Identifying Java Calls in Native Code via Binary Scanning

GEORGE FOURTOUNIS, University of Athens, Greece
LEONIDAS TRIANTAFYLLOU, University of Athens, Greece
YANNIS SMARAGDAKIS, University of Athens, Greece

Current Java static analyzers, operating either on the source or bytecode level, exhibit unsoundness for programs that contain native code. We show that the Java Native Interface (JNI) specification, which is used by Java programs to interoperate with Java code, is principled enough to permit static reasoning about the effects of native code on program execution when it comes to call-backs. Our approach consists of disassembling native binaries, recovering static symbol information that corresponds to Java method signatures, and producing a model for statically exercising these native call-backs with appropriate mock objects.

The approach manages to recover virtually all Java calls in native code, for both Android and Java desktop applications—(a) achieving 100% native-to-application call-graph recall on large Android applications (Chrome, Instagram) and (b) capturing the full native call-back behavior of the XCorpus suite programs.

CCS Concepts: • **Software and its engineering → Compilers**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: static analysis, Java, native code, binary

ACM Reference Format:
George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java Calls in Native Code via Binary Scanning. 1, 1 (May 2020), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over two decades ago, Java ushered in the era of portable, architecture-independent application development. The attempt to make portable mainstream applications was originally met with skepticism and became a critical point in Java adoption debates, as well as in the focus of the language implementors. Within a few years, the Java portability story was firmly established, and since then it has been paramount in the dominance of Java—the top ecosystem in current software development.

An often-overlooked fact, however, is that platform-specific (*native*) code is far from absent in the Java world. Advanced applications often complement their platform-independent, pure-Java functionality with specialized, platform-specific libraries. In Android, for instance, Almanee et al. [1] find that 540 of the 600 top free apps in the Google Play Store contain native libraries, at an average of 8 libraries per app! (The architectural near-monopoly of ARM in Android devices certainly does nothing to discourage the trend.) Desktop and enterprise Java applications seem to use native code much more sparingly, but native code still creeps in. Popular projects such as *log4j*, *lucene*, *aspectj*, or *tomcat* use native code for low-level resource access [11].

Authors' addresses: George Fourtounis, University of Athens, Athens, Post-Code1, Greece, gfour@di.uoa.gr; Leonidas Triantafyllou, University of Athens, Athens, Post-Code1, Greece, leotriantafyllou@gmail.com; Yannis Smaragdakis, University of Athens, Athens, Post-Code1, Greece, smaragd@di.uoa.gr.

The presence of native code in a Java application hinders static analysis, at any level. Failing to analyze the native parts of an application causes analysis *unsoundness* [12, 30]. Concretely, Sui et al. recently showed that native code is a core threat in call-graph analysis [38]. Native code can call back to Java code, introducing false negatives in *reachability* analysis—the static analysis that finds which parts of the code are reachable. Reachability analysis is, for instance, critical for Android: as part of packaging an Android application for deployment, unreachable (*dead*) code is eliminated via automated analysis. Modern Android development depends on a manually-guided workflow (via the ProGuard [23] configuration language) to explicitly capture the Java entry points used by native code, so that reachable code does not get optimized away.

Other than such manual "fixes" of the analysis results, there are few solutions to the problems of native-code-induced analysis unsoundness. Reif et al. [33] find that "none of the [state-of-the-art Java analysis] frameworks support cross-language analyses". In recent work, Lee proposes (as planned work) a hybrid Java/C static analysis [26] that addresses the issue. However, this heavyweight approach requires access and analysis of native source code, which is a severe burden in practice. Source code for third-party native libraries (and even metadata, such as DWARF information [8]) is typically unavailable to the Java developer. Furthermore, analyzing the source code of the native library is very hard—e.g., the code may be in any of several languages (C, C++, Rust, Go), many of which currently have no practically effective whole-program analysis infrastructure.

In this paper, we present a technique for finding the call-backs from native to Java bytecode,[1] via scanning of the binary libraries and cross-referencing the information with the Java code structure. Our approach recognizes uses of the Java Native Interface (JNI) API, which provides the bridge between native and Java code. Specifically, the technique identifies string constants that match Java method names and type signatures in native libraries, and follows their propagation (to find where method name strings are used together with type signature strings). In this way, the technique identifies entry points into Java code from native code, without fully tracking calls (i.e., call-graph edges) inside native code.

The resulting technique informs the static analyses of the Doop framework [5]. It is the first approach to effectively address unsoundness in static reachability analysis, in the presence of binary libraries. We evaluate the approach over large Android applications (Chrome, Instagram) and the native-code-containing programs in the XCorpus suite [11]. The two settings mandate different evaluation methodologies: for the Android applications, no native source code is available, yet the application has dynamic execution snapshots, showing Java methods called from native code. For the XCorpus programs, the bundled test suite does not exercise native call-backs, yet the native source is available for manual inspection. In both cases, our approach captures the full call-back behavior of the native code.

2 BACKGROUND

This section introduces the Java Native Interface specification (Section 2.1) and declarative static analysis (Section 2.2).

2.1 Java Native Interface

The Java Native Interface (JNI) [31] is an interface that enables native libraries written in other programming languages, such as C and C++, to communicate with the Java code of the application inside the Java Virtual Machine (JVM). The JNI is a principled form of a foreign function interface (FFI), a feature that mature programming languages usually incorporate as an escape hatch to third-party functionality or low-level operations. The JNI was first supported in JDK

[1] Java bytecode may not necessarily be produced from Java source code. For simplicity, we merely write "Java code" in the rest of the paper, with the understanding that the applicability of the technique extends transparently to all languages producing Java bytecode.

```
JNIEXPORT void JNICALL
Java_JNIExample_hello(JNIEnv *env,
 jobject thisObj, jobject arg) {
  printf("Hello World!\n");
  return;
}
```

Fig. 1. "Hello world" native function example.

Table 1. Java Method Signatures Examples.

Method	Signature
void m1()	()V
int m2(long)	(J)I
void m3(String)	(Ljava/lang/String;)V
String m4(String, int[])	(Ljava/lang/String;[I)Ljava/lang/String;

release 1.1, to improve the interplay of Java with native code (at a time when the JVM could itself be integrated with native code, especially Web browsers) [29].

The JNI allows programmers to use native code in their applications without requiring any change to the Java VM, which means that the native code can run inside any Java VM that offers JNI support. Via JNI, it is possible to create new Java objects and update them in native code functions, call Java methods of the same application from native code, and load classes and inspect them. This functionality is supported by an extensive API with appropriate methods and data structures that let native code interact with Java objects by using JVM concepts such as method and field descriptors. Such descriptors are full signatures for methods and fields, as they appear in bytecode, i.e. generics have been erased and types are represented by their low-level counterparts.

Figure 1 shows the "hello world" program in JNI, which exhibits the following features of the JNI API:

- The native function that implements native Java method *JNIExample.hello(Object arg)* is assumed to be named *Java_JNIExample_hello* and take a corresponding *jobject* argument.
- The native function also accepts a `JNIEnv` pointer for a reference to the JNI environment and a `jobject` for a reference to the receiver object (`this`). The `JNIEnv` argument points to a structure storing all JNI function pointers, which allow instantiation and use of objects, conversion between native strings and Java strings, and other functionality.
- The native function is decorated with macros that control the native code linking (`JNIEXPORT`) and call convention (`JNICALL`) for the specific platform for which the code will be compiled.

When using native code in an application, it is possible to call back Java methods from native functions. In order to call back a Java method, the programmer needs to find its method id object (of JNI type *jmethodID*). This object is looked up by giving the name of the containing class, the name of the method, and the low-level signature of the method (JVM method descriptor). The signature is a string of the form *(parameters)return-value* with some examples of methods and their signatures shown in Table 1.

The process of calling back a method starts by getting a reference to the object's class by using method `FindClass()` [24]. Then, the method name and signature are given as arguments in the function `GetMethodID()` of the class reference and the method id is returned. The method id can be used to call the Java method using the right function for the specific case, such as `CallVoidMethod()`, `Call<Primitive-type>Method()` and `CallObjectMethod()`. As for

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

Fig. 2. Call back Java method from native function example.

the type of the returned value of the called method, this can be `void`, <Primitive-type> and `Object`, respectively. An example of the process for calling a Java method that takes an `Object` argument and returns an integer through native code is shown in Figure 2.

2.2 Points-To Analysis in Datalog

Datalog is a declarative logic-based programming language which is designed to be used as a query language for deductive databases. Our analysis uses the Doop framework, implemented in Datalog [5], which provides a rich set of points-to analyses (e.g., context insensitive, call-site sensitive, object sensitive) for Java bytecode. However, because of the modular way of context representation in the framework, code built upon any such analysis can be oblivious to the exact choice of context (which is specified at run-time).

Soot [42] is a framework that is used by Doop and is responsible for generating input facts for an analysis as a pre-processing step. By using this framework, Doop expects as input the bytecode form of a Java program, which means the original source is not needed but only the compiled classes are necessary. This allows for analyzing programs whose source code is not available. The set of asserted facts for a program is called its EDB (Extensional Database) in Datalog semantics. The relations that are generated and directly produced from the input Java program, and any relation data added to the asserted facts by user defined rules, constitute the EDB predicates.

```
VarPointsTo(obj, var) :-
  AssignHeapAllocation(obj, var).
VarPointsTo(obj, to) :-
  Assign(to, from), VarPointsTo(obj, from).
```

Fig. 3. Simple Datalog example for IDB rules.

Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation (Figure 3). The Datalog code of the example consists of two simple rules known as IDB (Intensional Database) rules in Datalog semantics. These two rules are used to establish new facts from a conjunction of facts that are already established. The rule of the first line constitutes the base case of the computation and states that upon the assignment of an allocated heap object to a variable, this variable may point to that heap object. The second rule is the recursive case which states that if the value of a variable is assigned to another variable, then the second variable may point to any heap object the first variable may point to. For instance, the recursive rule of line 2 states that if `Assign(to, from)` and `VarPointsTo(obj, from)` are both true for some values of `from`, `to`, and `obj`, then that `VarPointsTo(obj, to)` is also true.

```java
public class HelloJNI {
    static {
        System.load("libhello.so");
    }

    // Declare a native method sayHello() that receives nothing and returns void
    private native void sayHello();
    private native Object newJNIObj();
    private native void callBack(Object obj);

    static Object sObj;

    // Test Driver
    public static void main(String[] args) {
        HelloJNI hj = new HelloJNI();
        hj.sayHello();  // invoke the native method
        Object obj = hj.newJNIObj();
        System.out.println(obj.toString());
        sObj = hj.newJNIObj();
        System.out.println(sObj.toString());
        hj.callBack(new Object());
    }

    public int helloMethod(Object obj1, Object obj2) {
        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());
        return 1;
    }
}
```

Fig. 4. Code of HelloJNI.java example file.

3 HELLOJNI EXAMPLE

This section describes our technique informally using an easy example: a toy Java/C program that uses few string constants and is easy to disassemble. We will use standard command-line tools to show the essence of our technique, without yet introducing the additional modeling and filtering (which will come in Section 4).

Assume we have a Java program (Figure 4) that defines native functions (Figure 5).[2] Further, assume we compile this code on Linux, on x86-64 hardware.

A pure-Java static analysis of the resulting program will miss the calls from the native code for methods `newJNIObj()` and `callBack()`. However, we observe that necessary parts of the target methods (names and signatures) appear in the native code as constant strings.

Investigating the problem, we first examine the resulting .so library, which is in ELF format. ELF (Executable and Linkable Format) [32] is a file format for binaries, libraries, and core files. In the ELF library, string constants reside in the `.rodata` section [27]. We use the `readelf` command [14] to view the ELF sections and find the address of section `.rodata` and then view the strings in `.rodata` (Figure 6). Since the section starts at address 2000, the strings "HelloJNI", "(Ljava/lang/Object;Ljava/lang/Object;)I", and "helloMethod" are at addresses 2035, 2050, and 2078 respectively.

Disassembling `Java_HelloJNI_callBack()` in Figure 7, shows `lea` instructions with a computed addresses in comments (computed by GDB[3]). These computed addresses are the references to the three strings found in the previous step. Thus, we can deduce that the native function uses these strings, one of which looks like a JVM signature. Also, these three

[2]Code adapted from online JNI tutorial [24].
[3]https://www.gnu.org/software/gdb/

```c
#include <jni.h>
#include <stdio.h>

// Implementation of native method sayHello() of HelloJNI class
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
   printf("Hello World!\n");
   return;
}

JNIEXPORT jobject JNICALL Java_HelloJNI_newJNIObj(JNIEnv *env, jobject thisObj) {
    jclass cls = (*env)->FindClass(env, "HelloJNI");
    jmethodID constructor = (*env)->GetMethodID(env, cls, "<init>", "()V");
    return (*env)->NewObject(env, cls, constructor);
}

JNIEXPORT void JNICALL Java_HelloJNI_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);
}
```

Fig. 5. Code of HelloJNI.c example file.

```
$ readelf --sections libhello.so
Section Headers:
[Nr] Name              Type            Address          Offset
     Size              EntSize         Flags  Link  Info  Align
...
[13] .rodata           PROGBITS        0000000000002000  00002000
     00000000000001ab  0000000000000000   A       0     0     8
...
$ 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
  ...
```

Fig. 6. Viewing section `.rodata` in the example program.

strings match the type, name, and JVM signature of an existing Java method `HelloJNI.helloMethod()`, thus the native function may be calling this Java method.

Finally, we can map the native function back to the original Java method. While this in general can be arbitrarily difficult, in this example, we assume the default JNI behavior where `Java_HelloJNI_callBack()` will be linked to an existing native Java method `HelloJNI.callBack()`. Thus, we can form a call-graph edge from `HelloJNI.callBack()` to `HelloJNI.helloMethod()`.

The above steps assume that (a) the input program is easy to disassemble (debugging metadata or other information offers function boundaries), (b) there is a way to statically compute the references to the strings that are used, and

```
0x00000000000011d6 <+31>:    lea    0xe58(%rip),%rsi        # 0x2035
0x00000000000011dd <+38>:    mov    %rdx,%rdi
0x00000000000011e0 <+41>:    callq  *%rax
...
0x00000000000011fc <+69>:    lea    0xe4d(%rip),%rcx        # 0x2050
0x0000000000001203 <+76>:    lea    0xe6e(%rip),%rdx        # 0x2078
0x000000000000120a <+83>:    callq  *%rax
```

Fig. 7. Disassembled native function `Java_HelloJNI_callBack()`, where string references are shown in comments.

(c) there is an easy way to map native functions back to their Java entry points. These assumptions may be violated: stripped or optimized binaries may be difficult to disassemble, address computation is platform- and compiler-dependent, and JNI linking can be complex. The next section presents the full technique, with more details on how to handle such difficult cases.

4 OUR TECHNIQUE

A summary of the steps of our technique follows:

- PRE A pre-processing step finds all method names and signatures in the Java code, forming a set $\mathcal{M}_0$.
- FILT The native code of each library n is scanned for strings that can be found in $\mathcal{M}$, resulting in set $\mathcal{M}^n \subseteq \mathcal{M}_0$.
- LOC The strings in $\mathcal{M}^n$ are localized: for each native function f in library n, $\mathcal{L}_f^n$ is the set of strings being used in the body of f.
- INVO For each function f, any method whose name and signature can be found in $\mathcal{L}_f^n$ is determined to be reachable from f, forming set $\mathcal{I}_f^n$.
- EDGE If the native function f implements a native Java method m, we form a call-graph edge from native method m to each method in $\mathcal{I}_f^n$.

4.1 Step PRE

This step takes place during the pre-processing stage of the static analysis. In the Doop framework, this stage consists of "fact generation" (implemented using either Soot [42] or WALA [10]): the extraction of database tables with input program information, for later processing by Datalog rules. During the PRE phase, every method encountered will save its name and JVM-level signature in set $\mathcal{M}_0$.

4.2 Step FILT

We find the native code of the application by reading all files recognized as dynamic libraries (e.g., with filenames ending in .so or .dll) from the input program. We extract the strings from the native code and only keep those that match actual method names or signatures in the program (by crossreferencing set $\mathcal{M}_0$, above). The resulting set is $\mathcal{M}$. As an optimization, if no method names or no signatures are found, we stop the analysis of this library (since the cross-product of names and signatures in the next step will be empty).

Finding strings is easy, as these are constants stored in special sections of the binary code (such as section ".rodata" in Linux ELF files). In practice, we can use the GNU "strings" utility, roll our own code to look for NULL-terminated strings, or use a special disassembler. We choose the last option and use Radare2,[4] a free and powerful reverse engineering

[4]https://rada.re/

```
.decl PossibleNativeCodeTargetMethod(method:Method, function:symbol, file:symbol)

PossibleNativeCodeTargetMethod(method, function, file) :-
  NativeMethodTypeCandidate(file, function, descriptor),
  NativeNameCandidate(file, function, name),
  Method_SimpleName(method, name),
  Method_JVMDescriptor(method, descriptor).
```

Fig. 8. Datalog rule to find possible Java method calls by matching method names and type descriptors used in the same native function.

```
// Mock arguments for methods called from native code.
MockValueConsMacro(mockId, frmType),
VarPointsTo(mockId, frm) :-
  PossibleNativeCodeTargetMethod(method, function, file),
  FormalParam(_, method, frm),
  Var_Type(frm, frmType),
  mockId = "<mock native object of type " + frmType + " from " + file + ":" + function + ">".

// Mock 'this' for methods called from native code.
MockValueConsMacro(mockId, type),
VarPointsTo(mockId, this) :-
  PossibleNativeCodeTargetMethod(method, function, file),
  ThisVar(method, this),
  Var_Type(this, type),
  mockId = "<mock receiver of type " + type + " from " + file + ":" + function + ">".
```

Fig. 9. Datalog rules to mock arguments and receivers of methods called from native code.

framework, which is also employed in other steps of our approach. As a side effect, Radare2 gives us support for multiple hardware targets (x86, x86_64, arm64, and armeabi) and operating systems (Linux, macOS, Windows).

This step also records global (or "static") strings stored in the binary (such as the strings in the ".data" section in Linux ELF files). These strings are also useful for resolving JNI functionality related to dynamic linking and may be used later, in step CALL.

4.3 Steps LOC and INVO

At this point, we can already perform step INVO crudely, to match all found strings against the method names and type signatures (a.k.a. "descriptors") of the Java code. If a method finds both its name and type signature in $\mathcal{M}$, then that method can be assumed to be called from the native code.

However, this can be too imprecise since a big native code library may contain many strings: strings used in different functions can accidentally match methods that will not be called in reality. Also, the resulting information is too basic: we can only determine the methods reachable from a native code library, which helps reachability computation but we cannot identify a call-site or other caller identity.

To address both above issues (imprecision and loss of invocation location), we perform step LOC, which finds which native code function uses which strings. Then, we can match name/signature strings per function, and solve the precision problem outlined above. Knowing the function can also help with the construction of a native call-graph edge $\mathcal{N}(f, m)$ from a native function f to a Java method m.

To find all places in the binary code where strings are referenced, we perform a "string cross-references" (a.k.a. "string x-refs") analysis using Radare2. Radare2 can analyze binary code and convert it to a stack-based IR, which then

```
// A filtered version of the cross-product.
.decl HighlyProbableNativeCodeTargetMethod(method:Method, function:symbol, file:symbol)
// Accept calls to non-constructors.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
  PossibleNativeCodeTargetMethod(method, function, file),
  !ClassConstructor(method, _).
// Accept calls to constructors if instance methods are reachable.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
  PossibleNativeCodeTargetMethod(method, function, file),
  ClassConstructor(method, type),
  Method_DeclaringType(instanceMethod, type),
  Reachable(instanceMethod),
  !Method_Modifier("final", instanceMethod).
// Accept calls to constructors if instance fields are used.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
  PossibleNativeCodeTargetMethod(method, function, file),
  ClassConstructor(method, type),
  Field_DeclaringType(field, type),
  ( StoreHeapInstanceField(field, _, _, _)
  ; LoadHeapInstanceField(_, _, field, _, _)),
   !Field_Modifier("final", field).
```

Fig. 10. Filtering the cross-product for constructors.

can be analyzed and partially evaluated so that string references can be computed [9]. A string x-refs analysis is needed since string references may not appear as local constants but be formed at runtime, by Position Independent Code [27, Chapter 8], by some layers of obfuscation, or by the hardware architecture at hand.

We should note here that static disassembly is not a solved problem [2]. For example, binary files may contain complex constructs, such as overlapping code and inline data in executable regions. In our technique, function boundaries (and the assignment of symbols to specific functions where they are used) may not be always recovered with full accuracy.

The output of the LOC step is $\mathcal{L}_f^n$: the set of string constants of library n being used in the body of native function f. At the analysis level, this is expressed as relations `NativeMethodTypeCandidate` and `NativeNameCandidate`, which relate string constants that match method names and type descriptors (e.g., `"(Ljava/lang/Object;Ljava/lang/Object;)I"` in Figure 5) to a library filename and native function identifier. The final step is a Datalog query (shown in Figure 8) that matches method names and descriptors appearing in the same native function.

4.3.1 Mock Objects. Finding methods callable from native code does not immediately imply their full treatment in the static analysis. To fully statically model the effects of the native-to-Java call-back, we need to provide appropriate values to the call's parameteres, including the implicit "`this`" receiver passed to non-static methods. We create "mock" objects, i.e., appropriate artificial objects, for such arguments, following a policy of single-object-per-type-and-call. This is illustrated in the rules of Figure 9.

The first rule in Figure 9 is used to create a mock id for every argument and its type of every reachable method by joining the predicate `PossibleNativeCodeTargetMethod(method, function, file)` and the predicates `FormalParam(_, method, frm)` and `Var_Type(frm, frmType)`. The second rule has a similar behavior, creating a mock object for the receiver ("`this`") of Java methods found in native code.

4.3.2 Constructor Filtering. Constructors are regular Java methods that are called when new objects are created. Native allocations are common in native code [38], so constructors are often called by native libraries. Since all constructors have the same low-level name (`<init>`), they only differ in signature and thus our technique may become too imprecise, lacking this core piece of distinguishing information. To address this issue, we employ a heuristic: we only accept

```
// Native allocations detected by the native scanner.
.decl NativeAllocation(constructor:Method, function:symbol, file:symbol, type:ReferenceType)

NativeAllocation(constructor, function, file, type) :-
  HighlyProbableNativeCodeTargetMethod(constructor, function, file),
  ClassConstructor(constructor, type).

// Native allocations trigger finalization code in the Android runtime. See:
// https://android.googlesource.com/platform/libcore/+/master/luni/src/main/java/java/lang/ref/
    FinalizerReference.java
ReachableMethodFromNativeCode(finalizerAdd) :-
  NativeAllocation(_, _, _, _),
  finalizerAdd = "<java.lang.ref.FinalizerReference: void add(java.lang.Object)>",
  isMethod(finalizerAdd).
```

Fig. 11. Native allocations.

```
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) {
  if (!registerNativeMethods(env, classPathName, method_table,
  sizeof(method_table) / sizeof(method_table[0]))) {
    return JNI_FALSE;
  }

  return JNI_TRUE;
}
```

Fig. 12. Example use of JNI function `RegisterNatives()`.

calls to Java constructors from native code, if instances of the constructed type are used. We assume that *type* is used if an instance method is already reachable or if an instance field is accessed (Figure 10). This heuristic is successful in practice, since native code constructs objects that will be passed to the Java program to be actually used there. The result of this filtering step, which runs in mutually recursive fashion with the rest of the analysis, is predicate `HighlyProbableNativeCodeTargetMethod(method, function, file)`.

4.3.3 Marking Native Allocations. Native allocations are easy to spot: creating an object in native code needs the native code to call a Java constructor method, so we can record such allocations in predicate `NativeAllocation(constructor, function, file, type)`, where a constructor method is called from a function in a library file, to construct some type (Figure 11). As a detail of the Java implementation, we also model the reachability of a method in the library class `java.lang.ref.FinalizerReference`.

4.4 Step EDGE

Knowing the native function that calls a Java method m is useful but we would like to also know if this function implements some Java method m' marked with the `native` keyword. If that is the case, then we should inform the pure-Java call graph and build an edge from m' to m. Step EDGE takes care of this computation by observing how JNI links native code functions to Java native method entry points.

There are essentially two approaches to this linking: (a) the automatic one, where the default JNI behavior automatically connects native and Java code, and (b) the programmable one, where the JNI code itself configures these entry points.

4.4.1 Automatic Linking. The automatic linking is easy to handle (and model as logic): the JNI specification follows a default naming convention that links a Java method with this declaration:

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

with function `Java_x_y_C_meth`, possibly with a signature suffix in the case of method overloading.

4.4.2 Configurable Linking. The JNI specification offers a method `RegisterNatives()` that allows code to programmatically link native methods against Java "native" method entry points [29, Section 8.3]. This functionality can be useful for overriding the default JNI naming convention for native functions as shown in Figure 12 (code taken from the Android-Terminal-Emulator Android application[5]). Another use of JNI programmable linking is to allow for relinking different native methods to the same Java method.

We observe that some JNI uses of `RegisterNatives()` depend on strings and structures defined outside functions, which go into a different section in the binary code (and gathered in step FILT).

On Android, applications may have to resort to a custom linker ("crazy linker"[6]) to allow for flexibility in finding native code and portability across different Android versions. In particular, the default Web browser on Android (Chrome) and its assorted system framework (WebKit[7]) depend on such configurable linking [19].

`RegisterNatives()` works by accepting triplets (name, signature, function), where name and signature describe the Java method and function is a pointer to the native function. Our technique can recover the data structures containing these triplets, for native code that is amenable to function boundary analysis and implicit-jump analysis.

5 DISCUSSION

This section discusses pragmatics concerning our technique's application: we show how to integrate with optimization tools used in Android development (Section 5.1), how the approach works in context-sensitive settings (Section 5.2), and the preconditions as well as variations of its applicability (Section 5.3).

5.1 Example Client Analysis: Code Optimization

An application of our technique (which we are actively exploring in a tool for wide use) is in the context of optimization of Android applications ("apps"). Android apps often use special tools (the two most common being ProGuard [23] and R8 [20]) to optimize the code that will be shipped to the "app store" and will then be downloaded by users. Two core tasks carried out by these tools are:

[5]https://github.com/jackpal/Android-Terminal-Emulator
[6]https://chromium.googlesource.com/android_tools/+/53862978424412e190e9bc40c7637a71fdd7d298/ndk/sources/android/crazy_linker/README.TXT
[7]https://developer.chrome.com/multidevice/webview/overview

(1) *Code shrinking*: Android app binaries are big by design, since they have to include any library they may use, if it is not part of the standard platform. Since smaller app sizes correlate with higher install conversion rates [41], this optimization is crucial.
(2) *Code obfuscation*: regardless of application size, Android developers often want to discourage reverse engineering of their code and thus resort to code obfuscation, even for small programs. As Wermke *et al.* found, obfuscation is performed in roughly 25% of apps in general and in 50% of the most popular apps [45]).

Both of the above program transformations cannot happen automatically for the case of native code calling Java code, since a Java-only analysis cannot find methods called from native libraries. Such missed methods will be marked as dead code and eliminated by the code shrinking phase or will be missed as entry points and renamed by the obfuscating phase; the outcome in both cases is run-time crashes due to missing methods.

Since these optimizations cannot happen automatically, both ProGuard and R8 use a configuration language that allows the developer to manually mark code that should not be eliminated or obfuscated. Maintaining these manual scripts is often a difficult and delicate process [45].

Our technique helps by automatically generating such configuration rules to be fed to the optimization tool, without the developer having to reason about the native code bundled in the application. We have implemented such a client analysis in Doop:[8] this is a reachability analysis that receives the configuration rules of an Android app and merges their effects with the results computed by the technique presented in this paper.

5.2 Context Sensitivity

Our approach fixes unsoundness in reachability metrics by finding entry points from native code. So far, these native-to-Java calls are assumed to happen in a context-insensitive way: the native code calls Java code using a hard-coded (constant) context.

Using our technique in a context-sensitive analysis requires that an existing context be propagated, which assumes the presence of call-graph information for the native code. Assuming that the native code is not hostile or stripped, Radare2 can compute this call graph and we can use the EDGE step (Section 4.4) to connect invocation contexts to native code, via this call graph, to the callback invocations to Java code. Thus, context sensitivity is fully supported for native code that is amenable to call-graph analysis.

In the case of native code that is not amenable to the above analysis, we can still compute hard-coded contexts for callbacks, without breaking the context-sensitive analysis of the rest of the code. We believe that having hard-coded contexts for *difficult programs* under context sensitivity is a reasonable best-effort approach.

5.3 Applicability

The approach described identifies entry points, where native code can invoke Java code. Importantly, the approach does not aim to recover a full native call-graph: calling patterns inside native code are not tracked. Assume the following call chain:

[8] Option `--keep-spec` in Doop 4.20.46.

m Java method
$\downarrow$
f Native function
$\downarrow$
f' Native function
$\downarrow$
m' Java method

The call from m to f is a JNI call to native code (uncovered by the EDGE step), while the call from f' to m' is a call-back (uncovered by the LOC step). For the EDGE step to find that m eventually calls m', it must be known that f calls f'. Our approach does not aim to recover such call-graph edges. However, such call-graph information can be extracted from Radare2, as long as native code is not hostile (i.e., obfuscated) or stripped. It is important to note that ignoring native call-graph edges does not affect the completeness of our technique: the Java entry point will be identified regardless.

Our approach does not depend on Android, Linux, or ELF, but works for every platform where an appropriate disassembler exists, allowing automatic discovery of referenced constant strings.

Our technique can be generalized to catch other string-based JNI functionality such as field reads or writes, to fix unsoundness in points-to analyses. Our approach can also be generalized to other programming languages that declare a foreign function interface (FFI) to native code that uses string constants for interoperability with the high-level language.

Finally, note that we do not attempt to detect the names of the enclosing classes of the callback methods. In our manual inspection, we see that method names and signatures are mainly constants in the native code. The classes or interfaces containing these methods are what is sometimes a parameter or a dynamic value (e.g., read from serialization data) and thus we choose to not rely on native code strings for type information.

Restrictions. Handling the general case of configurable linking (i.e., calls to `RegisterNatives()`) may fail, especially for stripped native code, where function pointers can go through relocations [27, Chapter 8].

Additionally, we do not handle JNI code that calls methods using dynamically-generated strings or interoperability with Java reflective method values (such as `FromReflectedMethod` [31, Chapter 4]).

Handling Uncooperative Native Code. We recap by accumulating, for reference purposes, all parts of our technique that may be affected by native code that is stripped, hostile, obfuscated, or otherwise not suitable for string x-refs analysis or call-graph analysis:

- configurable linking (Section 4.4.2)
- localization of call-back invocations (Section 4.3)
- context sensitivity (Section 5.2)

In handling these elements, our technique may lose precision but will not miss a call-back target that it would otherwise find. Notably, the string x-refs analysis (which attempts to compute which native functions use which strings) has a conservative fall-back. If a string is found to not be referenced by any function, we give it a default wildcard function, so that it will still be considered, albeit imprecisely.

```
/*
* Converts a WIN32_FIND_DATA to IFileInfo
*/
jboolean convertFindDataToFileInfo(JNIEnv *env, WIN32_FIND_DATA info, jobject fileInfo) {
  ...
  // select interesting information
  //exists
  mid = (*env)->GetMethodID(env, cls, "setExists", "(Z)V");
  if (mid == 0) return JNI_FALSE;
  (*env)->CallVoidMethod(env, fileInfo, mid, JNI_TRUE);

  // file name
  mid = (*env)->GetMethodID(env, cls, "setName", "(Ljava/lang/String;)V");
  if (mid == 0) return JNI_FALSE;
  (*env)->CallVoidMethod(env, fileInfo, mid, windowsTojstring(env, info.cFileName));

  // last modified
  mid = (*env)->GetMethodID(env, cls, "setLastModified", "(J)V");
  if (mid == 0) return JNI_FALSE;
  (*env)->CallVoidMethod(env, fileInfo, mid, fileTimeToMillis(info.ftLastWriteTime));
```

Fig. 13. Native code in AspectJ (from library *eclipse.platform.resources*).

6 EVALUATION

We evaluate our analysis on an independently-selected set of large Java programs, both desktop and Android. The programs include the subset of the XCorpus suite that contains native code (Section 6.1) and large Android applications (Section 6.2).

We integrated our analysis in Doop [5], version 4.20.46.[9] All experiments are run on a 64-bit machine with an Intel Xeon CPU E5-2667 v2 3.30GHz with 256 GB of RAM. We use the Soufflé compiler (v.1.5.1), which compiles Datalog specifications into binaries via C++ and run the resulting binaries in parallel mode using four jobs. Doop uses the Java 8 platform as implemented in Oracle JDK v1.8.0_121. All metrics are for Doop's default context-insensitive analysis.

In our experiments, we measure reachable methods from native code (that would be missed by a naive analysis) in the application, i.e., calls from native libraries bundled with the program, and not native code in the JDK or Android platform. The platform libraries certainly also contain native code, but since they are the same for all applications their behavior can be modeled explicitly. For instance, Doop already contains explicit models for a variety of JDK native methods, for the Android app lifecycle, as well as for frameworks involving native code, such as reflection [36] and `invokedynamic` [16].

6.1 XCorpus

XCorpus is a suite of executable Java programs containing features that are difficult to analyze by current tools [11]. We focus on the four benchmarks that the "feature analysis" tool of XCorpus reports as having native code [11, Table 3]: *aspectj-1.6.9*, *log4j-1.2.16*, *lucene-4.3.0*, and *tomcat-7.0.2*. We manually inspected the sources of each benchmark:[10]

- *aspectj* uses native code for filesystem functionality on Windows (example code: Figure 13).
- *log4j* contains native code that does not call back to Java code.
- *lucene*: on POSIX-style systems, native code constructs and returns a file descriptor object (example code: Figure 14).

[9] Our technique is freely available as (a) a front end (https://github.com/plast-lab/native-scanner) that extracts appropriate information from Java programs and (b) Datalog rules in the Doop framework (https://bitbucket.org/yanniss/doop).
[10] Available in https://bitbucket.org/gfour/xcorpus-native-extension.

```
/*
 * Class:     org_apache_lucene_store_NativePosixUtil
 * Method:    open_direct
 * Signature: (Ljava/lang/String;Z)Ljava/io/FileDescriptor;
 */
extern "C"
JNIEXPORT jobject JNICALL Java_org_apache_lucene_store_NativePosixUtil_open_1direct(JNIEnv *env,
    jclass _ignore, jstring filename, jboolean readOnly)
{ ...
  class_fdesc = env->FindClass("java/io/FileDescriptor");  ...
  // construct a new FileDescriptor
  const_fdesc = env->GetMethodID(class_fdesc, "<init>", "()V");  ...
  ret = env->NewObject(class_fdesc, const_fdesc);

  // poke the "fd" field with the file descriptor
  field_fd = env->GetFieldID(class_fdesc, "fd", "I");  ...
  env->SetIntField(ret, field_fd, fd);

  // and return it
  return ret;
}
```

Fig. 14. Native allocation in Lucene.

```
#define TCN_IMPLEMENT_CALL(RT, CL, FN)   JNIEXPORT RT JNICALL Java_org_apache_tomcat_jni_##CL##_##FN

TCN_IMPLEMENT_CALL(jlong, Pool, cleanupRegister)(TCN_STDARGS, jlong pool, jobject obj)
{ ...
  cls = (*e)->GetObjectClass(e, obj);
  cb->obj    = (*e)->NewGlobalRef(e, obj);
  cb->mid[0] = (*e)->GetMethodID(e, cls, "callback", "()I");

  apr_pool_cleanup_register(p, (const void *)cb, generic_pool_cleanup, apr_pool_cleanup_null);  ...
}

TCN_IMPLEMENT_CALL(jlong, SSL, newBIO)(TCN_STDARGS, jlong pool, jobject callback)
{ ...
  j = (BIO_JAVA *)BIO_get_data(bio); ...
  cls = (*e)->GetObjectClass(e, callback);
  j->cb.mid[0] = (*e)->GetMethodID(e, cls, "write", "([B)I");
  j->cb.mid[1] = (*e)->GetMethodID(e, cls, "read",  "([B)I");
  j->cb.mid[2] = (*e)->GetMethodID(e, cls, "puts",  "(Ljava/lang/String;)I");
  j->cb.mid[3] = (*e)->GetMethodID(e, cls, "gets",  "(I)Ljava/lang/String;");
  j->cb.obj    = (*e)->NewGlobalRef(e, callback);  ...
}
```

Fig. 15. Native code in Tomcat.

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→12729: 2.54%	108→291: 169.4%	55→56: 1.8%	295
tomcat-7.0.2	19661 (273)	1203→2088: 73.57%	59→218: 269.4%	61→95: 55.7%	308

Fig. 16. XCorpus benchmarks.

- *tomcat* uses a native library for performance [15] (example code: Figure 15).

Benchmark	App meths (native)	Base recall	Recall	+App-reachable	+Analysis time	+Factgen time	+Entry points
Chrome	37898 (1531)	7/83 = 8.43%	83/83 = 100.00%	17003→24060: 41.50%	469→505: 7.7%	46→255: 454.4%	4484
Instagram	43420 (348)	1/7 = 14.29%	7/7 = 100.00%	23921→32425: 35.55%	473→625: 32.1%	51→63: 23.5%	4669

Fig. 17. Android apps.

We found that our technique captures all call-back targets from the native code. Figure 16 shows the results of analyzing these four XCorpus benchmarks. For every program, we calculate the total number of application methods (*App methods*) and the increase in their reachability by our technique (*+App-reachable*). This increase is due to our scanning introducing a number of candidate Java methods as call-backs (*+Entry points*), which are added to the rest of the analysis as additional entry points for further analysis. We also measure the impact of our technique on analysis time (*+Analysis time*) and on fact generation time (*+Factgen time*). That last metric, fact generation time, includes the initial processing of the native library code (steps PRE, FILT, and LOC, described in Section 4).

Running times are for a single run, hence noisy. (A 5% variation between runs is common, in our experience.) Establishing statistically significant precision in running times is, however, far from the point: we intend to observe potential order-of-magnitude differences, not small variations. As seen, the extra processing is usually cheap, unless the program has a lot of binary code; in that case, it is close to the initial processing step for Java bytecode (and these two steps can happen in parallel, as they reason about different parts of the application).

We also see that our technique does not introduce noise in the benchmark where no native call-backs are to be found (*log4j*).

6.2 Android Applications

We evaluate (Figure 17) our technique on the Android apps from the benchmark suite of the HeapDL tool [21], which produces snapshots of the dynamic activity of Java applications. HeapDL is integrated with Doop and its published benchmarks[11] contain popular and complex Android apps (`Chrome`, `Instagram`, `Google Translate`, `pinterest`, `S PhotoEditor`, `androidterm`). We analyzed the dynamic activity logs of these apps and two of them (`chrome` and `Instagram`) exhibit call backs from native code to Java code. (It is likely that the rest of the apps also perform such call backs, but they are not exercised in the HeapDL dynamic executions.)

Our technique exhibits a 100% recall rate over the observed dynamic call-backs from native to Java code. This includes covering the full set of 83 dynamically-observed entry points in `Chrome`. As a result of adding these entry points, a much larger part of the application code becomes analysis-reachable. Accordingly, we also see a big increase in the fact generation time for Chrome: this is due to our Radare2 back-end requiring significant time to parse and analyze the bundled native library.

7 RELATED WORK

Our work connects declarative static analysis with information coming from a reverse engineering front end (Radare2), running on native code. Another declarative analysis tool working on native code is *ddisasm*, developed by GrammaTech, Inc. [13]. This framework also uses Datalog and achieves a high-level of accuracy, being capable to correctly reassemble its disassembled output. Our choice of Radare2 over ddisasm as a front end was pragmatic: ddisasm only supports

[11] https://bitbucket.org/yanniss/doop-benchmarks

the x86-64 platform,[12] while we require mature analysis on x86 32-bit code and the Android ARM targets (arm64 and armeabi [18]).

Redex is an open source Android bytecode optimizer developed by Facebook [25]. To improve performance and efficiency of Android apps, Redex applies optimizations such as dead code elimination, inlining, and minification, and removing unnecessary metadata. Redex also scans native libraries to find class names,[13] although it does not go as far as our technique to discover method call-backs and native allocations.

Although well-designed to balance security checks while allowing for low-level performance, the JNI is still a source of vulnerabilities [22, 28, 37, 40] due to native code being opaque and thus less amenable to static analysis. This has led in practice to a muiltitude of JNI sandboxes (some even in hardware) to contain the effects of native code running alongside Java code [6, 39].

Recently, Wei *et al.* showed how to do cross-language dataflow-based security analysis on Android applications that contain native code [43]. Similar to our approach, they model parts of the JNI API and use a reverse engineering front end (*angr* [35]). Their native code modeling is based on symbolic execution, as opposed to static recognition of JNI strings. This is a good fit for the intended goal of the approach: getting a precise model of native behavior, for security analysis purposes [44]. In contrast, our technique emphasizes more complete analysis, integrating generally with all existing Doop analyses, yet without a precise model of information flow through native code.

Furr and Foster also analyzed JNI strings in binaries for a different reason: type inference over the JNI boundary [17]. They also worked on the level of C sources, not compiled binaries. However, they "found that simple tracking of strings is sufficient", which is also a core idea of our approach.

Our analysis can be combined with other JNI-based analyses such as the security analysis of Li and Tan [28].

8 CONCLUSION

We showed that simple declarative analysis logic can be coupled with filtering logic to work on fixing false negatives in static reachability analysis. Our technique also uses an existing reverse engineering front end that does string x-refs analysis for added precision and reconstruction of programmable linking configurations.

Our technique is lightweight but can be a starting point for the incorporation of heavier static analyses on native code [3, 4, 34, 35]. Such tools (plus additional string analysis [7]) can offer more insight into specific uses of the JNI specification, such as handling JNI uses with on-the-fly construction of strings.

ACKNOWLEDGMENTS

We gratefully acknowledge support by the Hellenic Foundation for Research and Innovation, grant Dean-Block.

REFERENCES

[1] Sumaya Almanee, Mathias Payer, and Joshua Garcia. 2019. Too Quiet in the Library: A Study of Native Third-Party Libraries in Android. arXiv:cs.CR/1911.09716

[2] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 583–600.

[3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Compiler Construction*, Rastislav Bodik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–254.

[4] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–104.

[12] https://github.com/GrammaTech/ddisasm/issues/2

[13] https://github.com/facebook/redex/blob/master/libredex/RedexResources.cpp

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[6] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, and et al. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 569–583. https://doi.org/10.1145/3037697.3037725

[7] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. 2005. String Analysis for x86 Binaries. (2005), 88–95. https://doi.org/10.1145/1108792.1108814

[8] DWARF Standards Committee. [n. d.]. The DWARF Debugging Standard. http://dwarfstd.org/.

[9] Radare2 Contributors. 2020. ESIL - Radare2 Book. https://radare.gitbooks.io/radare2book/disassembling/esil.html.

[10] WALA Developers. 2019. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[11] Jens Dietrich, Henrik Schole, Li Sui, and Ewan D. Tempero. 2017. XCorpus - An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (2017), 1:1–24. https://doi.org/10.5381/jot.2017.16.4.a1

[12] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. 2017. On the Construction of Soundness Oracles. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3088515.3088520

[13] Antonio Flores-Montoya and Eric M. Schulte. 2019. Datalog Disassembly. *CoRR* abs/1906.03969 (2019). arXiv:1906.03969 http://arxiv.org/abs/1906.03969

[14] Free Software Foundation. 2017. GNU Binutils. https://www.gnu.org/software/binutils/.

[15] The Apache Software Foundation. 2019. Apache Tomcat Native Library - Documentation Index. http://tomcat.apache.org/native-doc/.

[16] George Fourtounis and Yannis Smaragdakis. 2019. Deep Static Modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs)*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 15:1–15:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.15

[17] Michael Furr and Jeffrey S. Foster. 2006. Polymorphic Type Inference for the JNI. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–324.

[18] Google. 2020. Android ABIs - Android NDK - Android Developers. https://developer.android.com/ndk/guides/abis.

[19] Google. 2020. Shared Libraries on Android. https://chromium.googlesource.com/chromium/src/+/master/docs/android_native_libraries.md.

[20] Google. 2020. Shrink, obfuscate, and optimize your app | Android Developers. https://developer.android.com/studio/build/shrink-code.

[21] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 68 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133892

[22] Y. Gu, K. Sun, P. Su, Q. Li, Y. Lu, L. Ying, and D. Feng. 2017. JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 427–438. https://doi.org/10.1109/DSN.2017.40

[23] Guardsquare. 2020. ProGuard - Official website - Java and Android Apps optimizer. https://www.guardsquare.com/en/products/proguard.

[24] Chua Hock-Chuan. 2018. Java Native Interface Tutorial. https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html.

[25] Facebook Inc. 2019. Redex - An Android Bytecode Optimizer. https://fbredex.com/.

[26] 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 2019)*. Association for Computing Machinery, New York, NY, USA, 448–451. https://doi.org/10.1145/3293882.3338990

[27] John R. Levine. 1999. *Linkers and Loaders* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] 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 (CCS '09)*. Association for Computing Machinery, New York, NY, USA, 442–452. https://doi.org/10.1145/1653662.1653716

[29] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Specification* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[30] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[31] Oracle. 2020. Java Native Interface Specification Contents. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[32] OSDev.org. 2019. ELF (Executable and Linkable Format). https://wiki.osdev.org/ELF.

[33] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

[34] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[36] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 485–503. https://doi.org/10.1007/978-3-319-26529-2_26

[37] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.

[38] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. To appear in ICSE 2020.

[39] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting Android Applications from Third-Party Native Libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2627393.2627396

[40] Gang Tan, Srimat Chakradhar, Raghunathan Srivaths, and Ravi Daniel Wang. 2006. Safe Java native interface. In *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*. 97–106.

[41] Sam Tolomei. 2017. Shrinking APKs, growing installs – How your app's APK size impacts install conversion rates. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2. (Nov. 2017).

[42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13.

[43] 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 (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1137–1150. https://doi.org/10.1145/3243734.3243835

[44] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security* 21, 3, Article 14 (April 2018), 32 pages. https://doi.org/10.1145/3183575

[45] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 222–235. https://doi.org/10.1145/3274694.3274726