JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis

Jordan Samhi[1], Jun Gao[1], Nadia Daoudi[1], Pierre Graux[1,3], Henri Hoyez[4], Xiaoyu Sun[2], Kevin Allix[1],
Tegawendé F. Bissyandé[1], Jacques Klein[1]

[1]SnT, University of Luxembourg, Luxembourg, firstname.lastname@uni.lu
[2]Monash University, Australia, firstname.lastname@monash.edu
[3]Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France, firstname.lastname@univ-lille.fr
[4]Technische Universität Kaiserslautern, Germany, firstname.lastname@sms-group.com

ABSTRACT

Native code is now commonplace within Android app packages where it co-exists and interacts with Dex bytecode through the Java Native Interface to deliver rich app functionalities. Yet, state-of-the-art static analysis approaches have mostly overlooked the presence of such native code, which, however, may implement some key sensitive, or even malicious, parts of the app behavior. This limitation of the state of the art is a severe threat to validity in a large range of static analyses that do not have a complete view of the executable code in apps. To address this issue, we propose a new advance in the ambitious research direction of building a unified model of all code in Android apps. The JuCify approach presented in this paper is a significant step towards such a model, where we extract and merge call graphs of native code and bytecode to make the final model readily-usable by a common Android analysis framework: in our implementation, JuCify builds on the Soot internal intermediate representation. We performed empirical investigations to highlight how, without the unified model, a significant amount of Java methods called from the native code are "unreachable" in apps' call-graphs, both in goodware and malware. Using JuCify, we were able to enable static analyzers to reveal cases where malware relied on native code to hide invocation of payment library code or of other sensitive code in the Android framework. Additionally, JuCify's model enables state-of-the-art tools to achieve better precision and recall in detecting data leaks through native code. Finally, we show that by using JuCify we can find sensitive data leaks that pass through native code.

1 INTRODUCTION

Android app analysis has been one of the most active themes of software engineering research in the last decade. Static analysis research, in particular, has produced a variety of approaches and tools that are leveraged in a variety of tasks, including bug detection, security property checking, malware detection, or empirical studies. The widely-used state-of-the-art approaches, such as FlowDroid [5], develop analyses that focus on the Dex bytecode in apps. Unfortunately, recent studies [1, 38, 50, 62, 69] have shown that malware authors often build on native code to hide their malicious operations (e.g., private data leak) or to implement sandbox evasion [64].

The need to account for native code within Android apps is becoming urgent as the usage of native code is growing within both benign and malicious apps. Our empirical investigation on apps from the AndroZoo [3] repository reveals that, in 2019, up to 62.9% of collected apps included native code within their packages. Yet, native code is scarcely considered in app security vetting [2, 69]. In the majority of static [5, 13, 16, 25, 34, 47, 78], dynamic [6, 49, 79] and machine learning based techniques [48, 55], native code is overlooked since it presents several challenges.

When researchers propose techniques to address native code such as with *JN-SAF* [69], *DroidNative* [2], *NativeGuard* [60], *TaintArt* [61] and others [1, 15, 31, 50], the integrated analyses (e.g., for taint tracking, native entry-point detection and machine learning feature extraction) are generally ad-hoc. Indeed, these works develop custom techniques to bridge native code and bytecode, typically by combining results of separate analyses of bytecode and native code. Therefore, they do not yield an explicit unified model of the app to which generic analyses can be applied to explore bytecode and native code altogether.

Our work aims to fill the gap in whole-app analysis by researching means to build a unified model of Android code. We propose JuCify, a step toward building a framework that breaks bytecode-native boundaries for Android apps and therefore copes with a common limitation of static approaches in the literature. To the best of our knowledge, JuCify is the first approach that targets the unification of Android bytecode and native code into a unified model and is instantiated in a standard representation [36]. We target the *Jimple* [67] Intermediate Representation as support for JuCify unified model. Jimple is the internal representation in the widely-used Soot framework and is indeed the representation that is considered in a large body of static analysis works [36]. By supporting Jimple, JuCify provides the opportunity for several analyses in the literature to readily account for native code.

This paper. JuCify is a multi-step static analysis approach that we implement as a framework for generating a unified model of apps taking into account native code. It ❶ relies on symbolic execution to retrieve invocations between both the Dex bytecode and the native worlds, ❷ pre-computes native call-graph, ❸ merges Dex bytecode and native call-graphs, and ❹ populates newly generated functions with heuristic-based defined Jimple statements using code instrumentation.

The main contributions of our work are as follows:

- We propose JuCify, an approach to build a unified model of Android app code for enabling enhanced static analyses. We have implemented JuCify to produce the Jimple code that unifies bytecode and native code within an app package;
- We conduct an assessment of the JuCify yielded model. We show that JuCify can significantly enhance Android apps' call-graphs. JuCify connects previously unreachable methods in Android apps call-graphs;
- We evaluate the unified model of app code in the task of data flow tracking. We show that JuCify can significantly boost the precision of the state-of-the-art FlowDroid, from 0% to 82% and its recall from 0% to 100% on a new benchmark targeting bytecode-native data flow tracking;
- We evaluate JuCify on a set of real-world Android apps and show that it can augment existing analysers, enabling them to reveal sensitive data leaks that pass through the native code which were previously undetectable.
- We release our open-source prototype JuCify to the community as well as all the artifacts used in our study at:

https://github.com/JordanSamhi/JuCify

The remainder of this paper is organized as follows. We first introduce background notions and motivate our work in Section 2. In Section 3, we present our JuCify approach. We evaluate JuCify in Section 4. In Sections 5 and 6, we present the limitations and the threats to validity of the current state of our approach. Finally, we overview the related work in Section 7 and conclude in Section 8.

2 BACKGROUND & MOTIVATION

Java and Kotlin are the two mainstream programming languages that support the development of Android apps. Their programs are compiled into Dex bytecode and included within app packages (in the form of DEX files). Nevertheless, thanks to *Java Native Interface* [23], native code functionalities are accessible in Android apps. They come in binary (e.g., .so share library) files compiled from input programs written in C/C++ for instance.

2.1 Java Native Interface (JNI)

JNI is an implementation of the *Foreign Function Interface* (FFI) [19] mechanism that allows programs written in a given language to invoke subroutines written in another language. JNI allows both Java to native and native to Java invocations.

2.1.1 Java to native code. Listing 1 presents an example where JNI capabilities are used to call a native function (here written in C++) from Java. First, a relevant Java method is defined with the keyword native (line 4). We will refer to it as a *Java native method*. Then, its

```
1   /*** JAVA WORLD ***/
2   public class MainActivity extends Activity {
3     static {System.loadLibrary("native-lib");}
4     public native String nativeGetImei(TelephonyManager tm);
5     @Override
6     protected void onCreate(Bundle savedInstanceState) {
7       super.onCreate(savedInstanceState);
8       TelephonyManager tm = (TelephonyManager) getSystemService("phone");
9       String imei = nativeGetImei(tm);
10      Log.d("IMEI", "" + imei);
11    }
12    public void malicious() {/* malicious code */}
13  }
14  /*** C++ WORLD ***/
15  JNIEXPORT jstring JNICALL
16  Java_MainActivity_nativeGetImei(JNIEnv *env, jobject thiz, jobject tm) {
17    jclass c = (*env).GetObjectClass(tm);
18    jmethodID m = (*env).GetMethodID(c, "getImei", "()Ljava/lang/String;");
19    jstring i = (jstring)(*env).CallObjectMethod(tm, m);
20    c = (*env).GetObjectClass(thiz);
21    m = (*env).GetMethodID(c, "malicious", "()V");
22    (*env).CallObjectMethod(thiz, m);
23    return i;
24  }
```

Listing 1: Code illustrating how an app can trigger native code. (Methods and code are simplified for convenience)

corresponding native function is registered to set up the mapping between them. Such a registration can be:

Static - the native function definition follows a naming convention based on specific JNI macros. For example, the Java native method nativeGetImei (line 4) corresponds to a native function named Java_com_example_nativeGetImei in C++ (line 16).

Dynamic - developers can arbitrarily name their native functions (in C++) as shown in Listing 2 (lines 10-13), but must inform JNI about how to map them with Java native methods. Thus, developers ❶ first map Java native methods to their counterpart native functions by using specific JNINativeMethod structures (lines 14-16 in Listing 2); ❷ overload a specific JNI Interface function [18], JNI_OnLoad, to register the mapping (lines 17-24 in Listing 2); and ❸ invoke RegisterNatives in JNI_OnLoad which will be called by Android VM (line 22 in Listing 2).

2.1.2 Native to Java. With JNI, developers can create and manipulate Java objects within the native code (e.g., written in C++). The fields and methods of Java objects are also accessible from the native code and can be invoked using specific JNI *Interface* functions. Eventually, likewise Java reflection [14], i.e., using strings to get methods and classes, the developer can invoke the Java methods (e.g., lines 17-19 in Listing 1).

Note that Listings 1 and 2 illustrate the interaction between Java and C++. However, JuCify, the approach proposed in this paper, works at the apk level. Therefore, the invocations are between bytecode and compiled native code.

2.2 Motivating Example

Binary static code analysis is in itself a challenge [43] since the compiled code is hard to represent for appropriate investigation [30].

Although current state-of-the-art Android static code analysis approaches are sophisticated [5, 34, 56, 70, 76], most of the time they overlook native code, with only a few of them considering it [31, 69].

With a simple example illustrated in Listing 1, we make the case that native code should be considered in static analysis approaches.

```
1   /*** JAVA WORLD ***/
2   public class MainActivity extends Activity {
3       static {System.loadLibrary("native-lib");}
4       public native String nativeMethod();
5       @Override
6       protected void onCreate(Bundle b) {nativeMethod();}
7   }
8   /*** C++ WORLD ***/
9   JNIEXPORT jstring JNICALL
10  jstring arbitrary_name(JNIEnv *e, jobject thiz) {
11      std::string str = "str";
12      return e->NewStringUTF(str.c_str());
13  }
14  static const JNINativeMethod m[] = {
15      {"nativeMethod", "()Ljava/lang/String;", (jstring*)arbitrary_name}
16  };
17  JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved){
18      JNIEnv* e = NULL;
19      if(vm->GetEnv((void**)&e, JNI_VERSION_1_4) != JNI_OK){return -1;}
20      jclass c = e->FindClass("com/example/MainActivity");
21      if (!c){return -1;}
22      if(e->RegisterNatives(c, m, sizeof(m)/sizeof(m[0]))){return -1;}
23      return 1;
24  }
```

Listing 2: Dynamic native function registration example. (Methods and code are simplified for convenience)

First, in the *onCreate()* method of the main `Activity`, a `String` is retrieved on line 9 from the method *nativeGetImei*, then this `String` is used as a parameter to the method *Log.d()*. From the point of view of taint tracking, there is a flow from the *getImei()* method (source) to the *Log.d()* method (sink). However, most state-of-the-art approaches will miss this flow due to technical limitations since the method *nativeGetImei* is not analyzed. Therefore the variable `imei` is not tainted, and the flow is not detected.

Second, the method *malicious()* (line 12) is never called in the Java code, thus, it will not appear in the call-graph since it is considered as *unreachable*. Hence it will not be analyzed, causing existing tools to fail to detect potential malicious code in the method.

Let us consider Figure 1, which presents the expected call-graph of this example. The current state-of-the-art approaches, such as [5, 17, 34, 35, 70], generally analyze the green nodes which are reachable from an entry point. However, the red nodes will only be considered by approaches able to analyze the native code. Approaches trying to overcome the challenge of native code analysis in Android apps, already exist (e.g., [31, 50, 52, 69]). However, they focus on specific analyses and propose custom solutions to bridge bytecode and native code. In contrast, in this paper, we aim at offering an explicit unified model of Android apps to which generic analysis could be applied to explore altogether bytecode and native code.
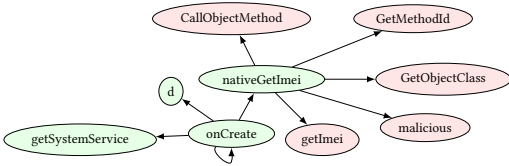

Figure 1: Unified call-graph representation for the code in Listing 1: Green nodes represent reachable nodes of existing static approaches, while Red ones represent the nodes unreachable with most of the existing static approaches

3 APPROACH

For a given Android app, JuCIFY aims to unify its Dex bytecode and native code into a unified model and instantiate this model in the Jimple representation (i.e., the intermediate representation of the popular Soot framework). In this section, we will first detail the overall JuCIFY conceptual approach, and then we will briefly present how we instrument the app to approximate the native behavior. However, due to space constraints, we will not present all technical details related to Jimple. We invite the interested reader to consider all our publicly-shared artifacts on the project Github repository[1]. JuCIFY implementation is fully open-sourced.

3.1 Call Graph as Unified Preliminary Model

To explain the overall functioning of JuCIFY, we will restrict our explanations to the notion of Call Graph (CG). A CG can be defined as $CG = (V, E)$, where V is a set of vertices representing functions, and $E \subseteq \{(u, v) \mid u, v \in V\}$ is a set of edges such as $\forall (u, v) \in E$, there is a call from u to v in the program.

JuCIFY is a multi-step static analysis framework whose overall architecture is depicted in Figure 2. First, a submodule called NATIVEDISCLOSER constructs the native callgraph and extracts the mutual invocations between bytecode and native code. Then, native callgraph is pruned and prepared to be SooT-compliant before being merged with the bytecode callgraph. Eventually, both callgraphs are unified thanks to information related to the bytecode-native method invocations. In the following with give more details about the different steps of our approach.

Step 0: Native Call Graph Construction

Native program call-graph construction is not trivial [21]. In fact, a large body of work tackled this problem and proposed several solutions to find function boundaries [21, 26, 44]. In this work, the call-graphs native libraries in Android apps are generated by ANGR [57], a well-known binary analysis framework, which is wrapped into our submodule NATIVEDISCLOSER.

Step 1: Bytecode-Native Code Invocations Extraction

This step is performed over 4 sub steps: ❶ Retrieve bytecode methods information; ❷ Extract entry method invocations (i.e., bytecode to native); ❸ Track native function calls and extract exit method invocations.

Step 1.1: Methods info extraction is a straightforward task that extracts information of bytecode methods, such as the class of a method, method signature. This step aims to complete the signature information required to perform the method invocations extraction task for statically registered functions. We perform this task by relying on ANDROGUARD [4].

Step 1.2: Entry method invocations extraction: An entry method invocation is a native method invocation from the bytecode (i.e., a bytecode-to-native "link"). As described in Section 2.1, for such an invocation, we need to match a "Java native method" (i.e., a method declared in Java with the `native` keyword, also called *entry method*) and an *entry function* (i.e., the counterpart native function). To perform this task, we have to take care of both static and dynamic registrations. The statically registered functions can be easily spotted via their naming conventions. However, as dynamic registration relies on JNI *interface* function calls, more sophisticated techniques are required. In our case, we rely on symbolic execution.

CG = Call-Graph, ● ● = Native CG Node, ● = Bytecode CG Node, ● = Previously Unreachable Bytecode CG Node
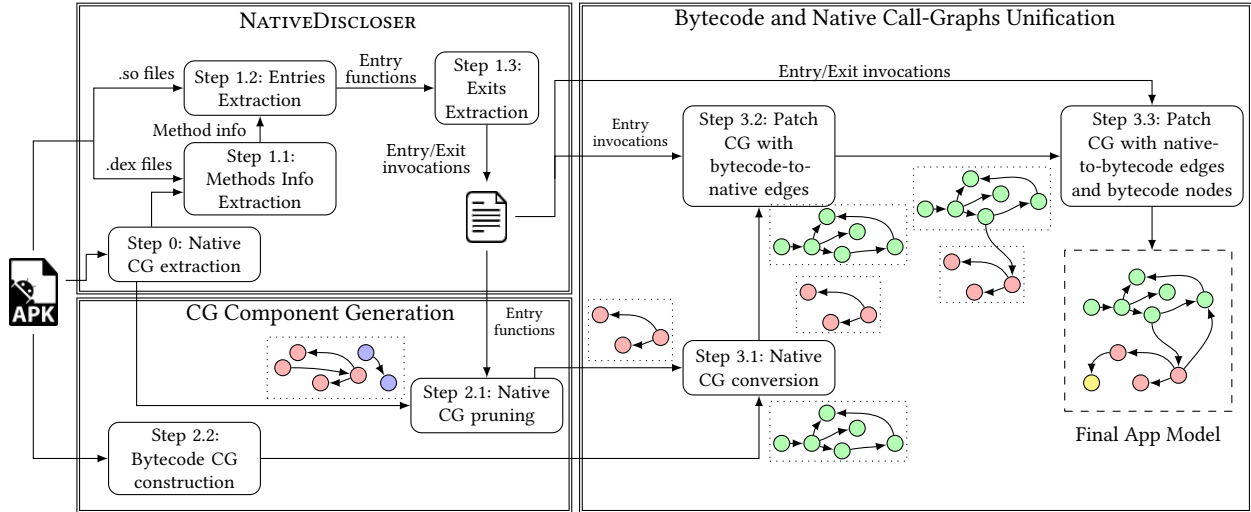

Figure 2: Overview of the JuCify Approach from the Angle of Call-Graph Construction

From a more technical point of view, NativeDiscloser takes as input the library (i.e., .so) files of an apk and the method information from the previous step. It first scans the symbol table of each binary to search for (1) statically registered native functions and (2) the JNI_OnLoad function for the case of dynamically registered functions. Then, if JNI_OnLoad exists, this function is symbolically executed to further detect dynamically registered native functions.

For symbolic execution, NativeDiscloser relies on Angr [57].

Step 1.3: Exit method invocations extraction: We are looking for the invocations of a bytecode method from the native code. We call *exit method* this bytecode method. In Section 2.1.2, we explained that this exit method is called by invoking certain JNI *Interface* functions in a chained manner. Collecting information related to this chain of JNI function invocations is challenging.

In practice, to overcome this challenge, NativeDiscloser executes all the entry functions acquired from step 1.2 symbolically to search for the exit method invocations and set up the relation mapping between entry and exit method invocations.

Furthermore, exit methods could be invoked deep down in a native function chain. However, the symbolic execution is not aware of the boundaries between native functions. Hence, we implemented a tracking mechanism during the search of exit methods. We rely on the starting address of each native function obtained from the native call-graph to maintain a stack of native functions and push a new function into the stack when its starting address is reached. Popping a function from the stack is triggered by the arrival of the return addresses of native functions, which can be obtained from a certain register or memory location based on architecture specifications (e.g., link register *LR* for *ARM*) during entering a native function. This allows us to know from which native function an exit method invocation occurs.

Step 2: CG Components Generation

Step 2.1: Native CG pruning. Since in .so libraries not all the functions are necessarily called in an app, we rely on a strategy to only keep relevant callgraph parts. To do so, we prune the obtained

native call-graphs constructed in Step 0 with the help of the entry functions passed in from Step 1. We only keep the sub-graphs starting from the entry functions (with all successor nodes) since the remaining parts will not be reachable from the bytecode.

Step 2.2: Bytecode CG construction. Our approach also requires the bytecode call-graph. For this purpose, we use FlowDroid [5] (itself based on Soot [66]) which leverages an advanced modeling of app components' life-cycle.

Step 3: Bytecode and native call-graphs unification

Step 3.1: Native CG conversion. In practice, the target is to load both native and bytecode call-graphs in Soot. Although this is straightforward for the bytecode call-graph, the native call-graph requires a conversion step to fit with Soot technical constraints. Once loaded, the sets of nodes and edges of both call-graphs are merged, but the call-graphs are not yet connected together.

Step 3.2: Patch CG with bytecode-to-native edges. Then, according to the entry invocations obtained from Step 1.2, edges between entry methods (in bytecode) and their counterpart entry functions (in native code) are added.

Step 3.3: Patch CG with native-to-bytecode edges and bytecode nodes. Finally, with the information of exit invocations and the relations with entry invocations from Step 1.3, edges between native functions to exit methods are added. This step allows uncovering previously unreachable bytecode callgraph nodes.

3.2 From CG to Jimple for a Unified Model

A call-graph is a useful model, but it is still limited because it does not contain enough information to perform static analysis (e.g., data flow analysis). Indeed, important information such as the statements present in each method is missing (i.e., the control flow graph (CFG)). A tool such as FlowDroid provides the CFG for each bytecode method where the method behavior is represented with Jimple statements. We will now explain how JuCify adds Jimple statements in specific native functions in a best-effort mode. After

this step, for a given APK, we obtain the Jimple representation of the apk with both bytecode and native code unified.

Native functions generation: JuCify relies on a *DummyBinaryClass* whose purpose is to incorporate any newly imported native function in the Soot representation. For each native function in the native call-graph, JuCify generates a new method in the *DummyBinaryClass* with appropriate signatures.

Bytecode method statements instrumentation: JuCify generates bytecode-to-native call-graph edges. It also has to replace the initial call to the native method at the statement level with a call to the newly generated native function. JuCify takes care of the returned value and the parameters to not fool any analysis based on the new built model.

Native function statements generation: There is no bijection between native code and Jimple code [67]. Moreover, bytecode and native code manipulate different notions (e.g., pointers) that cannot be translated directly. Therefore, we have to use heuristics based on the information at our disposal to put a first step toward reconstructing native function behavior.

Let us consider a native function named foo() containing at least one invocation to a bytecode method m. As explained in Section 3.1, the first step of JuCify aims to collect information about bytecode methods (full signature). Thanks to this, we can approximate the parameters used by m as well as its return values.

More specifically, in Listing 3, we detail the steps JuCify implements to populate the native function foo() that calls a bytecode method m. Let consider m is defined in a Java class named MyClass. In line 1, JuCify starts with the empty method foo(). Then:

Step 1 in Listing 3: If the bytecode method m should return a value, JuCify generates a new local variable with the same type as the method's return type (line 4).

Step 2 in Listing 3: JuCify generates the declaration of a variable of type MyClass, the class in which m is defined (line 8). In line 9, JuCify creates a new MyClass instance (if there is not one usable as a base for the bytecode call).

Step 3 in Listing 3: Regarding the parameters that should be used for the invocation of m, JuCify scans foo() for local variables and parameters whose types match the types of the parameters of m. If, for a given type, no local variable, nor parameter of foo() is found, JuCify generates one (e.g., line 15). Then, it generates all the permutations of these variables with a given length (i.e., the number of parameter of m) and retains only those matching the types' order of the parameters of m (($i1, s$), and ($i2, s$) in Listing 3). Each retained permutation corresponds to a possible call to the bytecode method in the native function as an over-approximation. Nevertheless, these calls cannot be generated sequentially since they correspond to different realities. Hence, we rely on opaque predicates (*if* statements whose predicate cannot be evaluated statically) so that each control flow path is considered identically (lines 16-17).

Step 4 in Listing 3: If the native function returns a value (from the signature of foo()), JuCify should generate return statements. To do so, it operates as for m. It relies on opaque predicates. Indeed, first, JuCify scans the body of the current native function to find any local variable corresponding to the type of the return value (even those newly generated local variables that could be returned). If no variable is found, JuCify generates such a variable. Else, for

```
1   public boolean foo(int i1, int i2, boolean b1){}
2   // STEP 1
3   public boolean foo(int i1, int i2, boolean b1){
4       boolean b2;
5   }
6   // STEP 2
7   public boolean foo(int i1, int i2, boolean b1){
8       boolean b2; MyClass jc;
9       jc = new MyClass();
10  }
11  // STEP 3
12  public boolean foo(int i1, int i2, boolean b1){
13      boolean b2; MyClass jc; String s;
14      jc = new MyClass();
15      s = new String();
16      if(opaque_predicate) {b2 = jc.m(i1, s);}
17      else if(opaque_predicate) {b2 = jc.m(i2, s);}
18  }
19  // STEP 4
20  public boolean foo(int i1, int i2, boolean b1){
21      boolean b2; MyClass jc; String s;
22      jc = new MyClass();
23      s = new String();
24      if(opaque_predicate) {b2 = jc.m(i1, s);}
25      else if(opaque_predicate) {b2 = jc.m(i2, s);}
26      if(opaque_predicate) {return b1;}
27      else if(opaque_predicate) {return b2;}
28  }
```

Listing 3: JuCify's process to populate native functions

each of found local variables, JuCify generates return statements with opaque predicates so that each path can be equally considered (lines 26-27 in Listing 3).

Finally, JuCify yields a unified model of Android apps on which analysts can perform any static analysis.

4 EVALUATION

We investigate the following research questions to assess the importance of our contributions:

RQ1: What is the proportion and evolution of native code usage in both real-world benign and malicious apps?

RQ2: To what extent our bytecode-native invocation extraction step (named NativeDiscloser) yields better results than the state-of-the-art ?

RQ3: Can JuCify boost existing static data flow analyzers?

RQ4: How does JuCify behave in the wild? We address this question both at the quantitative and qualitative levels:
- **RQ4.a:** To what extent can JuCify augment apps' call-graphs and reveal previously unreachable Java methods?
- **RQ4.b:** Can JuCify reveal previously unreachable data leaks that pass through native code in real-world apps?

4.1 RQ1: Native code usage in the wild

This section presents general statistics about the usage of native code in both benign and malicious Android apps. We also perform an evolutionary study of this usage.

Dataset: We rely on the AndroZoo repository [3] to build ❶ a dataset of 2 641 194 benign apps (where we consider an app as benign if no Antivirus in VirusTotal [65] has flagged it - score 0); and ❷ a dataset of 174 342 malicious apps (where we consider an app as malicious when at least 10 Antivirus engines in VirusTotal have flagged it). Both datasets contain all the apps from 2015 to 2020 that we were able to collect from AndroZoo with the mentioned VirusTotal constraints.

Empirical study: Android programming with the Native Development Kit (NDK) suggests developers to integrate native libraries

(i.e., `.so` files) whose code can be invoked from the Java world. Therefore, to study the extent of native code usage in Android apps, as a preliminary study, for each app, we check if it contains at least one .so file in its APK file. However, since native libraries can be present in apps but never used, we also check for each app if Java native methods (cf. Section 2.1) are declared in the bytecode.

	Goodware			Malware		
	# Apps	w/ .so files	w/ native methods	# Apps	w/ .so files	w/ native methods
2015	632 279	220 934 (34.9%)	216 329 (34.2%)	89 542	65 221 (72.8%)	63 275 (70.7%)
2016	1 103 899	405 209 (36.7%)	404 357 (36.6%)	48 358	35 601 (73.6%)	34 240 (70.8%)
2017	277 690	143 463 (51.7%)	143 183 (51.6%)	15 141	8742 (57.7%)	8539 (56.4%)
2018	304 746	191 491 (62.8%)	184 447 (60.5%)	10 890	8415 (77.3%)	8018 (73.6%)
2019	179 309	113 433 (63.3%)	112 873 (62.9%)	9773	8993 (92.0%)	8311 (85.0%)
2020	143 271	81 755 (57.1%)	81 111 (56.6%)	638	446 (69.9%)	274 (42.9%)
Total	2 641 194	1 156 285 (44%)	1 142 300 (43%)	174 342	127 418 (73%)	122 657 (70%)

Table 1: Number and proportion of Android apps that contain at least one "`.so` file" / "Java native method" (w/ = with).

Results of our empirical study are presented in Table 1. They indicate that, overall, 1 156 285 benign apps (i.e., 44%) contain at least one .so file, and 1 142 300 (i.e., 43%) contain at least one Java native method declaration. This means that 98.8% of apps with native libraries contain Java native method declaration in their bytecode. Regarding malware, 127 418 (i.e., 73%) of apps contain native libraries and 122 657 (i.e., 70%) Java native method declarations. Hence, 96.3% of malware with native libraries contain Java native method declarations. Overall, these results show that native code is, in proportion, more used in malicious apps.

Regarding usage evolution in benign apps, the rate increases until 2018 to reach a plateau at around 60%. The trend regarding malware is much more erratic (with sharp decreases in 2017 and 2020). However, for each year, malicious apps use significantly more native code than benign apps.

> **RQ1 answer:** Native code is definitely pervasive in Android apps. While both benign and malicious code leverage native code, native invocations are substantially more common in malware (70% vs. 43%).
>
> These results indicate that ignoring native code is a serious threat to validity in Android static code analysis.

4.2 RQ2: Bytecode-Native Invocation Extraction Comparison

Identifying native-to-bytecode and bytecode-to-native code invocations are key steps towards code unification. Our objective is to estimate to what extent the corresponding building block in JuCify is effective against a benchmark and against the state of the art.

Native to Bytecode: Fourtounis *et al.* [15] proposed an approach to detect exit invocations (i.e., native to bytecode invocations, c.f., Section 3.1) in native code via binary scanning. Their tool named Native-Scanner [46] has been developed as a plugin of a framework called DOOP [45]. Briefly, their tool scans binary files for string constants that match Java method names and Java VM type signatures and follows their propagation. In this way, they consider all matches as new entry points back to bytecode.

To compare our NativeDiscloser with Native-Scanner, we developed and released 16 benchmark apps. All these apps are executable Android apps and have been tested on a *Nexus* 5 phone with Android version 8.1.0. We design these apps to cover different

situations such as dynamic/static registration, chained invocations in native functions, parameter passing via structures and classes, string accessing via arrays and function returns, string obfuscation, etc. Table 2 presents the results obtained with both tools.

TP = True Positive, FP = False Positive, FN = False Negative

Benchmark	Native-Scanner			NativeDiscloser		
	TP	FP	FN	TP	FP	FN
bm_{1-5}, bm_7, bm_{10-12}[†]	1	0	1	2	0	0
bm_6, bm_8	1	0	2	3	0	0
bm_9	0	0	2	0	0	2
bm_{13}	0	1	5	5	0	0
bm_{14}	1	4	1	2	0	0
bm_{15}, bm_{16}	1	0	1	1	0	1
Precision	73.68%			100%		
Recall	37.84%			89.19%		

[†] due to space limitation, we put together apps with same results. E.g., NativeDiscloser detects 3 TP for each app bm_6 and bm_8.

Table 2: Comparison of Tools

These results show that Native-Scanner misses a high number of exit invocations. We realized that Native-Scanner seems not to consider Android framework APIs (the tool misses the API invocations in all benchmark apps). Note that Native-Scanner is not specific to Android. This could explain why it does not consider Android APIs. The tool is also challenged by constant string obfuscation (app bm_9), which is also the case for NativeDiscloser. $bm14$ implements fake method string constants in the native part. For this app, we can observe the over-estimation of Native-Scanner (i.e., a high number of false-positive) while NativeDiscloser is not affected. Finally, NativeDiscloser also failed with string constants passing via arrays and function returns as implemented in $bm15$ and $bm16$ respectively. Limitations of Angr could cause this in parsing pointer of pointers. Overall, compared to Native-Scanner, NativeDiscloser obtains significantly higher precision and recall.

Bytecode to native: We were unable to compare NativeDiscloser with Native-Scanner. Unlike our tool, Native-Scanner does not investigate (1) bytecode to native entry invocations and (2) the relations mapping between entry and exit invocations.

Note, however, that on our benchmark of 16 apps, NativeDiscloser yields 100% precision in finding both the entry invocations and the entry-to-exit relations and achieves a recall of 95.59% and 89.19% respectively.

> **RQ2 answer:** Compared to the state-of-the-art Native-Scanner, our NativeDiscloser extracts exit invocations with better precision and recall. Besides, it can provide extra information, including entry invocations (i.e., bytecode to native invocations) and the relations with exit invocations, which is essential to generate comprehensive call-graphs.

4.3 RQ3: Can JuCify boost static data flow analyzers?

In Section 3, we described how JuCify could approximate the behavior of native functions based on the information retrieved from signatures, parameters, return type, and bytecode methods called from native code via JNI. In this RQ, we check if this first step approximation helps perform advanced static analyses such as data

leak detection on a well-defined benchmark. We will assess the capability of JuCify on real-world applications in RQ4.

The benchmark that we built for RQ3 contains 11 apps that we plan to integrate into DroidBench, an open test suite that contains hand-crafted Android apps to assess taint analyzers. Among these apps, 9 contain a flow going through the native world, and 2 do not contain any data flow (to detect potential false positives). Then, we apply the state-of-the-art FlowDroid taint-analysis engine before and after applying JuCify in our benchmark apps, to show that FlowDroid can, likewise in [56], be *boosted*. FlowDroid detects paths from well-defined sources (e.g., getDeviceId()) and sinks (e.g., sendTextMessage()) methods in Android apps.

Benchmark construction: We identified 4 cases on which we built our 11 benchmark apps to assess the ability of tools in detecting data leak via native code:

a) Getter: Source in native code and sink in Java code
b) Leaker: Source in Java code and sink in native code
c) Proxy: Source in Java code and sink in Java code
d) Delegation: Source in native code and sink in native code

Note that "Source/Sink in native code" means that the call to a sensitive method is actually performed in native code, but the sensitive method is always a method from the Android framework accessed with JNI (e.g., calling with JNI the getDeviceId() from the native code). For each of these cases, at least one step happens in native. Figure 3 illustrates these four cases. The red dots represent tainted information from a source method, and the red arrows represent how this information flows in the program. The *Getter* use-case allows developers to get sensitive data from the native code to leak it in the Java world. The *Leaker* use-case allows developers to get sensitive data from the Java world to leak it in the native world. Regarding the *Proxy* use-case, the sensitive information is retrieved in the Java world, sent to the native world to "break" the flow, and sent back to the Java world to be leaked. Concerning the *Delegation* use-case, a simple native function is called from the Java world, and the sensitive information is retrieved and leaked in the native world.

Our benchmark apps has been built, upon these four cases that we identified, to be representative of these cases, with combination of multiple cases.

Results: Table 3 provides the results of our experiments. FlowDroid is clearly limited and not designed to handle native code. Therefore its inferior performances are not surprising. Indeed, FlowDroid gets a precision and recall of 0% on this benchmark.

Nevertheless, we can see that after applying JuCify, FlowDroid performance is significantly boosted. Indeed, it can detect all the leaks present in the benchmark, hence achieving a recall score of 100%. Regarding apps *getter_string* and *leaker_string*, FlowDroid reports for both of them a false positive alarm leading to a precision of 82% on this benchmark. In these apps, a string is sent outside the apps, not sensitive data. This is easily explained by the fact that when JuCify reconstructs the native function's behavior, it uses opaque predicates to approximate what variable can be returned by the current function given its signature. Therefore, there is a path in which the sensitive data is considered, whereas it is not leaked.

● = Tainted Information, ⌁⌁⌁▸ = Call Edge, → = Taint Propagation,
● = Method entrypoint

(a) Getter   **(b) Leaker**

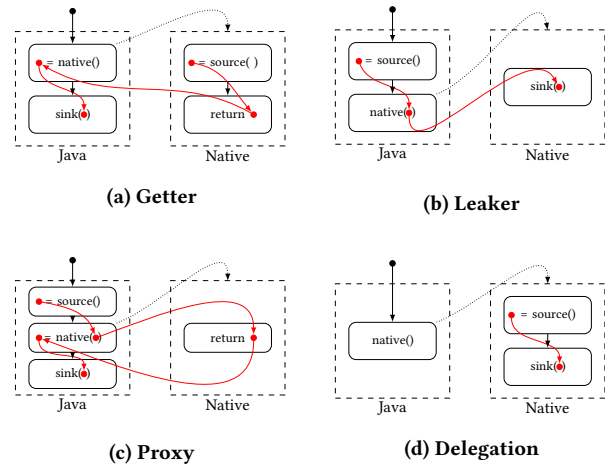**(c) Proxy**   **(d) Delegation**

Figure 3: Four propagation scenarios through native code

Ⓧ = true-positive, ★ = false-positive, ◯ = false-negative

Test Case	Leak	FlowDroid	JuCify
getter_imei	●	◯	Ⓧ
leaker_imei	●	◯	Ⓧ
proxy_imei	●	◯	Ⓧ
delegation_imei	●	◯	Ⓧ
getter_string	○		★
leaker_string	○		★
proxy_double	●	◯	Ⓧ
delegation_proxy	●	◯	Ⓧ
getter_leaker	●	◯	Ⓧ
getter_proxy_leaker	●	◯	Ⓧ
getter_imei_deep	●	◯	Ⓧ
Sum, Precision, Recall			
Ⓧ, higher is better		0	9
★, lower is better		0	2
◯, lower is better		9	0
Precision $p = Ⓧ/(Ⓧ+★)$		0%	82%
Recall $r = Ⓧ/(Ⓧ+◯)$		0%	100%
F_1-score = $2pr/(p+r)$		0%	90%

Table 3: Results of data leak detection through native code in bench apps. FlowDroid column represents the results of running FlowDroid alone. JuCify column represents the results of running FlowDroid after applying JuCify

RQ3 answer: Jucify is essential for boosting state-of-the-art static analyzers such as FlowDroid to take into account native code. On our constructed benchmark, FlowDroid, which failed to discover any leak, is now able to precisely identify leaks in a high number of samples (F1-score at 90%).

4.4 RQ4: JuCify in the wild

In this section, we evaluate JuCify in the wild from two points of view: ❶ a quantitative assessment in section 4.4.1; and ❷ a qualitative assessment in section 4.4.2.

4.4.1 RQ4.a: To what extent can JuCify augment apps' call-graphs and reveal previously unreachable Java methods?

To assess to what extent call-graphs are augmented by JuCify, we applied it on two sets of Android apps: 1) 1000 benign apps; 2) 1000 malware. Note that we only selected apps that contain at least one .so file. The results reported concern apps for which JuCify succeeded to make call-graph changes. The reasons for which there are apps without changes is related to the absence of bytecode-to-native links (i.e., for 559 goodware and 384 malware) and/or JuCify reaching the 1h-timeout (i.e., for 15 goodware and 51 malware).

Number of nodes and edges in call-graphs: We first report the average number of nodes (i.e., the number of methods) and edges (i.e., the number of potential invocations) in the call-graphs obtained before and after having applied JuCify.

The call-graph augmentations brought by JuCify are visible in Table 4. Column *# apps* represents the number of apps for which JuCify made callgraph changes, i.e., they did not reach the timeout and contained bytecode-native links. We notice that about half of the apps' call-graphs are impacted by JuCify (426 and 565 for goodware and malware respectively). We then notice that the number of nodes and edges added by JuCify is higher for goodware than for malware: 270 vs. 197 on average per app for nodes, and 778 vs. 446 for edges. This shows that classical static analyzers that do not take into account the native code, overlook a significant amount of nodes and edges in their call-graph.

		Before JuCify		After JuCify		Difference	
	# apps	# Nodes	# Edges	# Nodes	# Edges	Added Nodes	Added Edges
Goodware	426	4515	18 287	4784	19 065	270 (+5.9%)	778 (+4.2%)
Malware	565	3056	14 266	3253	14 712	197 (+6.4%)	446 (+3.1%)

Table 4: Average numbers of nodes and edges before and after JuCify on 426 goodware and 565 malware

Number of binary functions in the augmented call-graph: Newly added nodes can be explained by the binary functions (i.e., functions in the native code part) that are now considered in the unified call graph yielded by JuCify. Figure 4 details the distributions of the number of binary functions for both datasets. We notice that benign apps tend to have more added binary function nodes (median = 172, and mean = 269.7) in the call-graph than malicious apps (median = 162, and mean = 197.2). Both distributions are significantly different, as confirmed by a Mann-Whitney-Wilcoxon (MWW) test [41] (significance level set at 0.05).
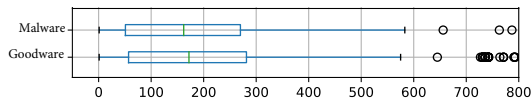

Figure 4: Distribution of the number of binary functions nodes in benign and malicious Android apps

Number of bytecode-to-native call-graph edges: Newly created edges can originate from native function invocations in bytecode methods (i.e., entry invocations). We compute the number of bytecode-to-native edges in apps' call-graph and detail their

distributions over our datasets in Figure 5. The difference between malware and goodware is significant, with a median equal to 14 for malware and 8 for goodware. Overall, JuCify reveals a total of 6758 bytecode-to-native invocations in the malware dataset and 29 908 in the goodware dataset.
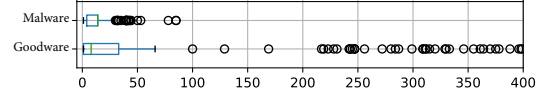

Figure 5: Distribution of the number of bytecode-to-native edges in benign and malicious Android apps

Number of native-to-bytecode call-graph edges: Newly added edges can also originate from bytecode methods invoked in native functions (i.e., exit invocations with reflection-like mechanisms as explained in Section 2.1.2). The median of number of edges is significantly low for both goodware and malware. Indeed, the median of native-to-bytecode edges is equal to 3 for both datasets, the distribution is available in Figure 6. Overall, JuCify reveals a total of 261 native-to-bytecode invocations in the entire goodware set and 4288 in the malware set. The conclusion that can be drawn from these results is the following:

the low numbers of native-to-bytecode edges in goodware shows that benign apps make little use of reflection-like mechanisms to invoke Java methods from native code, compared to malware.
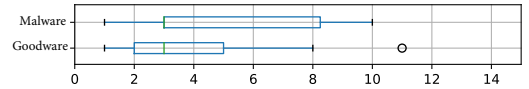

Figure 6: Distribution of the number of native-to-bytecode edges in benign and malicious Android apps

New previously unreachable bytecode methods: By considering native code, JuCify can reveal previously unreachable bytecode methods that are now reachable (because called from the native part). The number of previously unreachable bytecode methods is highly linked to the number of native-to-bytecode call-graph edges discussed in the previous paragraph. However, a new edge from native to bytecode can simply end to a previously reachable node, which does not present an interest here. Indeed, newly reachable nodes are interesting since they allow static analyzers to not consider them as dead code anymore. In Section 4.3, we give a concrete example of the importance of this metric.

Overall, JuCify can reveal 34 previously unreachable bytecode methods in 18 benign apps (with a maximum of 5 for one given app). For malicious apps, JuCify reveals 122 previously unreachable bytecode methods called from native code in 54 apps. This accounts for 13% of native-to-bytecode invocation in goodware and 2.8% for malware. This suggests that in most cases when Android app developers invoke bytecode methods from native code, it is to trigger bytecode methods that are already reachable from the bytecode. However, this shows that a non-negligeable proportion of bytecode invocation from the native in goodware and malware are overlooked by classical static analyzers since they account for non-reachable nodes in original bytecode callgraph.

Goodware vs. Malware native/bytecode calls: To better understand the difference between goodware and malware, we inspected the native functions invoked from the bytecode and the

bytecode methods invoked from the native code. Results indicate that in 82.7% of the cases, the native function `Java_mono_android_-Runtime_register` is invoked from the bytecode in goodware. In fact, most of the top invoked native functions in goodware are from the `mono` framework, which is used by Xamarin [74]. The same method is, however, not found in the malware dataset. The top invoked native functions in malware is composed of different elements such as `Java_com_seleuco_mame4all_Emulator_-setPadData`, `Java_com_shunpay210_sdk_CppAdapter210_pay`, or more suspicious functions: `Java_iqqxF_TZfff_ggior` and `Java_-glrrx_efgnp_twCJN`.

From native to bytecode, we note some interesting insights: while benign apps invoke from the native code, in the majority of cases, bytecode methods like `Context.getPackageName` (14.2%), or `ThreadLocal.get` (8.2%), malicious apps invoke methods such as `TelephonyManager.getDeviceId` (2.4%), or `TelephonyManager.-getSubscriberId` (4.3%) which can indicate suspicious behaviors.

Our results become more convincing by focusing on bytecode methods that were previously unreachable in call-graphs and called from native code. While most of the bytecode methods that were previously unreachable and called in the native code in goodware are Mono framework methods, in malware, the situation is different. Indeed, the most used bytecode methods in native code are dedicated to payment libraries (e.g., `com.shunpay208.sdk.Shun-Pay208`), and sensitive methods such as `getDeviceId`.

> **RQ4.a answer:** JuCify helps to discover new paths in app behaviour. It augments call-graphs with about 5-6% new nodes in both benign and malware apps. Overall, apps tend to use much more bytecode-to-native invocations than native-to-bytecode. However, malware seems to use bytecode invocations from native to perform suspicious activities.

4.4.2 RQ4.b: Can JuCify reveal previously unreachable sensitive data leaks that pass through native code in real-world apps?

With this RQ, our goal is to assess JuCify from a qualitative point of view. In particular, we check whether the call-graphs augmented by JuCify with previously unseen nodes are relevant. To that end, we run JuCify and FlowDroid on real-world apps to check if FlowDroid can detect sensitive data leaks through the native code.
Experimental setup: To assess JuCify in the wild, we selected malicious applications since the intuition is that malicious apps tend to leak sensitive data more than goodware. Therefore, we randomly selected 1800 malicious apps (i.e., VirusTotal score > 20) from Androzoo [3] that contain .so files. Besides, to detect data leaks, we used the default sources and sinks provided by FlowDroid. For each of these 1800 apps, we set a 1-hour timeout (30 min for the symbolic execution and 30 min for FlowDroid).
Findings: Among the 1800 malicious apps, 1460 contained Java native methods declaration(s) in the code. In total, JuCify was able to augment the call-graph of 1066 (i.e., 73%) of the 1460 apps that contain both .so files and Java native method declaration in bytecode. From these 1460 apps, FlowDroid revealed sensitive data leaks that take advantage of the native code in 14 apps. These 14 apps were manually checked and confirmed to contain sensitive

data leaks that goes through the native code. Note that this number is highly linked to the source and sink methods used.

In the following, we discuss two case studies where JuCify was able to reveal sensitive data leaks that pass through native code. Both Android apps were manually checked by the authors to confirm the presence of a leak detected by FlowDroid.

4.4.3 Getter-Scenario Case Study. In Figure 3a we illustrated an example of how malware developers can rely on native code to hide, from static analyzers, the <u>retrieval</u> of sensitive data from static analyzers. JuCify revealed an Android malware [2] implementing this specific behavior. JuCify reconstructed the `A()` native method from the `com.y` class as the following: "`<DummyBinaryClass: java-.lang.String Java_com_y_A(android.content.Context)>`". In this native function, the IMEI number of the device is obtained via the JNI interface and returned as a result. This reconstructed method is called in method `b()` of class `com.cance.b.q` to store the IMEI number. The resulting IMEI number is then wrapped and transferred to a method to log it.

After examining the VirusTotal report of this app, we found that the flags raised by antiviruses refer to Trojan behavior and explicitly mention the retrieval of sensitive information from the device as well as the use of native code in the implementation of the malicious behaviour. To some extent, this corroborates that JuCify contributed to uncover a malicious behaviour that is hidden through exploiting native-to-bytecode links (which state of the art static analyzers could not be aware of).

4.4.4 Leaker-Scenario Case Study. In Figure 3b, we illustrated how app developers can rely on native code to hide the <u>leakage</u> of sensitive data. JuCify revealed an Android malware [3] with this behavior.

First, the IMEI number is obtained in the `getOperator()` method of the `com.umeng.adutils.AppConnect` class and stored in the `imei` field of the same class. Then, in the `processReplyMsg()` method of this class (method which is triggered when an SMS is received), the IMEI number is wrapped in another string and sent to the native method "`stringFromJNI()`" as a parameter. JuCify's instrumentation engine constructed the following method from this native method: "`<DummyBinaryClass: java.lang.String Java_com_umeng_adutils_SdkUtils_stringFromJNI(android.-app.PendingIntent,java.lang.String,java.lang.String)>`". This latter has been populated with the information given by the symbolic execution and revealed that the `sendTextMessage()` method from the `android.telephony.SmsManager` class is called with the valued derived from the IMEI number as parameter.

To summarize, a value derived from the IMEI number is sent out of the device using an SMS through the native code. Doing so, the leak would have remained undetected without JuCify.

As in the previous case study, we examined the VirusTotal report of this app. In their majority, antiviruses flag it as a Trojan app. Some reports even explicit tag the use of `getDeviceId()` and of native code for the malicious operations. Thus, with JuCify we enabled an existing analyzer to uncover a leak being performed through native code.

---

[2]SHA-256: 54DAFDF3635B18C0FD9F5CE89FE14C072D75AB4687B376FBADF370388574DC14
[3]SHA-256: A0B7BFBC272B462A2F59CC09ACC8B75114137CF7A2B391201C14C1A90EA7E369

RQ4.b answer: JuCify is effective for highlighting data flows across native code that were previously unseen. Indeed, its enhanced call-graphs enable static analysers to reveal sensitive data leaks within real-world Android apps.

5 LIMITATIONS

Our approach is a step towards realizing the ambition of full code unification for Android static analysis. Our current prototype of JuCify, despite promising performances, presents a few limitations: First, our implementation relies on existing tools to extract native call-graphs and mutual invocations between bytecode and native code. Limitations of these tools are therefore carried over to JuCify. Such limitations include the exponential analysis time for symbolic execution, the limitation in finding the boundaries of native functions, the unsoundness in app modeling with FlowDroid due to reflective calls [35], multi-threading [40], and dynamic loading [75].

Second, our prototype currently relies on symbolic execution which is known to be non-scalable in the general case. Therefore, as described in Section 4.4.1, the call-graph of some Android apps was not augmented due to the symbolic execution that did not return native-bytecode links and/or due to the timeout.

Third, a major limitation of JuCify lies in the fact that it does not yet reconstruct native functions behavior with high precision. Indeed, as described in Section 3.2, for the native functions that represent Java native function, JuCify considers a partial list of statements: it employs opaque predicates to guide static analyzers into considering every possible path during analyses. Moreover, JuCify overlooks native functions that are not explicitly targeted by JNI Java calls since it cannot approximate their behavior in the current implementation. As a result, JuCify cannot generate native functions' control flow graphs with Jimple statements that cover the full behavior of functions. This limitation implies that if, for instance, a leak is performed by using Internet communication implemented "purely" in C (e.g., with a socket), then this leak would not be detected with FlowDroid even after JuCify processing. Also, during the reconstruction phase described in Section 3.2, in some cases where the number of parameter is important, the number of parameter combination can explode. This can lead to methods being extremely long that might not represent reality. We plan to address this limitation in future work.

6 THREATS TO VALIDITY

Manual Checking. To check the correctness of the results, we manually checked a hundred Android apps. To do so, we relied on Java bytecode decompilers and native code decompilers such as Jadx [58] and RetDec [29]. Although native code manual checking is challenging, we were able to confirm that the native nodes added by JuCify matched the nodes from the native callgraph constructed by NativeDiscloser. Regarding bytecode-to-native links, as the symbols were always available for the apps we checked (since native methods were statically registered), we were able to confirm the correctness of those links in the callgraph generated by JuCify. We reverse-engineered these apps and were able to reach the same conclusions. Regarding native-to-bytecode links, the method names are represented as strings, which are not directly available in the native code. Therefore, we faced a challenge to check if the symbolic

execution yielded correct links. One way to verify would be to execute the code part to trigger the native code and ensure that the correct information are yielded by NativeDiscloser, but this is a challenge per se and it is out of the scope of this study. Therefore, we made the hypothesis that the symbolic yields correct results.

7 RELATED WORK

Static analysis of Android apps. Static analysis of Android apps is widely explored to assess app properties. Less than 10 years after the introduction of Android, a systematic literature review [36] has shown that over one hundred papers presented static approaches to analyze Android apps. The review highlights that Android apps' security vetting is one of the main concerns for analysts, who assess properties such as sensitive data leak detection [5, 34, 56], or check for maliciousness [20, 33, 72]. Static approaches have also been implemented to identify functional and non-functional defects [10, 73] and towards fixing runtime crashes [27, 63]. Static analysis is also further leveraged to collect information in apps towards improving dynamic testing approaches [28, 42, 59, 77]. Given these fundamental usages of static analysis, it is essential to take into account all code that implements any part of the app behavior. Therefore, the fact that many analyses are reduced to focus on the bytecode (while leaving out native code within app packages) constitutes a severe threat to validity in many studies.

Binary analysis. Binary analysis techniques have been applied for different platforms, using static [7, 9, 12, 21], dynamic [6, 8, 37], hybrid [11, 22, 54] and machine-learning-based [32, 39, 68, 71] approaches. A recent work [15] tackles the challenging task of analyzing binaries by combining declarative static analysis (using Datalog declarative logic-based programming language) with reverse-engineering techniques to perform x-refs analysis in native libraries using Radare2 [51]. In the Android realm, analysis of binaries can be essential to cope with obfuscation [24].

Cross-language analysis. Several researchers have also acknowledged the presence of native code alongside bytecode in their analysis of Android apps. For instance, in 2016, Alam et al. [2] presented *DroidNative* which can perform Android malware detection considering both the bytecode and the native code. *NDroid* [50] and *TaintArt* [61] were proposed for dynamic taint analysis to track sensitive information flowing through JNI. *JN-SAF* [69] is also proposed as an inter-language static analysis framework to detect sensitive data leaks in Android apps, taking into account native code. All the aforementioned tools, however, are task-specific. They also, typically, perform their analyses separately for bytecode and native code, and later post-process and merge the outputs to present unified analysis results. In contrast, JuCify proposes to unify the representation before task-specific analyses. This enables other analyses to be built upon the output of JuCify. For experimental assessment of JuCify representation for data flow analysis (RQ-5), we envisioned a comparison with JN-SAF. Unfortunately, two co-authors independently failed to run the tool.

Overall, there are various approaches and studies [1, 31, 53, 60] in the literature that investigate the possibility to analyze apps by account for the different language-specific artifacts in the package. Although the approaches described are promising for cross-language analysis, they do not generally offer a practical framework to unify

the representation of both the bytecode and the native code into a single model that standard static analysis pipelines can leverage. Our prototype JuCify does bring such a unified model and targets the Jimple intermediate representation, which is the default internal representation of Soot. Therefore, by pushing in this research direction, we expect to provide the community with a readily usable framework, which will allow to (re)perform their analyses on whole code in Android apps.

8 CONCLUSION

We contribute in the ambitious research agenda of unifying bytecode and native code to support comprehensive static analysis of Android apps. We presented JuCify, as a significant step towards this unification: it generates a native call-graph that is merged with the bytecode call-graph based on links retrieved via symbolic execution. In this model (i.e., the unified call-graph), we are able to heuristically populate specific native functions with Jimple statements. The Jimple intermediate representation was selected to readily support existing static analysers based on the Soot framework.

We first empirically showed that JuCify significantly improves Android apps call-graphs, which are augmented (to include native code nodes) and enhanced (to reveal previously unreachable methods). Then, we showed that JuCify holds its promise in supporting state-of-the-art analyzers such as FlowDroid in enhancing their taint tracking analysis. Finally, we discuss how JuCify can reveal sensitive data leaks that pass through the native code in real-world Android apps, which were previously undetectable.

9 DATA AVAILABILITY

For the sake of Open Science, we provide to the community all the artifacts used in our study. In particular, we make available the datasets used for our experimentations, the source code of JuCify, the JuCify executable, and our benchmark apps. All artifacts (code, benchmarks, results) are available at:

https://github.com/JordanSamhi/JuCify

10 ACKNOWLEDGMENT

REFERENCES

[1] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*. 1–15.

[2] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. *Computers & Security* 65 (2017), 230 – 246. https://doi.org/10.1016/j.cose.2016.11.011

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In

Proceedings of the 13th International Conference on Mining Software Repositories (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[4] Androguard. [n.d.]. https://androguard.readthedocs.io. Accessed April 2021.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[6] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. 2006. Dynamic analysis of malicious code. *Journal in Computer Virology* 2, 1 (2006), 67–77.

[7] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. 1999. Static analysis of binary code to isolate malicious behaviors. In *Proceedings. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99)*. 184–189. https://doi.org/10.1109/ENABL.1999.805197

[8] Young-Hyun Choi, Min-Woo Park, Jung-Ho Eom, and Tai-Myoung Chung. 2015. Dynamic binary analyzer for scanning vulnerabilities with taint analysis. *Multimedia Tools and Applications* 74, 7 (2015), 2301–2320.

[9] C. Cifuentes and A. Fraboulet. 1997. Intraprocedural static slicing of binary executables. In *1997 Proceedings International Conference on Software Maintenance*. 188–195. https://doi.org/10.1109/ICSM.1997.624245

[10] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. 2019. Do Energy-oriented Changes Hinder Maintainability?. In *The 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)*.

[11] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques* 13, 1 (2017), 1–12.

[12] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.

[13] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. 2016. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2016.7792435

[14] Ira R Forman, Nate Forman, and John Vlissides Ibm. 2004. Java reflection in action. (2004).

[15] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java Calls in Native Code via Binary Scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 388–400. https://doi.org/10.1145/3395363.3397368

[16] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. 377–396. https://doi.org/10.1109/SP.2016.30

[17] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[18] JNI Functions. [n.d.]. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html. Accessed April 2021.

[19] Michael Furr and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 62–72. https://doi.org/10.1145/1065010.1065019

[20] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 281–294.

[21] Laune C. Harris and Barton P. Miller. 2005. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News* 33, 5 (Dec. 2005), 63–68. https://doi.org/10.1145/1127577.1127590

[22] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu. 2018. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 104–114. https://doi.org/10.1109/ICSME.2018.00019

[23] JNI. [n.d.]. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/. Accessed April 2021.

[24] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Daniel Xiapu Luo. 2019. Automated deobfuscation of Android native binary code. *arXiv preprint arXiv:1907.06828* (2019).

[25] Hyunjae Kang, Jae wook Jang, Aziz Mohaisen, and Huy Kang Kim. 2015. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 479174. https://doi.org/10.1155/2015/479174 arXiv:https://doi.org/10.1155/2015/479174

[26] Joris Kinable and Orestis Kostakis. 2011. Malware classification based on call graph clustering. *Journal in computer virology* 7, 4 (2011), 233–245.

[27] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. 2019. Mining Android crash fixes in the absence of issue-and change-tracking systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 78–89.

[28] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[29] J. Křoustek and P. Matula. 2018. RetDec: An Open-Source Machine-Code Decompiler. [talk]. Presented at Pass the SALT 2018, Lille, FR.

[30] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[31] S. Lee, H. Lee, and S. Ryu. 2020. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 127–137.

[32] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, and Ki-Woong Park. 2017. Learning binary code with deep learning to detect software weakness. In *KSII The 9th International Conference on Internet (ICONI) 2017 Symposium*.

[33] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2015. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*.

[34] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[35] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.

[36] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67 – 95. https://doi.org/10.1016/j.infsof.2017.04.001

[37] Lixin Li and Chao Wang. 2013. Dynamic analysis and debugging of binary code for security applications. In *International Conference on Runtime Verification*. Springer, 403–423.

[38] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE, 3–17.

[39] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 288–308.

[40] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. *ACM SIGPLAN Notices* 49, 6 (2014), 316–325.

[41] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[42] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[43] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[44] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (April 1998), 158–191. https://doi.org/10.1145/279310.279314

[45] DOOP Github page. [n.d.]. https://bitbucket.org/yanniss/doop/src/master/. Accessed April 2021.

[46] Native Scanner Github page. [n.d.]. https://github.com/plast-lab/native-scanner. Accessed April 2021.

[47] Dorottya Papp, Levente Buttyán, and Zhendong Ma. 2017. Towards semi-automated detection of trigger-based behavior for software security assurance. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[48] N. Peiravian and X. Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 300–305. https://doi.org/10.1109/ICTAI.2013.53

[49] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*. 1–6.

[50] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. 2014. On Tracking Information Flows through JNI in Android Applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 180–191. https://doi.org/10.1109/DSN.2014.30

[51] Radare2. [n.d.]. https://github.com/radareorg/radare2. Accessed April 2021.

[52] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[53] Claudio Rizzo. 2020. *Static Flow Analysis for Hybrid and Native Android Applications*. Ph.D. Dissertation. Royal Holloway – University of London.

[54] Kevin A Roundy and Barton P Miller. 2010. Hybrid analysis and control of malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 317–338.

[55] J. Sahs and L. Khan. 2012. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*. 141–147. https://doi.org/10.1109/EISIC.2012.34

[56] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1398–1409. https://doi.org/10.1109/ICSE43902.2021.00126

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

[58] Skylot. [n.d.]. *JadX: Dex to Java decompiler, https://github.com/skylot/jadx*. Accessed August 2021.

[59] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

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

[61] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 331–342. https://doi.org/10.1145/2976749.2978343

[62] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors.. In *Ndss*.

[63] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 187–198.

[64] Oguzhan Topgul. [n.d.]. *Android Malware Evasion Techniques - Emulator Detection*. https://www.oguzhantopgul.com/2014/12/android-malware-evasion-techniques.html Accessed December 2020.

[65] Virus Total. 2021. *Virus total free online virus, malware and url scanner*. https://www.virustotal.com/en

[66] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[67] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[68] S. Wang, P. Wang, and D. Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 388–398. https://doi.org/10.1109/ICSME.2017.59

[69] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1137–1150. https://doi.org/10.1145/3243734.3243835

[70] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[71] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[72] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*. IEEE, 62–69.

[73] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction*. 185–195.

[74] XAMARIN. [n.d.]. https://dotnet.microsoft.com/apps/xamarin. Accessed April 2021.

[75] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. 2017. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security* 12, 7 (2017), 1529–1544.

[76] Z. Yang and M. Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. https://doi.org/10.1109/WCSE.2012.26

[77] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2016. Automated test generation for detection of leaks in Android applications. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 64–70.

[78] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1106–1120.

[79] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. 93–104.