IPSJ Transactions on Programming Vol.5 No.3 23–28 (Aug. 2012)

Regular Paper

SEAN: Support Tool for Detecting Rule Violations in JNI Coding

Haruna Nishiwaki[1] Tomoharu Ugawa[2] Seiji Umatani[1,a)]
Masahiro Yasugi[3] Taiichi Yuasa[1]

Received: December 16, 2011, Accepted: April 24, 2012

Abstract: A static analysis tool has been developed for finding common mistakes in programs that use the Java Native Interface (JNI). Specific rules in JNI are not caught by C++ and other compilers, and this tool is aimed at rules about references to Java objects, which are passed to native methods as local references. Local references become invalid when the native method returns. To keep them valid after the return, the programmer should convert them into global references. If they are not converted, the garbage collector may malfunction and may, for example, fail to mark referenced objects. The developed static analysis tool finds assignments of local references to locations other than local variables such as global variables and structure fields. The tool was implemented as a plug-in for Clang, a compiler front-end for the LLVM. Application of this tool to native Android code demonstrated its effectiveness.

Keywords: program verification tool, escape analysis, JNI, Android, experience

1. Introduction

Java is a platform-independent programming language and thus enables the efficiency of software development to be improved and maintenance costs to be reduced. However, functions that depend on the execution environment cannot be called directly in Java. The Java Native Interface (JNI) [3] is used to call such functions.

JNI is a facility of Java that enables Java applications to interoperate with native applications. Native applications and libraries written in native languages such as C++ can be called from Java by using JNI. However, care must be taken when developing applications using JNI because JNI has specific rules.

Java objects are passed to native methods as local references in JNI. These local references are invalid after the execution of the native method finishes. These local references should thus be converted into global ones if they are to be used after the native method returns. Though most programmers know about this rule, they tend to forget about it and violate it. To make matters worse, compilers of native languages such as C++ cannot detect these violations.

We are working on enhancing Dalvik VM [13], a virtual machine that executes bytecode converted from a Java program. Dalvik VM runs on Android, which is an OS for smart phones. We found that even the application framework library provided with Android has violations of the JNI rules, and these violations prevented the VM from running correctly after we modified it. We began to check the source code in order to correct these violations but soon realized that it has too many lines to check.

Warnings about such violations have been posted on web pages for Android developers [5], and they have been given as an example of common mistake in section 10 of the JNI programmer's guide [11]. This is evidence that this problem affects not only Android developers but also all developers who use JNI.

To help us detect missing conversions from local references into global ones, we developed a static analysis tool called "SEAN" (Static Escape Analyzer for Native code). This tool identifies lines of source code where local references may be stored to somewhere other than local variables. Use of this tool greatly reduces the effort needed to find such violations because only those lines identified need to be manually checked.

This paper is organized as follows. In Section 2, we explain the pattern of violations on which this paper is focused. In Section 3, we describe the design and implementation of SEAN. In Section 4, we describe how we verified its operation, and in Section 5, we discuss related work. We conclude in Section 6 with a brief summary and a look at future work.

2. Problem Description

JNI provides two kinds of references, local and global, enabling native methods to handle Java objects. The objects are passed to native methods as local references. Local references are valid only during execution of the native method. They become invalid when the native method returns. Therefore, a programmer cannot write native methods that store local references somewhere (such as in global variables) and then use those references after the return. However, programmers often store local

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[2] Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[3] Department of Artificial Intelligence, Kyushu Institute of Technology, Fukuoka 820–8502, Japan
[a)] umatani@kuis.kyoto-u.ac.jp

© 2012 Information Processing Society of Japan

```
static jobject savedReference;

JNIEXPORT void JNICALL Java_Foo_someMethod(JNIEnv *env, jobject obj){
    // local reference is assigned to global variable
    savedReference = obj;
    ...
}

JNIEXPORT void JNICALL Java_Foo_someOtherMethod(JNIEnv *env, jobject obj){
    jclass clazz;
    // local reference may be invalid
    clazz = env->GetObjectClass(savedReference);
    ...
}
```

Fig. 1 Example of using an invalid local reference.

```
static jobject savedReference;

JNIEXPORT void JNICALL Java_Foo_someMethod(JNIEnv *env, jobject obj){
    // local reference is converted into global reference
    savedReference = env->NewGlobalRef(obj);
    ...
}

JNIEXPORT void JNICALL Java_Foo_someOtherMethod(JNIEnv *env, jobject obj){
    jclass clazz;
    // savedReference is valid until explicitly deleted
    // because it was converted to a global reference
    clazz = env->GetObjectClass(savedReference);
    ...
}
```

Fig. 2 Example of using a global reference.

references in global variables. **Figure 1** shows an example of this common violation. Native method `Java_Foo_someMethod` receives local reference `obj` and stores it in global variable `savedReference`. However, the saved reference is no longer valid after the native method returns. Thus,

```
clazz = env->GetObjectClass(savedReference);
```

may not return a valid value. To enable the program to use the object after the native method returns, the programmer must convert the local reference into a global one. In contrast to local references, global ones remain valid until they are explicitly deleted. Programmers can convert a local reference into a global one by explicitly using the `NewGlobalRef` function. The violation illustrated in Fig. 1 is corrected in **Fig. 2** using `NewGlobalRef`. The global reference stored in `savedReference` remains valid after `Java_Foo_someMethod` returns as long as it is not deleted. Thus, `GetObjectClass` returns the correct class of the object received by `Java_Foo_someMethod`. Programmers can delete a global reference by calling `DeleteGlobalRef` with the global reference as its argument. An object that is created in a native method is passed to the native method as a local reference. Thus, the reference must be converted into a global one in the same way as those that are passed as arguments if the program is to use it after the native method returns.

3. Design and Implementation

3.1 Design

The necessary condition for the rule violation described above is that the local reference *escapes* from the native method. That is, the local reference leaves the local environment of the native method. Our analyzer detects the following three cases as program points where such escapes may occur.

(1) A reference is assigned to a global variable, a member variable of a structure, a member variable of a class instance, or a static member variable (hereafter, we refer to these variables collectively as "global variables") with an assignment expression.
(2) A reference is stored in a member variable using the corresponding member initializer.
(3) The value of a *reference type* (`jobject` type or one of its subclass types) is converted to the value of a non-reference type by using a type cast, etc.

The rule violation illustrated in Fig. 1 is detected as an instance of case (1). More specifically, the following assignment to a global variable is detected:

```
savedReference = obj; // savedReference is a
                      // global variable.
```

Besides assignments to global variables, assignments to member variables may be rule violations in some cases. An assignment of some value to a member variable does not mean that the value escapes if the instance (or structure) to which the value is assigned is allocated on the stack. However, to simplify the implementation, we decided to detect any assignments to member variables regardless of their location.

C++ provides member initializers as well as assignment expressions that may store local references in member variables. Rule violations that use member initializers are detected as instances of case (2). The Android framework in fact includes a rule violation that uses a member initializer:

Table 1 Verification environment.

Computer	CPU	Memory	OS
Mac Book Air	Intel Core 2 Duo 2.13 GHz	4 GB	Mac OS X Ver 10.7.2

```
class Foo{
public:
    Foo(JNIEnv* env, jobject obj)
        : otherObj(obj){ ... } // value is passed
                               // using member
                               // initializer
    ˜Foo(){ ... }
        ...
private:
    jobject otherObj;
};
```

In this code, parameter `obj` is stored in member variable `otherObj` when constructor `Foo` is called.

Cases (1) and (2) are detected using type information (the detection methods are described in the next section). In case (1), an assignment expression is detected if the right-hand side of the expression is of a reference type. This means that such cases cannot be detected if local references are cast to non-reference types before their assignment. Moreover, once a local reference is cast to a non-reference type, it may be passed to any C++ library functions for which definitions are not written in the same source file. Therefore, we detect such escapes as instances of case (3). For example, we can detect a rule violation in which a local reference that is implicitly cast to the void-pointer type is then assigned to a member variable of a structure:

```
static someFoo *some;
class UserFoo{
public:
    void UserFooMemberFunc(void *any){
        some = new someFoo();
        // local reference cast to void-pointer
        // type is assigned to member
        // variable of structure
        some->foo = any;
    }
};
jobject FuncFoo(JNIEnv *env, jobject obj){
    UserFoo some = new UserFoo;
    // local reference is cast to void-pointer
    // type implicitly
    some->UserFooMemberFunc(obj);
}
```

3.2 Implementation

We implemented the SEAN static analyzer as a plug-in for Clang, which is a C language family front-end for the LLVM compiler [7].

We implemented detection for case (1) (described in the previous section) by detecting every assignment expression in which the left-hand side is a global variable and the right-hand side is of a reference type. We implemented the detection for case (2) by detecting every member initializer with at least one reference-type argument. We implemented detection for case (3) by detecting every expression that casts a reference-type expression to the void-pointer type. Strictly speaking, we should detect every expression that casts an expression to a non-reference type. However, for simplicity, we cover only those that cast to the void-pointer type. In the Android source code, there is no cast to non-reference types other than the void-pointer type.

For simplicity, the current implementation of SEAN does not check whether each Java object is a local reference. Therefore, it sometimes mistakenly detects rule violations. For example, for case (1), SEAN detects assignment expressions in which the right-hand side expressions are Java objects converted in advance to global references by the `NewGlobalRef` function.

Table 2 Number of detections.

Case	Reported	Actual
(1)	72	12
(2)	9	9
(3)	24	11

4. Verification

To verify the operation of our proposed static analyzer, we applied it to C++ source files in the `frameworks` directory of the Android source distribution. The version of Android we used is GingerBread 2.3.7. The `frameworks` directory contains the source code of frameworks for developing applications and consists of 1,037 files (500,877 lines of code). **Table 1** shows the computer settings used for the verification.

Table 2 summarizes the detections. The processing time was 92 seconds, the total number of detections for all cases was 105, and 32 of them were actual violations. Judgement of whether a reported violation was an actual violation was done manually by the four authors, three of whom have been working to improve Dalvik VM for six months.

The detected patterns of JNI rule violations correspond to two (P1 and P2) of the following four patterns; they are described on a web page for Android developers [5] as typical rule violations related to local references.

(P1) Forgetting to call `NewGlobalRef` when stashing a `jobject` in a native peer
(P2) Mistakenly assuming `FindClass` returns global references
(P3) Calling `DeleteLocalRef` and continuing to use the deleted reference
(P4) Calling `PopLocalFrame` and continuing to use a popped reference

Figure 3 shows one of the detected rule violations that do not correspond to any of the patterns above. In the code, the argument `thiz` is cast to the void-pointer type and then stored as a unique key. We cannot simply compare the source code of GingerBread (the target version of the verification) with that of IceCreamSandwich 4.0.3 (the latest version at the time of writing) due to the difference in file/directory structures of the source code. However, we see that the detected rule violations corresponding to (P1) and (P2) are corrected in IceCreamSandwich while that in Fig. 3 is not.

There are many code patterns in which, before a local reference is assigned to a global variable, the `GlobalNewRef` function

```
// frameworks/base/core/jni/android_hardware_Camera.cpp
static void android_hardware_Camera_native_setup(
    JNIEnv *env, jobject thiz, jobject weak_this, jint cameraId){
    ...
    // local reference cast to void-pointer type implicitly at the following line
    context->incStrong(thiz);
    ...
}
// frameworks/base/libs/utils/RefBase.cpp
void RefBase::incStrong(const void *id) const{
    ...
    // void-pointer type is stored as unique key
    refs->addWeakRef(id);
    ...
}
```

Fig. 3 Example violation detected in Android sources using JNI.

is called for it from other functions or macros. Although these patterns are not rule violations, our current SEAN implementation mistakenly detected them as rule violations.

5. Related Work

Programmers should strictly follow the rules of the foreign function interface (FFI) when writing programs in multiple languages using FFI. Even expert programmers often make a mistake in using FFI because such rules do not exist in programming using a single language. In fact, we found a case in which local references were cast to void-pointers and used as unique values. This case is not listed on the web page for Android developers [5].

Avoiding bugs due to violations of the FFI rules is a subject of active investigation because common compilers cannot detect such violations. The approaches taken can be classified as

- detecting violations by static analysis,
- detecting violations by runtime checking, and
- improving FFI.

Our tool detects violations by static analysis.

5.1 Static analysis

Furr and Foster [1] presented a multi-language type inference for checking the type safety of OCaml programs together with C functions called across FFI. They applied the inference to JNI and developed a system that detects in compile time a bug where a C function called across JNI calls a Java method with arguments of the wrong class [2]. Li and Tan [9], [10] proposed an analysis that detects C functions called across JNI that may throw exceptions that the calling Java methods do not expect. These studies targeted different violations than the one we did. SEAN is thus worth using with these systems. Moreover, these systems do not support C++ programs and thus cannot be used for verification of the Android framework.

Kondoh and Onodera [6] proposed static analyses that detect three kinds of violations from among the several kinds in the JNI programmer's guide [3]. One of their analyses is aimed at the same target as ours, that is, references that are no longer valid. Their analysis detects the lines of code where the program stores those values into global variables that are returned from invocations of functions except `NewGlobalRef` and `NewWeakGlobalRef`, which return global references by JNI specification. The analysis is a syntactic checking but not a data-flow analysis. Therefore, it cannot detect lines where the program stores those references into global variables that come through local variables or arguments. In contrast, SEAN detects such lines as well though it does not use a data-flow analysis. Instead, it uses types of variables described in C++ programs in such a way that it detects lines where values of reference types such as `jobject` are stored in global variables.

5.2 Dynamic analysis

Most Java VMs have a command line option such as `"-Xcheck:jni"` to detect JNI rule violations at run time. When this option is enabled, JNI verifies references passed from C functions as arguments of Java method calls or return values. This facility is designed and implemented by VM developers. Lee et al [8] developed a tool, Jinn, that synthesizes verification code for runtime checking from FFI rules.

As is shown in Section 2 of the paper by Lee et al [8], runtime checking can detect many kinds of violations with few false positives. However, it is impossible in general to detect all violations using only runtime checking since runtime checking cannot detect violations unless the problematic code is executed.

5.3 Improving FFI

SafeJNI [12] and Jeannie [4] are alternatives to direct use of JNI. Programs using these interfaces are finally converted into programs using JNI. These interfaces are designed to make it easy to detect violations statically, and violations are detected when the programs are converted.

This approach is useful when developing new programs. However, a violation detecting tool is still needed to find violations in the enormous amount of source code that has already been described using the JNI such as Android framework.

6. Summary

Our static analysis tool helps programmers find JNI coding rule violations as a plug-in module of Clang. Application of this tool to Android native code showed that JNI rule violations that cannot be detected by a compiler for the native language can be detected using static analysis.

Since the proposed method is simple, there are false positives. Nevertheless, in our demonstration experiment, it enabled us to focus on the potential violations of about 100 lines, rather than having to check half a million lines of code.

Future work includes improving the detection accuracy, possibly by preventing the tool from reporting typical patterns of false positives.

Future work also includes equipping the tool with data-flow analysis so that it can both reduce the false positives and detect (P3) and (P4) violations listed on the web page for Android developers [5]. Applying the tool to other applications besides Android that use JNI is also future work.

Acknowledgments I appreciate the feedback offered by reviewers. This work was partly supported by JSPS KAKENHI 22700026.

References

[1] Furr, M. and Foster, J.S.: Checking type safety of foreign function calls, *Proc. PLDI '05*, pp.62–72 (2005).
[2] Furr, M. and Foster, J.S.: Polymorphic type inference for the JNI, *Proc. ESOP '06*, pp.309–324 (2006).
[3] Gordon, R.: *Essential Java Native Interface* (*Essential Java*), Prentice Hall Ptr (1998).
[4] Hirzel, M. and Grimm, R.: Jeannie: Granting Java Native Interface Developers Their Wishes, *Proc. OOPSLA '07*, pp.19–38 (2007).
[5] Hughes, E.: JNI Local Reference Changes in ICS (2011), available from ⟨http://android-developers.blogspot.com/2011/11/jni-local-reference-changes-in-ics.html⟩.
[6] Kondoh, G. and Onodera, T.: Finding Bugs in Java Native Interface Programs, *Proc. ISSTA '08*, pp.109–117 (2008).
[7] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. 2004 International Symposium on Code Generation and Optimization* (*CGO'04*), pp.75–86 (2004).
[8] Lee, B., Wiedermann, B., Hirzel, M., Grimm, R. and McKinley, K.S.: Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces, *Proc. PLDI '10*, pp.36–49 (2010).
[9] Li, S. and Tan, G.: Finding bugs in exceptional situations of JNI programs, *Proc. CCS '09*, pp.442–452 (2009).
[10] Li, S. and Tan, G.: JET: Exception Checking in the Java Native Interface, *Proc. OOPSLA '11*, pp.345–357 (2011).
[11] Liang, S.: *Java Native Interface: Programmer's Guide and Specification*, pp.61–72, Addison-Wesley (1999).
[12] Tan, G., Appel, A.W., Chakradhar, S., Raghunathan, A., Ravi, S. and Wang, D.: Safe Java Native Interface, *Proc. ISSSE '06*, pp.97–106 (2006).
[13] Dalvik Technical Information, Android developers (2007), available from ⟨http://source.android.com/tech/dalvik/⟩.

Haruna Nishiwaki was born in 1988, and received a B.E. degree in electronic engineering from Kyoto Institute of Technology in 2011. Since 2011, she has studying informatics at the postgraduate program of Kyoto University. Her current research interests include programming languages and compilers.

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. Since 2008, he is an assistant professor at the University of Electro-Communications, Tokyo, Japan. His research interests include program languages, language processing systems, and system software.

Seiji Umatani was born in 1974, and received a B.E. degree in information science, and M.E. and Ph.D. degrees in informatics from Kyoto University, Kyoto, Japan, in 1999, 2001, and 2004, respectively. In 2004–2005, he was a research staff member in the Graduate School of Informatics at Kyoto University, and he was appointed to an assistant professor position in 2005. His current research interests include programming languages, compilers, and parallel/distributed systems. He is a member of ACM and the Japan Society for Software Science and Technology.

Masahiro Yasugi was born in 1967. He received a B.E. in electronic engineering, an M.E. in electrical engineering, and a Ph.D. in information science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he has been working at Kyushu Institute of Technology as a professor. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of ACM and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.

Taiichi Yuasa received the Bachelor of Mathematics degree in 1977, the Master of Mathematical Sciences degree in 1979, and the Doctor of Science degree in 1987, all from Kyoto University, Kyoto, Japan. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a professor at the Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current area of interest includes symbolic computation and programming language systems. Dr. Yuasa is a member of ACM, IEEE, the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science and Technology.

© 2012 Information Processing Society of Japan