) for interaction between Java and C. Using JNI, developers can improve Java program performance or reuse existing libraries implemented in C. At the same time, differences between the languages can lead to various kinds of unexpected bugs when developers do not understand the differences or comprehensive interoperation semantics completely. Furthermore, existing program analysis techniques do not cover the interoperation, which can reduce the quality of JNI programs.

We propose a JNI program analysis technique that analyzes Java and C code of JNI programs using analyzers targeting each language respectively. The C analyzer generates a semantic summary for each C function callable from Java and the Java analyzer constructs call graphs using the semantic summaries and Java code. In addition to the call graph construction, we extend the analysis technique to detect four bug types that can occur in the interoperation between the languages. We believe that our approach would be able to detect genuine bugs as well as improve the quality of JNI programs.

CCS CONCEPTS

• **Software and its engineering → Operational analysis**;

KEYWORDS

Java Native Interface; multilingual program analysis; JNI bugs

ACM Reference Format:
Sungho Lee. 2019. JNI Program Analysis with Automatically Extracted C Semantic Summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338990

1 INTRODUCTION

Java Native Interface (JNI) is one of the features of Java runtime environments, which enables developers to implement programs in both Java and C code. Developers can implement computationally expensive logic in C to improve performance and use JNI for the interoperation between other parts implemented in Java and the C code. Besides the performance reason, because plenty of C libraries are reusable in Java programs, JNI is officially supported by most Java runtime environments, such as Oracle JVM and Android Runtime [4, 9].

However, since Java and C have different semantics each other, the interoperation between the languages can lead to unexpected behaviors. Java adopts a garbage collection technique for memory management, for example, but developers have to explicitly allocate and deallocate memory in C. Also, Java uses the `try-catch` statement to handle exceptions, but C does not have an exception handling mechanism. Because even a compiler does not give warnings or errors for the JNI interoperation, those differences are the main difficulty in JNI program development as well as make JNI programming error-prone.

Researchers suggested various static or dynamic analysis techniques for JNI programs [1, 6–8, 13]. However, most of them focused on security vulnerability, which is not suitable for bug detection. Some of them suggested analysis techniques finding bugs in JNI programs, but they analyzed only C code or targeted to detecting traditional C bugs not related to the JNI interoperation.

In this paper, we propose a static analysis technique using both Java and C analyzers for JNI program analysis. At first, a C analyzer analyzes C code of JNI programs and extracts semantic summaries for each function accessible from Java. Using the semantic summaries and Java code, a Java analyzer constructs call graphs for the JNI programs, which represents program control flows not only in Java but also in native functions implemented in C. Then, as a mitigation for the possible bugs, we define four bug types and extend the analysis technique to detect the bugs.

Our expected contributions are followed:

- Suggesting a new static analysis technique for JNI program analysis.
- Defining four bug types that can occur in the JNI interoperation between Java and C.
- Developing a JNI static analysis tool and detecting bugs in real-world JNI programs.

2 BACKGROUND AND RELATED WORK

2.1 Background

Figure 1 represents a high-level overview of the interoperation between Java and C. JNI supports bidirectional communication, which Java and C can invoke the other side functions and methods. JNI maps C functions to Java native methods declared with the `native` keyword using a specialized naming convention. For example, a native method `foo` of `org.issta.Example` class is mapped to a

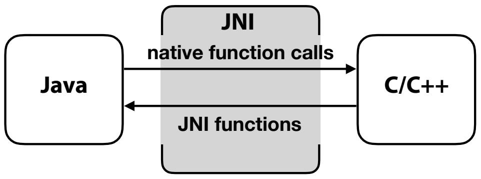

Figure 1: Overview of Java and C interoperation

`Java_org_issta_Example_foo` C function. Java invokes C functions by calling the corresponding native methods. C invokes Java methods and manipulates a Java environment using a reflection like way via JNI functions predefined `jni.h`.

```
package com.example;

class App{
  static{ System.loadLibrary("lib"); }
  void exec(){ callJava(this); }
  void foo(){ /* do something... */ }
  void bar(){ /* do something... */ }
  native void callJava(App app);
}
```

(a) Java code

```
void Java_com_example_App_callJava(JNIEnv* env, jobject
    /*this*/, jobject app){
  jclass klass = env->GetObjectClass(env, app);
  jmethodID mid = env->GetMethodID(env, klass, "foo", "()V");
  env->CallVoidMethod(env, app, mid);
}
```

(b) C code in `lib.c`

Figure 2: Sample JNI communication in JNI programs

Figure 2 shows a concrete example, where (a) is Java code and (b) is C code in the `lib.c` file. Class App has four methods, `exec`, `foo`, `bar`, and `callJava`. The `callJava` method is a native method corresponding to `Java_com_example_App_callJava` in the `lib.c` while the others are Java methods. When invoking the `exec` method, it calls the native method, `callJava`, and the Java runtime environment passes the program control to the entry of the corresponding native function, `Java_com_example_App_callJava`. The native function calls method `foo` of class App via a JNI function call sequence; `GetObjectClass` function is for getting class information of Java object app, `GetMethodID` function is for getting an identifier of method `foo` of class App, and `CallVoidMethod` function is for invoking method `foo`, propagating Java object app as a `this` argument.

Existing Java program analyzers are not able to check the reachability of the `foo` and `bar` methods correctly. Because they focus on only Java code and it is not possible to pre-define analysis models for native functions developers defined, native function calls are just ignored unsoundly, or excessively over-approximated. In the unsound approach, the analyzers handle native method calls as no-operations, so they tag both `foo` and `bar` methods as unreachable. On the other hand, since the sound over-approximation supposes that native functions would invoke all Java methods, both `foo` and `bar` methods are tagged as reachable.

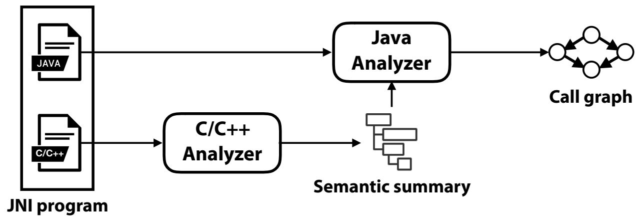

Figure 3: An overview of JNI program analysis

2.2 Related Work

ILEA project analyzed null-related bugs in JNI programs by compiling C code to JVML [12]. Since expressiveness of C is different from Java, they extended JVML to include the most features of C. In the opposite direction, DroidNative compiled Java to native code and detected malicious patterns using a signature-based detection technique [1].

Various Java static analysis research handles native methods dynamically [5, 10, 11]. They analyze Java code statically and abstract a C function semantics as its runtime behavior at the callsites, which analyzes JNI programs with partial native function semantics. However, those analysis techniques do not include native function semantics soundly as well as miss any control and data flow from C to Java.

The most related approach is JN-SAF that analyzes JNI programs using both Java and binary code analyzers based-on symbolic execution [13]. Since its target is Java and binary code, the approach is widely adaptable for all the JNI programs in which C disappears after compilation. However, because symbolic execution suffers from the path explosion problem, they evaluated JN-SAF for only small size programs. Also, it focused on data flow analysis only to detect security vulnerabilities, which is not generally useful to detect diverse bugs that possibly exist in JNI programs.

3 RESEARCH

We propose a general approach to analyze JNI programs using Java and C analyzers. Figure 3 shows an overview of our approach. At first, a C analyzer performs a modular static analysis to extract semantic summaries of C code from JNI programs. Using the semantic summaries instead of C code, a Java analyzer performs a whole program analysis to construct call graphs of JNI programs including all the control flows not only in Java but also in C and their interoperation. We describe each step as follows.

3.1 Phase 1: C Semantic Summary Extraction

A semantic summary is a collection of JNI functions called in C functions and values of the arguments. Because C code invokes Java methods and manipulates the Java environment only via JNI functions, the JNI function calls and the arguments are all the information used to affect control and data flow from C to Java.

Figure 4 shows an example semantic summary of the C function `Java_com_example_App_callJava` in Figure 2, where the left-hand side and the right-hand side show JNI function callsites and heap states at the callsites, respectively. `@i` denotes the *i*-th index of a JNI function callsite, v_i is a variable used as the *i*-th argument

`@1 GetObjectClass(`v_1`, `v_2`)`	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
`@2 GetMethodID(`v_1`, `v_2`, `v_3`, `v_4`)`	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	$ret_{@1}$
	v_3	$\mapsto$	`"foo"`
	v_4	$\mapsto$	`"()V"`
`@3 CallVoidMethod(`v_1`, `v_2`, `v_3`)`	v_1	$\mapsto$	arg_1
	v_2	$\mapsto$	arg_3
	v_3	$\mapsto$	$ret_{@2}$

Figure 4: Example semantic summary of C function `Java_com_example_App_callJava` **in Figure 2**

of a JNI function call, arg_i is the i-th argument of the native function `Java_com_example_App_callJava`, and $ret_{@i}$ is a return value at the `@i` JNI function callsite. The `CallVoidMethod` is one of JNI functions calling a Java method from a native function. It receives three arguments; the first is the Java environment, the second is the target Java method id, and the third is an argument of the target Java method. From the summary, we can extract Java method call information, `CallVoidMethod(arg`$_1$`, arg`$_3$`, GetMethodID(arg`$_1$`, GetObjectClass(arg`$_1$`, arg`$_3$`), "foo", "()V"))`, which means that the native function calls the `foo()V` Java method of `App`, propagating `app` as a `this` argument. Thus, a Java analyzer adds an indirect call edge from `exec()V` to `foo()V` at the native function callsite and resumes analysis from the entry of `foo()V`.

To extract semantic summaries for C functions, we extend a modular analysis technique Dillig et al. suggested and implement the C analyzer on top of Infer that is a modular analysis framework developed by Facebook [2, 3]. Because of the lack of analysis entry and callsite information, a whole program analysis approach requires a special model to analyze C code from an analysis entry. Instead, we use a modular approach representing analysis results as relations between inputs and outputs for each function accessible from Java, without any special models.

3.2 Phase 2: Call Graph Construction for JNI Programs using Semantic Summaries

A Java analyzer gets two inputs, Java code and the semantic summaries, to construct call graphs of JNI programs. The Java analyzer analyzes Java code as the same as existing whole program analyzers. However, at native function callsites, it handles native function calls using the semantic summary instead of ignoring or extensively over-approximating them.

To use semantic summaries in general, we transform semantic summaries of C functions to Java source code preserving JNI function call semantics. For example, the semantic summary in Figure 4 is compiled as follows:

```
void native callJava(Object app){ app.foo(); }
```

which folds the sequence of three JNI function calls to a Java method call according to JNI function semantics. By the transformation from semantic summaries to Java source code, semantic summaries are completely separated from a Java analyzer and used by all Java analyzers to improve the precision of the analysis.

3.3 Phase 3: Bug Detection

We briefly describe four bug types that possibly lead to unexpected behaviors in the interoperation between languages as follows.

a) **Missing Exception Handling** An exception thrown in C must be handled explicitly via JNI function calls such as `ExceptionOccured`, `ExceptionCheck`, and `ExceptionClear` in C. Because of the lack of exception handling mechanism, an exception thrown in C is just pending and the execution continues until the program control is passed to Java. When a pending exception exists, following JNI function calls may lead to unexpected results and the pending exception is finally propagated to Java.

b) **Descriptor Mismatching** C can fail to invoke Java methods or manipulate Java environment, because of mismatching between descriptors C uses with declarations in Java. To access Java environment, C uses a reflection like mechanism based-on string values for class, field and method descriptors. When a descriptor is mismatched to the declaration, Java runtime fails to find a correct target class, field, or method and an exception is thrown in C.

c) **Nondeterministic Native Function Call** A Java native method can be mapped to multiple C functions and the callee is randomly determined when calling the Java native method. While functions cannot be the same name in the same scope in C, it is possible to have relations between multiple C projects to a Java class, which makes one-to-n mapping between a Java native method and C functions having the same name. Besides of the case, JNI supports a special naming convention that attaches the types of parameters to the end of the function name for method overloading of Java native methods, which makes one-to-n mapping as well. For example, a Java native method `foo` with an integer parameter can be mapped to both C functions `foo` and `foo_I`. When one-to-n mapping is made, Java runtime nondeterministically chooses one of them as the callee of the native function call.

d) **Use of Garbagable Java Objects** Java objects in C environment are possibly freed by the garbage collector, which causes use-after-free bugs in C. Because Java runtime does not monitor C environment, the garbage collector counts only the references from other Java objects. When it finds a unreferenced object, it frees the object regardless of whether C variables are pointing to the object or not. To prevent an object from being a garbage, developers must register the object as a global reference via `NewGlobalRef` JNI function, and unregister it via `ReleaseGlobalRef` at the end of the use to avoid memory leakage.

To detect those JNI interoperation bugs, we design and implement checkers in the Java analyzer. The two types of bugs, b) and c), are syntactically detectable by comparing Java class, field and method declarations to semantic summaries. For a), we use an exception analysis suggested by Li and Tan [8]. Since the approach handles only the control flows between JNI function calls in C, we extend it to the Java exception control flows to trace unhandled exceptions propagated from Java to C. For d), we use the flow-sensitive pointer analysis to check reference relations for an object at the use-sites in C functions.

3.4 Application for Unused Code Removal in JNI Programs

If both Java and C/C++ analyzers are sound, we can utilize our approach of a JNI program not only to detect bugs but also to remove unused code from it. We currently expect that two optimizations are possible.

Unreachable Methods Removal. Unreachable methods are safely removable from a JNI program using call graphs. All reachable methods are included in a call graph. In other words, methods that are defined but not included in a call graph are unreachable from program entries. Thus, we can remove unreachable methods safely from JNI programs to reduce code size and complexity. For example, in Figure 2, the `foo` method is reachable but the bar method is unreachable. In our approach, since a call graph includes only the `foo` method, we can remove the `bar` method from the class `App`.

Opaque Predicates Removal. A JNI program call graph is useful to remove opaque predicates related to exception checking in C functions. As briefly described above, when a C function calls a Java method and an exception occurs in the Java method, the exception is pending and propagated to the native function. Because of the lack of an exception handling mechanism in C, developers must check the existence of an exception explicitly every time after calling a Java method. However, if a Java method does not have any logic to produce an exception, exception checking after calling it is redundant. For example, we assume that the below code is after `env->CallVoidMethod(...)` in Figure 2 (b):

```
if(env->ExceptionOccurred()){ ...
}
```

The `ExceptionOccured` JNI function returns `true` if there is a pending Java exception, or `false` otherwise. If the `foo` method does not introduce any exceptions, `env->ExceptionOccured()` is an opaque predicate that always returns `false`. Thus, we can remove the predicate from the `Java_com_example_App_callJava` function.

4 EVALUATION PLAN

To show the usefulness of our approach, we plan to analyze both JNI benchmarks suggested by JN-SAF and real-world Android apps using JNI interoperations [13]. Since the benchmarks are for private data leakages, we will show the private data leakage detection results using built-in data flow analysis of Java static analyzers. For the real-world Android apps, we will show the usefulness in two aspects; One is the effectiveness of call graph construction and the other one is bug detection results. The number of resolving native call sites and reachable methods via C functions are suitable to show how much our approach analyzes more than naive Java analyzers in JNI programs. The goal of bug detection is to detect JNI interoperation bugs in real-world JNI programs. We plan to analyze the Android apps using our checkers, confirm true/false positives, and report true positives to the developers.

5 CONCLUDING REMARKS

This paper proposes a new static analysis approach for JNI programs using both Java and C analyzers. The C analyzer extracts semantic summaries of C functions, and the summaries are transformed to Java source code to being an input of the Java analyzer. Using the semantic summaries instead of C functions, the Java analyzer constructs call graphs including not only directly reachable methods from program entries, but also indirectly reachable methods via C functions. In addition to the call graph construction, we define four types of bugs that possibly exist in JNI programs and plan to extend the analyzer to detect them in real-world applications. We believe that our approach would be a front-end of Java analyzers to handle native method calls and helpful to developers to make high-quality JNI programs.

ACKNOWLEDGMENT

This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

[1] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *computers & security* 65 (2017), 230–246.
[2] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 567–577.
[3] Facebook. 2019. Infer. https://fbinfer.com.
[4] Google. 2016. Android NDK. https://developer.android.com/ndk?hl=en.
[5] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 2 (2007), 11.
[6] 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*. ACM, 109–118.
[7] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S McKinley. 2010. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. *ACM Sigplan Notices* 45, 6 (2010), 36–49.
[8] Siliang Li and Gang Tan. 2009. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 442–452.
[9] Oracle. 2018. Java Native Interface Specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.
[10] Nastaran Shafiei and Franck van Breugel. 2014. Automatic handling of native methods in Java PathFinder. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. ACM, 97–100.
[11] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. 2015. Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security* 14, 2 (2015), 141–153.
[12] Gang Tan and Greg Morrisett. 2007. ILEA: Inter-language analysis across Java and C. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 39–56.
[13] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1137–1150.