NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Scanning native binaries to resolve unsoundness in static analysis of mixed Java-native code

Leonidas S. Triantafyllou

Supervisors: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Research Associate NKUA

ATHENS

OCTOBER 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σάρωση δυαδικών αρχείων για πληρέστερη στατική ανάλυση μικτού Java-native κώδικα

Λεωνίδας Σ. Τριανταφύλλου

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Ερευνητικός Συνεργάτης ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2019

UNDERGRADUATE THESIS

Scanning native binaries to resolve unsoundness in static analysis of mixed Java-native code

Leonidas S. Triantafyllou
S.N.: 1115201400202

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Research Associate NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σάρωση δυαδικών αρχείων για πληρέστερη στατική ανάλυση μικτού Java-native κώδικα

Λεωνίδας Σ. Τριανταφύλλου
Α.Μ.: 1115201400202

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Ερευνητικός Συνεργάτης ΕΚΠΑ

ABSTRACT

Many real-world applications contain native code written in C and/or C++, which interacts with Java code. Analyzing native files, it is possible to estimate how Java code is called by native code and furthermore resolve the unsoundness in static analyses of such applications.

We present an analysis that finds Java method calls in native code by scanning disassembled binary files of native code. The main challenge in analyzing these binary files is the difficulty of finding function boundaries in order to determine which native function calls which Java method of the same application.

The analysis was written in the Java and Datalog languages and is based on the Doop framework. Specifically, the implementation of a Java utility for scanning binary files combined with a specific analysis logic in Datalog demonstrates `Doop`'s capabilities in creating concise and expressive static analyses.

SUBJECT AREA: Static program analysis

KEYWORDS: static program analysis, doop framework, java native interface, java, datalog

ΠΕΡΙΛΗΨΗ

Πολλές εφαρμογές πραγματικού κόσμου σε Java περιέχουν εγγενή κώδικα γραμμένο σε C και/ή C++, ο οποίος αλληλεπιδρά με τον κώδικα Java. Αν αναλύσουμε τα εγγενή αρχεία, είναι δυνατόν να υπολογίσουμε τον τρόπο με τον οποίο ο κώδικας Java καλείται από τον εγγενή κώδικα και επιπλέον να επιλύσουμε την αβεβαιότητα στις στατικές αναλύσεις τέτοιων εφαρμογών.

Παρουσιάζουμε μια ανάλυση η οποία βρίσκει κλήσεις μεθόδων Java σε εγγενή κώδικα με σάρωση αποσυναρμολογημένων δυαδικών αρχείων εγγενούς κώδικα. Η κύρια πρόκληση στην ανάλυση αυτών των δυαδικών αρχείων είναι η δυσκολία εύρεσης ορίων μεθόδων προκειμένου να βρεθεί ποια εγγενής μέθοδος καλεί ποιες μεθόδους Java της ίδιας εφαρμογής.

Η ανάλυση γράφτηκε σε γλώσσα Java και Datalog και βασίζεται στο Doop framework. Συγκεκριμένα, η υλοποίηση ενός Java utility για τη σάρωση δυαδικών αρχείων σε συνδυασμό με μια συγκεκριμένη λογική ανάλυσης σε Datalog επιδεικνύει τις δυνατότητες του Doop στη δημιουργία περιεκτικών και εκφραστικών στατικών αναλύσεων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, doop framework, java native interface, java, datalog

To my family.

ACKNOWLEDGMENTS

I would like to express my gratitude to prof. Yannis Smaragdakis for suggesting the idea of my thesis and for giving me the chance to work on such an interesting topic.

I would also like to thank both my supervisors, Yannis Smaragdakis and George Fourtounis for their motivation and their continuous support by sharing their expertise on the subject and providing help and suggestions throughout this project.

October 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis aims to make an assessment of how Java code is called by native code of the same application in order to resolve unsoundness in the static analysis of such applications. It was developed as my undergraduate thesis between March 2019 and October 2019 at the Department of Informatics and Telecommunications of the University of Athens.

1. INTRODUCTION

Pointer Analysis or Points-to Analysis [1] is a fundamental static program analysis. The objective of this analysis is to compute an approximation of the set of program objects referred to by a pointer variable or expression. Doop [2] is a declarative framework for static analysis that performs Pointer Analysis for Java programs, using the Datalog language.

Many Android applications cannot be implemented entirely in Java. Sometimes it is necessary to handle different platforms and support their features or use existing non-Java libraries without rewriting them. There are also situations in which the performance of an Android application is improved by using code developed in a different programming language that allows to overcome specific Java constraints. As a result, there are parts of Java applications that need to be programmed in other high-level languages (HLL), most likely in C and/or C++, and are included in binaries called **native libraries**.

In this thesis, we develop a Java utility in order to find Java method calls in disassembled native binaries of applications and to estimate how Java code is called by native code. This estimation is important for making the analysis of Doop framework more complete.

The rest of the thesis is organized as follows:

1. In Chapter 2 we (a) give an overview of the development of Android applications containing native code and (b) introduce the reader to pointer analysis in the Doop framework.
2. In Chapter 3 we describe the process of finding Java method calls in native libraries using the Doop framework.
3. In Chapter 4 we present our experimental evaluation.
4. In Chapter 5 we present other important tools related to scanning native libraries.
5. In Chapter 6 we summarize our conclusions.

2. BACKGROUND

This chapter gives an overview of (a) developing Android Java applications containing native code (Section 2.1) and (b) declarative points-to analysis in the Doop framework (Section 2.2).

2.1 Native Code Development in Android Java Applications

This section shows how Java applications can bundle native code on Android. We describe three core technologies: the Java Native Interface, the operating system ABI, and the Android Native Development Kit.

2.1.1 Java Native Interface

The Java Native Interface (JNI) [3] 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). 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 their information.

In order to write a native library for an application, programmers need to be familiar with JNI. First of all, there is a specific formula for defining native code functions which is shown in Figure 1. When a Java method calls a native code function, in addition to Java variable arguments, it also has to pass two other arguments, a `JNIEnv` pointer for a reference to **JNI** environment and a `jobject` for a reference to “this” Java object [4]. The `JNIEnv` type is a pointer 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.

Naming Convention

`Java_{package_and_classname}_{function_name}(JNI_arguments).`

`The dot in package name is replaced by underscore.`

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

Figure 1: ”Hello World” Native Function Example.

In JNI there are equivalent types that correspond to Java types. For example, for the

Java primitive types `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, JNI defines `jboolean`, `jbyte`, `jchar`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble`, and `jchar` respectively. Other basic types are `jstring` for `java.lang.String`, `jobject` for `java.lang.Object` and `jclass` for `java.lang.Class`. In addition, JNI also supports Java arrays using the `jarray` type and there are nine type of arrays which are `jbooleanArray`, `jbyteArray`, `jcharArray`, `jshortArray`, `jintArray`, `jlongArray`, `jfloatArray`, `jdoubleArray`, and an array of objects `jobjectArray`.

The communication between Java and native code can be implemented in a hassle-free way, as the JNI interface provides all the necessary functions. When a native function is called by the Java code, the arguments that are passed to it are in JNI types. JNI types are converted or copied to local native types, which are used from the native functions. The returned value at the end of a function's execution is of a JNI type.

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 specify its name and signature. The name is the Java method's name, whereas the signature is a string of the form **(parameters)return-value** of the Java method.

Table 1: Java (JNI) and native code type mapping.

C type	Java type	Type signature
unsigned char	jboolean	Z
signed char	jbyte	B
unsigned short	jchar	C
short	jshort	S
int	jint	I
long	jlong	J
float	jfloat	F
double	jdouble	D
void	void	V

Taking into consideration the type signatures, it is possible to create method signatures. Some examples of methods and their signatures are given in Table 2:

Table 2: Java Method Signatures Examples.

Method	Signature
void method1()	()V
int method2(long)	(J)I
void method3(String)	(Ljava/lang/String;)V
String method4(String, int[])	(Ljava/lang/String;[I)Ljava/lang/String;

The process of calling back a method starts by getting a reference to the object's class by using method `GetObjectClass()` [4]. 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, which are `CallVoidMethod()`, `Call<Primitive-type>Method()` and `CallObjectMethod()`. As for the type of the returned value of the called method, it is void, <Primitive-type> and Object, respectively. An example of the process for calling a simple Java method that takes no arguments and returns no value through native code is shown in Figure 2.

```
JNIEXPORT void JNICALL Java_JNIExample_callBack(JNIEnv *env, jobject thisObj) {
   jclass cls = (*env)->FindClass(env, "JNIExample");
   jmethodID helloWorldMethod = (*env)->GetMethodID(env, cls, "helloWorld",
      "()V");
   jint i = (*env)->CallIntMethod(env, obj, helloWorldMethod, obj, obj);
   printf("callBack(): i = %d\n", i);
}
```

Figure 2: Call back Java method from native function example.

2.1.2 ABI Management

Android is an operating system that is designed primarily for handheld devices, such as smartphones. To run an application on such a device, it is necessary to define an interface that determines how the application code will interact with the device. This interface in the case of Android is called the Application Binary Interface (ABI). Android supports many different instruction sets because of the different CPUs it is able to run on, resulting in the existence of many different ABIs. An ABI contains information about the supported information, such as the CPU instruction set, the endianness of memory, the format of binaries, how system and code pass data, how system manages registers and stack in function calls and the list of function symbols of the machine code. Examples of ABIs in Android are armeabi, armeabi-v7, arm64-v8a, x86, and x86-64.

2.1.3 Android Native Development Kit

The Android Native Development Kit (NDK) [3] is a set of tools enabling programmers to add C and C++ code in Android applications. This type of code is called native code and its existence is useful for Android. The NDK can be used to build native static libraries and dynamic or shared libraries [5]. Native code included in the application can interact with the Java code of the same application using the Java Native Interface (JNI). Moreover, at the time application is running, the Application Binary Interface (ABI) is responsible for determining how the program in machine language instructions will be executed by the CPU of the system. The NDK supports several architectures, such as x86, x86-64, 32-bit ARM and AArch64 to name a few.

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 for Datalog [2], which provides a rich set of points-to analyses (e.g. context insensitive, call-site sensitive, object sensitive). 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 runtime).

Doop's defining feature is the use of Datalog for its analyses and its explicit representation of relations as tables of a database (i.e., all tuples of a relation are represented as an explicit table), instead of using Binary Decision Diagrams(BDDs), which have been considered necessary for scalable points-to analysis in the past [6, 7].

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [8, 9] and for high-level [10, 11] analyses. The essence of Datalog is its ability to define recursive relations. Mutual recursion is the source of all complexity in program analysis. For a standard example, the logic for computing a call-graph depends on having points-to information for pointer expressions, which, in turn, requires a call-graph. Such recursive definitions are common in points-to analysis.

Soot [12] 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 is important because it enables the use of libraries whose source code is not public. 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.

Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

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

Figure 3: Simple Datalog example for IDB rules.

The Datalog code of the example in Figure 3 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(?heap, ?from) are both true for some values of ?from, ?to and ?heap, then that VarPointsTo(?heap, ?to) is also true.

3. NATIVE SCANNER

Static disassembly is a problem that has not been solved completely yet [13]. It is a difficult process because its capabilities and limitations are not always defined in a clear way or are not fully understood by the researchers. Furthermore, binary files may contain complex constructs, such as overlapping code and inline data in executable regions. For example, function boundaries is a construct that cannot be recovered accurately.

In this chapter we describe the tools we use in our analysis and the heuristic techniques of the process we follow in order to find Java method calls in the native code of the same application. As we have already stated there are many different ABIs in Android (e.g. armeabi, armeabi-v7, arm64-v8a, x86 and x86_64), so we use different disassembling tools as the set of features provided by each tool differs for different ABIs. In our analysis, we use different disassemblers to examine native libraries of the following architectures: x86, x86_64, aarch64, armeabi and armeabi-v7a.

3.1 Packed Files in APK Files

Many APK files also offer support for packed files, which can contain native code. To be more specific, files with extensions .xzs and .zstd that include native library code can be decompressed and this code can also be scanned. Our analysis is able to analyze these files in search for Java method calls. The commands to decompress .xzs (need to change extension to .xz first) and .zstd files are shown in Figure 4 and Figure 5 respectively.

```
xz --decompress <xz_file>
```

Figure 4: Command to decompress .xz files.

```
zstd -d -o <zstd_file>
```

Figure 5: Command to decompress .zstd files.

3.2 HelloJNI Example

In order to find possible Java method calls in the native code of applications, we started our analysis by disassembling a small modified example from an online tutorial [4] and then we tested our utility in real-world applications. The example's source code is found in Figure 6 and Figure 7. By using a Toolchain provided with the Android NDK independently, we were able to compile the C code and create native libraries in different architectures.

```java
public class HelloJNI {
   static {
      String libName = "libhello.so";
      URL url = HelloJNI.class.getResource("/" + libName);
      try {
         File tmpDir = Files.createTempDirectory("libhello").toFile();
         tmpDir.deleteOnExit();
         File libTmpFile = new File(tmpDir, libName);
         libTmpFile.deleteOnExit();
         try (InputStream in = url.openStream()) {
            Files.copy(in, libTmpFile.toPath());
            System.load(libTmpFile.getCanonicalPath());
         }
      } catch (IOException ex) {
         System.err.println("Could not load " + libName);
      }
   }

   // 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());

      File f = new File(".");
      FileSystem ufs = FileSystems.getDefault();
      System.out.println("ufs.getClass() = " + ufs.getClass().getName());
      String[] list = f.list();
      System.out.println("== Contents of current dir ==");
      for (String entry : list) {
         System.out.println(entry);
      }
      hj.callBack(new Object());
   }

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

Figure 6: Code of HelloJNI.java example file.

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

// Test method to inspect parts of the jobject data structure. This
// only examines the first two elements pointed to by <obj>, plus the
// first two elements pointed to in each case. We assume these pairs
// usually exist
// (https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.7).
void inspect_jobject(jobject obj) {
    printf("obj = %p\n", obj);
    int i, j;
    for (i = 0; i < 2; i++) {
        printf("obj[%d] = %p\n", i, (void*)((long*)obj)[i]);
        for (j = 0; j < 2; j++)
            printf("obj[%d][%d] = %p\n", i, j, (void*)((long**)obj)[i][j]);
    }
}

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

JNIEXPORT jobject JNICALL Java_HelloJNI_newJNIObj(JNIEnv *env, jobject thisObj) {
    inspect_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) {
    inspect_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);
}
```

Figure 7: Code of HelloJNI.c example file.

3.3 Executable and Linkable Format

Executable and Linkable Format (ELF) [14] is a file format for binaries, libraries, and core files. The structure of an ELF file consists of two parts, the ELF header and the FILE data. The ELF header contains valuable information about the file, whereas the FILE data consists of the program headers or segments, the section headers or sections and the data of the file. We are interested in the sections of the ELF file because in our analysis we search for constant strings in the `.rodata` section of the FILE data [5].

A useful command which displays information about ELF format object files is the **readelf** command [15]. This program can be used like `objdump` but its output is more detailed. By using `readelf`, we can see the sections of the ELF file and information about every section (size of section, offset align, etc).

```
There are 24 section headers, starting at offset 0x16d0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.android.ide NOTE            00000134 000134 000098 00 A    0   0  4
  [ 2] .dynsym           DYNSYM          000001cc 0001cc 0000c0 10 A    3   1  4
  [ 3] .dynstr           STRTAB          0000028c 00028c 0000cc 00 A    0   0  1
  [ 4] .hash             HASH            00000358 000358 000044 04 A    2   0  4
  [ 5] .gnu.version      VERSYM          0000039c 00039c 000018 02 A    2   0  2
  [ 6] .gnu.version_d    VERDEF          000003b4 0003b4 00001c 00 A    3   1  4
  [ 7] .gnu.version_r    VERNEED         000003d0 0003d0 000020 00 A    3   1  4
  [ 8] .rel.dyn          REL             000003f0 0003f0 000018 08 A    2   0  4
  [ 9] .rel.plt          REL             00000408 000408 000028 08 A    2   0  4
  [10] .plt              PROGBITS        00000430 000430 000050 00 AX   0   0  4
  [11] .text             PROGBITS        00000480 000480 000368 00 AX   0   0  4
  [12] .ARM.exidx        ARM_EXIDX       000007e8 0007e8 000008 08 AL  11   0  4
  [13] .rodata           PROGBITS        000007f0 0007f0 000098 01 AMS  0   0  1
  [14] .fini_array       FINI_ARRAY      00001f18 000f18 000008 00 WA   0   0  4
  [15] .dynamic          DYNAMIC         00001f20 000f20 0000e0 08 WA   3   0  4
  [16] .data             PROGBITS        00002000 001000 000004 00 WA   0   0  4
  [17] .got              PROGBITS        00002004 001004 000020 00 WA   0   0  4
  [18] .comment          PROGBITS        00000000 001024 000108 01 MS   0   0  1
  [19] .note.gnu.gold-ve NOTE            00000000 00112c 00001c 00      0   0  4
  [20] .ARM.attributes   ARM_ATTRIBUTES  00000000 001148 000036 00      0   0  1
  [21] .symtab           SYMTAB          00000000 001180 0002b0 10     22  32  4
  [22] .strtab           STRTAB          00000000 001430 0001b5 00      0   0  1
  [23] .shstrtab         STRTAB          00000000 0015e5 0000eb 00      0   0  1
```

Figure 8: Sections of armeabi libhello.so.

In Figure 8, the starting address and the size of the `.rodata` section can be seen. We can also use the `readelf` command to dump the contents of a specific section as strings. By dumping the `.rodata` section, we are able to find the offset of every constant string used in the native library and furthermore calculate its address by adding its offset to the starting address of the `.rodata` section (Figure 9). The result of dumping the `.rodata` section of our example is represented in Figure 10.

```
readelf -p .rodata libhello.so
```

Figure 9: Command to dump contents of the `.rodata` section as strings.

```
String dump of section '.rodata':
  [     0]  obj = %p^J
  [     a]  obj[%d] = %p^J
  [    18]  obj[%d][%d] = %p^J
  [    2a]  Hello World!^J
  [    38]  HelloJNI
  [    41]  <init>
  [    48]  ()V
  [    4c]  helloMethod
  [    58]  (Ljava/lang/Object;Ljava/lang/Object;)I
  [    80]  callBack(): i = %d^J
```

Figure 10: Dumping contents of the `.rodata` section of armeabi libhello.so as strings.

Then, we can use the disassembling tools to analyze native libraries and search for string references in the disassembled code. We can also find in which native function the strings are located.

3.4 Disassembling Native Libraries

The point of our analysis is to examine disassembled native libraries of real-word applications in order to find constant strings that represent Java method names or Java method signatures and then find possible Java method calls in native code. Depending on the architecture of the native library we use different disassemblers. Our primary tools are the GNU Project Debugger (GDB) and the GNU tools `objdump` and `nm`.

We start the analysis with GNU `nm` which is used to list symbols from object files. It can also be used to demangle/decode low-level symbol names into user-level names. For example, in C++ and Java it is possible to declare functions with the same name, but different types of parameters (overloading). To ensure that these functions are distinguished, they are given a unique low-level assembler name to be identified (mangling). A program, such as `c++filt`, is necessary to demangle/decode these names into readable names for programmers. An example of calling `nm` can be seen in Figure 11 while the respective output is represented in Figure 12.

```
nm --demangle libhello.so
```

Figure 11: Decode (demangle) low-level symbol names into user-level names.

```
0000000000004038 b completed.7287
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000001070 t deregister_tm_clones
00000000000010e0 t __do_global_dtors_aux
0000000000003e10 t __do_global_dtors_aux_fini_array_entry
0000000000003e18 d __dso_handle
0000000000003e20 d _DYNAMIC
000000000000134c t _fini
0000000000001120 t frame_dummy
0000000000003e08 t __frame_dummy_init_array_entry
0000000000002190 r __FRAME_END__
0000000000004000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002098 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
0000000000001129 T inspect_jobject
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
000000000000129a T Java_HelloJNI_callBack
000000000000120f T Java_HelloJNI_newJNIObj
00000000000011f0 T Java_HelloJNI_sayHello
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
00000000000010a0 t register_tm_clones
0000000000004038 d __TMC_END__
```

Figure 12: Demangled symbols of x86_64 HelloJNI library.

The GNU Project debugger (GDB) is well known among programmers for the debugging and disassembling capabilities it offers. GDB enables programmers to find how their programs are executed and examine the stacktrace after a crash. Moreover, it can also be used as a disassembler giving the ability to users to see the disassembled code of their applications. GDB supports all the following languages: Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal and Rust. In our analysis, we use GNU gdb (GDB) Fedora 8.2-6.fc29 (License GPLv3+: GNU GPL version 3 or later) for disassembling x86_64 native libraries.

Another disassembler that we use for x86 and ARM native libraries is the GNU `objdump`. The `objdump` is used to display information about object files like the `readelf` command does, but it can also be used to provide the disassembled code of native libraries.

3.4.1 X86 Native Libraries

In our approach, we use the `objdump` disassembler to find string references in x86 native libraries. In order to find these references we search for `lea` instructions and furthermore for a specific pattern in the disassembled code. Figure 14 presents the disassembled code of Java_HelloJNI_sayHello function of our libhello example.

```
      objdump -j .text -d libhello.so
```

Figure 13: Objdump disassembling function of x86 native code.

```
00001228 <Java_HelloJNI_sayHello>:
    1228:  55                   push  %ebp
    1229:  89 e5                mov   %esp,%ebp
    122b:  53                   push  %ebx
    122c:  83 ec 04             sub   $0x4,%esp
    122f:  e8 45 01 00 00       call  1379 <__x86.get_pc_thunk.ax> # eax = 1234
    1234:  05 cc 2d 00 00       add   $0x2dcc,%eax # eax = 1234(eax) + 2dcc = 4000
    1239:  83 ec 0c             sub   $0xc,%esp
    123c:  8d 90 2a e0 ff ff    lea   -0x1fd6(%eax),%edx # eax(4000) - 1fd6 = 202A
    1242:  52                   push  %edx
    1243:  89 c3                mov   %eax,%ebx
    1245:  e8 16 fe ff ff       call  1060 <puts@plt>
    124a:  83 c4 10             add   $0x10,%esp
    124d:  90                   nop
    124e:  8b 5d fc             mov   -0x4(%ebp),%ebx
    1251:  c9                   leave
    1252:  c3                   ret
```

Figure 14: Disassembled function Java_HelloJNI_sayHello of x86 libhello.so.

```
Sections
Idx  Name    Size     VMA      LMA      File off Algn
...
 12 .rodata  00000094 00002000 00002000 00002000 2**2
...
 20 .got.plt 0000001c 00004000 00004000 00003000 2**2
...
```

Figure 15: Information about the `.rodata` and `.got.plt` section of x86 libhello.so.

```
String dump of section '.rodata':
  [     0]  obj = %p^J
  [     a]  obj[%d] = %p^J
  [    18]  obj[%d][%d] = %p^J
  [    2a]  Hello World!
  [    37]  HelloJNI
  [    40]  ()V
  [    44]  <init>
  [    4c]  (Ljava/lang/Object;Ljava/lang/Object;)I
  [    74]  helloMethod
  [    80]  callBack(): i = %d^J
```

Figure 16: Dumping contents of the `.rodata` section of x86 libhello.so as strings.

From the disassembled function in Figure 14, we can see that there is a pattern that is followed for the string references. The call instruction in line 6 of the code loads the address 1234 into the %eax register. This instruction is used in Position Independent Code [16] on x86 and enables access to global objects that have a fixed offset from that register as an offset. In line 7, the value 0x2dcc is added in the %eax register and the new value is 4000 which is the address of the .got.plt section as we can observe from the Figure 15. Therefore, in line 9, there is a lea instruction that subtract the value 0x1fd6 from the %eax register and the value of %eax register becomes 202a which is the address of ”Hello World!” string. This can be easily verified if we subtract from %eax register the starting address of .rodata section (15).The result of this arithmetic operation is 2a which is the offset of ”Hello World!” string from the starting address of .rodata section (16).

It is worth noting that if we compile a native library with gcc and the option `-shared` instead of `-FPIC` then the pattern we presented in the previous paragraph does not exist and the addresses of the strings of the .rodata section can be found directly in the disassembled code (Figure 17). In line 6, we can easily spot the reference to the string ”Hello World!”. The `-FPIC` option can be used to build position independent code, so that a shared library can be loaded at any address in memory.

```
1  000011e3 <Java_HelloJNI_sayHello>:
2      11e3:  55                   push  %ebp
3      11e4:  89 e5                mov   %esp,%ebp
4      11e6:  83 ec 08             sub   $0x8,%esp
5      11e9:  83 ec 0c             sub   $0xc,%esp
6      11ec:  68 2a 20 00 00       push  $0x202a   # string reference: "Hello World!"
7      11f1:  e8 fc ff ff ff       call  11f2 <Java_HelloJNI_sayHello+0xf>
8      11f6:  83 c4 10             add   $0x10,%esp
9      11f9:  90                   nop
10     11fa:  c9                   leave
11     11fb:  c3                   ret
```

Figure 17: Disassembled function Java_HelloJNI_sayHello of x86 libhello.so compiled with -shared option.

3.4.2 X86_64 Native Libraries

In our analysis, we use GDB to disassemble x86_64 native libraries. GDB allows us to disassemble whole binaries. Nevertheless, we can disassemble and examine functions in isolation, as we know the names of the functions from the execution of the GNU `nm`. For instance, we can use the argument “-ex” combined with the “disassemble” command and the name of the function we want to disassemble as can be seen in Figure 18. The output of the command is presented in Figure 19.

```
    gdb -batch -ex "disassemble Java_HelloJNI_sayHello" libhello.so
```

Figure 18: GDB disassembling function of x86_64 native code.

```
Dump of assembler code for function Java_HelloJNI_sayHello:
   0x00000000000011f0 <+0>: push %rbp
   0x00000000000011f1 <+1>: mov %rsp,%rbp
   0x00000000000011f4 <+4>: sub $0x10,%rsp
   0x00000000000011f8 <+8>: mov %rdi,-0x8(%rbp)
   0x00000000000011fc <+12>: mov %rsi,-0x10(%rbp)
   0x0000000000001200 <+16>: lea 0xe23(%rip),%rdi    # 0x202a
   0x0000000000001207 <+23>: callq 0x1030 <puts@plt>
   0x000000000000120c <+28>: nop
   0x000000000000120d <+29>: leaveq
   0x000000000000120e <+30>: retq
End of assembler dump.
```

Figure 19: Disassembled function Java_HelloJNI_sayHello of x86_64 libhello.so.

```
[Nr]   Name      Type      Address          Offset   Size             ...
...
[13]   .rodata   PROGBITS  0000000000002000 00002000 0000000000000098 ...
...
```

Figure 20: Information about the `.rodata` section of x86_64 libhello.so.

```
String dump of section '.rodata':
  [     0] obj = %p^J
  [     a] obj[%d] = %p^J
  [    18] obj[%d][%d] = %p^J
  [    2a] Hello World!
  [    37] HelloJNI
  [    40] ()V
  [    44] <init>
  [    50] (Ljava/lang/Object;Ljava/lang/Object;)I
  [    78] helloMethod
  [    84] callBack(): i = %d^J
```

Figure 21: Dumping contents of the `.rodata` section of x86_64 libhello.so as strings.

From the disassembled function in Figure 19, we can see that `lea` instructions contain the address of the constant string we are interested in. We also see that the `.rodata` section of the file starts at address 2000 (Figure 20) and by adding the offsets in the starting address we can find the referenced strings. For example, adding 2000 and the offset 2a of the “Hello World!” string (Figure 21) we have the hexadecimal value 202a which is found in a `lea` instruction of the address 1200 of the disassembled code. By repeating this procedure, we can find all the references of constant strings.

3.4.3 AArch64 Native Libraries

The same process can be followed in AArch64 native libraries. Instead of `lea` instructions though, we search for a small set of instructions, namely `adrp`, `add`, and `mov` instructions. Figure 22 shows the disassembled code of Java_HelloJNI_sayHello function.

```
Dump of assembler code for function Java_HelloJNI_sayHello:
   0x0000000000000698 <+0>: sub sp, sp, #0x20
   0x000000000000069c <+4>: stp x29, x30, [sp, #16]
   0x00000000000006a0 <+8>: add x29, sp, #0x10
   0x00000000000006a4 <+12>: adrp x8, 0x0
   0x00000000000006a8 <+16>: add x8, x8, #0x866
   0x00000000000006ac <+20>: str x0, [sp, #8]
   0x00000000000006b0 <+24>: str x1, [sp]
   0x00000000000006b4 <+28>: mov x0, x8
   0x00000000000006b8 <+32>: bl 0x510 <printf@plt>
   0x00000000000006bc <+36>: ldp x29, x30, [sp, #16]
   0x00000000000006c0 <+40>: add sp, sp, #0x20
   0x00000000000006c4 <+44>: ret
End of assembler dump.
```

Figure 22: Disassembled function Java_HelloJNI_sayHello of AArch64 libhello.so.

```
[Nr] Name    Type      Address          Offset   Size             ...
...
[10] .rodata PROGBITS 000000000000083c 0000083c 0000000000000094 ...
...
```

Figure 23: Information about the `.rodata` section of AArch64 libhello.so.

```
String dump of section '.rodata':
  [     0] obj = %p^J
  [     a] obj[%d] = %p^J
  [    18] obj[%d][%d] = %p^J
  [    2a] Hello World!^J
  [    38] HelloJNI
  [    41] <init>
  [    48] ()V
  [    4c] helloMethod
  [    58] (Ljava/lang/Object;Ljava/lang/Object;)I
  [    80] callBack(): i = %d^J
```

Figure 24: Dumping contents of the `.rodata` section of AArch64 libhello.so as strings.

From the two instructions in addresses 0x6a4 and 0x6a8 (Figure 22) we observe that there is a reference to a constant string at address x866 of the `.rodata` section. Using the `readelf` command we locate the beginning of the `.rodata` section at address 0x83c (Figure 23) and the “Hello World!” string is at 2a offset (Figure 24). The result of adding 0x83c and 2a is the address 0x866 which we also found in the disassembled code.

3.4.4 Armeabi Native Libraries

In applications which contain native code of the armeabi architecture we use the `objdump` disassembler. In our analysis, we search for a small set of instructions, such as `ldr`, `add`, and `mov` instructions. In order to get the disassembled code of the binary file we issue a command as presented in Figure 25.

```
objdump -j .text -d libhello.so
```

Figure 25: Disassembling HelloJNI native library using `objdump`.

```
1   000005e4 <Java_HelloJNI_sayHello>:
2   5e4: e92d4800  push {fp, lr}
3   5e8: e1a0b00d  mov  fp, sp
4   5ec: e24dd010  sub  sp, sp, #16
5   5f0: e50b0004  str  r0, [fp, #-4]
6   5f4: e58d1008  str  r1, [sp, #8]
7   5f8: e59d0008  ldr  r0, [sp, #8]
8   5fc: ebffff99  bl 468 <inspect_jobject@plt>
9   600: e59f0014  ldr  r0, [pc, #20] ; 61c <Java_HelloJNI_sayHello+0x38>
10  604: e08f0000  add  r0, pc, r0
11  608: ebffff93  bl 45c <printf@plt>
12  60c: e58d0004  str  r0, [sp, #4]
13  610: ebffff97  bl 474 <aaaa@plt>
14  614: e1a0d00b  mov  sp, fp
15  618: e8bd8800  pop  {fp, pc}
16  61c: 00000212  .word 0x00000212
```

Figure 26: Disassembled function java_HelloJNI_sayHello of armeabi library.

Figure 26 shows that at address 600 there is a ldr instruction that loads the word of the address 61c in r0 register. In line 10, we can see an add instruction which adds the pc (usually in armeabi the program counter is the current address plus 8) in r0 register, so the value of r0 register at address 604 of the disassembled code is 81a. By subtracting the starting address 7f0 of the `.rodata` section (Figure 8) from 81a we get the value 2a which is the offset of ”Hello World” string we search for (Figure 10).

3.4.5 Armeabiv7a Native Libraries

For armeabiv7a libraries, we follow a similar process. We use the `readelf` command to see the contents of the `.rodata` section and find the address of every constant string of the file we analyze. Then, we use `objdump` to get the disassembled code and by searching for a small set of instructions, such as load, add, and move instructions, we can find the string references. For example, in armeabiv7a an address of a contant string is the result of adding an offset to a loaded address. By looking up the contents of the `.rodata` section we can finally find which constant string is at the calculated address.

3.5 Possible Java Method Calls

In Section 3.4, we described some heuristics which help us find possible Java method names and Java method signatures. In this chapter, we describe some simple processes for checking if the combination of names and signatures found refer to methods of the Java code. If such combinations exist, then it is possible that these Java methods are called by native code.

As we have already stated it is sometimes difficult to find function boundaries. By taking this into consideration, in our approach, we have three options to find possible Java method calls. These options have different advantages and drawbacks and are the following:

- Finding Java method calls by examining the result of the command ”readelf -p .rodata” without knowing the function boundaries. This option lacks in precision because it is not possible to find the native function that each found Java method belongs in.
- Finding Java method calls by examining the result of the command ”readelf -p .rodata” without knowing the function boundaries and considering as Java method calls the Java methods that their name and signature are close enough in the `.rodata` section using an offset. This option has a better precision than the previous one, but it is not possible to find in which native function each found Java method belongs.
- Finding Java method calls utilizing the heuristics we described in Section 3.4. This option has the best precision as we find to which native function each found Java method belong, but it may miss some Java methods as it is based on heuristics.

An efficient way to find possible Java method calls is by using Datalog language. With the help of Datalog, we are able to implement simple rules in order to check if the found names and signatures refer to Java methods and the result of our analysis can be easily accessed by Doop framework.

```
.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).
```

Figure 27: Datalog rule to find possible Java method calls.

Using the Datalog rule in Figure 27 we can find possible Java methods. To be more precise, we store the Java method names and the Java method signatures with the function in which they were found as facts and can be accessed by using the rules _NativeNameCandidate(?file, ?function, ?name,_) and _NativeMethodTypeCandidate(?file, ?function, ?descriptor, _) respectively. By joining two tables of the database, we can find names and signatures of the same native code function. We also use the rule Method_SimpleName(?method, ?name) to get the simplified name of the method and join all the rules together with the Method_JVMDescriptor(?method, ?descriptor) in order to check if the Java method with the name ?method and the signature ?descriptor exists. In case we do not know the function in which we found a Java method or a Java signature we use the symbol ”-” as function in order to join these found strings and find possible Java methods that we missed.

```
PossibleNativeCodeTargetMethod(?method, ?function, ?file) :-
  _NativeMethodTypeCandidate(?file, ?function, ?descriptor, ?offset1),
  _NativeNameCandidate(?file, ?function, ?name, ?offset2),
  Method_SimpleName(?method, ?name),
  Method_JVMDescriptor(?method, ?descriptor),
  (?offset1 - ?offset2) <= V,
  (?offset1 - ?offset2) >= -V.
```

Figure 28: Datalog rule to find possible Java method calls using an offset.

The Datalog rule in Figure 28 is a bit different than the previous one and is for the second option we stated before. The only difference is that we use the offsets of the Java method names and signatures in order to find the combinations of names and signatures that are close enough and refer to Java methods. This is done by having a threshold with value V for the difference of the offsets of the names and signatures we found.

3.6 Mock Objects

In this work we present an analysis that makes use of mock objects in an attempt to make the Doop framework analyze the possible Java methods called by native code. This step is important because the arguments of the Java methods found in native libraries are not created which means that we need to create them in order to run a Doop analysis. To achieve the creation of mock objects, we did not implement new Datalog rules, but instead we used existing rules of the Doop Framework (Figure 29 and Figure 30). By using these rules we are able to create mock objects for every argument and ”this” object of the found methods in order for this methods to be included in the static analysis of Doop.

```
// Reachability for methods discovered by the native code scanner.
.decl ReachableContextFromNative(?ctx:configuration.Context, ?method:Method,
    ?function:symbol, ?file:symbol)
ReachableContextFromNative(?immCtx, ?method, ?function, ?file) :-
  basic.PossibleNativeCodeTargetMethod(?method, ?function, ?file),
  isImmutableContext(?immCtx).
```

Figure 29: Datalog rule for reachability of methods called from native code.

```
// Mock arguments for methods called from native code.
MockValueConsMacro(?mockId, ?frmType),
VarPointsTo(?immHCtx, ?mockId, ?ctx, ?frm) :-
  ReachableContextFromNative(?ctx, ?method, ?function, ?file),
  FormalParam(_, ?method, ?frm),
  _Var_Type(?frm, ?frmType),
  isImmutableHContext(?immHCtx),
  ?mockId = cat("<mock native object of type ", cat(?frmType, cat(" from ",
      cat(?file, cat(":", cat(?function, ">")))))).

// Mock 'this' for methods called from native code.
MockValueConsMacro(?mockId, ?type),
VarPointsTo(?immHCtx, ?mockId, ?ctx, ?this) :-
  ReachableContextFromNative(?ctx, ?method, ?function, ?file),
  ThisVar(?method, ?this),
  _Var_Type(?this, ?type),
  isImmutableHContext(?immHCtx),
  ?mockId = cat("<mock receiver of type ", cat(?type, cat(" from ", cat(?file,
      cat(":", cat(?function, ">")))))).
```

Figure 30: Datalog rules to mock arguments and ”this” of methods called from native code.

The above Datalog rules are necessary in order to create mock objects for the Java methods found in native libraries. The rule ReachableContextFromNative(?ctx, ?method, ?function, ?file) in Figure 29 is responsible for making every Java method reachable for the analysis of Doop. The first rule in Figure 30 is used to create a mock id for every argument and its type of every reachable method by joining the predicate ReachableContextFromNative(?ctx, ?method, ?function, ?file) and the predicates FormalParam(_, ?method, ?frm) and _Var_Type(?frm, ?frmType). The second rule has a similar behavior to the first rule as it creates a mock id for ”this” object of Java methods found in native code.

3.7 Applicability

Our approach does not depend on Android, Linux, or ELF but it works for every platform where an appropriate disassembler exists, allowing automatic discovery of referenced constant strings (this information is also known as “string cross-references” or “string x-refs”). This means that our technique also works, for instance, on Microsoft Windows or macOS. Moreover, our approach can also work on other programming languages that have a native language interface with similar principles as JNI.

A restriction of the technique described in this thesis is that we do not handle JNI code that calls methods using `FromReflectedMethod` to convert Java reflective method values [17].

4. EXPERIMENTAL EVALUATION

This Chapter presents the evaluation of the analyses performed by the Doop framework for some benachmarks using the native scanner utility we presented in Chapter 3 and we comment on the experimental results of our analysis.

Table 3 shows the total sizes of the applications analyzed, the size of their (uncompressed) Dalvik bytecode and native code, the number of methods of the applications and their native code architecture.

Table 3: Sizes of test APK files and their (uncompressed) contents.

Benchmark	Application	Dalvik Bytecode	Native Code	App Methods	Architecture
Chrome	99,0 MB	72,4 MB	6,5 MB	37898	x86
Instagram	24,3 MB	21,5 MB	8,0 MB	43420	x86

In our evaluation, we have included the results for the base analysis (an analysis without the use of native scanner utility) and for the analysis using various modes of the native scanner utility. These modes are the following:

- -default. In this mode, we utilize the heuristics we described in Section 3.4 to find Java method calls.
- -dist-<offset>. In this mode, we examine the result of the command ”readelf -p .rodata” and we consider as Java method calls the Java methods that their name and signature are close enough in the `.rodata` section using an offset value equal to <offset>. We do not know the function boundaries in this mode.

4.1 App-reachable Methods Increase

Using the native scanner utility, there is an increase in reachable methods of applications. Table 4 presents the percentage of increase between the amount of application’s reachable methods found in base analyses and the amount of the application’s reachable methods found using the native scanner utility.

Table 4: App-reachable methods increase over base.

Benchmark	Mode	Base	Scanner	Percentage	Entry points
Chrome	-default	17003	23781	39.86%	4275
Chrome	-dist-40	17003	24060	41.50%	4484
Instagram	-default	23921	32425	35.55%	4669
Instagram	dist-40	23921	32425	35.55%	4669

4.2 Heap Snapshot vs. Our Approach

In this section, we measure the *recall* of our technique, i.e., the percentage of runtime behavior that our technique captures (compared to an analysis without our technique). We use HeapDL [18] in order to get heap snapshots for some Android applications running on a x86 emulator and our HelloJNI example. By analysing the heap snapshots, we can see what Java methods were called from native code at runtime and check if our analysis finds them too.

Let B, N, H be the set of call-graph edges between native functions and Java methods in base analysis, in native scanner analysis and in HeapDL analysis respectively. Therefore, *base recall BR* is $BR = \frac{(B \cap H)}{H}$ and *recall R* of our technique is $R = \frac{(N \cap H)}{H}$.

Table 5: Base Recall and Recall of test APK files.

Benchmark	Mode	HeapDL∩ Base	HeapDL∩ Scanner	HeapDL	Base Recall	Scanner Recall
Chrome	-default	7	83	83	8.43%	100.00%
Chrome	-dist-40	7	83	83	8.43%	100.00%
Instagram	-default	1	7	7	14.29%	100.00%
Instagram	-dist-40	1	7	7	14.29%	100.00%

4.3 Time Increase

In this section we present the time needed for the analysis of the APK files and the time needed for fact generation. We also present the percentage changes of both the analysis time (Table 6) and the fact generation time (Table 7) between base analyses and different modes of the analyses using the native scanner utility. Because of extra computations, almost in every analysis using the native scanner there is an increase in time.

Table 6: Analysis time increase.

Benchmark	Mode	Base	Scanner	Percentage
Chrome	-default	456	472	3.51%
Chrome	-dist-40	456	526	15.35%
Instagram	-default	543	572	5.34%
Instagram	-dist-40	543	650	19.71%

Table 7: Fact generation time increase.

Benchmark	Mode	Base	Scanner	Percentage
Chrome	-default	49	258	426.53%
Chrome	-dist-40	49	260	430.61%
Instagram	-default	52	87	67.31%
Instagram	-dist-40	52	61	17.31%

5. RELATED WORK

An interesting approach for finding string references in x86 closed-source executable files using a static analysis is presented in the work ”String analysis for x86 binaries” [19]. In this work, two analyses are applied to get some information about types of the binary, a backward analysis called `string-inference analysis` and a forward analysis called `stack-height analysis`. By making the assumptions that strings in x86 binaries are represented using the C-style encoding and all string operations use the libc string functions, the `string-inference analysis`, using the strcat signature for instance, can infer that the registers before the call of strcat are C-style strings. On the other hand, the `stack-height analysis` relies on the assumptions that x86 binaries use _cdel calling convention and the caller function passes arguments to the callee using the stack, thus push instructions can be combined with function calls. Last but not least, an `alias analysis` is used in order to overcome C-style strings' mutability because of string operations that may affect the memory. For example, the `mov` or `lea` instruction establish an alias between two registers.

A tool that is relevant to our work is the disassembler called Ddisasm which is developed by GrammaTech, Inc. [20]. Ddisasm is implemented in Datalog and it is a disassembler for machine code that produces reassembleable assembly by combining heuristics and novel static analyses. It is fast, precise and can handle large files of real-world applications. After disassembling a file the resulting code after disassembling contains accurate symbolic information and provides cross references. This means that Ddisasm can be used to find constant strings in functions of the native code of applications and more precisely Java method calls.

Redex is an open source Android bytecode (dex) optimizer that was developed at Facebook [21]. In order to improve performance and efficiency of Android applications, Redex tries to reduce .dex file size with optimizations, such as dead code elimination, inlining, minification. Furthermore, it removes unnecessary metadata and it performs analysis to remove unnecessary interfaces. Redex scans also native libraries which are concentrated into .xz and .zstd files to perform optimizations.

6. CONCLUSIONS

Using Datalog and the Doop framework, we were able to express a compact and succinct analysis aiming to resolve the unsoundness in static analysis of applications containing Java and native code. The key of our process is that we are able to scan native libraries of real-world applications for which we do not have the source code using disassemblers. Through the examination of the disassembled code we can find how Java methods are called by native functions. Last but not least, using the Doop framework, we were also able to decouple the implementation of the analysis from the context chosen for the points-to analysis.

Our analysis is based on heuristic techniques that nevertheless had positive results. However, further work is still necessary because our research is focused on finding only constant strings that could refer to Java method calls. So, our estimation of how native code interacts with the Java code of the application cannot be considered complete in any way. We hope that our research could constitute the basis for future work that could lead to better results.

ACRONYMS AND ABBREVIATIONS

HLL	High-Level Languages
JNI	Java Native Interface
JVM	Java Virtual Machine
ABI	Application Binary Interface
NDK	Android Native Development Kit
BDDs	Binary Decision Diagrams
IDB	Intensional Database
EDB	Extensional Database
APK	Android Application Package

REFERENCES

[1] Yannis Smaragdakis and George Balatsouras (2015), “Pointer Analysis”, Foundations and Trends® in Programming Languages: Vol. 2: No. 1, pp 1-69. http://dx.doi.org/10.1561/2500000014

[2] Martin Bravenboer and Yannis Smaragdakis. “Strictly declarative specification of sophisticated points-to analyses”, in Proceedings of the 24th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, OOPSLA ’09, New York, NY, USA, 2009. ACM.

[3] “Android Native Development Kit”. https://developer.android.com/ndk

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

[5] Levine, John R. “Linkers and Loaders”, Morgen Kaufmann: 10-12, 2000.

[6] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam, “Using Datalog with binary decision diagrams for program analysis”, in Kwangkeun Yi, editor, APLAS, volume 3780 of Lecture Notes inComputer Science, pages 97–118. Springer, 2005.

[7] Ondrej Lhotak and Laurie Hendren, “Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation”, ACM Trans. Softw. Eng. Methodol., 18(1):1–53, 2008.

[8] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, DzintarsAvots, Michael Carbin, and Christopher Unkel, “Context-sensitive program analysis as database queries”, in PODS ’05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12, NewYork, NY, USA, 2005. ACM.

[9] Thomas Reps, “Demand interprocedural program analysis using logic databases”, in R. Ramakrishnan, editor, Applications of Logic Databases, pages 163–196. Kluwer Academic Publishers, 1994.

[10] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini, “Defining andcontinuous checking of structural program dependencies”, in ICSE ’08: Proc. of the 30th int. conf. on Software engineering, pages 391–400, New York, NY, USA, 2008. ACM.

[11] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor, “Codequest: Scalable source code queries with Datalog”, in Proc. European Conf. on Object-OrientedProgramming (ECOOP), pages 2–27. Spinger, 2006.

[12] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, “The Soot framework for Java program analysis: a retrospective”, in Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011.

[13] Dennis Andriesse, Xi Chen, and Victor van der Veen, Asia Slowinska, and Herbert Bos. “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries”, in SEC’16 Proceedings of the 25th USENIX Conference on Security Symposium, pages 583-600, Austin, TX, USA — August 10 - 12, 2016.

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

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

[16] Position Independent Code on i386 – what is it, why is it needed, how to write, how to exploit https://web.archive.org/web/20140911162958/http://www.greyhat.ch/lab/downloads/pic.html

[17] Chapter 4: JNI Functions, Java Documentation, Oracle https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html

[18] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. “Heaps don’t lie: countering unsoundness with heap snapshots”, in Proceedings of the ACM on Programming Languages, Volume 1, Issue OOPSLA, October 2017, Article No. 68.

[19] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. “String analysis for x86 binaries”, in Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, September 05-06, 2005, Lisbon, Portugal.

[20] Antonio Flores-Montoya and Eric Schulte. “Datalog Disassembly”. Last revised 12 Jun 2019: `https://arxiv.org/abs/1906.03969`

[21] Facebook Inc. “Redex - An Android Bytecode Optimizer”. `https://fbredex.com/`