ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/363739852

Reach Me if You Can: On Native Vulnerability Reachability in Android Apps

Conference Paper · September 2022

DOI: 10.1007/978-3-031-17143-7_348

CITATIONS
0

READS
18

6 authors, including:

Luca Borzacchiello
Sapienza University of Rome
8 PUBLICATIONS **46** CITATIONS

SEE PROFILE

Emilio Coppa
Sapienza University of Rome
28 PUBLICATIONS **650** CITATIONS

SEE PROFILE

Davide Maiorca
Università degli studi di Cagliari
34 PUBLICATIONS **2,738** CITATIONS

SEE PROFILE

Camil Demetrescu
Sapienza University of Rome
120 PUBLICATIONS **2,771** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Adversarial Machine Learning View project

https://www.cyberroad-project.eu View project

All content following this page was uploaded by Emilio Coppa on 22 September 2022.

The user has requested enhancement of the downloaded file.

Reach Me if You Can: On Native Vulnerability Reachability in Android Apps

Luca Borzacchiello[1[0000−0001−7198−5175]], Emilio Coppa[1[0000−0002−8094−871X]], Davide Maiorca[2[0000−0003−2640−4663]], Andrea Columbu[2], Camil Demetrescu[1[0000−0002−4686−6745]], and Giorgio Giacinto[2[0000−0002−5759−3017]]

[1] Sapienza University of Rome, Italy
{borzacchiello, coppa, demetres}@diag.uniroma1.it
[2] University of Cagliari, Italy
{davide.maiorca,giacinto}@unica.it

Abstract. Android applications ship with several native C/C++ libraries. Research on Android security has revealed that these libraries often come from third-party components that are not kept up to date by developers, possibly posing security concerns. To assess if known vulnerabilities in these libraries constitute an immediate security problem, we need to understand whether vulnerable functions could be reached when apps are executed (we refer to this problem as function reachability).
In this paper, we propose DroidReach, a novel, static approach to assess the reachability of native function calls in Android apps. Our framework addresses the limitations of state-of-the-art approaches by employing a combination of heuristics and symbolic execution, allowing for a more accurate reconstruction of the Inter-procedural Control-Flow Graphs (ICFGs). On the top 500 applications from the Google Play Store, DroidReach can detect a significantly higher number of paths in comparison to previous works. Finally, two case studies show how DroidReach can be used as a valuable vulnerability assessment tool.

Keywords: Android · Static Analysis · Mobile Security.

This paper is dedicated to the memory of Camil Demetrescu, a brilliant researcher and a great teacher that will be missed by many students and colleagues.

Luca, Emilio, Davide, and Giorgio

1 Introduction

The Android ecosystem has significantly evolved over the years. Applications have become more user-friendly and feature functionalities such as advanced graphics, database management, and modern encryption. While many features can be directly implemented with Java code, developers rely on C/C++ libraries (via the Java Native Development Kit) to achieve greater speed and flexibility.

The analysis of Java code has dominated the Android security scene, as malicious samples typically resort to Java components to carry out their operations [40,8]. Conversely, the focus of research on the Android Native Environment has been limited. Nonetheless, the Android Native Environment conceals more issues than what can be superficially assumed, as recent works showed a significant presence of vulnerabilities in native code [3]. The problem becomes significant as such vulnerabilities are mostly due to not-updated versions of libraries that are continuously employed even in very popular applications (featuring millions of downloads). However, the presence of vulnerabilities alone does not immediately translate into a security problem because it depends on *whether they could be concretely exploited*. While this question can be extremely difficult to answer, especially in large-scale environments such as Android, we can study whether *such vulnerable functions could be reached when apps are executed*. We refer to this problem as *function reachability*.

Previous works on Android native code have proposed dataflow techniques that work either statically or dynamically. Static approaches [38,39] mainly exploited symbolic execution – a powerful program analysis that struggles to scale over complex and large Android apps – leading to results that are incomplete and thus inaccurate when considering function reachability. Dynamic approaches [42] can accurately analyze single execution paths but they still can hardly scale over apps featuring even thousands of classes, leading to the so-called *path explosion* problem. In this sense, it becomes crucial to find a proper balance between the *time* needed for the analysis and the *precision* of the attained results.

In this paper, we propose DroidReach, a static approach to establish the reachability of native methods in Android apps starting from the application Java entry points. In particular, we propose the following contributions:

1. We discuss the technical limitations that hamper the analysis capabilities of current analysis tools. In particular, we show limitations in: (a) properly mapping Java `native` methods to Java Native Interface (JNI) methods, (b) handling *nested* native libraries, and (c) accurately building the ICFGs.
2. We present the methodology underlying DroidReach: it combines several heuristics and ICFG construction techniques to mitigate the limitations mentioned in the previous point. In this way, DroidReach can accurately and effectively reconstruct possible paths to potentially vulnerable native calls.
3. We perform an evaluation considering 500 popular applications featuring complex native libraries and ICFGs (with an average of $2,000,000$ native instructions and $1,000,000$ Java instructions). We show that DroidReach can reach more instructions than Argus-SAF [38], which is the state-of-the-art static framework for analyzing the Java and native layers in Android.
4. We propose two real, practical case studies where we show how DroidReach can be helpful to assess the reachability of vulnerable functions.

To foster further research, we make our contributions available at https://github.com/season-lab/DroidReach. We believe that DroidReach represents a step forward for the Android community as it can provide valuable insights to security experts in presence of large and complex apps.

2 Background and Related Work

Android Apps. Android applications are zipped `.apk` (Android application package) archives containing: *(i)* The `AndroidManifest.xml` file and other `.xml` files, which provide the application metadata and layout; *(ii)* One or more `classes.dex` files, which contain the executable bytecode of the Java/Kotlin classes; *(iii)* External resources, such as images or native libraries.

The Android NDK. The Android Native Development Kit (NDK) is an ensemble of tools that allow for the implementation of parts of Android apps in native (C/C++) code. Such a code is typically employed to guarantee faster performance in comparison to traditional Java code. The interface between the Java and the native layer is called Java Native Interface (JNI). JNI essentially defines how functions receive parameters or provide return values. Native libraries can be loaded with the `System.loadLibrary` method. Then, the native methods to be invoked are declared in the Java code by using the `native` keyword.

Native Libraries Analysis. Native libraries have been especially studied in the context of vulnerability identification, i.e., understanding the *presence* of vulnerabilities in native code. More specifically, prominent works concerned the study of the JNI interface vulnerabilities [35,36,29,23,27,25], while others involved the identification of possible vulnerabilities in Android libraries. Derr et al. [17] conducted a test with 200 developers in which they showed that many libraries embedded in apps are outdated and contain security vulnerabilities. Various approaches have been proposed to detect them, based on machine learning [21], similarities between functions [20,41], and hybrid analysis [31,30].

Recently, Almanee et al. [3] proposed an extended assessment of the presence of vulnerable functions in Android native libraries. In particular, they showed that applications contain libraries that have not been updated even for two years, thus exposing vulnerabilities that typically require a long time to be fixed. We base the beginning of our analysis on the results of this work, as it depicts a critical scenario where various applications may feature critical security issues.

Dataflow Analysis. Dataflow analysis has been extensively studied in Android, with a focus on how data propagates in Java code. This problem has been addressed with static and dynamic approaches. Regarding static approaches, FlowDroid [7] was among the first to introduce proper handling of the Android callbacks. Other works improved FlowDroid in many aspects, such as proper dataflow tracking for intents [26,22,28,13]. Amandroid [39] is deemed as the current reference point for static dataflow analysis in Android. Wei et al. [38,39] expanded Amandroid by releasing JN-SAF (now known as Argus-SAF), which introduced the analysis of the information flows between the Java and the native layer. The approach employs symbolic execution to handle the native layer. In particular, Argus-SAF uses `CFGEmulated` from Angr [33] to reconstruct ICFGs of the native code (§3.1) and compute approximate dataflow facts. Argus-SAF will be the main reference point for the analyses discussed in this paper.

Dynamic approaches employ code instrumentation and execution to perform taint analysis. Droidscope and TaintDroid [44,19] are among the first approaches

to have adopted this technique. Subsequent works built upon and improved TaintDroid [34,43,42] by using, e.g., concolic execution [14,12,11]. Unfortunately, a challenge is *how* to generate the *right* executions that will reach a function.

Input Generation. Several works [6,4,15,24,1,40,45,37] aim at generating *user inputs* that can lead to the execution of specific functions. Systems based on static analysis allow for faster code coverage, but they can lack precision. Conversely, approaches based on dynamic analysis can be much more precise, but they can often be unfeasible due to the so-called *path explosion* problem. One notable example is Intellidroid [40], an approach that uses static and dynamic analysis to generate those inputs that allow for reaching specific calls. Intellidroid only focuses on the Java layer without exploring the native layer.

Automatically finding the right set of stimulations for an app remains an open research problem, especially when considering large and complex apps. DroidReach cannot find the inputs able to reproduce a specific path but can provide insights about the existence of a path toward a specific target point.

3 DroidReach

In this section, we present the main ideas behind DroidReach. First, we discuss the problem targeted by DroidReach and the challenges that affect existing approaches. Then, we present the design and the components of DroidReach.

3.1 Problem Statement and Reachability Challenges

Terminology. In the following, we define a few terms used across the paper:

- A **code point** p for our analysis is an instruction inside the set of instructions from the Java layer (J) or the native layer (N) of an app, i.e., $p \in (J \cup N)$.
- A **Control-Flow Graph (CFG)** is a graph representation of possible paths that can be taken during the execution of a function. Each node represents a contiguous sequence of code points. Edges represent jumps across nodes.
- A **Call Graph (CG)** is a graph that represents the calls across different functions of an app.
- An **Inter-procedural Control-Flow Graph (ICFG)** connects the CFGs of different functions using the information from the CG. In practice, it can encode all the app's paths starting from a specific entry point.
- A **source** p_s is a code point in an app that could start the execution of some Java code. Hence, a source could be seen as an entry point for the Java layer.
- A **sink** p_t is a code point inside a native library, i.e., $p_t \in N$. The sink identifies an interesting point that we would like to reach during the execution.

Goal. Given the Java instructions J, the native instructions N, and a sink p_t, our goal is to identify at least one path starting from one source p_s and ending in the sink p_t. The path is represented as the sequence of points traversed in the ICFG from the source p_s. Identifying a path in the ICFG can be valuable for several program analyses and security tasks. This paper focuses on sinks that could be associated with vulnerable code points within native libraries.

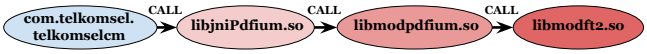

Fig. 1: Example of *nested* libraries.

Reachability Challenges. Argus-SAF (§2) is the state-of-the-art solution for statically analyzing both the Java and the native layers of an Android app. When testing it on real-world apps for our goals, we have identified a few critical challenges which affect its accuracy (and also of other existing works):

C1 **Mapping Java native methods to JNI methods.** To execute the code of a native library, the Java code invokes a Java `native` method. Java `native` methods at running time could be seen as *jumps* to JNI methods, which are the *entry points* for the native layer. Unfortunately, statically identifying the *mapping* between Java `native` methods and JNI methods is not always trivial. State-of-the-art solutions may fail (§4) to *resolve* a large number of these mappings, possibly ignoring several entry points of the native layer.

C2 ***Nested* native libraries.** A JNI method is part of a shared library, e.g., `libA.so`. A shared library may call methods of other libraries, i.e., one library may link another one (e.g., `libB.so`). State-of-the-art solutions may not perform analyses across multiple native libraries in the case of *nesting*. This is crucial on Android, since apps often: (a) integrate open-source libraries, which may rely on other ones, and (b) devise *wrappers* to work with libraries that were not originally written for Android and thus do not implement the JNI API. An example of nested libraries is given in Figure 1.

C3 **Scalability versus accuracy.** Argus-SAF uses symbolic execution to analyze the native code. While this technique can be very accurate during the ICFG construction, allowing the tool to even compute data flow facts, it does not scale on complex libraries. Indeed, Argus-SAF trades accuracy for scalability, halting its analysis when the call depth is larger than, e.g., 5, which is not enough in several cases, thus generating incomplete ICFGs. Approaches based on traditional binary frameworks [32], may provide better scalability but then generate less accurate ICFGs, e.g., in the presence of indirect jumps.

To help the reader grasp the technical aspects behind these challenges, we show them in the context of a running example in the remainder of the section. However, we first present at a high level the design behind DroidReach.

3.2 Architecture of DroidReach

Figure 2 depicts the main steps performed by DroidReach:

S1 **Static analysis of the Java layer.** The first step builds the ICFGs of the Java code, identifying sources and calls to Java `native` methods.

S2 **Analysis of interactions between Java and native layer.** After detecting the Java `native` methods that could be reached during an execution, DroidReach identifies the mappings between Java `native` methods and JNI methods. This step is thus designed to tackle challenge C1.

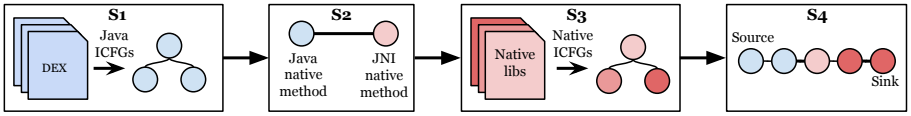

Fig. 2: Main steps of DroidReach.

S3 **Static analysis of the native layer.** Given a list of JNI methods that could be reached during the execution, DroidReach builds the ICFGs of the native libraries. This step aims at challenges C2 and C3, combining different techniques to target a nice trade-off between accuracy and scalability.

S4 **Reachability analysis.** The last step is where DroidReach puts together the pieces constructed in the previous stages. It first merges the ICFGs of the Java layer with the ICFGs of the native layer and then evaluates whether there exists a path from a source p_s to a user-defined sink p_t.

In the remainder of this section, we review in detail these steps. Throughout our discussion, we use a running example: Figure 3 shows an excerpt of its code. This is an app with two activities, where the first one (`LoginActivity`, omitted from the code) checks the credentials of the user, while the second one (`JavaLayerActivity`) runs some tasks using some native libraries (`native-lib.so` and `other-native-lib.so`) when the user clicks a button. Differently from dynamic approaches, DroidReach can directly focus on `JavaLayerActivity` without necessarily satisfying the execution requirements of `LoginActivity`, which could be arbitrarily hard to automatically identify and satisfy.

3.3 Static analysis of the Java Layer

Different state-of-the-art frameworks already exist to analyze the Java layer, providing different trade-offs in terms of accuracy and scalability. The current implementation of DroidReach can work with Androguard [18] and FlowDroid [7], while support for Amandroid [39] is being worked on. Regardless of the specific framework in use, DroidReach performs three stages:

1. **Identification of sources.** DroidReach looks for sources by considering the class methods of several Android components, following the guidelines and suggestions proposed in previous works [18,7].
2. **ICFG construction.** For each source, DroidReach builds an ICFG considering the entire Java code of the app, connecting the CFGs of the methods based on their *caller-callee* relationships.
3. **Identification of Java native methods.** Finally, this step identifies Java `native` methods which are invoked in the ICFGs of the Java layer. At this stage, a call to a Java `native` method is not yet mapped to its JNI method, which contains the actual binary implementation of the Java `native` method.

Running example. When considering the `JavaLayerActivity` class, there are two sources: `onCreate` (`J1`), which is executed at the activity startup; the

```
// Java layer: classes.dex
    public class JavaLayerActivity
            extends AppCompatActivity {
J0.  static {
        System.loadLibrary("native-lib"); }

J1.  protected void onCreate(Bundle state) {
J2.   super.onCreate(state);
J3.   setContentView(
        R.layout.activity_native_buttons);
J4.   findViewById(R.id.btn_1).setOnClickListener
J5.     (v -> { execMethod(); });
     }

J6.  private static native void execMethod();
    }

// Native layer: native-lib.so
N0. JNIEXPORT jint JNI_OnLoad(JavaVM* vm,
                              void* reserved) {
     JNIEnv* env; vm->GetEnv(&env, JNI_VERSION_1_6);
     jclass c = env->FindClass("JavaLayerActivity");
     static const JNINativeMethod mappings[] =
      { {"execMethod", "()", execMethod} };
```

```
// Native layer: native-lib.so (cont'd)
     /* some complex code */
     return env->RegisterNatives(c, mappings,
      sizeof(methods)/sizeof(JNINativeMethod));
    }
N1. static void execMethod(JNIEnv* env,
                           jclass clazz) {
N2.  Handler* h = build();
N3.  h->callback(); // use of fn pointer
    }
N4. Handler* build() {
N5.  auto* h = new Handler();
N6.  h->init(); // virtual call
N7.  return h;
    }

N8.  void Handler::init()
      { this->callback = &foo1; }

// Native layer: other-native-lib.so
N9.  void foo1() { foo2(); }
N10. void foo2() { foo3(); }
N11. void foo3() { foo4(); }
N12. void foo4() { bug(); }
```

Fig. 3: Running example.

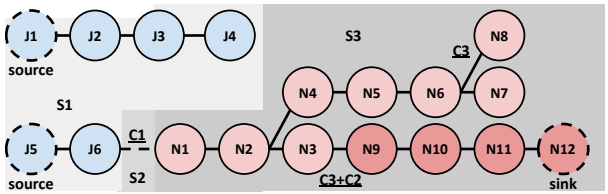

Fig. 4: ICFGs of the running example: shades of gray highlight different steps.

anonymous handler (J5) for button events. Figure 4 shows in blue the Java code points for the ICFGs starting from these two sources. J6 is a Java `native` method. An additional *implicit* source, considered by DroidReach but omitted from Fig. 4, is J0, which triggers the execution of `JNI_OnLoad` (see next section).

3.4 Analysis of interactions between Java and native layer

Each Java `native` method is mapped to a JNI method in the native layer. The mapping can be defined statically or dynamically, as described in the following.

Static mapping. The name of the JNI method is a symbol exported by the library that follows a specific mangling scheme, allowing the dynamic linker to uniquely identify the Java `native` method associated with it. For instance, the native method `com.lyrebirdstudio.lyrebirdlibrary.EffectFragment.shadows` in Figure 5 maps to the JNI method `Java_com_lyrebirdstudio_lyrebirdlibrary_EffectFragment_shadows`. As in previous works [38], DroidReach uses a decoder of the mangling scheme to resolve statically defined JNI methods.

Dynamic mapping. When the dynamic loader loads a library, it runs the `JNI_OnLoad` function exported by the library. This function may dynamically de-

```
// Java layer: classes.dex
package com.lyrebirdstudio.lyrebirdlibrary;
public class EffectFragment extends Fragment {
  private static native void shadows(Bitmap arg0, float arg1)
  { /* statically resolved by dlsym() based on the name of the JNI method */ }

// Native layer: libfilter.so
void Java_com_lyrebirdstudio_lyrebirdlibrary_EffectFragment_shadows(
    JNIEnv *env, jclass c, jfloat arg1) { /* native implementation of the JNI method */ }
```

Fig. 5: A statically defined JNI method.

```
// Java layer: classes.dex
package com.aviary.android.feather.headless.moa;
public class Moa {
  static native void n_applyActions() { /* dynamically resolved by JNI_OnLoad() */ }

// Native layer: libaviary_native.so
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved) { ...;
 jclass c = env->FindClass("com/aviary/android/feather/headless/moa/Moa");
 static const JNINativeMethod m[] =
  { ..., {"n_getEffects", "()[Ljava/lang/string;]", (void*)(n_getEffects)}, ... };
 env->RegisterNatives(c, m, sizeof(m) / sizeof(JNINativeMethod)); ...; }

jobjectArray n_getEffects(JNIEnv *env, jclass c) { /* native implementation of the method */ }
```

Fig. 6: A dynamically defined JNI method.

fine mappings between Java methods and native functions using the JNI primitive `RegisterNatives`, which takes as one of its arguments a pointer to an array of `JNINativeMethod`. This `struct` is defined as:

```
typedef struct {
    char *name;      // ex: "nativeCtor"
    char *sign;      // ex: "(Ljava/lang/String;)J"
    void *fnPtr;     // function code pointer
} JNINativeMethod;
```

The `struct` states that the native implementation of the Java native method `name` having the signature `sign` (which defines, in `smali`, the types of the method arguments and the return value) is available at the address `fnPtr`. Figure 6 shows how a real app is defining the mapping for the JNI method `n_getEffects`.

Previous works [38] perform symbolic execution from `JNI_OnLoad` to identify the array passed to `RegisterNatives`. This strategy has two downsides: (a) the exploration is halted when reaching a given call depth (e.g., 5 in Argus-SAF) to mitigate path explosion, possibly failing to reach `RegisterNatives`, and (b) the exploration may incur a large overhead when `JNI_OnLoad` is not trivial.

For these reasons, our approach devises a more scalable heuristic to detect dynamic mappings. The key idea is that several developers follow the guidelines of Android [5] and statically define the `JNINativeMethod` array at compilation time, placing it in the global data section. DroidReach thus scans the data section of a library, looking for an array with elements following the pattern:

1. Pointer to a valid string (`name`).
2. Pointer to a valid string (`sign`).
3. Pointer to a function in the `text` section (`fnPtr`).

Since some apps may instead allocate and initialize the array during the execution of `JNI_OnLoad`, DroidReach fallbacks to symbolic execution when:

1. The heuristic fails to identify mappings for a library containing `JNI_OnLoad`.
2. The heuristic identifies some mappings, but there are clashes on the pair (`name`, `signature`) of some methods, e.g., there are multiple Java methods from different classes called `name` with the same `signature`, requiring to inspect additional arguments of `RegisterNatives` to solve the ambiguity.

Hence, DroidReach fallbacks to a more heavyweight analysis only when there are insights that the heuristic is not working correctly.

Running example. When loading the `JavaLayerActivity` class, the loader is executed due to `J0`, processing `native-lib.so`. The `JNI_OnLoad` function of this library defines the mapping (`J6`, `N1`). DroidReach identifies it by analyzing the array `mappings`. Symbolic execution analyses may instead struggle when `JNI_OnLoad` integrates some complex code before the call to `RegisterNatives`.

3.5 Static analysis of the native layer

After identifying the JNI mappings, DroidReach constructs the ICFGs for the native layer considering each JNI method as a possible entry point. To cope with challenge C2, DroidReach recursively builds the ICFG if one function of a library is calling a function of another library. Additionally, to cope with challenge C3, our approach combines the ICFGs built by different techniques.

ICFG construction. For each JNI method, DroidReach builds the CFGs and the CGs of the native functions to obtain the ICFGs. Our implementation uses the Ghidra reversing framework [32], as it worked particularly well when considering libraries found in Android apps. The ICFGs derived in this stage include only code points from the same shared object of the starting JNI method.

Library dependency graph. Given the ICFG for a JNI method, our approach analyzes the `calls` to imported functions, i.e., calls to functions from other libraries. To represent this information for all ICFGs, it defines a *library dependency graph*, where the nodes represent libraries and the edges are calls across different libraries. Each edge is annotated with a list of caller-callee tuples to track the different calls that may involve the same pair of libraries.

ICFG refinement: nested libraries. Using the library dependency graph, DroidReach refines the ICFG of each JNI method to include code points from nested libraries. Since this stage may need to build the ICFG of methods never met before (or it may discover new calls to other imported functions), our approach iteratively repeats the two previous stages until a fixed point is reached.

ICFG refinement: symbolic exploration. The previous stages can build ICFGs that may traverse several libraries, potentially representing paths able to reach even *deep* code points in an execution path. However, the previous stages may still miss some critical edges in the ICFG: e.g., in the presence of a callee that performs an indirect call using a target defined by its caller. While reverse engineering frameworks, such as Ghidra, have reduced the need for

heavyweight dataflow analyses significantly, there are still several cases where they may be needed to accurately build an ICFG. For this reason, for each JNI method, our approach performs a symbolic exploration using `CFGEmulated` of ANGR [33] to recover the missed edges. To control path explosion, the exploration stops its analysis when the call stack contains more than 5 nested calls (as in ARGUS-SAF). Moreover, path branches are not evaluated, thus skipping most symbolic queries. After running this refinement step, we repeat the two previous stages until a fixed point is reached. Hence, DROIDREACH combines two different techniques for building the ICFGs: the first one is more scalable but less accurate, while the second one is more accurate but less scalable. Previous approaches, such as ARGUS-SAF, have favored ICFG construction approaches based on symbolic execution, which however can struggle at reaching *deep* code points.

Running example. Starting from N1, DROIDREACH builds one ICFG with code points {N1, N2, N3, N4, N5, N6, N7}. Indirect jumps (N6, N8) and (N3, N9) are not discovered by GHIDRA but can be recovered using a symbolic exploration. Since N3 calls a function of `other-native-lib.so`, DROIDREACH builds the library dependency graph, analyzes this library, and adds {N9, N10, N11, N12}.

3.6 Reachability Analysis

The last step is in charge of computing a path from a source to a sink.

Defining the sink. In general, the sink is a user-defined choice that is tightly connected to the goal targeted by an analysis. In this paper, given a vulnerability report, we define the sink as the *closest* code point (or even the set of code points if there is not a unique choice) that the app execution should reach in order to reproduce the bug described in the report. To identify the open-source project related to a library from an Android app, including the adopted release, we refer to solutions, such as [3], that have proposed effective binary similarity techniques.

Merging ICFGs. Given a sink, DROIDREACH exploits the JNI mappings to connect ICFGs of the Java layer to the ICFGs of the native layer.

Finding a path from a source to a sink. Finally, for each source p_s and for each sink p_t, our approach evaluates whether there exists a path from p_s to p_t. In practice, since there could be several alternative paths between p_s and p_t, our current implementation by default emits the shortest one as it typically is the simplest to check for a user. However, alternative paths can be requested.

Running example. Assuming that N12 is part of a known vulnerability in `other-native-lib.so`, DROIDREACH builds the ICFGs in Figure 4 and quickly computes the path {J5, J6, N1, N2, N3, N9, N10, N11, N12}.

4 Experimental Evaluation

In this section, we evaluate the efficacy of DROIDREACH. Due to lack of space, we omit the discussion of step S1 as it involves well-known mainstream Java analysis frameworks, which we did not alter in DROIDREACH. Experiments were

conducted in a Ubuntu 20.04 Docker container, using two Intel Xeon E5-4610v2 CPUs and 256 GB of RAM. APK hashes of evaluated apps can be found at [9].

4.1 Microbenchmarks

To validate DroidReach, we considered existing benchmarks from the Android literature. DroidBench [7] does not involve native libraries. On NativeFlowBench [38], DroidReach performs consistently with Argus-SAF when considering the reachability goal. Since NativeFlowBench ignores the challenges from §3.1, we designed a new benchmark suite composed of 13 apps that exhibit these aspects from different perspectives (see Table 6 in the Appendix). The source code and a detailed discussion of this suite can be found in a dedicated repository [10]. On 12 out of 13 apps, DroidReach correctly builds accurate native ICFGs, improving on Argus-SAF (8 out of 13). The failing case involves an indirect jump at *deep* call depth: this app was specifically designed to highlight that DroidReach cannot solve, in general, the scalability issues that inherently affect static analyses, and it can only help to mitigate them (hopefully in several cases). Further details on the results are available at [10].

4.2 Real-World Dataset

Dataset. To evaluate the efficacy of DroidReach on real-world apps, we collected the top-20 apps from each category of the Google Play Store, keeping the ones containing ARMv7 libraries. Overall, we obtained 500 apps, whose *popularity* ranges from a minimum of 100K downloads to more than 1 billion downloads. Such selection choice has also been guided by the idea of representing apps whose vulnerability may have a very large impact on the end-users. The average *complexity* of these apps is very high, as detailed in Table 5 from the Appendix, in terms of the number of Java and native instructions (more than 2.3 million of native LoC on average), the number of Java `native` methods (more than 204 methods on average) and of ARMv7 libraries (at least 5 on average).

Fine-grained evaluation. To evaluate the correctness of DroidReach, we need to analyze the false negatives (code points that are missing from the ICFGs) and the false positives (code points that are wrongly inserted in the ICFGs).

For the false negatives, we randomly picked 15 apps from our dataset and then manually stimulated them in the Android emulator as a user would do in a short usage session, recording the executed native function entry points. Any recorded code point should thus be contained in the ICFGs. While our sample set may seem small, the effort for validating the results took more than 1.5 man-months. Table 1 divides the 15 apps into three groups: apps where DroidReach was able to identify more than 95% of the executed code points are in the first group, more than 50% of the code points in the second group, less than 50% of code points in the third group, respectively. DroidReach significantly outperforms Argus-SAF and Ghidra (when used in step S3 in place of DroidReach) on several apps. This result comes from the effective combination of different techniques: Argus-SAF fails to scale its analysis and Ghidra misses indirect jumps

APK	# of executed code points found in the native ICFGs from		
	Argus-SAF	Ghidra	DroidReach
com.sec.android.easyMover	33/33	33/33	33/33
com.jb.zcamera	11/11	11/11	11/11
com.mi.android.globalFileexp.	31/48	47/48	47/48
com.space.cleaner.smart.tool	52/75	65/75	75/75
com.soundcloud.android	57/102	72/102	72/102
video.like	197/518	272/518	320/518
com.zentertain.photocollage	186/331	174/331	239/331
com.picsart.studio	442/1282	203/1282	736/1282
shareit.lite	33/60	44/60	47/60
com.imangi.templerun	60/326	67/326	248/326
com.amazon.mp3	92/395	218/395	239/395
com.cam001.selfie	282/344	211/344	322/344
com.tripadvisor.tripadvisor	2/42	30/42	30/42
com.yodo1.crossyroad	3/79	25/79	25/79
com.king.candycrushjellysaga	5/54	6/54	6/54

Table 1: Analysis of false negatives: executed code points found in the ICFGs.

APK	Validation mode	Confidence	# code points to validate	# validated code points
com.imangi.templerun	Dynamic	High	2357	1565
com.picsart.studio	Dynamic	High	1307	748
com.cam001.selfie	Dynamic	High	974	326
com.king.candycrushjellysaga	Dynamic	High	168	63
com.amazon.mp3	Mixed	Medium	625	441
shareit.lite	Static	Medium	12	12
com.sec.android.easyMover	Static	Medium	107	60

Table 2: Analysis of false positives: validated code points in the ICFGs.

that could be recovered with a symbolic execution analysis, while DroidReach shows the best of the two approaches (see [9] for detailed debug results). However, there are some apps where even DroidReach is unable to statically recover some executed code points. On some apps, slightly increasing the maximum call depth in the symbolic exploration can lead to better results (e.g., from 5 to 10 allows to find +9% of executed code points in `com.amazon.mp3`). Similarly, increasing the maximum analysis timeout can improve the accuracy, but there is a trade-off that must be taken into account between accuracy and analysis time. Even when extending the analysis time, DroidReach cannot cope with some patterns (§5): e.g., `com.yodo1.crossyroad` loads a library using a custom loader and `com.imangi.templerun` indirectly executes code from the Mono framework.

For the false positives, the evaluation is significantly harder as it requires to *exhaustively* stimulate an app, which can hardly be done *automatically* for most apps. Nonetheless, we attempted to still validate at least a subset of the code points. In particular, we compared the ICFGs from DroidReach to the ones from Ghidra and Argus-SAF, extracting the code points detected only by our approach and then keeping only the function entry points. To keep the evaluation sustainable, we considered a subset of the 15 apps and analyzed how to stimulate their JNI methods based on the reports from FlowDroid (S1). We then executed each app under a debugger during an extended usage session, tracking which function entry points found by DroidReach were actually executed. Table 2 shows the results of our experiments. On four applications, we were able to validate a large fraction of the selected code points, bringing high

	# Recovered Mappings		Analysis Time (secs)	
	ARGUS-SAF	DROIDREACH	ARGUS-SAF	DROIDREACH
Static JNI mappings	4,610	4,610	8,542	8,542
Dynamic JNI mappings	765	1,912	259,136	20,345
Both	5,375	6,522	267,678	28,887

Table 3: Resolved JNI mappings during step S2.

DROIDREACH vs	# JNI methods processed by both	% apps for which DROIDREACH has code points than competitor: less	same	more	# code points found by DROIDREACH (ratio w.r.t. competitor): total	total ratio	geo. mean ratio
GHIDRA	5,623	2.8%	9.8%	87.3%	64,818,031	1.24×	1.95×
ARGUS-SAF	4,711	1.3%	7.2%	91.5%	54,901,175	7.58×	5.64×
ARGUS-SAF-MLIB	4,527	1.6%	8.5%	89.9%	51,618,223	6.51×	5.09×

Table 4: ICFG results on methods analyzed using different approaches in S3.

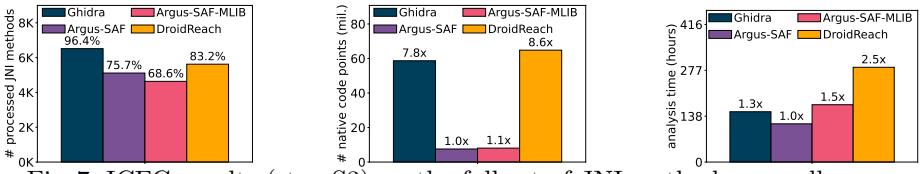

Fig. 7: ICFG results (step S3) on the full set of JNI methods over all apps.

confidence in their correctness. On three other applications, we could not dynamically validate most code points. This is not unexpected as several program behaviors depend on external events (e.g., server-side interactions) and specific usage patterns that cannot always be reproduced. For instance, the considered code points from `com.sec.android.easyMover` are within a library related to USB OTG functionalities in Samsung devices, which we could not stimulate. On `shareit.lite`, the code points are mostly related to C++ exception handling, making them hard to trigger. On `com.amazon.mp3`, we experienced some crashes when inserting the breakpoints in some libraries, allowing us to dynamically validate only a few points. For these three applications, we thus also performed a *manual* static validation by analyzing a subset of their code points with IDA Pro, validating whether the points are *reasonable*, i.e., they are likely reachable within an execution, reporting, however, lower confidence as we did not validate them by running the app. Regarding unvalidated code points, they should not necessarily *all* be seen as false positives, as proving or disproving their correctness is hard: static analyses are often proposed when automatic dynamic analyses cannot exhaustively cover the program code. We provide additional details at [9]. Overall, the effort for this validation was more than 1.5 man-months.

Coarse-grained evaluation. To get a wider evaluation of DROIDREACH, we now consider the full dataset. We focus our discussion on steps S2 and S3. Step S1 brings the same results for all tools as they can use the same analysis framework. Similarly, step S4 can be implemented in the same way for all compared tools.

A crucial challenge tackled by DROIDREACH during step S2 is the identification of mappings between Java native methods and JNI methods. In our dataset, step S1 identifies $7,463$ *reachable* Java `native` methods. This is quite interesting since the total number of Java `native` methods in our dataset is $113,316$: this

suggests that even if an app contains some code, then it may not necessarily execute it. Our manual investigation has confirmed that most apps are integrating third-party frameworks, but they often only use a subset of their functionalities.

Table 3 reports the cumulative number of mappings successfully resolved by DroidReach compared to Argus-SAF, and the cumulative analysis time for these two approaches. Overall, a large fraction (61.8% of $7,463$) of the mappings are statically defined by the apps and can be resolved by both frameworks. The remaining 2853 (38.2%) mappings are defined dynamically though `RegisterNatives`: DroidReach performs significantly better than Argus-SAF on these methods, resolving 2.5× dynamic mappings. DroidReach is also significantly more efficient: the analysis time is reduced by a factor of 9.3×.

Overall, DroidReach has resolved 87.4% of the methods (compared to 72.0% in Argus-SAF), suggesting that 941 (12.6%) methods do not follow the implementation patterns expected by DroidReach. While the symbolic exploration helped resolve 79 methods, it still failed on the 941 unresolved methods. In these cases, the `JNI_OnLoad` function was too complex and the exploration was aborted after a 15-minutes timeout. Users can customize this timeout in DroidReach to possibly increase the accuracy of S2. While there are still some unresolved mappings, the improvement from DroidReach can be quite significant in practice: when considering the Adobe PDF reader (`com.adobe.reader`), all mappings were found exclusively by DroidReach, meaning that Argus-SAF would completely skip any analysis on the native layer for this app.

After finding the JNI mappings for our dataset, we evaluate the effectiveness and performance of DroidReach during the ICFG construction (step S3). Besides DroidReach, we consider: (a) Ghidra, as it is internally used by DroidReach, (b) Argus-SAF, which is the main competitor, and (c) Argus-SAF-MLIB, a variant of Argus-SAF that we developed, which can continue its analysis even in the presence of nested libraries (while the original approach would ignore them). This is important since 340 (68%) apps out of 500, have at least one nested library and some apps may even have a *nested chain* with up to three libraries. Each solution was executed for 2 hours for each application, reconstructing in sequence the ICFGs of the reachable JNI methods. To make a fair comparison, all tools received the same output from step S2.

Since different tools come with different trade-offs in terms of accuracy and performance, leading to a very different number of JNI methods processed within the 2-hour experiment, we first present in Table 4 a pairwise comparison between DroidReach and the other solutions considering the common set of JNI methods which were processed by each pair of frameworks. When considering the $5,623$ JNI methods analyzed by both DroidReach and Ghidra, DroidReach can identify more code points in 87% of the apps. On average for each app, DroidReach finds 1.95× code points than Ghidra. When considering the $4,711$ JNI methods analyzed by both DroidReach and Argus-SAF, our approach identifies more code points in 91% of the apps. On average for each app, DroidReach finds 5.64× code points than Argus-SAF. When considering our custom variant Argus-SAF-MLIB, DroidReach is still more effective.

Figure 7 summarizes the results when considering all JNI methods from all apps: one approach could be less accurate but more efficient on one method, thus having the chance to process more methods within the 2-hour per-app timeout. The left chart shows that Ghidra was able to process more methods than the competitors, followed by DroidReach. The right chart confirms that DroidReach is indeed slower than Ghidra. However, the center chart shows that the number of code points is still in favor of DroidReach. This is expected: DroidReach is performing the same work as Ghidra, plus additional analyses. Hence, its running time is always larger than Ghidra, leading to some apps reaching the 2-hour per-app timeout before processing the entire set of methods.

When comparing DroidReach to Argus-SAF and Argus-SAF-MLIB, the results in Figure 7 show that DroidReach was able to process more methods than these two solutions, detecting $\sim 8\times$ their number of code points but requiring also a larger analysis time. Indeed, Argus-SAF (and Argus-SAF-MLIB) are generally faster (-60%) than Ghidra (and thus DroidReach) for a large set ($\sim 60\%$) of methods but: (a) these solutions are significantly less accurate, identifying fewer code points on this large set, and (b) on the other methods, these solutions fail to generate any ICFG as they reach the timeout or saturate very early the memory (25GB in our experiments) due to path explosion. When attempting to increase the maximum call depth in Argus-SAF-MLIB, we observed a crucial increase in the number of timeouts and out-of-memory events.

Finally, the average analysis time of DroidReach per app was 0.7 hours, 0.4 for Ghidra, 0.3 for Argus-SAF, and 0.4 for Argus-SAF-MLIB.

4.3 Case Studies

Establishing that apps contain vulnerable libraries *does not mean* that such functions constitute necessarily an *immediate* security concern. We present two case studies where DroidReach can be used as an aid in evaluating the *impact* of vulnerable functions. These apps were considered by a previous study [3,2].

Case study A: reachable function. We consider the function `BN_bn2dec` from `libcrypto.so`. This function is used in Amazon Alexa (`com.amazon.dee.app`) and is vulnerable in OpenSSL $\leq$ 1.1.0 (CVE-2016-2182 [16]) with a score of 7.5. DroidReach finds the following path (depicted in Fig. 8 in the Appendix):

- The Java layer loads the `OnStartCommand` function belonging to the `com.here.android.mpa.service.MapService` class. This function loads the (name obfuscated) `a` method from the `com.nokia.maps.SSLCertManager` class.
- This method calls the `x509_NAME_HASH` `native` function that belongs to the `com.nokia.maps.CryptUtils` class, which is statically mapped to the JNI method `com_nokia_maps_CryptUtils_X509_1NAME_1HASH` in `libMAPSJNI.so`.
- The JNI method calls `X509_free`, which is a function from `libcrypto_here.so`, that in turn invokes `ASN1_item_free`, which calls a stripped function at offset `0x5fdd4` (after reversing, it appears to be `asn1_item_combine_free`).
- From this function, the static exploration becomes challenging. There are no direct jumps that connect the function to the target sink. However, DroidReach identifies a reachable offset `0x5ba20` (which would allow for

further exploration towards the sink). A deeper inspection shows that such an offset is *indirectly* calculated and jumped to by accessing dedicated data structures. This is the reason why the connection between the offsets was not immediately evident. Moreover, it demonstrates the capability of DroidReach to identify non-obvious paths that do not involve direct jumps.
- The function at offset `0x5ba20` calls `X509_NAME_ONELINE`, which invokes `i2t_ASN1_OBJECT`. Such a function invokes `OBJ_obj2txt`, which calls `BN_bn2dec`.

After having statically identified a path, we tried to stimulate it dynamically. Unfortunately, reproducing it in the emulator is not easy: besides registering an account and performing several interactions, additional events must be faked to execute the interesting Java class. Nonetheless, we successfully reproduced a similar path in `com.nokia.maps`, which includes the same third-party library. Argus-SAF and Ghidra miss some crucial edges, failing to find the path.

Case study B: unreachable function. The goal of this case study is to ascertain whether there is *no path* to a target vulnerable function. We consider Zoom (`us.zoom.video meetings`) and the function `SRP_VBASE_get_by_user` in `libcrypto.so` (CVE-2016-0798, score 7.8), for which DroidReach could *not* find a path. To validate our claim, we directly patched the native library function with an interrupt `svc 11` instruction to see whether the function was invoked. Then, we tested all possible functionalities. The application showed no signs of a crash, meaning that the target function was not invoked during the execution. Although we *cannot* guarantee that the function will *never* be invoked, we believe that it cannot be executed by a normal user under normal conditions.

5 Limitations

Our current implementation of DroidReach has a few limitations:
- Like Argus-SAF, DroidReach is currently tuned for ARMv7 code. However, from the methodological side, nothing is tight to a specific architecture.
- DroidReach looks for native libraries in standard locations: fixes may be needed in the case of a custom loader or packed libraries.
- DroidReach cannot prove the feasibility of a path, i.e., it does not currently generate the *inputs* or *stimulations* that can reproduce the execution path. Unfortunately, existing static solutions [38] do scale on large apps
- DroidReach represents the structure of the code using ICFGs. This representation may be inadequate in the presence of frameworks that deviate significantly from the traditional Android programming environment.

6 Conclusions

DroidReach statically analyzes Android apps to assess the *reachability* of native functions. Understanding this aspect can be crucial to assess the security of apps featuring libraries with known vulnerabilities, as vulnerable but not reachable functions may not represent an *immediate* threat. Our experiments show that DroidReach can reconstruct more accurate ICFGs than other solutions and that it can be a valuable tool for an analyst during a security evaluation.

Acknowledgments

This work was partially supported by the project PON AIM Research and Innovation 2014–2020 - Attraction and International Mobility, funded by the Italian Ministry of Education, University and Research.

References

1. Abraham, A., Andriatsimandefitra, R., Brunelat, A., Lalande, J., Tong, V.V.T.: Grodddroid: a gorilla for triggering malicious behaviors. In: 10th Int. Conf. on Malicious and Unwanted Software. MALWARE '15 (2015). https://doi.org/10.1109/MALWARE.2015.7413692
2. Almanee, S.: Librarian dataset (2021), https://github.com/salmanee/Librarian
3. Almanee, S., Ünal, A., Payer, M., Garcia, J.: Too quiet in the library: An empirical study of security updates in android apps' native code. In: 43rd IEEE/ACM Int. Conf. on Software Engineering. ICSE '21 (2021). https://doi.org/10.1109/ICSE43902.2021.00122
4. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI Ripping for Automated Testing of Android Applications. In: Proceedings of the 27th IEEE/ACM Int. Conf. on Automated Software Engineering. ASE '12 (2012). https://doi.org/10.1145/2351676.2351717
5. Android: Native libraries (2021), https://developer.android.com/training/articles/perf-jni#native-libraries
6. Android Developers: UI/Application Exerciser Monkey (2021)
7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (2014). https://doi.org/10.1145/2594291.2594299
8. Bello, L., Pistoia, M.: Ares: Triggering payload of evasive android malware. In: Proc. of the 5th Int. Conf. on Mobile Software Engineering and Systems. MOBILESoft '18 (2018). https://doi.org/10.1145/3197231.3197239
9. Borzacchiello, L.: DroidReach (2022), https://github.com/season-lab/DroidReach
10. Borzacchiello, L.: DroidReach Benchmarks (2022), https://github.com/season-lab/DroidReachBenchmarks
11. Borzacchiello, L., Coppa, E., Demetrescu, C.: Fuzzing Symbolic Expressions. In: Proc. of the 43rd Int. Conf. on Soft. Eng. ICSE '21 (2021). https://doi.org/10.1109/ICSE43902.2021.00071
12. Borzacchiello, L., Coppa, E., Demetrescu, C.: FUZZOLIC: mixing fuzzing and concolic execution. Computers & Security (2021). https://doi.org/10.1016/j.cose.2021.102368
13. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In: Proc. of the 2017 ACM on Asia Conf. on Computer and Communications Security (2017). https://doi.org/10.1145/3052973.3053004
14. Chen, T., Zhang, X.S., Guo, S.Z., Li, H.Y., Wu, Y.: State of the art: Dynamic symbolic execution for automated test generation. Future Gener. Comput. Syst. (2013). https://doi.org/10.1016/j.future.2012.02.006

15. Choi, W., Necula, G., Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In: Proc. of the 2013 ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications. OOPSLA '13 (2013). https://doi.org/10.1145/2509136.2509552
16. CVE: CVE-2016-2182 (2016), https://www.cvedetails.com/cve/CVE-2016-2182/
17. Derr, E., Bugiel, S., Fahl, S., Acar, Y., Backes, M.: Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. CCS '17 (2017). https://doi.org/10.1145/3133956.3134059
18. Desnos, A.: Androguard (2021), https://github.com/androguard/androguard
19. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Transactions on Computer Systems (2014). https://doi.org/10.1145/2619091
20. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: Efficient cross-architecture identification of bugs in binary code. In: 23rd Annual Network and Distr. Sys. Sec. Symp. (2016). https://doi.org/10.14722/ndss.2016.23185
21. Gao, J., Yang, X., Fu, Y., Jiang, Y., Shi, H., Sun, J.: Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In: Proc. of the 2018 Eur. Soft. Eng. Conf. and Symp. on the Foundations of Soft. Eng. ESEC/FSE 2018 (2018). https://doi.org/10.1145/3236024.3275524
22. Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., Rinard, M.: Information-Flow Analysis of Android Applications in DroidSafe. In: Proceedings 2015 Network and Distributed System Security Symposium (2015). https://doi.org/10.14722/ndss.2015.23089
23. Gu, Y., Sun, K., Su, P., Li, Q., Lu, Y., Ying, L., Feng, D.: 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). pp. 427–438 (2017). https://doi.org/10.1109/DSN.2017.40
24. Hao, S., Liu, B., Nath, S., Halfond, W.G., Govindan, R.: Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In: Proc. of the 12th Annual Int. Conf. on Mobile Systems, Applications, and Services. MobiSys '14 (2014). https://doi.org/10.1145/2594368.2594390
25. Hwang, S., Lee, S., Kim, J., Ryu, S.: JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In: 2021 43rd Int. Conf. on Soft. Eng. (ICSE '21) (2021). https://doi.org/10.1109/ICSE43902.2021.00151
26. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proc. of the 3rd ACM SIGPLAN Int. Workshop on the State of the Art in Java Program Analysis. SOAP '14 (2014). https://doi.org/10.1145/2614628.2614633
27. Lee, S., Lee, H., Ryu, S.: Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In: 2020 35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '20) (2020). https://doi.org/10.1145/3324884.3416558
28. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In: 37th IEEE Int. Conf. on Soft. Eng (ASE '15) (2015). https://doi.org/10.1109/ICSE.2015.48
29. Li, S., Tan, G.: Finding bugs in exceptional situations of jni programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 442–452. CCS '09 (2009). https://doi.org/10.1145/1653662.1653716

30. Liao, Y., Cai, R., Zhu, G., Yin, Y., Li, K.: MobileFindr: Function Similarity Identification for Reversing Mobile Binaries. In: ESORICS 2018: Computer Security (2018). https://doi.org/10.1007/978-3-319-99073-6_4
31. Ming, J., Xu, D., Jiang, Y., Wu, D.: BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In: 26th USENIX Security Symposium (USENIX Security 17) (2017)
32. NSA: Ghidra. https://ghidra-sre.org/ (2016)
33. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE SP'16 (2016). https://doi.org/10.1109/SP.2016.17
34. Sun, M., Wei, T., Lui, J.C.: TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In: Proc. of the 2016 Conf. on Comp. and Com. Sec. CCS '16 (2016). https://doi.org/10.1145/2976749.2978343
35. Tan, G., Chakradhar, S., Srivaths, R., Wang, R.D.: Safe java native interface. In: In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering. pp. 97–106 (2006)
36. Tan, G., Croft, J.: An Empirical Security Study of the Native Code in the JDK. In: Proc. of the 17th Conf. on Security Symposium. SS '08, USENIX (2008). https://doi.org/10.5555/1496711.1496736
37. Wang, X., Zhu, S., Zhou, D., Yang, Y.: Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In: Proc. of the 33rd Annual Computer Security Applications Conference (2017). https://doi.org/10.1145/3134600.3134601
38. Wei, F., Lin, X., Ou, X., Chen, T., Zhang, X.: Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. CCS '18 (2018). https://doi.org/10.1145/3243734.3243835
39. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Transactions on Privacy and Security (2018). https://doi.org/10.1145/3183575
40. Wong, M.Y., Lie, D.: IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In: Proceedings 2016 Network and Distributed System Security Symposium (2016). https://doi.org/10.14722/ndss.2016.23118
41. Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs. In: Proc. of the 29th ACM SIGSOFT Int. Symp. on Software Testing and Analysis. ISSTA '20 (2020). https://doi.org/10.1145/3395363.3397361
42. Xue, L., Qian, C., Zhou, H., Luo, X., Zhou, Y., Shao, Y., Chan, A.T.: NDroid: Toward Tracking Information Flows Across Multiple Android Contexts. IEEE Transactions on Information Forensics and Security (2019). https://doi.org/10.1109/TIFS.2018.2866347
43. Xue, L., Zhou, Y., Chen, T., Luo, X., Gu, G.: Malton: Towards on-device non-invasive mobile malware analysis for ART. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association (2017)
44. Yan, L.K., Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: 21st USENIX Security Symposium (USENIX Security 12) (2012)

45. Yuanchun Li, Ziyue Yang, Yao Guo, Xiangqun Chen: DroidBot: a lightweight UI-Guided test input generator for android. In: 2017 IEEE/ACM 39th Int. Conf. on Software Engineering (2017). https://doi.org/10.1109/ICSE-C.2017.8

Appendix

Downloads Range	# apps	Avg # Java insns.	Avg # Java `native` methods	Avg native insns	Avg # ARMv7 libs
100K-1M	32	1,228,452	212.81	2,308,222	18
1M-10M	89	1,533,235	229.24	2,654,673	10.42
10M-100M	132	1,849,999	204.86	2,554,607	7.52
100M-500M	201	1,515,141	209.48	2,841,205	7.81
500M-1B	28	1,649,847	235.43	2,511,424	9.82
1B+	18	1,945,265	565.94	2,392,670	5.39

Table 5: Statistics for the apps selected for the evaluation.

Challenge	ID	Description
C1	0	JNI mapping through static name mangling.
C1	1	JNI mapping through static name mangling and method overloading.
C1	2	JNI mapping dynamically defined using the `RegisterNatives` API.
C1	3	JNI mapping dynamically defined using the `RegisterNatives` API but with *clash* in the class name.
C1	4	JNI mapping dynamically defined using the `RegisterNatives` API but without following the Android guidelines.
C1	5	JNI mapping dynamically defined using the `RegisterNatives` API with a hard-to-analyze `JNI_OnLoad`.
C2	6	JNI Method calls a function from a *nested* library.
C3	7	The target function is called at a *high* calldepth.
C3	8	The target function is called after an indirect call (C++ virtual call, lazy initialization).
C3	9	The target function is called after an indirect call (C++ virtual call, callback).
C3	10	The target function is called after an indirect call (function pointer).
C3	11	The target function is called at a *high* calldepth after an indirect call (*small* calldepth after the indirect call).
C3	12	The target function is called at a *high* calldepth after an indirect call (*high* calldepth after the indirect call).

Table 6: Description of the microbenchmarks [10].

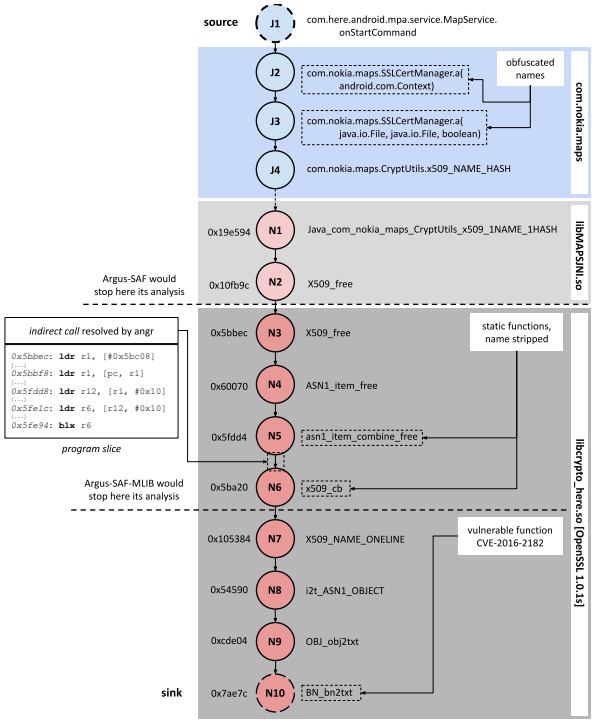

Fig. 8: Path found in the Amazon Alexa app (`com.amazon.dee.app`) that can reach the vulnerable function `BN_bn2txt` from OpenSSL (CVE-2016-2182 [16]).

View publication stats