Short Presentation: Static Verification of Global Heap References in Java Native Libraries

Florian Brandner, Dietmar Ebner, Andreas Krall and Christian Thalinger
Institut für Computersprachen
TU Wien
{brander,ebner,andi,twisti}@complang.tuwien.ac.at

ABSTRACT

Explicit memory management of Java objects is a frequent source of programming errors in Java native libraries. In this article we present a static verification tool for the automatic detection of frequent kinds of errors such as missing registration of global references. The tool implements a control-flow aware interprocedural escape analysis in order to determine which references have to be explicitly registered for the garbage collector followed by an analysis that makes sure that those references are created on all possible control flow paths. The tool has been applied to some large native libraries and we were able to detect a fairly large number of programming errors.

1. INTRODUCTION

Via the Java Native Interface (*JNI*) [?] Java allows to call methods written in other programming languages like C or C++. A Java Virtual Machine (*JVM*) [?] has automatic memory management. But automatic memory management does not work correctly anymore when references to Java objects are stored in global variables in native libraries. The programmer is responsible to register and unregister such references explicitly for the garbage collector (*GC*).

JNI distinguishes between local and global references. The creation of a Java object in a native function results in a local reference. This local reference is valid only in the creating thread until the scope of the native function ends. However, those references can be prematurely invalidated by a call of the *JNI* function `DeleteLocalRef`, thereby enabling the *GC* to reclaim the occupied memory space earlier. References passed as arguments to native functions or created locally can be stored in global (static) variables and can thus also be accessed by other threads. In this case it is necessary to register the reference with the *JNI* function `NewGlobalRef` to prevent the *GC* from reclaiming the memory space of a particular object too early. `NewGlobalRef` stores a reference to an object into a data structure which is known to the *GC* and prevents the *GC* thereby from marking them for collection. Those references han be removed later on by a call to `DeleteGlobalRef`.

To be presented at the third workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE), January 2006, Charleston, South Carolina

The explicit memory management in native functions is a source of different programming errors. Missing calls to `NewGlobalRef` or wrong placed `DeleteGlobalRef` calls lead to dangling references. Missing `DeleteGlobalRef` calls create memory leaks. Wrong pairing of `NewGlobalRef` and `DeleteGlobalRef` calls or multiple `DeleteGlobalRef` or `DeleteLocalRef` calls can lead to run time exceptions in some *JVM*s. These errors are very hard to detect. The *GC* is called at unpredictable intervals. Some *conservative GC*s also scan the global memory and do not produce an error when `NewGlobalRef` calls are missing. However, for *exact GC*s, the registration of Java objects stored in global variables in native code is essential.

In this work we present a tool to catch some frequent sources of errors in Java native libraries. We implement a control-flow aware interprocedural escape analysis in order to determine which references have to be explicitly registered for the garbage collector followed by an analysis that makes sure that those references are created on all possible control flow paths.

Related work is presented in Section 2. Section 3 explains our interprocedural algorithm, and computational results on some big Java native libraries are presented in Section 4.

2. RELATED WORK

The problem to determine if an object leaves a certain scope is known as escape analysis.

In 1988, Ruggieri and Murtagh developed an interprocedural analysis for determining the lifetime of dynamically allocated objects [?]. They partition the heap into subheaps for procedures and determine the objects whose lifetime is contained in the lifetime of the procedures.

Goldberg and Park [?] introduced the term escape analysis and proposed an algorithm to determine if arguments of functions have a greater lifetime than the function call itself.

Bruno Blanchet extended a Java-to-C compiler by an escape analysis [?]. The analysis transforms Java into *SSA* form, builds equations, and solves them with an iterative fixpoint solver.

John Whaley and Martin Rinard combined pointer and escape analysis [?]. They analyze arbitrary regions of incomplete programs obtaining complete information for objects which do not escape these regions.

Kotzmann and Mössenböck developed a very fast escape analysis for Suns Java HotSpot client compiler [?]. The analysis operates on an intermediate representation in *SSA* form and introduces equi-escape sets for the efficient propagation of escape information. The results are used for scalar replacement of fields and for stack allocation of objects.

Shaham et al. present a framework for statically reasoning about temporal heap properties [?]. They developed a conservative analysis which proves for certain program points that a memory object of a heap reference is not needed any more.

3. ANALYSIS

Our analysis makes use of the *gcc* [1] compiler framework and is based on its intermediate representation *GIMPLE*, which is basically a simple three address language without high level flow structures. No *GIMPLE* statement has implicit side effects and lexical scopes are represented as containers. Moreover, variables residing in memory are never used directly in expressions but loaded into a temporary which is used instead. Furthermore, we make use of a framework for interprocedural optimization recently introduced in *gcc*. However, due to technical limitations, this currently excludes the use of the Static Single Assignment form (*SSA*), which would simplify our analysis. Therefore, we have to take into account that references in the intermediate representation can be redefined. A project aiming to drop this restriction in *gcc* is currently scheduled for version 4.2.

Once the call graph is build, we iterate its nodes in post-order, thereby ensuring we analyze all callees before their callers as long as they are available in the current compilation unit and there are no backward edges. During the intraprocedural analysis, each function is annotated with the following information:

(a) Parameter declarations of a *JNI* type are marked, if they point to an object that escapes the current function and there is at least one path from the entry block to the escape site without passing the objects address to `NewGlobalRef`.

(b) A static function whose address is not taken is marked, if it returns a reference to a garbage collected object which is created within the function and for which no global reference has been created.

In general, pointers are assumed to escape from a function, if they are assigned to global (static) variables, passed to functions whose parameter is marked to escape as described in *(a)*, or returned by a function whose address was taken or that is declared non-static (otherwise, the calling functions will be analyzed later on and the returned object is tracked there).

In a first step, we heuristically determine the set of garbage collected objects. Therefore, we assume that all *JNI* entry points are dereferenced function pointers contained in a global environment structure (`JNIEnv`) and test their return type against a set of predefined types. The set of tracked objects is determined by those calls and by references returned from static functions as described in *(b)*. Furthermore, we can treat function parameters of appropriate type just like newly created objects while — instead of creating a warning — marking the corresponding parameter as described in *(a)* as long as the reference is not passed to `NewGlobalRef`.

[1] `http://gcc.gnu.org`

For each of those objects, a *object tag* is created that represents the global object. Each variable in the intermediate representation is annotated with an object tag if it points to a garbage collected object at a particular point. For each basic block, this information is stored in an in- and out-set while the out-set is computed from the in-set by traversing the basic block and applying the following rules:

- For a function call returning a global object, *i.e.*, a call to a *JNI* function of appropriate type or a static function annotated with this information, the left hand side of the expression is set to point to the associated object tag.
- For a simple copy- or cast-operation, the left hand side is set to the object tag associated with the right hand side.
- For any assignment overwriting the points-to information of the left hand side, *e.g.*, `var=NULL`, the associated object tag is removed.

Furthermore, in-sets of a particular basic block are computed by merging the out-sets of its predecessor. However, if a particular variable points to different object tags in its predecessors, a *virtual tag* is created and associated with the current variable. A virtual tag is annotated with a set of "real" tags and encodes the information "one of those, I do not know which". Merging two different virtual tags results in a new virtual tag while the set of real tags is the union of both. For the rest of the analysis, virtual tags and real tags are mostly treated the same. This procedure is repeated for each basic block of the current function until all in- and out-sets remain stable, *i.e.*, the points-to information has been fully propagated within the current function.

Next, each object tag o is associated with a mark $m_{o,b} \in \{\mathtt{m}, \mathtt{u}, \top\}$ for each basic block b and initialized to $\top$. This mark encodes the information, if an object escaping within the current basic block is ensured to be registered at the garbage collector, *i.e.*, it is registered either on all control-flow paths entering the block or on all paths from the current block to the exit block. Again, $m_{o,b}$ for a particular basic block b is computed by merging the information of the predecessors and the procedure is iterated until the marks remain stable. Algorithm 3.1 shows how $m_{o,b}$ is computed for all object tags o of a particular basic block b. Object tags for formal parameters are considered to be defined in the entry block of the function. The operator $\oplus$ is defined as given in Table 1.

$\oplus$	m	u	$\top$
m	m	u	m
u	u	u	u
$\top$	m	u	$\top$

Table 1: Operator $\oplus$. The three states encode if a particular tag can be proven to be registerd at the *GC* (`m`), unregistered on at least one path (`u`), or this information is not yet known ($\top$).

In a final step, all escape sites for each basic block b are checked and a warning is generated, if $m_{o,b} \neq \mathtt{m}$ for a particular object tag o. If o is a virtual tag, each of its associated

Algorithm 3.1 `combine_mark(basic_block b)`

```
1: for all object tags $o$ do
2:   if $o$ is created in $b$ then
3:     if type($o$) = virtual then
4:       let $m_1, \ldots, m_n$ be the marks of associated tags in their source block
5:       $m_{o,b} = m_1 \oplus m_2 \cdots \oplus m_{n-1} \oplus m_n$
6:     else
7:       $m_{o,b} = \mathtt{u}$
8:   if $o$ is passed to NewGlobalRef within $b$ then
9:     $m_{o,b} = \mathtt{m}$
10:  else
11:    $m_{o,b} = \top$
12:    for all $p \in$ pred($b$) do
13:      $m_{o,b} = m_{o,b} \oplus m_{o,p}$
14:    if $m_{o,b} \neq \mathtt{m}$ then
15:      let $\{s_1, \ldots, s_m\}$ be the set of successors of $b$
16:      if $m_{o,s_1} = \cdots = m_{o,s_m} = \mathtt{m}$ then
17:        $m_{o,b} = \mathtt{m}$
```

real tags are processed instead and in the case o was created for a formal parameter, this parameter is marked to escape without being registered in the function annotation instead of emitting a warning.

4. EXPERIMENTAL RESULTS

The algorithm has been tested on a number of prominent open source libraries making extensive use of *JNI* code, *i.e.*, the *GNU classpath* project (version 0.19, 31.000 lines of code (*loc*)), the *The Standard Widget Tookit* (*SWT*) (version 3.2M3, 22.000 *loc*), and *libgtk-java* (version 2.6.2, 27.000 *loc*).

	classpath	SWT	*libgtk-java*
`jclass`	5/7	61/61	0/0
`jmethodID`	35/43	1/1	22/22
`jfieldID`	5/8	498/498	0/0
others	0/38	0/0	0/158
Σ	45/96	560/560	22/180
`NewGlobalRef` calls	28	2	17

Table 2: Computational results for three prominent Java libraries. Numbers indicate the number of bugs in comparison to the number of generated warnings and the number of explicit references created.

Table 2 presents the number of bugs and the number of emitted warnings for each of the three test cases. Although many warnings for field- and method IDs are emitted, those instances are harmless in many cases since they are valid as long as the particular class is loaded. However, one should at least create a global reference to the corresponding class to prevent the VM from prematurely unloading the class, since this might result in dangling references. We therefore count only those instances as bugs where neither a global reference for the class nor for a corresponding object has been created.

Furthermore, most false positives (50.0% for *GNU classpath*, 86.07% for gtk-java) result from return expressions of native functions that are mostly called from within the VM and usually do not cause problems. However, we cannot rule out that those functions are called from other native functions in different modules and therefore emit a warning in these cases. Note that such occasions can be easily filtered since those functions follow a common naming convention.

The huge number of bugs in *SWT* results from code that is automatically generated and omits necessary calls to `NewGlobalRef`. Patches for most of the identified bugs have already been submitted to the project maintainers.

5. CONCLUSION AND FURTHER WORK

Static verification for *JNI* code has proven to be a powerful tool for the detection of frequent sources of error and we plan to incorporate analysis for some further frequent error patterns, *i.e.*, deletion of local references before the allocation of a global reference, unnecessary allocation of global references for local variables, and - as far as possible - static analysis of missing removals of allocated global references.

Acknowledgements

This work was supported in part by the Christian Doppler Forschungsgesellschaft, Infineon, and OnDemand Microelectronics.