DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps

Li Li[α], Tegawendé F. Bissyandé[α], Damien Octeau[β], Jacques Klein[α]
[α] SnT, University of Luxembourg, Luxembourg
[β] CSE, Pennsylvania State University, USA
{li.li, tegawende.bissyande, jacques.klein}@uni.lu, octeau@cse.psu.edu

ABSTRACT

Android developers heavily use reflection in their apps for legitimate reasons, but also significantly for hiding malicious actions. Unfortunately, current state-of-the-art static analysis tools for Android are challenged by the presence of reflective calls which they usually ignore. Thus, the results of their security analysis, e.g., for private data leaks, are inconsistent given the measures taken by malware writers to elude static detection. We propose the DroidRA instrumentation-based approach to address this issue in a non-invasive way. With DroidRA, we reduce the resolution of reflective calls to a composite constant propagation problem. We leverage the COAL solver to infer the values of reflection targets and app, and we eventually instrument this app to include the corresponding traditional Java call for each reflective call. Our approach allows to boost an app so that it can be immediately analyzable, including by such static analyzers that were not reflection-aware. We evaluate DroidRA on benchmark apps as well as on real-world apps, and demonstrate that it can allow state-of-the-art tools to provide more sound and complete analysis results.

CCS Concepts

•Software and its engineering → Software notations and tools;

Keywords

Android; Static Analysis; Reflection; DroidRA

1. INTRODUCTION

Reflection is a property that, in some modern programming languages, enables a running program to examine itself and its software environment, and to change what it does depending on what it finds [1]. In Java, reflection is used as a convenient means to handle genericity or to process Java annotations inside classes. Along with many Java features, Android has inherited the Java Reflection APIs which are packaged and included in the Android SDK for developers to use. Because of the fragmentation of the Android ecosystem, where many different versions of Android are concurrently active on various devices, reflection is essential as it allows developers, with the same application package, to target devices running different versions of Android. Indeed, developers may use reflection techniques to determine, at runtime, if a specific class or method is available before proceeding to use it. This allows the developer to leverage, in the same application, new APIs where available while still maintaining backward compatibility for older devices. Reflection is also used by developers to exploit Android hidden and private APIs, as these APIs are not exposed in the developer SDK and consequently cannot be invoked through traditional Java method calls.

Unfortunately, recent studies on Android malware have shown that malware writers are using reflection as a powerful technique to hide malicious operation [2,3]. In particular, reflection can be used to hide the real purpose, e.g., by invoking a method at runtime to escape static scanning, or simply to deliver malicious code [4]. We have conducted a quick review of recent contributions on static analysis-based approaches for Android, and have found that over 90% of around 90 publications [5] from top conferences (including ICSE and ISSTA) do not tackle reflection. Indeed, most state-of-the-art approaches and tools for static analysis of Android simply ignore the use of reflection [6, 7] or may treat it partially [8, 9]. By doing so, the literature has produced tools that provide analysis results which are *incomplete*, since some parts of the program may not be included in the app call graph, and *unsound*, since the analysis does not take into account some hidden method invocations or potential writes to object fields. In this regard, a recent study by Rastogi et al. [10] has shown that reflection has made most of the current static analysis tools perform poorly on malware detection.

Tackling reflection is however challenging for static analysis tools. There exist ad-hoc implementations (e.g., in [9]) for dealing with specific cases of reflections patterns. Such approaches cannot unfortunately be re-exploited in other static analysis tools. However, there is a nascent commitment in the Android research community to propose solutions for improving the analysis of reflection. For example, in a recent work [11], Barros *et al.* propose an approach for resolving reflective calls in their Checker static analysis framework [12]. Their approach however 1) requires application source code (which is not available for most Android apps), 2) targets specific check analyses based on developer

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931044

annotations (which thus needs additional developer efforts, e.g., learn the target framework, find the right place to annotate, etc.) and 3) does not provide a way to directly benefit existing static analyzers, i.e., to support them in performing reflection-aware analyses.

Our aim is to deal with reflection in a non-invasive way so that state-of-the-art analysis tools can benefit from this work to better analyze application packages from app markets. To that end, we present in this paper the DroidRA instrumentation-based approach for automatically taming reflection in Android apps. In DroidRA, the targets of reflective calls are determined after running a constraint solver to output a regular expression satisfying the constraints generated by an inter-procedural, context-sensitive and flow-sensitive static analysis. A Booster module is then implemented to augment reflective calls with their corresponding explicit standard Java calls. Because some code can be loaded dynamically, before our reflection analysis, we also use heuristics to extract any would-be dynamically-loaded code (e.g., a jar file with *classes.dex* inside) into our working space. Indeed, our reflection taming approach hinges on the assumption that all the code that exist in the app package may be loaded at runtime and thus should be considered for analysis.

This paper makes the following contributions:

- We provide insights on the use of reflection in Android apps, based on an analysis of 500 apps randomly selected from a repository of apps collected from Google Play, third-party markets, as well as malware samples [13]. Our findings show that 1) a large portion of Android apps relies on reflective calls and that 2) reflective calls are usually used with some common patterns. We further show that reflective calls can be discriminated between malicious and benign apps.
- We designed and implemented DroidRA – an approach that aims at boosting existing state-of-the-art static analysis for Android by taming reflection in Android apps. DroidRA models the use of reflection with COAL [14] and is able to resolve the targets of reflective calls through a constraint solving mechanism. By instrumenting Android apps to augment reflective calls with their corresponding explicit standard Java calls, DroidRA complements existing analysis approaches.
- We evaluated DroidRA on a set of real applications and report on the coverage of reflection methods that DroidRA identifies and inspects. We further rely on well-known benchmarks to investigate the impact that DroidRA has on improving the performance of state-of-the-art static analyzers. In particular, we show how DroidRA is useful in uncovering dangerous code (e.g., sensitive API calls, sensitive data leaks [15, 16]) which was not visible to existing analyzers.
- We release DroidRA and the associated benchmarks as open source [17], not only to foster research in this direction, but also to support practitioners in their analysis needs.

2. MOTIVATION

Millions of Android apps are spread in different markets with more and more security and privacy alerts from antivirus vendors. A recent report from Symantec shows that in 2014 they classified about 1 million of Android apps as malware [18] on a total of 6.3 millions of apps analyzed. These facts urge for practical and scalable approaches and tools targeting the security analysis of large sets of Android apps. As example of such approaches, static taint analyzers aim at tracking data across control-flow paths to detect potential privacy leaks.

```
1  TelephonyManager telephonyManager = //default;
2  String imei = telephonyManager.getDeviceId();
3  Class c =
       Class.forName("de.ecspride.ReflectiveClass");
4  Object o = c.newInstance();
5  Method m = c.getMethod("setIme" + "i",
       String.class);
6  m.invoke(o, imei);
7  Method m2 = c.getMethod("getImei");
8  String s = (String) m2.invoke(o);
9  SmsManager sms = SmsManager.getDefault();
10 sms.sendTextMessage("+49 1234", null, s, null,
       null);
```

Listing 1: Code excerpt of *de.ecspride.MainActivity* from DroidBench's `Reflection3.apk`.

Let us consider the FlowDroid [15] state-of-the-art approach as a concrete example. FlowDroid is used to detect private data leaks from sensitive sources, such as contact information or device identification numbers, to sensitive sinks, such as sending HTTP posts or short messages. FlowDroid has demonstrated promising results, however, they suffer from limitations inherent to the challenges of static analysis in Android for taking into account reflection, class loading or native code support. In this paper, we focus on taming reflection in Android apps to allow state-of-the-art tools such as FlowDroid to significantly improve their results. Reflection breaks the traditional call graph construction mechanism in static analysis, resulting in an incomplete control-flow graph (CFG) and consequently leading to insufficient results. Dealing with reflection in static analysis tools is however challenging. Even the Soot Java optimization framework, on top of which most state-of-the-art approaches are built, does not address the case of reflective calls in its analyses. Thus, overall, taming reflection at the app level will enable better analysis by state-of-the-art analysis tools to detect security issues for app users.

We consider the case of an app included in the DroidBench benchmark [15, 19]. The Reflection3 benchmark app is known to be improperly analyzed by many tools, including FlowDroid, because it makes use of reflective calls. In this example app (Listing 1), class `ReflectiveClass` is first retrieved (line 3) and initialized (line 4). Then, two methods (*setImei()* and *getImei()*) from this class are reflectively shipped and invoked (lines 5-8). *setImei()*, which is matched by concatenating two strings, will store the device ID, which was obtained at line 2, into field *imei* of class `ReflectiveClass` (line 6). *getImei()*, similarly, gets back the device ID into the current context so that it can be sent outside the device via SMS to a hard-coded (i.e., not provided by user) phone number (line 10).

The operation implemented in this code sample is malicious since the device ID is taken as sensitive private information. The purpose of the reflective calls, which appear between the obtaining of the device ID and its leakage outside the device, is to elude any taint tracking by confusing the traditional flow. Thus, statically detecting such leaks becomes non trivial. For example, analyzing string patterns can be challenged intentionally by developers, as it was done in line 5. Furthermore, simple string analysis

```
1  //Example (1): providing genericity
2  Class collectionClass;
3  Object collectionData;
4  public XmlToCollectionProcessor(Str s, Class c) {
5   collectionClass = c;
6   Class c1 = Class.forName("java.util.List");
7   if (c1 == c) {
8    this.collectionData = new ArrayList();
9   }
10  Class c2 = Class.forName("java.util.Set");
11  if (c2 == c){
12   this.collectionData = new HashSet();
13 }}
14
15 //Example (2): maintaining backward compatibility
16 try {
17  Class.forName("android.speech.tts.TextToSpeech");
18 } catch (Exception ex) {
19  //Deal with exception
20 }
21
22 //Example (3): accessing hidden/internal API
23 //android.os.ServiceManager is a hidden class.
24 Class c =
       Class.forName("android.os.ServiceManager");
25 Method m = c.getMethod("getService", new Class[]
       {String.class});
26 Object o = m.invoke($obj, new String[] {"phone"});
27 IBinder binder = (IBinder) o;
28 //ITelephony is an internal class.
29 //The original code is called through reflection.
30 ITelephony.Stub.asInterface(binder);
```

Listing 2: Reflection usage in real Android apps.

is not enough to resolve reflective calls, because not only the method name (e.g., *getImei* for method $m2$) but also the method's declaring class name (e.g., *ReflectiveClass* for $m2$) are needed. These values must therefore be matched and tracked together: this is known as a composite constant propagation problem.

3. REFLECTION IN ANDROID APPS

We now investigate whether reflection is a noteworthy problem in the Android ecosystem. To this end, we mainly investigate why and to what extent reflection is used. More specifically, in Section 3.1, we first report on the common (legitimate) reasons that developers have to use reflection techniques in their code. Then, we investigate, in Section 3.2, the extent of the usage of reflection in real-world applications.

3.1 (Legitimate) Uses of Reflection

We have parsed Android developer blogs and reviewed some apps to understand when developers need to inspect and determine program characteristics at runtime leveraging the Java reflection feature.

Providing Genericity. Just like in any Java-based software, Android developers can write apps by leveraging reflection to implement generic functionality. Example (1) in Listing 2) shows how a real-world Android app implements genericity with reflection. In this example, a fiction reader app, *sunkay.BookXueshanfeihu* (`4226F8`[1]), uses reflection to produce the initialization of Collection List and Set.

Maintaining Backward Compatibility. In an example case, app *com.allen.cc* (`44B232`, an app for cell phone bill management) exploits reflection techniques to check at runtime the *targetSdkVersion* of a device, and, based on its value, to realize different behaviors. A similar use scenario consists in checking whether a specific class exists or not, in order to enable the use of advanced functionality whenever possible. For example, the code snippet (Example (2) in Listing 2), extracted from app *com.gp.monolith* (`61BF01`, a 3D game app), relies on reflection to verify whether the running Android version, includes the text-to-speech module. Such uses are widespread in the Android community as they represent the recommended way [20] of ensuring backward compatibility for different devices, and SDK versions.

Reinforcing App Security. In order to prevent simple reverse engineering, developers separate their app's core functionality into an independent library and load it dynamically (through reflection) when the app is launched: this is a common means to obfuscate app code. As an example, developers usually dynamically load code containing premium features that must be shipped after a separate purchase.

Accessing Hidden/Internal API. In development phase, Android developers write apps that use the `android.jar` library package containing the SDK API exposed to apps. Interestingly, in production, when apps are running on a device, the used library is actually different, i.e., richer. Indeed, some APIs (e.g., *getService()* of class `ServiceManager`) are only available in the platform SDK as they might still be unstable or were designed only for system apps. However, by using reflection, such previously hidden APIs can be exploited at runtime. Example (3), found in a wireless management app –*com.wirelessnow* (`314D51`)–, illustrates how a hidden API can be targeted by a reflective call.

3.2 Adoption of Reflection in Android

To investigate the use of reflection in real Android apps, we consider a large research repository of over 2 millions apps crawled from Google Play, third-party markets and known malware samples [13]. We randomly select 500 apps from this repository and parse the bytecode of each app, searching for reflective calls. The strategy used consists in considering any call to a method implemented by the four reflection-related classes[2] as a reflective call, except such methods that are overridden from `java.lang.Object`.

3.2.1 Overall Usage of Reflection

Our analysis shows that reflection usage is widespread in Android apps, with 87.6% (438/500) of apps making reflective calls. On average, each of the flagged apps uses 138 reflective calls. Table 1 summarizes the top 10 methods used in reflective calls.

Table 1: Top 10 used reflection methods and their argument type: either (C): `Class`, (M): `Method` or (F): `Field`.

Method (belonging class)	# of Calls	# of Apps
getName (C)	12,588	283 (56.6%)
getSimpleName (C)	5,956	87 (17.4%)
isAssignableFrom (C)	4,886	164 (32.8%)
invoke (M)	3,026	223 (44.6%)
getClassLoader (C)	2,218	163 (32.6%)
forName (C)	2,141	227 (45.4%)
getMethod (C)	1,715	135 (27.0%)
desiredAssertionStatus (C)	1,218	202 (40.4%)
get (F)	1,139	177 (35.4%)
getCanonicalName (C)	1,115	388 (77.6%)
Others	24,708	4 (8%)
Total	60,710	438 (87.6%)

We perform another study to check whether most reflective calls are only contributed by common advertizement

[1]In this paper, we represent an app with the last six letters of its sha256 code.

[2]`java.lang.reflect.Field`, `java.lang.reflect.Method`, `java.lang.Class`, and `java.lang.reflect.Constructor`.

libraries. We thus exclude reflective calls that are invoked by common ad libraries[3]. Our results show that there are still 382 (76.4%) apps whose non-ad code include reflective calls, suggesting the use of reflection in primary app code.

3.2.2 Patterns of Reflective Calls

In order to have a clear picture of how one can spot and deal with reflection, we further investigate the sequences of reflective calls and summarize the patterns used by developers to implement Android program behaviour with reflection. We consider all method calls within the 500 apps and focus on the reflection-related sequences that are extracted following a simple, thus fast, approach considering the natural order in which the bytecode statements are yielded by Soot[4]. We find 34,957 such sequences (including 1 or more reflective call). An isolated reflective call is relatively straightforward to resolve as its parameter is usually a String value (e.g., name of class to instantiate). However, e.g., when a method in the instantiated class must be invoked, other reflective calls may be necessary (e.g., to get the message name in object of class), which may complicate the reflection target resolution. We found 45 distinct patterns of sequences containing at least three reflective calls. Table 2 details the top five sequences: in most cases, reflection is used to access methods and fields of a given class which may be identified or loaded at runtime. This confirms the fundamental functionality of reflective calls which is to access methods/fields.

Table 2: Top 5 patterns of reflective calls sequences.

Sequence pattern	Occurences
Class.forName() → getMethod() → invoke()	133
getName() → getName() → getName()	120
getDeclaredMethod() → setAccessible() → invoke()	110
getName() → isAssignableFrom() → getName()	92
getFields() → getAnnotation() → set() → ...	88

We further investigate the 45 distinct patterns to focus on reflective calls that are potentially dangerous as they may change program state. Thus we mainly focus on sequences that include a method invocation (sequences 1 and 3 in Table 2) or access a field value in the code (sequence 5). Taking into account all the relevant patterns, including 976 sequences, we infer the common pattern which is represented in Figure 1. This pattern illustrates how the reflection mechanism allows to obtain methods/fields dynamically. These methods and fields can be used directly when they are statically declared (solid arrows in figure 1); they may otherwise require initializing an object of the class, e.g., also through a reflective call to the corresponding constructor (dotted arrows). With this common pattern, we can model most typical usages of reflection which can hinder state-of-the-art static analysis approaches.

The model yielded allows to consider different cases in reflective call resolution: In some simple cases, a string analysis is sufficient to extract the value of the call parameter; In other cases however, where class objects are manipulated to point to methods indicated in field values, simple string analysis cannot be used to help mapping the flow of a malicious operation. Finally, in some cases, there is a need to track back to the initialization of an object by another reflective call to resolve the target.

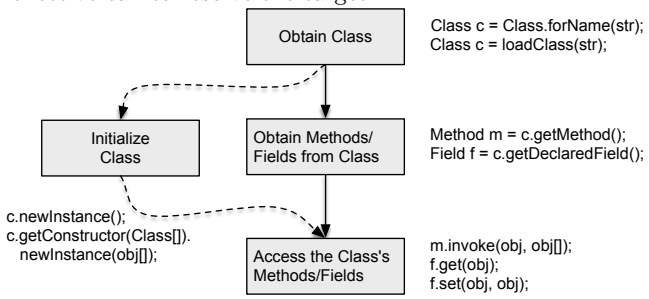

Figure 1: Abstract pattern of reflection usage and some possible examples.

4. TAMING REFLECTION

Our work is directed towards a twofold aim: (1) to resolve reflective call targets in order to expose all program behaviours, especially for analyses that must track private data; (2) to *unbreak* app control-flow in the presence of reflective calls in order to allow static analyzers to produce more precise results.

Figure 2 presents an overview of the architecture of the DroidRA approach involving three modules. (1) The first module named *JPM* prepares the Android app to be properly inspected. (2) The second module named *RAM* spots reflective calls and retrieves the values of their associated parameters (i.e., class/method/field names). All resolved reflection target values are made available to the analysts for use in their own tools and approaches. (3) Leveraging the information yielded by the RAM module, the *BOM* module instruments the app and transforms it in a new app where reflective calls are augmented with standard java calls. The objective of BOM is to produce an equivalent app whose analysis by state-of-the-art tools will yield more precise results [24, 25].

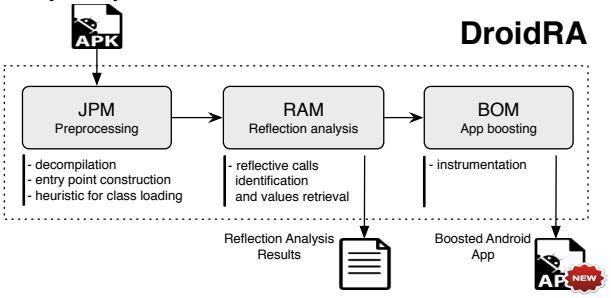

Figure 2: Overview of DroidRA.

4.1 JPM – Jimple Preprocessing Module

Android programming presents specific characteristics that require app code to be preprocessed before Java standard analysis tools can be used on it. First, an Android app is distributed as an *apk* file in which the code is presented in the form of Dalvik bytecode, a specific format for Android apps. Our analysis and code instrumentation will however manipulate code in Jimple, the intermediate representation required by Soot [26], a Java optimization framework. As a result, in a first step JPM leverages the Dexpler [27] translator to decompile the apk and output Jimple code.

Second, similarly to any other static approaches for Android, DroidRA needs to start analysis from a single entry-

[3]We take into account 12 common libraries, which are published by [21] and are also used by [22]. We believe that a bigger library set like the one provided by Li et al. [23] could further improve our results.

[4]One of the most popular open-source framework that supports static analysis of Java/Android apps.

point. Unfortunately, Android apps do not have a well-defined entry-point, e.g., *main()* in Java applications. But instead, they have multiple entry-points since each component that declares `Intent Filter`s (which defines the capabilities of a component) is a possible entry-point. To address this challenge, we use the same approach as in the FlowDroid [15] state-of-the-art work on Android analysis, that is, to artificially assemble a dummy main method, taking into account all possible components including their lifecycle methods (e.g., *onCreate()* and *onStop()*) and all possible callback methods (e.g., *onClick()*). This enables the static analyzer to build an inter-procedural control-flow graph and consequently to traverse all the app code.

Third, we aim to analyze the entire available app code, including such code that is dynamically loaded (e.g., at runtime). Dynamic Code Loading (DCL), however, is yet another challenge for static analysis, as some would-be loaded classes, which would be added at runtime (e.g., downloaded from a remote server), may not exist at all at static analysis time. In this work, we focus on dynamically loaded code that is included in the apk file (although in a separated archive file) and which can then be accessed statically. We assume that this way of storing locally the code to be dynamically is the most widespread. In any case, Google Play policy explicitly states that an app downloaded from Google Play may not modify, replace or update its own APK binary code using any method other than Google Play's update mechanism [28].

In practice, our DCL analysis is performed through heuristics: Given an app a, we first unzip[5] it and then traverse all its embedded files, noted as set F. For each file $f \in F$, if it is a Java archive format (the file extension could vary from *dat*, *bin* to *db*), then we recursively look into it, to check whether it contains a *dex* file through its magic number (*035*). All retrieved *dex* (usually come with *classes.dex*) files are then taken into consideration for any further analysis of the app.

We tested this heuristics-based process for finding DCL code by analyzing 1,000 malicious apps randomly selected from our data set. We found that 348 (34.8%) apps contain additional code, which could be dynamically loaded at runtime. Among the 348 apps, we collect 1,014 archives that contain an extra *classes.dex* file, giving an average of 2.9 "archives with code" per app. We also found that the 1,014 archives are redundant in many apps: there are actually only 74 distinct archive names. For example, library *bootablemodule.jar* (which contains a *classes.dex* file) has been used by 115 apps. This library package was recently studied in a dynamic approach [29].

4.2 RAM – Reflection Analysis Module

The Reflection Analysis Module identifies reflective calls in a given app and maps their target string/object values. For instance, considering the motivating example from the DroidBench app presented in Listing 1, the aim with RAM is to extract not only the method name in the `m2.invoke(o)` reflective call (line 8 in Listing 1), but also the class name that $m2$ belongs to. In other words, we have to associate $m2$ with $getImei$, but also o with $de.ecspride.ReflectiveClass$. To that end, based on the motivation example and our study of reflective call patterns, we observe that the reflection problem can be modeled as a constant propagation problem within an Android Inter-procedural Control-Flow Graph. Indeed, mapping a reflective call eventually consists in resolving the value of its parameters (i.e., name and type) through a context-sensitive and flow-sensitive inter-procedural data-flow analysis. The purpose is to obtain highly precise results, which are very important since the app will be automatically instrumented without any manual check of the results. Let us consider the resolution of the value of $m2$ in line 8 ('$String\ s = (String)\ m2.invoke(o)$' in Listing 1) as an example: if we cannot precisely extract the class name that $m2$ belongs to, say, our RAM tells that $m2$ belongs to class *TelephonyManager*, rather than the right class *ReflectiveClass*, then, during instrumentation, we will write code calling $m2$ as a member of *TelephonyManager*, which would yield an exception at runtime (e.g., no such method error), and consequently fail the static analysis.

To build a mapping from reflective calls to their target string/object values, our static analysis adopts an inter-procedural, context-sensitive, flow-sensitive analysis approach leveraging the composite COnstant propAgation Language (COAL) [14] for specifying the reflection problem. In order to use COAL, the first step is to model the reflection analysis problem independently from any app using the abstract pattern of reflective call inferred in Section 3.2.2. This generic model is specified by composite objects, e.g., a reflective method is being specified as an object (in COAL) with two fields: the method name and its declaring class name. Once reflection analysis has been modeled, we build on top of the COAL solver to implement a specific analyzer for reflection. This analyzer then performs composite constant propagation to solve the previously defined composite objects and thereby to infer the reflective call target values.

COAL-based Reflection Analysis. We now illustrate a simple example shown in Listing 3 to better explain the constant propagation of reflection-related values for class `Method`. Specifications for all other reflection-related classes are defined similarly. All specifications will be open-sourced eventually. Based on the specification shown in Listing 3, the COAL solver generates the semilattice that represents the analysis domain. In this case, the `Method` has two string fields, where Class types (strings of characters) are modeled as fully qualified class names. In the COAL abstraction, each value on an execution path is represented by a tuple, in which each tuple element is a field value. More formally, let $\mathcal{S}$ be the set of all strings in the program and let $B = (S \cup \{\omega\}) \times (S \cup \{\omega\})$, where ω represents an unknown value. Then the analysis domain is the semilattice $L = (2^B, \subseteq)$, where for any set X, 2^X is the power set of X, and elements in 2^B represent the set of values of `Method` variables across all execution paths. Semilattice L has a bottom element $\perp = \varnothing$ and its top element is the set of all elements in B. For example, the following equation models the value of object `m` at line 10 of Listing 3:

$$\{(\texttt{first.Type}, \texttt{method1}), (\texttt{second.Type}, \texttt{method2})\} \quad (1)$$

The first tuple in Equation (1) represents the value of Method object `m` contributed by the first branch of the `if` statement. The second tuple, on the other hand, models the value on the fall-through branch.

In order to generate transfer functions for the calls to `getMethod`, the COAL solver relies on the specification presented in lines 15-17 of Listing 3. The modifier `mod` statement specifies the signature of the `getMethod` method and it

[5]The format of an *apk* is actually a compressed ZIP archive.

```
1  //Java/Android code
2  Class c; Method m;
3  if (b) {
4    c = first.Type.class;
5    m = c.getMethod("method1");
6  } else {
7    c = second.Type.class;
8    m = c.getMethod("method2");
9  }
10 m.invoke(someArguments);
11 //Simplified COAL specification (partial)
12 class Method {
13  Class declaringClass_method;
14  String name_method;
15  mod gen <Class: Method
        getMethod(String,Class[])>{
16   -1: replace declaringClass_method;
17   0: replace name_method; }
18  query <Method: Object invoke(Object,Object[])>{
19   -1: type java.lang.reflect.Method; }
20 }
```

Listing 3: Example of COAL-based reflection analysis for class *Method*. Similar specifications apply for all other reflection classes

describes how the method modifies the state of the program. The `gen` keyword specifies that the method generates a new object of type Method (i.e., it is a factory function). Statement `-1: replace declaringClass_method` indicates that the name of the Class object on which the method is called (e.g., `first.Type` at line 4) is used as the field `declaringClass_method` of the generated object. Note that in this statement the special `-1` index indicates a reference to the instance on which the method call is made, for example object `c` at line 5. Finally, statement `0: replace name_method` indicates that the first argument (as indicated by index 0) of the method is used as the `name_method` field of the generated object.

At the start of the propagation performed by the COAL solver, all values are associated with $\perp$. Then the COAL solver generates transfer functions that model the influence of program statements on the values associated with reflection. Following the formalism from [14], for any $v \in L$, we define function $init_v$ such that $init_v(\perp) = v$. By using the specification at lines 15-17, the COAL solver generates function $init_{\{(\texttt{first.Type,method1})\}}$ for the statement at line 5. The function that summarizes the statement at line 8 is defined in a similar manner as $init_{\{(\texttt{second.Type,method2})\}}$. Thus, when taking the join of $init_{\{(\texttt{first.Type,method1})\}}(\perp)$ with $init_{\{(\texttt{second.Type,method2})\}}(\perp)$, we obtain the value given by Equation (1).

The COAL specification in Listing 3 includes a `query` statement at lines 18-19. This causes the COAL solver to compute the values of objects of interest at specific program points. In our example, the `query` statement includes the signature for the `invoke` method. The `-1: type Method` statement specifies that objects on which the `invoke` method is called have type `Method`. Thus using this specification the COAL solver will compute the possible values of object `m` at line 10 of Listing 3.

Improvements to the COAL Solver. In the process of this work, we have contributed to several improvements of the COAL solver that now enables it to perform efficiently for resolving targets of reflective calls. At first, we extended both the COAL language and the solver to be able to query the values of objects on which instance calls are made. For example, this allowed us to query the value of object m in statement *m.invoke(obj, args)*. Second, we added limited support for arrays of objects such that the values of object

```
1 Object[] objs = new Object[2];
2 objs[0] = "ISSTA";
3 objs[1] = 2016;
4 m.invoke(null, objs);
5 //m(String,int)
```

Listing 4: Example of use of a varargs parameter.

arrays can be propagated to array elements. More specifically, if an array a is associated with values v_1, v_2, ..., v_n, for any i array element $a[i]$, we mark it as potentially containing all the values (from v_1 to v_n). While this may not be precise in theory, in the case of reflection analysis, the arrays of constructors, returned by method *getConstructors()*, that we consider typically only have a few elements. Thus, this improvement, which ensures that the propagation of constructors is done, is precise enough in practice. Finally, we performed various optimizations to improve overall performance of the COAL solver[6].

We now detail an example of difficulty that we have encountered to retrieve the string/object values. The difficulty is due to the fact that some reflection calls such as *m.invoke(Object, Object[])* take as parameter a varargs [30]. The problem here is that the object array is not the real parameter of the method m. Indeed, the parameters are instead the elements of the array. This keeps us from extracting the appropriate method for instrumentation.

Let us consider the example code snippet in Listing 4. By only looking in line 4, we would infer that the parameter of the method m is $objs$. Whereas actually m has two parameters: a *String* and an *int* (as showed in line 5). To solve this problem and infer the correct list of parameters, we perform a backward analysis for each object array. For example, from $objs$ in line 4, we go back to consider both line 2 and line 3, and infer that 1) the first parameter of m is a *String* whose value is $ISSTA$, 2) the second parameter is an *int* whose value is 2016.

4.3 BOM – Booster Module

The Booster Module considers as input an Android app represented by its Jimple instructions and the reflection analysis results yielded by the RAM module. The output of BOM is a new *reflection-aware analysis-friendly* app where instrumentation has conservatively augmented reflective calls with the appropriate standard Java calls: reflective calls will remain in the app code to conserve its initial behaviour for runtime execution, while standard calls are included in the call graph to allow only static exploration of once-hidden paths. For example, in the case of Listing 1, the aim is to augment "$m.invoke(o, imei)$" with "$o.setImei(imei)$" where o is a concrete instance of class $de.ecspride.ReflectiveClass$ (i.e. explicitly instantiated with the *new* operator). Boosting approaches have been successful in the past in state-of-the-art frameworks for improving analysis of specific software by reducing the cause of analysis failures. TamiFlex [31] deals with reflection in standard Java software in this way, while IccTA [16] explicitly connects components, to improve Inter-Component Communication analysis.

Let us consider again our motivation example presented in Listing 1 to better illustrate the instrumentation proposed by BOM. Listing 5 presents the boosting results of Listing 1. Our instrumentation tactic is straightforward: for instance if a reflection call initializes a class, we explicitly represent the statement with the Java standard *new* operator (line 4

[6]https://github.com/siis/coal.git

```
1   Class c =
      Class.forName("de.ecspride.ReflectiveClass");
2   Object o = c.newInstance();
3 + if (1 == BoM.check())
4 +   o = new ReflectiveClass();
5   m.invoke(o, imei);
6 + if (1 == BoM.check())
7 +   o.setImei(imei);
8   String s = (String) m2.invoke(o);
9 + if (1 == BoM.check())
10+   s = (String) o.getImei();
```

Listing 5: The boosting results of our motivating example.

in Listing 5). If a method is reflectively invoked (lines 5 and 8), we explicitly call it as well (lines 7 and 10). This instrumentation is possible thanks to the mapping of reflective call targets yielded by the RAM module. The target resolution in RAM indeed exposes that (1) object c is actually an instance of class `ReflectiveClass`; (2) object m represents method *setImei* of class `ReflectiveClass` with a String parameter *imei*; (3) object $m2$ represents method *getImei* of class `ReflectiveClass`.

This example illustrates why reflection target resolution is not a simple string analysis problem. In this case, the support of composite object-analysis in RAM is warranted: In line 1 of Listing 5, c is actually an object, yet the boosting logic requires information that this represents class name "ReflectiveClass".

Observant readers may have noticed that the new injected code is always guarded by a conditional to add a path for the traditional calls. The *check()* method is declared in an interface whose implementation is not included for static analysis (otherwise a precise analyzer could have computed its constant return value). However for runtime execution, *check()* always returns `false`, preventing paths added by BOM from ever being executed. Thus, The opaque predicate keeps the new injected code from changing the app behavior, while all sound static analysis can safely assume that the path can be executed.

Additional Instrumentations. BOM performs additional instrumentations that are not directly related to the Reflection problem. Nevertheless, these instrumentations are useful to improve the completeness of other static analyses. We remind that the goal of our approach is to enable existing analyzers such as FlowDroid to perform reflection-aware static analysis in a way that improves their security results. For instance FlowDroid aims at detecting data leaks with taint-flow static analysis. In the presence of dynamic class loading, FlowDroid stops its analysis when a class has to be loaded. We already explained how DroidRA tackles this problem with the JPM module (cf. Section 4.1). However, not all the classes which have to be loaded are accessible. One of the reasons is that some files are encrypted, which prevents the analysis from statically accessing them. For example, app *com.ivan.oneuninstall* contains an archive file called *Grid_Red_Attract.apk*, which contains another archive file called *tu.zip* that has been encrypted. Because it is unrealistic to implement a brute-force technique to the password, we simply exclude such apps from our analysis. However, to allow tools such as FlowDroid to continue their analyses, we propose an instrumentation that conservatively solves this problem: we explicitly mock all the classes, methods and fields that are reported by the RAM module[7] but are not existing in the current class path (i.e. they are neither present in the initial code of the apk, nor in the code "extracted" by the JPM module).

Let us take an example to illustrate our instrumentation. Consider the instruction "`result=o.inc(a_1, a_2)`" where the method *inc* is not accessible and where a_1 is tainted. Without any modification of this code, a standard analyzer would stop its analysis. Our instrumentation consists in creating the method *inc* (and the associated class if required) in a way that the *taints* of a_1 and a_2 can be propagated. Concretely, the instrumented method *inc* will contain the following instruction: *return* (*Object*) ($a_1.toString() + a_2.toString()$), assuming that the type of *result* is *Object*.

[7]This means that we only take into account reflective calls.

5. EVALUATION

Overall, our goal was to enable existing state-of-the-art Android analyzers to perform reflection-aware static analysis, thus improving the soundness and completeness of their approaches. The evaluation of DroidRA thus investigates whether this objective is fulfilled. To that end, we consider answering the following research questions:

RQ1 What is the coverage of reflection calls that DroidRA identifies and inspects?

RQ2 How does DroidRA compare with state-of-the-art approaches for resolving reflective call targets in Android?

RQ3 Does DroidRA support existing static analyzers to build sounder call graphs of Android apps?

RQ4 Does DroidRA support existing static analyzers to yield reflection-aware results?

5.1 RQ1: Coverage of Reflective Calls

The goal of our reflection analysis is to provide necessary information for analysts (or other approaches) to better understand how reflections are used by Android apps. Thus, instead of considering all reflection-related methods, in this experiment, we select such methods that are most interesting for analysts. These include: 1) methods that acquire *Method*, *Constructor* and *Field* objects. Those method call sequences are falling in our common pattern (cf. Figure 1) and are critical as they can be used, e.g., to exchange data between normal explicit code and reflectively hidden code parts. For these calls, we perform a composite analysis and inspect the related class names and method/field names if applicable; and 2) methods that contain at least one string parameter. For these methods, we explore their string parameter's possible values.

We use the corpus of 500 apps selected in Section 3.2, to investigate the coverage of reflection calls. From each app with reflective calls we extract two information:

1. **Reached:** The total number of reflective calls that are identified by our RAM reflection analysis module.
2. **Resolved:** The number of reflective calls that are successfully resolved (i.e., the values of relevant class, method and field names can be extracted) by our reflection analysis.

Our experimental results are shown in Figure 3, which illustrates with boxplots the performance of DroidRA in reaching reflective calls from the dummy main, and in resolving their targets. Compared to the total number of reached

reflective calls, in average, DroidRA is able to correctly resolve 81.2% of the targets.

These off-targets are mainly explained by 1) the limitations of static analysis, where runtime values (e.g., user configuration or user inputs) cannot be solved statically at compile time; 2) the limitations of our COAL solver, e.g. currently it is not able to fully propagate arrays of objects, although we have provided a limited improvement on this.

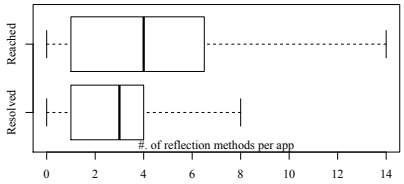

Figure 3: Results of the coverage of reflection methods.

5.2 RQ2: Comparison with Checker

The approach proposed by Barros et al. [11] is the closest work to ours. Their recent publication presents an approach, hereon referred to as *Checker*, to address reflection in the Information Checker Framework (IFC) [12]. We thus compare both approaches using their evaluation dataset which consists of 10 apps from the F-Droid open-source apps repository [32]. Table 3 lists the 10 apps and provides comparative results between Checker and DroidRA. Checker has been evaluated by providing statistics on methods and constructors related to reflective invocations. We thus consider the same settings for the comparison. Moreover, note that we apply DroidRA directly on the bytecode of the apps while Checker is applied on source code. Additionally, our approach does not need extra developer efforts while Checker needs manual annotations, e.g., one has to pinpoint good places to put appropriate annotations.

Overall, as shown in Table 3, DroidRA resolves 9 more method/constructors than Checker. Now we give more details on these results. For app *RemoteKeyboard*, DroidRA missed one method and Checker reports that it is not able to resolve it as well. Our further investigation shows that it is impossible for static approaches to resolve the reflective call in this case as the reflection target is a runtime user input (a class name for a shell implementation). For app *VimTouch*, DroidRA refuses to report a reflective call, namely method *Service.stopForeground*, because its caller method *ServiceForegroundCompat.stopForeground* is not invoked at all by other methods, letting it becomes unreachable from our entry method.

For app *ComicsReader*, DroidRA has resolved one more reflective method than Checker. We manually verify in the source code that the additional reflective call is a True Positives of DroidRA. However, with *ComicsReader*, DroidRA missed one method[8], although it resolved two additional reflective calls that Checker missed. This missed method is actually located in a UI-gadget class which is not an Android component (e.g., Activity). Since our dummy main only considers Android components of an app as potential entry-points, DroidRA further failed to reach this method from its dummy main.

Last but not the least, we have found 10 more constructors located in libraries embedded in the studied apps. Because Checker only checks the source code of apps, it could not reach and resolve them.

Table 3: The comparison results between DroidRA and Checker, where cons means the number of resolved constructors.

App	Checker		DroidRA	
	methods	cons	methods	cons
AbstractArt	1	0	1	0
arXiv	14	0	14	0
Bluez IME	4	2	4	2
ComicsReader	6	0	7	0
MultiPicture	1	0	1	0
PrimitiveFTP	2	0	2	7
RemoteKeyboard	1^{α}	0	0	3
SuperGenPass	1	0	1	0
VimTouch	3^{β}	0	2	0
VLCRemote	1	0	1	0

$^{\alpha}$ Reached but not resolved.
$^{\beta}$ One from dead code.

5.3 RQ3: Call Graph Construction

An essential step of performing precise and sound static analysis is to build at least a complete program's method call graph (CG), which will be used by static analyzers to visit all the reachable code, and thus perform a sound analysis. Indeed, methods that are not included in the CG would never be analyzed since these methods are unreachable from the analyzer's point of view. We investigate whether our DroidRA is able to enrich an app's CG. To that end we build the CG of each of the apps before and after they are instrumented by BOM. Our CG construction experiments are performed with the popular Soot framework [26]: we consider the CHA [33] algorithm, which is the default algorithm for CG construction in Soot, and the more recent Spark [34] algorithm which was demonstrated to improve over CHA. Spark was demonstrated to be more precise than CHA, and thus producing fewer edges in its constructed CG.

On average, in our study dataset of 500 apps, for each app, DroidRA improves by 3.8% and 0.6% the number of edges in the CG constructed with Spark and CHA respectively. Since CHA is less precise than Spark, CHA yields far more CG edges, and thus the proportion of edges added thanks to DroidRA is smaller than for Spark.

We highlight the case of three real-world apps from our study dataset in Table 4. The CG edges added (i.e., Diff column) vary between apps. We have further analyzed the added edges to check whether they reach sensitive API methods[9] (e.g., `ActivityManager.getRunningTasks(int)`) which are protected by a system permission (e.g., `GET_TASKS`). The recorded number of such newly reachable APIs (see Perm column in Table 4) further demonstrates how taming reflection can allow static analysis to check the suspicious call sequences that are hidden by reflective calls. We confirmed that this app is flagged as malicious by 24 anti-virus products from VirusTotal.

Case Study: `org.bl.cadone`. We consider the example of app *org.bl.cadone* to further highlight the improvement in CG construction. We have computed the call graph (CG) of this app with CHA and, for the benefit of presentation clarity, we have simplified it into a class dependency graph (CDG) where all CG edges between methods of two classes are transformed into a single CDG edge where all nodes representing methods from a single class are merged into a single node representing this class.

[8] *View.setSystemUiVisibility()*.

[9] The list of sensitive API methods are collected from PScout [35].

Table 4: The call graph results of three apps we highlight for our evaluation. Perm column means the number of call graph edges that are actually empowered by DroidRA and are accessing permission protected APIs.

Package	Algo	Original	DroidRA	Diff	Perm
com.boyaa.bildf	Spark	714	22,867	22,153	3
	CHA	172,476	190,436	17,960	51
org.bl.cadone	Spark	694	951	257	0
	CHA	172,415	187,079	14,664	16
com.audi.light	Spark	6,028	6,246	218	0
	CHA	174,007	174,060	53	0

Figure 4 presents the CDG with 14,664 new edges added after applying DroidRA. Black edges represent nodes and edges that were available in the original version of the app. The new edges (and nodes) have been represented in green. Some of them however reach sensitive APIs, and are highlighted in red. We found that, among the 8 permissions that protect the 16 sensitive APIs (included in 8 classes) that are now reachable, 6 (i.e., 75%) are of the *dangerous* level [36], which further suggests that the corresponding reflective calls were meant to hide dangerous actions.

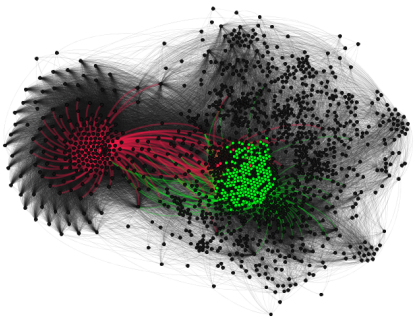

Figure 4: The class dependency graph (simplified call graph) of app *org.bl.cadone* (based on CHA algorithm). Black color shows the originally edges/nodes, green color shows edges/nodes that are introduced by DroidRA while red color shows new edges that are further protected by permissions.

5.4 RQ4: Improvement of Static Analysis

We consider the state-of-the-art tool FlowDroid and its ICC-based extension called IccTA for assessing to what extent DroidRA can support static analyzers in yielding reflection-aware results. Our experiments are based on benchmark apps, for which the ground truth of reflective calls is known, and on real-world apps. Finally, on the real-word apps, we check whether the runtime performance of DroidRA will not prevent its use in complement to other static analyzers.

DroidRA on Benchmark Apps. We assess the efficacy of DroidRA on 13 test case apps for reflection-based sensitive data leaks. 4 of these apps are from the Droidbench benchmark where they allowed to show the limitations of FlowDroid and IccTA. We further consider 9 other test cases to include other reflective call patterns (e.g., the top used sequences , cf. Table 2). Since the test cases are handcrafted, the data leak (e.g., leak of *device id* via *SMS*), are known in advance. In 12 of the apps, the leak is intra-component, while in the 11^{th} it is inter-component.

Table 5 provides details on the reflective calls and whether the associated data-leak is identified by the static analysis of IccTA and/or DroidRA-supported IccTA. Expectedly, IccTA alone only succeeds on the first test case, $Reflection_1$ in DroidBench, where the reflective calls are not in the data-leak path, thus not requiring a reflective call resolution for the taint analysis to detect the leak. However, IccTA fails on all other 12 test cases. This was expected since IccTA is not a reflection-aware approach. When reflection is tamed in the test cases by DroidRA, IccTA gains the ability to detect a leak on 11 out of 12 test cases. In test case 13, the reflective call is not resolved because the reflection method *getFields()* returns an array of fields that the current implementation of constant propagation cannot manage to resolve. Indeed, we have enhanced COAL with limited support to propagate array elements, complex field arrays are not addressed. Nevertheless, constructor arrays can now be resolved, allowing DroidRA to tame reflection in test case 6.

DroidRA on Real-world Apps. To investigate the impact of DroidRA on the static analysis results of real-word apps, we consider a random set of 100 real-world apps that contain reflective calls and at least one sensitive data leak (discovered by IccTA). Comparing to using IccTA on original apps, the instrumentation by DroidRA impacts the final results by allowing IccTA to report on average (median) 1 more leak in a reflection-aware setting.

Runtime Performance of DroidRA. We investigate the time performance of DroidRA to check whether time overhead of DroidRA app will not be an obstacle to practical usage in complement with state-of-the-art static analyzers. On the previous set of 91 apps, we measure the time performance of the three modules of DroidRA. The median value for JPM, RAM, BOM on the apps are 24 seconds, 21 seconds and 8 seconds respectively leading to a total median value of 53 seconds for DroidRA. This value is reasonable in comparison with the execution of tools such as IccTA or FlowDroid which can run for several minutes and even hours on a given app.

6. THREATS TO VALIDITY

The main threats to validity of DroidRA is carried from the COAL solver: at the moment, the composite constant propagation cannot fully track objects inside an array. We have provided limited support in our improved version of the COAL solver and we plan to address this further in future work. Besides, the conservative setting, where a string is represented by a regular expression ("*") if COAL cannot statically infer its value, which could be taken as everything and thus may also introduce false positives. Applying a probabilistic model could potentially mitigate this threat [37]. Another threat is related to Dynamic Class Loading. Although we have used heuristics to include external classes, some other would-be dynamically loaded code (e.g., downloaded at runtime) can be missed during the reflective call resolution step. However, our objective in this paper was not to solve the DCL problem. Other approaches [29, 38] can be used to complement our work.

The single entry-point method (the dummy main method) that we build may not cover all the reflective calls, which means that some reflective calls may not be reachable from the call graph of RAM. Finally, the call graph built by RAM is leveraging the implementation of the Spark algorithm in Soot, which also comes with specific limitation [39].

Finally, DroidRA handles neither native code, nor multi-threads. These are challenges that most current Android static analysis approaches ignore, but are out of the scope of this paper.

Table 5: The 13 test cases we use in our in-the-lab experiments. These 13 cases follow the common pattern of Figure 1 and each case contains exactly one sensitive data leak.

Case	Source	Reflection Usage	IccTA	DroidRA+IccTA
1	DroidBench	forName() → newInstance()	✓	✓
2	DroidBench	forName() → newInstance()	✗	✓
3	DroidBench	forName() → newInstance() → m.invoke() → m.invoke()	✗	✓
4	DroidBench	forName() → newInstance()	✗	✓
5	New	forName() → getConstructor() → newInstance()	✗	✓
6	New	forName() → getConstructors() → newInstance()	✗	✓
7	New	forName() → getConstructor() → newInstance() → m.invoke() → m.invoke()	✗	✓
8	New	loadClass() → newInstance()	✗	✓
9	New	loadClass() → newInstance() → f.set() → m.invoke()	✗	✓
10	New	forName() → getConstructor() → newInstance() → f.get()	✗	✓
11	New	startActivity() → forName() → newInstance() → m.invoke() → m.invoke()	✗	✓
12	New	forName() → getConstructor() → newInstance() → f.set() → f.get()	✗	✓
13	New	forName() → getConstructor() → newInstance() → getFields() → f.set() → f.get()	✗	✗

7. RELATED WORK

Research on static analysis of Android apps presents strong limitations related to reflection handling [40–43]. Authors of recent approaches explicitly acknowledge such limitations, indicating that they ignore reflection in their approaches [14, 16, 44] or failing to state whether reflective calls are handled [45] in their approach.

The closest work to ours was concurrently proposed by Barros et al. [11] within their Checker framework. Their work differs from ours in several ways: first, the design of their approach focus on helping developers checking the information-flow in their own apps, using annotations in the source code; this limits the potential use of their approach by security analysts in large markets of Android apps such as GooglePlay or AppChina. Second, they build on intra-procedural type inference system to resolve reflective calls, while we build on an inter-procedural precise and context-sensitive analysis. Third, our approach is non-invasive for existing analysis tools who can now be boosted when presented with apps where reflection is tamed.

Reflection, by itself, has been investigated in several works for Java applications. Most notably, Bodden et al. [31] have presented TamiFlex for aiding static analysis in the presence of reflections in Java programs. Similarly to our approach, TamiFlex is implemented on top of Soot and includes a Booster module, which enriches Java programs by "materializing" reflection methods into traditional Java calls. However, DroidRA manipulates Jimple code directly while TamiFlex works on Java bytecode. Furthermore, DroidRA is a pure static approach while TamiFlex needs to execute programs after creating logging points for reflection methods to extract reflection values. Finally, although Android apps are written in Java, TamiFlex cannot even be applied to Android apps as it uses a special Java API that is not available in Android [29].

Another work that tackles reflection for Java has been done by Livshits et al. [46], in which points-to analysis is leveraged to approximate the targets of reflection calls. Unlike our approach, their approach needs users to provide an per-app specification in order to resolve reflections, which is difficult to apply for a large scale analysis. Similarly, Braux et al. [47] propose a static approach to optimize reflection calls at compile time, for the purpose of increasing time performance.

Regarding Dynamic Code Loading in Android, Poeplau et al. [38] have proposed a systematic review on how and why Android apps load additional code dynamically. In their work, they adopt an approach that attempts to build a super CFG by replacing any *invoke()* call with the target method's entry point. This approach however fails to take into account the *newInstance()* reflective method call, which initializes objects, resulting in a context-insensitive approach, potentially leading to more false positives. StaDynA [29] was proposed to address the problem of dynamic code loading in Android apps at runtime. This approach requires a modified version of the Android framework to log all triggering actions of reflective calls. StaDynA is thus not market-scalable, and present a coverage issue in dynamic execution. Our approach, DroidRA, provides a better solution for reflective method calls, can be leveraged to compliment these approaches, so as to enhance them to conduct better analysis.

Instrumenting Android apps to strengthen static analysis is not new [48]. For example, IccTA [16], a state-of-the-art ICC leaks analyzer, instruments apps to bridge ICC gaps and eventually enable inter-component static analysis. AppSealer [49] instruments Android apps for generating vulnerability-specific patches, which prevent component hijacking attacks at runtime. Other approaches [50, 51] apply the same idea, which injects shadow code into Android apps, to perform privacy leaks prevention.

8. CONCLUSION

This paper addresses a long time challenge that is to perform reflection-aware static analysis on Android apps. We have presented DroidRA, an open source tool, to perform reflection analysis, which models the identification of reflective calls as a composite constant propagation problem through the COAL declarative language, and leverages the COAL solver to automatically infer reflection-based values. We remind the reader that these reflective-based values can be directly used as basis for many whole-program analyses of Android apps. We further illustrate this point by providing a booster module, which is based on the previously inferred results to augment apps with traditional Java calls, leading to a non-invasive way of supporting existing static analyzers in performing reflection-aware analysis, without any modification or configuration. Through various evaluations we have demonstrated the benefits and performance of DroidRA.

9. ACKNOWLEDGMENTS

We thank Paulo Barros, René Just and Michael Ernst, from the Checker [11] team, for sharing detailed results of their approach to deal with reflection. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

10. REFERENCES

[1] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[2] Mikhail Kazdagli, Ling Huang, Vijay Reddi, and Mohit Tiwari. Morpheus: Benchmarking computational diversity in mobile malware. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, pages 3:1–3:8, New York, NY, USA, 2014. ACM.

[3] Martina Lindorfer, Matthias Neugschw, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis- 1,000,000 apps later: A view on current android malware behaviors.

[4] Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, 2014. https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation.

[5] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. Technical Report ISBN 978-2-87971-150-8 TR-SNT-2016-3, 2016.

[6] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE*, 2014.

[7] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, May 2014.

[8] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. 2015.

[9] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2014.

[10] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[11] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Armorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE, Lincoln, Nebraska, 2015.

[12] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *CCS*, pages 1092–1104, Scottsdale, AZ, USA, November 4–6, 2014.

[13] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *The 13th International Conference on Mining Software Repositories, Data Showcase track*, 2016.

[14] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[15] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[16] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[17] Droidra. https://github.com/serval-snt-uni-lu/DroidRA.git. Accessed: 2015-08-22.

[18] Symantec. Internet security threat report. Volume 20, April 2015.

[19] Droidbench benchmarks, Aug. 2014. http://sseblog.ec-spride.de/tools/droidbench/.

[20] Backward compatibility for android applications. http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html. Accessed: 2015-08-22.

[21] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *International Conference on Software Engineering (ICSE)*, 2015.

[22] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *QRS*, 2015.

[23] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[24] Li Li. Boosting static analysis of android apps through code instrumentation. In *ICSE-DS*, 2016.

[25] Li Li, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Towards a generic framework for automating extensive analysis of android applications. In *The 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016)*, 2016.

[26] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[27] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, June 2012.

[28] Google play developer program policies. https://play.google.com/about/developer-content-policy.html. Accessed: 2015-07-22.

[29] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2015.

[30] Varargs. http://docs.oracle.com/javase/7/docs/technotes/guides/language/varargs.html. Accessed: 2015-08-22.

[31] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250. ACM, 2011.

[32] F-droid. https://f-droid.org. Accessed: 2015-08-22.

[33] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[34] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Compiler Construction*, pages 153–169. Springer, 2003.

[35] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[36] Android permission element. http://developer.android.com/guide/topics/manifest/permission-element.html. Accessed: 2015-08-22.

[37] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[38] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[39] Missing call edges (for spark, not cha). https://www.marc.info/?l=soot-list&m=142350513016832. Accessed: 2015-08-22.

[40] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[41] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[42] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[43] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[44] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behavior Under Contexts. In *International Conference on Software Engineering (ICSE)*, 2015.

[45] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[46] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Programming Languages and Systems*, pages 139–160. Springer, 2005.

[47] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. *ACM SIGPLAN Notices*, 34(11):2–11, 1999.

[48] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *Runtime Verification*, pages 364–381. Springer, 2013.

[49] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[50] Julian Schutte, Dennis Titze, and JM De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *TrustCom*, pages 370–379. IEEE, 2014.

[51] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *AsiaCCS*, 2014.