Identifying Java Calls in Native Code via Binary Scanning

George Fourtounis
Leonidas Triantafyllou
Yannis Smaragdakis

University of Athens

ISSTA 2020

Native Code in Java Programs

- **Java aims for platform independence, yet native code is pervasive, esp. in Android**
  - *540 of 600 top free apps in Google Play Store contain native libraries, average of 8 libs/app [Almanee et al. 2019]*
- **Native code is opaque for Java static analysis tools**
  - Core threat in call-graph analysis *[Sui et al. 2020]*
  - Android apps often require manual marking of native call-backs

This Work

- Detect call-backs from native code to Java
- Fix unsoundness in reachability computation
- Inform call-graph computation
- Key feature: Lightweight
  - avoid full-blown dual Java-native analysis
  - start from strings found in binaries

Java Native Interface

- Native code interacts with Java via the JNI API
- In practice, using constant strings

```
JNIEXPORT void JNICALL
Java_JNIExample_callBack(JNIEnv* env, jobject obj) {
  jclass cls = (*env)->FindClass(env, "HelloJNI");
  jmethodID helloMethod = (*env)->GetMethodID(env, cls,
                          "helloMethod",
                          "(Ljava/lang/Object;Ljava/lang/Object;)I");
  jint i = (*env)->CallIntMethod(env, obj, helloMethod, obj, obj);
  printf ("callBack(): i=%d\n", i);
}
```

Java Native Interface

- Native code interacts with Java via the JNI API
- In practice, using constant strings

```
JNIEXPORT void JNICALL
Java_JNIExample_callBack(JNIEnv* env, jobject obj) {
  jclass cls = (*env)->FindClass(env, "HelloJNI");
  jmethodID helloMethod = (*env)->GetMethodID(env, cls,
                                  "helloMethod",
                                  "(Ljava/lang/Object;Ljava/lang/Object;)I");
  jint i = (*env)->CallIntMethod(env, obj, helloMethod, obj, obj);
  printf ("callBack(): i=%d\n", i);
}
```

Our Approach

```
$ readelf -p .rodata libhello.so
String dump of section '.rodata':
...
[28] Hello World!
[35] HelloJNI
[3e] ()V
[42] <init>
[50] (Ljava/lang/Object;Ljava/lang/Object;)I
[78] helloMethod
```

- The strings still exist and can be classified as *names* and *signatures*
- *Call-backs* = *names* x *signatures*
- Filter call-backs to improve precision
- Determine call-graph edges

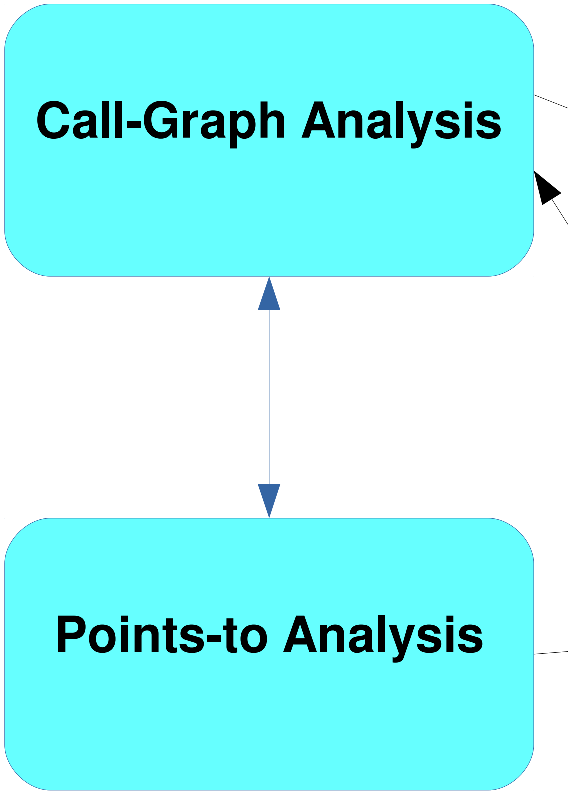
Problem 1:
Naive String Product does not Scale

- Big native libraries contain many common strings (“read”, “write”, “id”) and method signatures tend to be simple (“void()”, “int(int)”)
- Too many false positives
- For precision, combine strings per-native-function
  - Must analyze native code to find function boundaries (via Radare2)
  - Not always possible, revert to unfiltered mode for hostile native code

Problem 2: Constructor Call-Backs

- **Constructor call-backs = native allocations**
  - Native allocations are a major source of unsoundness *[Sui et al. 2020]*
- **Constructor methods are all “`<init>`”, only signatures differ**
  - Again, too many false positives
- **Apply precision heuristic:**
  - A constructor is called if an instance of its class is used (instance field read/written, instance method invoked)
  - Informed by points-to/call-graph analysis
    - Easy to express in Doop's mutually-recursive logic

Constructor Fitering, Mutually-Recursive




```
// Accept calls to constructors if instance methods are reachable.
HighlyPossibleNativeCodeTargetMethod(?method, ?function, ?file) :-
  basic.PossibleNativeCodeTargetMethod(?method, ?function, ?file),
  basic.ClassConstructor(?method, ?type),
  Method_DeclaringType(?instanceMethod, ?type),
  Reachable(?instanceMethod),
  !Method_Modifier("static", ?instanceMethod).
// Accept calls to constructors if instance fields are used.
HighlyPossibleNativeCodeTargetMethod(?method, ?function, ?file) :-
  basic.PossibleNativeCodeTargetMethod(?method, ?function, ?file),
  basic.ClassConstructor(?method, ?type),
  Field_DeclaringType(?field, ?type),
  ( StoreHeapInstanceField(?field, _, _, _, _)
  ; LoadHeapInstanceField(_, _, ?field, _, _)).
// Make call-backs reachable.
Reachable(?method) :- HighlyPossibleNativeCodeTargetMethod(?method, _, _).
```

Problem 3:
Inform Java Call Graph

- A Java call graph edge: `native` $m \to m'$
- So far, we know that m is called (i.e., fix reachability unsoundness) by some native function ***F***
- But how do we know that ***F*** corresponds to m? (fix call-graph unsoundness)
- Solution: model JNI linking
  - Automatic
  - Configurable (difficult!)

JNI Linking: Automatic

```
package x.y;
class C {
  native meth (Object obj);
}
```

- Native code: `Java_x_y_C_meth`
- Optional signature suffix

JNI Linking: Configurable

- `RegisterNatives()`
  - May be called from outside functions
  - Android, “crazy linker”
- Handling: read more strings, from more sections, recover linking triplets
- Not always possible (hostile binaries)

```
static const char *classPathName =
  "jackpal/androidterm/compat/FileCompat$Api8OrEarlier";
static JNINativeMethod method_table[] = {
  { "testExecute",
    "(Ljava/lang/String;)Z",
    (void *) testExecute
  }};


int init_FileCompat(JNIEnv *env) {
  registerNativeMethods(env, classPathName,
    method_table,
    sizeof(method_table) / sizeof(method_table[0]))
}
```

Problem 3:
Inform Java Call Graph

- Our handling of linking is shallow
- If `native` $m \rightarrow \boldsymbol{F} \rightarrow \boldsymbol{F'} \rightarrow m'$, we do not recognize $m \rightarrow m'$
  - unless we extract the call graph from Radare2

Evaluation 1: XCorpus

Benchmark	App methods (native)	+App-reachable	+Analysis time	+Factgen time	+Entry points
aspectj-1.6.9	41749 (8)	13034→13454: 3.22%	229→249: 8.7%	74→78: 5.4%	47
log4j-1.2.16	3423 (3)	961→961: 0.00%	60→58: -3.3%	47→49: 4.3%	0
lucene-4.3.0	33393 (9)	12414→12612: 1.59%	117→260: 122.22%	57→58: 1.75%	182
tomcat-7.0.2	19661 (273)	1204→2037: 69.19%	61→227: 272.13%	61→95: 55.7%	246

- Ground truth: the XCorpus “feature report”
  - reports native methods
- All call-backs are detected
- Control benchmark: log4j (no call-backs, no added entry points)

Evaluation 2: HeapDL Android Apps

Benchmark	App methods	Base recall	Recall	+App-reachable	+Analysis time	+Factgen time	+Entry points
Chrome	37898	7/83 = 8.43%	83/83 = 100.00%	17003→24060: 41.50%	469→505: 7.7%	46→255: 454.4%	4484
Instagram	43420	1/7 = 14.29%	7/7 = 100.00%	23921→32425: 35.55%	473→625: 32.1%	51→63: 23.5%	4669

- Two big Android benchmarks that called back from native code in their dynamic runs
- All call-backs are detected
- “Base recall”, some call-backs may be:
  - reachable from other places
  - false positives due to analysis over-approximation

Application: Finding Native Calls in Android apps

- Android apps resort to program optimizers (ProGuard/R8) to shrink/obfuscate code
  - Manual maintenance of “keep” rules to avoid removing/obfuscating native call-backs
- We automatically calculate these rules
- Example: Wikipedia app
  - 224 rules by default, +26 rules for native code
- Available in Doop and the Clyze packager

Conclusion

- Use strings found in JNI binaries to find call-backs to Java code
- Uncover missed methods in reachability computation
- Fix missing edges in call-graph analysis
- 100% success in real-world programs

Thank you!