Taming Reflection: An Essential Step Towards Whole-Program Analysis of Android Apps

XIAOYU SUN, Monash University, Australia
LI LI*, Monash University, Australia
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg
JACQUES KLEIN, University of Luxembourg, Luxembourg
DAMIEN OCTEAU†, Pennsylvania State University, USA
JOHN GRUNDY, Monash University, Australia

Android developers heavily use reflection in their apps for legitimate reasons. However, reflection is also significantly used 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 incomplete, given the measures taken by malware writers to elude static detection. We propose a new instrumentation-based approach to address this issue in a non-invasive way. Specifically, we introduce to the community a prototype tool called DroidRA, which reduces the resolution of reflective calls to a composite constant propagation problem and then leverages the COAL solver to infer the values of reflection targets. After that, it automatically instruments the app to replace reflective calls with their corresponding Java calls in a traditional paradigm. Our approach augments an app so that it can be more effectively statically analyzable, including by such static analyzers that are not reflection-aware. We evaluate DroidRA on benchmark apps as well as on real-world apps, and demonstrate that it can indeed infer the target values of reflective calls and subsequently allow state-of-the-art tools to provide more sound and complete analysis results.

ACM Reference Format:
Xiaoyu Sun, Li Li, Tegawendé F. Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2020. Taming Reflection: An Essential Step Towards Whole-Program Analysis of Android Apps. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 35 pages. https://doi.org/10.1145/3440033

1 INTRODUCTION

Reflection is supported by some modern programming languages for enabling a running program to examine itself and its software environment, and to change what it does depending on what it finds [27]. In Java, reflection is used as a

*Li Li is the corresponding author.
†Damien Octeau is now with Google Inc., USA

Authors' addresses: Xiaoyu Sun, xiaoyu.sun@monash.edu, Monash University, Australia, Wellington Rd, Clayton, VIC, 3800; Li Li, li.li@monash.edu, Monash University, Australia, Wellington Rd, Clayton, VIC, 3800; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg, 2 Avenue de l'Universite, 4365 Esch-sur-Alzette, Luxembourg; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg, 2 Avenue de l'Universite, 4365 Esch-sur-Alzette, Luxembourg; Damien Octeau, docteau@google.com, Pennsylvania State University, USA; John Grundy, john.grundy@monash.edu, Monash University, Australia, Wellington Rd, Clayton, VIC, 3800.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

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. It allows developers, with the same application package, to target devices running different versions of Android. Developers often 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 [22, 48]. 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 of Android malware have shown that malware writers are using reflection as a powerful technique to hide malicious operations [34, 49, 53]. In particular, reflection can be used to hide the real purpose of an app e.g., by invoking a method at runtime to escape static scanning, or simply to deliver malicious code [12]. Most state-of-the-art approaches and tools for static analysis of Android simply ignore the use of reflection [26, 33] or may treat it partially [30, 62]. A review of recent contributions on static analysis-based approaches for Android shows that over 90% of around 90 publications [47] from top conferences (including ICSE and ISSTA) do not tackle this reflection problem. This means that many research and practice tools that provide analysis results 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. [61] has shown that reflection makes most current static analysis tools perform poorly on malware detection tasks.

Handling reflection is however challenging for static analysis tools. There exist ad-hoc implementations for dealing with specific cases of reflection patterns [62]. Such implementations cannot unfortunately be re-used in other static analysis tools. Nevertheless, there is a commitment in the Android research community to propose solutions for improving the analysis of reflection. For example, in a recent work [17], Barros *et al.* propose an approach for resolving reflective calls in their *Checker* static analysis framework [25]. Their approach however 1) requires application source code, which is not available for most Android apps, 2) targets specific check analyses based on developer annotations, which requires additional developer effort to 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 better analyze application packages from app markets. To that end, we present our new DroidRA instrumentation-based approach for automatically resolving 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. This is done by scanning all the files disassembled from APKs, such as a jar file with *classes.dex* inside, into our working space. Indeed, our reflection resolving approach hinges on the assumption that all the code that exists in the app package may be loaded at runtime and thus should be considered for analysis.

We make the following key contributions:

- We provide insights on the use of reflection in Android apps, based on an analysis of 1000 apps randomly selected from a repository of apps collected from Google Play, third-party markets, as well as malware samples [11]. Our findings show that 1) a large portion of Android apps rely on reflective calls, and that 2) reflective calls are usually used with some common patterns. We further show that we can discriminate between reflective calls in benign and malicious apps.
- We designed and implemented DroidRA – an approach that aims to boost existing state-of-the-art static analysis for Android by resolving reflection in Android apps. DroidRA models the use of reflection with COAL [57] 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, such as sensitive API calls, sensitive data leaks [14, 41], that was not previously visible to existing analyzers.
- We release DroidRA and its associated benchmarks as open source [4], not only to foster research in this direction, but also to support practitioners in their analysis needs.

This paper is an extended and improved version of previous work presented at the 2016 International Symposium on Software Testing and Analysis (ISSTA). In this extension, we have improved DroidRA in three aspects:

- **Improvement #1**: We introduce a probabilistic-like approach to strengthen our static reflection analysis, aiming at providing more sound and comprehensive reflection analysis results to support whole-program analysis of Android apps. Previously, when DroidRA cannot correctly infer the value of specific reflective calls, which is quite common because of the drawbacks of static analysis, DroidRA would yield a "*" as output indicating that this value can be anything. This output is not useful for users. Hence, in this improvement, when "*" occurs, we try to mitigate it by predicting its possible values using domain knowledge on top of static analysis results. Our experimental results show that this improvement can indeed help in resolving more reflective targets, compared to that of the previous version of DroidRA.
- **Improvement #2**: We further take *Fragment* into consideration for the reflection analysis process. Fragment is a special element that can be placed in an Activity component to form a piece of an app's user interface. A Fragment element has its own lifecycle methods for which each of them can access into reflective calls. If Fragment elements are overlooked, an inevitable part of code will be missed, leading to less reflective calls characterized. After integrating Fragment into our analyzing process, our tool is capable of reaching reflective calls that could not be reached previously, and thereby results in more reflective calls discovered and resolved.
- **Improvement #3**: Since static analysis is known to be time consuming, we introduce a feature into DroidRA to support customized analysis of Android apps. If a given component is known to not include any reflective calls, there is no need to include this component for reflective analysis. As a result, less code will be analyzed, and hence better performance can be achieved. As revealed in our experiments, the customized module can (1) reduce computational costs while keeping the reflection analysis results unchanged for such apps that are successfully analyzed before, and (2) enable the analysis of some apps that cannot be successfully analyzed if not customized.

```
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`. We remind the readers that while our approach proposed in this work works at the bytecode level of Android apps, we present as examples Java code snippets directly for the sake of readability.

Apart from tool enhancements, which are detailed in the approach section (i.e., sec.approach), we have also improved the manuscript in the following aspects. First of all, we conduct extensive new experiments to evaluate the prototype tool improvements. Specifically, we have doubled the number of selected apps, and all the apps are randomly selected from a dataset with the latest apps. We have also compared our approach with another state-of-the-art tool called DINA, except for comparing against the Checker framework. Second, apart from updating the results of the original research questions, we have also added two new research questions (i.e., RQ2 and RQ4) for evaluating the effectiveness of the newly introduced functions in the DroidRA enhancements. Third, we have now included a discussion section to discuss the impact of timeout setting on our experiments. Except for giving 10 minutes as timeout, which has been used in the experiments of the conference version and subsequently this extension, we further repeat the experiments with different timeouts (i.e., 1, 5, 20, 30, and 50 minutes). Our exploratory results show that 10 minutes is a reasonable timeout for applying DroidRA to analyze Android apps. Finally, we have significantly extended the related work discussion.

The remainder of this paper is organized as follows. Section 2 motivates our work through a concrete example and Section 3 investigates the use of reflection in Android apps. Section 4 presents our approach DroidRA, including its design and implementing details. Section 5 reports on the evaluation of DroidRA while Section 6 presents the limitations of DroidRA. Later, Section 7 discusses the related work and finally Section 8 concludes this paper.

2 MOTIVATION

Millions of applications are freely available for download by Android users. However, malware developers keep targeting the platform and trying to get their abusive applications into the Android ecosystem [28, 32, 44]. For example, 360 Security recently reported that over 500,000 new mobile malware variants targeting the Android platform were found in China in the first quarter of 2019 [65]. Also, as of early 2020, 1 out of every 1000 app installs from Google's official Play store is from a potentially harmful application, as shown in the recent Google Transparency Report.[1] These numbers demand practical and scalable approaches and tools to support security analysis of large sets of Android apps [28, 32, 54].

As example of such approaches, static taint analyzers aim at tracking data across control-flow paths to detect potential privacy leaks. Consider the state-of-the-art FlowDroid [14] 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 many promising results. However,

[1] https://transparencyreport.google.com/android-security/overview

it suffers from limitations inherent to the challenges of performing static analysis in Android where the presence of reflection, dynamic class loading and native code calling, makes static analysis very challenging. Reflection breaks the traditional call graph construction mechanism in static analysis, resulting in an incomplete control-flow graph (CFG) and consequently leading to incomplete analysis results. For example, a dynamically loaded class and reflectively called method can not usually be detected. In this paper, we focus on resolving reflection in Android apps to allow state-of-the-art tools such as FlowDroid to significantly improve their results. Dealing with reflection in static analysis tools is challenging. The Soot Java optimization framework, on top of which most state-of-the-art approaches are built, does not address the presence of reflective calls in its analyses. Thus, overall, resolving reflection at the app level will enable better analysis by state-of-the-art analysis tools to detect more potential security issues for app users.

Consider the case of an app included in the DroidBench benchmark suite [3, 14]. The Reflection3 benchmark app is known to be improperly analyzed by many tools, including FlowDroid, because it makes heavy use of reflective calls. Listing 1 shows part of this app. Here, class `ReflectiveClass` is first retrieved (line 3) and initialized (line 4). Then, two methods (*setImei()* and *getImei()*) from this class are reflectively invoked (lines 5-8). *setImei()*, which is selected by concatenating two strings at run-time, will store the device ID, obtained on 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 the user – phone number (line 10).

The operation implemented in this code sample is malicious as the sensitive, private information of device ID is sent to a number unknown to the user. 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 traditional control flow analysis. Thus, statically detecting such leaks becomes very hard. For example, bypassing property and method name detection by using run-time constructed string patterns, as done in line 5. Furthermore, even using simple string analysis is not enough to identify such malicious reflective calls. This is because both the method name (e.g., *getImei* for method *m2*) and its 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 investigated whether reflection is truly a noteworthy problem in the Android ecosystem. To this end, we investigated why and to what extent reflection is used in Android apps. In Section 3.1, we 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 run-time, by 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`[2]), uses reflection to effect the initialization of Collection List and Set.

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

```
//Example (1): providing genericity
Class collectionClass;
Object collectionData;
public XmlToCollectionProcessor(Str s, Class c) {
 collectionClass = c;
 Class c1 = Class.forName("java.util.List");
 if (c1 == c) {
  this.collectionData = new ArrayList();
 }
 Class c2 = Class.forName("java.util.Set");
 if (c2 == c){
  this.collectionData = new HashSet();
}}

//Example (2): maintaining backward compatibility
try {
 Class.forName("android.speech.tts.TextToSpeech");
} catch (Exception ex) {
 //Deal with exception
}

//Example (3): accessing hidden/internal API
//android.os.ServiceManager is a hidden class.
Class c = Class.forName("android.os.ServiceManager");
Method m = c.getMethod("getService", new Class[] {String.class});
Object o = m.invoke($obj, new String[] {"phone"});
IBinder binder = (IBinder) o;
//ITelephony is an internal class.
//The original code is called through reflection.
ITelephony.Stub.asInterface(binder);
```

Listing 2. Reflection usage in real Android apps.

Maintaining Backward Compatibility. In app *com.allen.cc* (`44B232`, an app for cell phone bill management), reflection techniques are used to check at run-time the *targetSdkVersion* of a device, and, based on its value, to realize different behaviors. A similar use scenario includes checking whether a specific class exists or not, in order to enable the use of advanced functionality when 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 [2] of ensuring backward compatibility for different devices and SDK versions.

Protecting Intellectual Property. 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 APIs. In the development phase, Android developers write apps that use the `android.jar` library package containing the SDK API exposed to apps. In contrast, when in production and 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 [46].

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 such 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 [11].

We randomly selected 500 apps from this repository and parsd the bytecode of each app, searching for the use of reflective calls. The strategy used consists in considering any call to a method implemented by the four reflection-related classes[3] 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 these selected apps making use of reflective calls. In fact, on average, each of the selected 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%)

However, many real-world ad libraries make extensive use of reflection. Thus, we performed another study to check whether or not most reflective calls are only contributed by common advertisement libraries. We thus exclude reflective calls that are invoked by common ad libraries[4]. Our results show that there are still 382 (76.4%) apps whose non-ad code includes reflective calls, suggesting the use of reflection in primary app code.

3.2.2 Patterns of Reflective Calls. In order to have a clearer picture of how one can find reflection in apps, we further investigate the sequences of reflective calls used and summarize the patterns used by developers to implement Android program behaviour with reflection. We consider all method calls within the selected 500 apps and focus on the reflection-related sequences that are extracted. We used a simple and fast approach considering the natural order in which the bytecode statements are yielded by Soot[5]. We find 34,957 such sequences (including 1 or more reflective calls). An isolated reflective call is relatively straightforward to resolve as its parameter is usually a String value (e.g., name of class to instantiate or method to call). However, when a method in the instantiated class must be invoked, other

[3] `java.lang.reflect.Field`, `java.lang.reflect.Method`, `java.lang.Class`, and `java.lang.reflect.Constructor`.
[4] We take into account 12 common libraries, which are published by [16] and are also used by [40]. We believe that a bigger library set like the one provided by Li et al. [45] could further improve our results.
[5] One of the most popular open-source framework that supports static analysis of Java/Android apps.

reflective calls may be necessary (e.g., to get the method name or parameter object type), which complicate reflection target resolution. We found 45 distinct patterns of sequences containing at least three reflective calls. Table 2 details the top five sequences of reflection calls in these apps. 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 identify the reflective call patterns that are potentially dangerous, as they may change program state. We 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 an app 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 us to consider different cases for reflective call resolution. In some simple cases, a string analysis is sufficient to extract the value of the call parameter. However, in other cases 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.

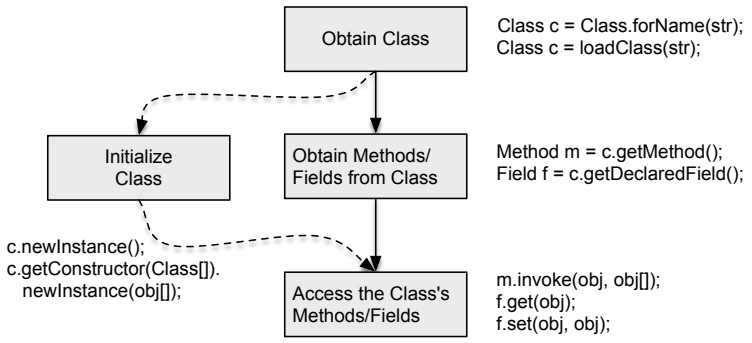


Fig. 1. Abstract pattern of reflection usage and some possible examples.

4 RESOLVING REFLECTION

We have two key aims: (1) to resolve reflective call targets in order to expose all program behaviors for static analysis, especially for analyses that must track private data, to produce more complete results; and (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 inspected. (2) The second module named *RAM* locates 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 into a new app where reflective calls are now 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 and complete results [38].

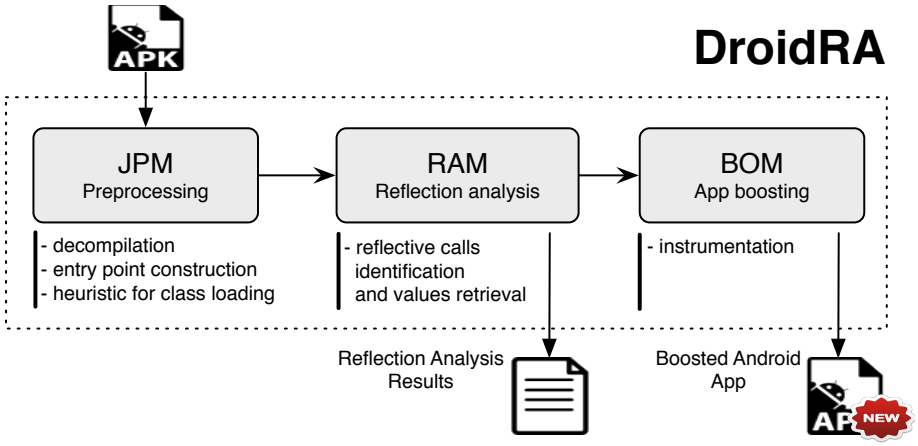


Fig. 2. Overview of DroidRA.

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 [36], a Java optimization framework. As a result, in a first module in DroidRA, JPM, leverages the Dexpler [18] translator to decompile the apk and output Jimple code.

Second, similarly to any other static analysis approaches for Android, DroidRA needs to start analysis from a single entry-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 Filters`, which define the capabilities of a component, is a possible entry-point. To address this challenge, we use the same approach as in FlowDroid [14]. This is to artificially assemble a dummy main method, taking into account all possible components including all possible callback methods (e.g., *onClick()*), and their lifecycle methods (e.g., *onCreate()* and *onStop()*). In

Android, there are four types of components: Activities, Services, Broadcast Receivers, and Content Providers. Each of these four types of components has its own lifecycle that is different from others and hence needs to be separately modeled and analyzed. Indeed, as an example of an activity's lifecycle shown in Fig. 3, all the methods, both lifecycle and callback, are not connected at the code level. They therefore need to be artificially connected in the dummy main method in order to allow the static analyzers to reach them.

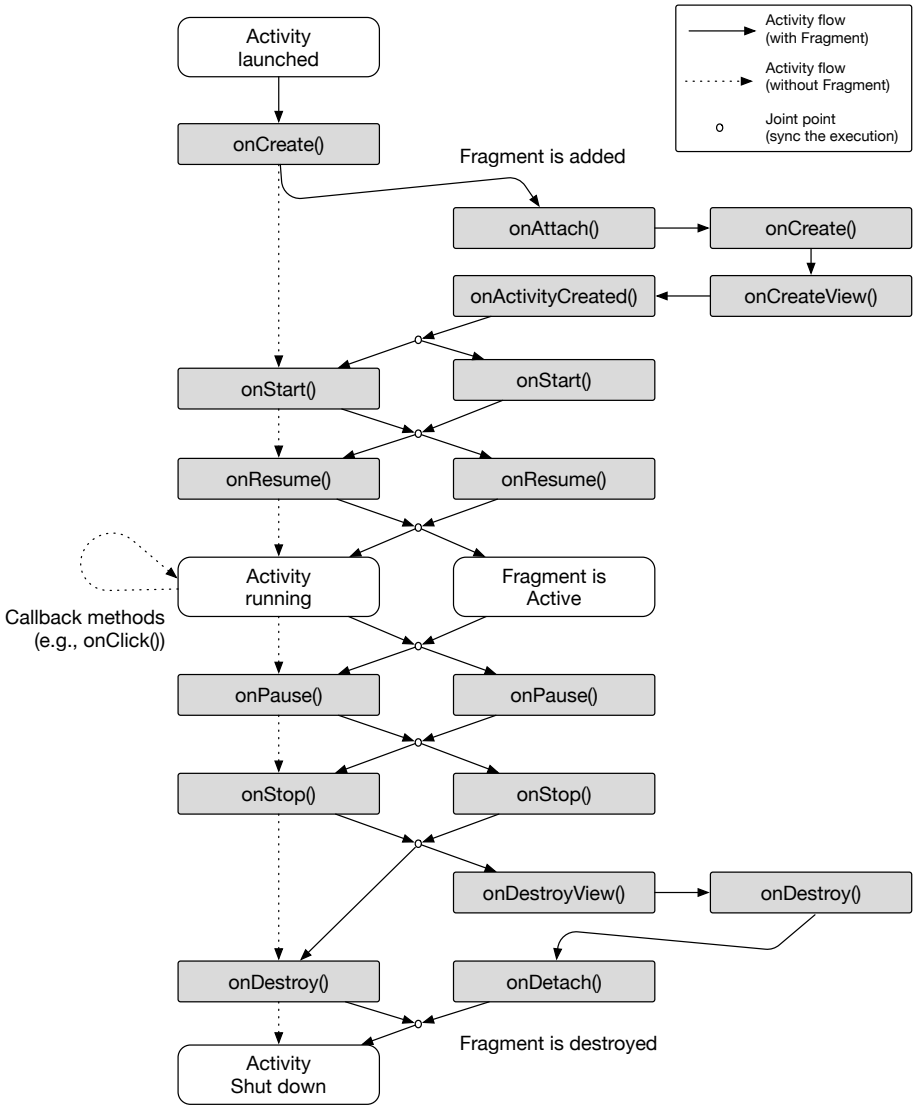


Fig. 3. The lifecycle of an activity.

Apart from these four components, there is a special element called the Fragment that also introduces challenges to static analysis. A Fragment, although it cannot be run independently, can be placed in an Activity component,

responsible for an app's user interface, to form a piece of the app's user interface and can be added or removed while the host Activity is running. Fragment is introduced to Android apps for achieving the following advantages: (1) Modularity: cohesive UI code can be encapsulated into a fragment rather than scattered in an Activity; (2) Reusability: a dedicated fragment can be leveraged by multiple Activities; and (3) Adaptability: according to the screen size or screen orientation of the hardware, different layouts can be adapted for fragments to enable better user experience.

Because of these, the Fragment concept has been heavily leveraged by app developers to implement Android apps. However, reflection might also be used in the implementation of Fragment, and therefore there is a need to take into account Fragments in our analysis. Listing 3 illustrates a Fragment-related code snippet extracted from a real-world Android app. Developers have to extend the *Fragment* class to create concrete Fragment implementations (e.g., *SmartBarFragment*). Reflection might also be used in the implementation of Fragments. Indeed, as shown in Listing 3, reflection (e.g., lines 28-31) has been leveraged by *getNavBarOverride()* to access system properties.

Unfortunately, when developing the initial version of DroidRA, the authors were not aware of Fragments and their potential impacts on static analysis of Android apps (i.e., less code reached). As a result, Fragments had been ignored, resulting in Fragment-related code blocks not being visited, and consequently, reflective calls leveraged by those code blocks not reached. To this end, there is a strong need to also take Fragments-related code into consideration. However, it is non-trivial to implement this as Fragments (like Android components) also introduce their own lifecycle and callback methods and these may also access reflective calls. Indeed, let us take Listing 3 again as an example, there are three lifecycle methods (i.e., *onCreate()*, *onCreateView*, *onViewCreated*) declared in the *SmartBarFragment* class. These lifecycle methods are not directly connected at the code level. Their execution sequences are also determined by the Android framework (hard to decide by statically parsing this code).

Hence, these lifecycle methods also need to be included in the generated dummy main method. Fig. 3 illustrates the lifecycle of an activity where it involves a fragment and its lifecycle methods (including the aforementioned three lifecycle methods). In this work, we include these methods into the dummy main method following the call relations shown in Fig. 3. When a Fragment is identified in Android activities, our approach will attach the Fragment's lifecycle and callback methods into the lifecycle of the activities. For example, there will be a call flow from the *onCreate()* method of the activity to the *onAttach()* method of the Fragment. When there is a joint point (e.g., after the *onActivityCreated()* method), an if-statement will be created in the dummy main method, which will subsequently generate two branches covering all the possible execution sequences of the lifecycle methods (e.g., either the *onStart()* method of the activity or the *onStart()* method of the Fragment). This enables the static analyzer to build an inter-procedural control-flow graph and consequently to traverse all the app code.

In order to support customized analysis of Android apps, we introduce an option allowing users to specify how the dummy main method should be built. Users can leverage this option to exclude such classes that have already been analyzed previously and have not been changed at the time of the new analysis if a regular reflective analysis is planned. This option further provides opportunities for the analysis to remove such methods that do not use reflective calls. To ensure a given method is reflection-free, we check not only the method itself but also its accessed methods, following the call graph constructed for the analyzed Android app. We believe this option will be helpful when the analyzed apps are large. Indeed, some complicated apps may cause static reflective analysis to be unable to terminate within a certain period of time. To overcome this problem, which is commonly encountered by many static analysis tools, users can leverage this option to exclude part of the codebase from the app, thereby only analyzing a part of the app code. In this way, the reflection analysis can finish and could yield useful results. Although these results are only partial for the whole app, it is nonetheless better than the former case where no results are obtained.

```java
public class SmartBarFragment extends Fragment {
 @Override
 public void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  mActivity = getActivity();
  mHandler = new Handler();
 }
 @Override
 public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
    savedInstanceState) {...}
 @Override
 public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);
  Log.d(TAG, "navigationHeight:" + UiUtils.getNavigationBarHeight(mActivity));
 }
}
public final class UiUtils {
 public static int getNavigationBarHeight(Context context) {
  hasNavBar(context);
  ...
 }
 private static boolean hasNavBar(Context context) {
  String sNavBarOverride = getNavBarOverride();
 }
 private static String getNavBarOverride() {
  String sNavBarOverride = null;
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
   try {
    Class c = Class.forName("android.os.SystemProperties");
    Method m = c.getDeclaredMethod("get", String.class);
    m.setAccessible(true);
    sNavBarOverride = (String) m.invoke(null, "qemu.hw.mainkeys");
   } catch (Throwable e) {}
  }
  return sNavBarOverride;
}}
```

Listing 3. Fragment example with simplified code snippets extracted from a real-world Android app.

Third, we aim to analyze the entire available app code, including such code that is dynamically loaded (e.g., at runtime). However, such dynamic Code Loading (DCL) is yet another challenge for static analysis. This is because some would-be loaded classes that are added at runtime e.g., downloaded from a remote server, may not exist at all at static analysis time. 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 at static analysis time. We assume that this way of locally storing the code to be dynamically loaded is most widespread. Indeed, in our previous work in understanding the piggybacking behaviors of Android apps, we have empirically found that a significant number of DEX files are distributed (inside the APK) via resource files such as image or XML files. 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 [7].

In practice, our DCL analysis is performed through heuristics. Given an app a, we first unzip[6] it and then traverse all its embedded files, noted as the 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 open it and check whether it contains a *dex* file through its magic number (*035*). All retrieved *dex* files, usually with *classes.dex*, are then taken into consideration for any further analysis of the app.

We tested our 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 that can be dynamically loaded at runtime. Among these 348 apps, we collected 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, is used by 115 apps. This library package was recently studied in a dynamic approach [73].

4.2 RAM – Reflection Analysis Module

The Reflection Analysis Module of DroidRA identifies reflective calls in a given app and maps their target string/object values. For example, consider the motivating example from the DroidBench app presented in Listing 1. The aim of 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 this motivating example and our study of reflective call patterns described previously, we observe that this can be modeled as a constant propagation problem within an Android Inter-procedural Control-Flow Graph. Mapping a reflective call eventually consists of resolving the value of its parameters – name and type – through a context-sensitive and flow-sensitive inter-procedural data-flow analysis. The purpose is to obtain highly precise results. This is very important since the app will be automatically instrumented without any manual check of these 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, then our RAM module might tell us that $m2$ belongs to class *TelephonyManager*, rather than the right class *ReflectiveClass*. During instrumentation, we will then write code calling $m2$ as a member of *TelephonyManager*, which would yield an exception at runtime i.e. 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. We leverage the composite COnstant propAgation Language (COAL) [57] for specifying this reflection problem. In order to use COAL, the first step is to model the reflection analysis problem independently from any app. We use the abstract patterns of reflective calls described in Section 3.2.2. This generic model is specified by composite objects. For example, a reflective method is 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 resolve 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 4 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 4,

[6]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 4. Example of COAL-based reflection analysis for class *Method*. Similar specifications apply for all other reflection classes

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 4:

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

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 4. The modifier `mod` statement specifies the signature of the `getMethod` method and it 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 [57], 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.

```
1 Object[] objs = new Object[2];
2 objs[0] = "TOSEM";
3 objs[1] = 2020;
4 m.invoke(null, objs);
5 //m(String,int)
```

Listing 5. Example of use of a varargs parameter.

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). For string analysis, the COAL solver introduces a flow-sensitive and interprocedural approach. This first gathers the dataflow facts for string variables and then determines regular sets that satisfy these facts. This analysis is implemented based on the flow-sensitive use-def analysis provided by the Soot framework [36]. By leveraging the Single Static Assignment (SSA) intermediate program representation, the analysis traverses all the instructions in all reachable functions to determine the corresponding constraints related to strings.

The COAL specification in Listing 4 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 4.

Improvements to the COAL Solver. We have contributed to several improvements of the COAL solver in this work. These improvements now enable it to perform efficiently for resolving the targets of reflective calls. At first, we extended the COAL language and its solver to be able to query the values of objects on which instance calls are made. 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 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 to use in practice.

We detail an example of a difficulty that we 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 [9]. 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 5. 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 the assignments to array elements on both lines 2 and 3. We thus infer that 1) the first parameter of m is a $String$ whose value is $TOSEM$, and 2) the second parameter is an int whose value is 2020.

Refinements to the COAL Outputs. Finally, although we have improved the COAL Solver in various aspects, specific reflective calls still cannot be resolved by COAL. In such a context, COAL will yield a star (i.e., "`*`") as output, indicating that the reflective call could refer to any method. Indeed, as revealed by Barros et al. [17], in some cases, it can be impossible for static approaches to resolve reflective calls because the reflective target can be a runtime user

```
1  public void submit(String editorDecision) {
2    Method action = TOSEM.class.getMethod(editorDecision, String.class, String.class);
3    String title = ...;
4    String author = ...;
5    action.invoke(null, title, author);
6  }
7
8  public class TOSEM {
9    public static void publish(String title, String authors) {}
10   public static void accept(int paperID) {}
11   public static void reject(int paperID) {}
12 }
```

Listing 6. Example of an unresolvable reflective method.

input. Furthermore, as argued by Octeau et al. [57], the COAL Solver shares the traditional limitations of static analysis on Java programs in that it does not handle native code and reflection. As a result, the constant propagation traces may be broken and thereby lead to unknown results (i.e., "*"). These unknown results provide meaningless information to users. To improve prior work and so as to refine the outputs of the COAL solver, we go one step further to approximate the possible values of "*" when our approach fails to reveal them. The approaches introduced by Octeau et al. [56], propose to combine static analysis with probabilistic models to improve static analyzers. Those provided by Smaragdakis et al. [64] and Li et al. [50] [51] propose to infer reflective targets by leveraging code information such as reflective API semantics and type systems in Java. In this work, we present a similar approach (i.e., essentially a subset of the inference system introduced by Li et al. [50] [51]) to "guess" the unknown reflective targets. The resolved targets are then integrated back to the results of the COAL solver to support further analyses. In this work, DroidRA rewrites the app by representing the resolved reflective calls with traditional Java calls. It subsequently generates a semantically equivalent new app for supporting reflection-free static app analyses.

More specifically, given a method called via reflection, ideally, we would need to infer the class for which the method belongs to, the method's name, the method's parameter numbers and types. Only by this we can be sure which method is called reflectively. Unfortunately, in practice, it may not be always possible to resolve all the aforementioned targets for a reflectively called method. In such cases where only partial targets are resolved, we propose to predict the unresolved targets based on domain knowledge i.e. their relationship to the *resolved* targets. In Android, we resort to the scheme defined in the app code, including that specified in the Android framework, to build domain knowledge. By statically scanning the code of a given app, we obtain the following code scheme i.e., domain knowledge: (1) the list of all involved classes; (2) given a class, except its name, we know all the methods and fields it declares, including its declared constructors i.e., methods that share the same name of their class; (3) given a method, we know the class it belongs to and the parameter numbers and types it contains; and (4) given a field, we know the class it belongs to and the type it is defined for.

To demonstrate the usefulness of leveraging the above domain knowledge, which can be obtained before the reflection analysis, for approximating the value of reflective targets, we use a concrete example to depict this refinement to COAL outputs. Listing 6 presents a sample code showing the basic usage of reflection in Android. In the beginning, a method is extracted from class *TOSEM* via reflection (line 2) and then reflectively invoked (line 5). Lines 8-12 enumerates the code structure of the TOSEM class. Unfortunately, since the method name of the reflective call is given as run-time input (line 1), our tool cannot infer its value correctly, resulting in "*" for the method name from COAL analysis.

```
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 7. The boosting results of our motivating example (Augmented app code lines are highlighted by the # symbol).

Nonetheless, except for the method name, our tool can correctly resolve the other relevant targets: (1) the method's class *TOSEM* and (2) the method has two parameters, and their types are both *java.lang.String*. We can then compare these resolved targets with our domain knowledge. The only match in our codebase would be the *publish* method, with the same class name and the same argument numbers and types (line 9). Therefore, we can confidently refine the "*" value to *publish*, as the output of the COAL solver. Note that in some cases the refined results might not be unique, and in such a case we report all of the possible results.

4.3 BOM – Booster Module

The Booster Module for DroidRA takes 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 appropriate standard Java calls. All reflective calls 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 [20] deals with reflection in standard Java software in this way, while IccTA [41] explicitly connects components, to improve Inter-Component Communication analysis.

Consider again our motivating example presented in Listing 1 to better illustrate the instrumentation done by BOM. Listing 7 presents the boosting results of Listing 1. Our instrumentation tactic is straightforward: for an instance where a reflection call initializes a class, we explicitly represent the statement with the Java standard *new* operator (line 4 in Listing 7). If a method is reflectively invoked (lines 5 and 8), we explicitly call it (lines 7 and 10). This instrumentation is possible thanks to the mapping of reflective call targets yielded by the RAM module. In this example the target resolution in RAM 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 needed: In line 1 of Listing 7, c is actually an object, yet the boosting logic requires information that this represents class name "ReflectiveClass".

Note also that the new injected code is always guarded by a conditional to construct a guarded control flow 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 run-time execution, *check()* always returns `false`, preventing paths added by BOM from ever being executed. Thus, this 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 some additional instrumentations that are not directly related to the Reflection problem. Nevertheless, these instrumentations are useful to improve the completeness of other static analyses. 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 to detect 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 explained above how DroidRA tackles this problem with its JPM module (cf. Section 4.1). However, not all classes that have to be loaded are actually accessible. One reason 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 find the password, we simply exclude such apps from our analysis. However, to allow tools such as FlowDroid to continue their analyses, we use 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 do not exist 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.

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 of creating the method *inc* – and its associated declaring 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 $Object$.

5 EVALUATION

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

RQ1 What is the coverage of reflection calls that DroidRA identifies and inspects?
RQ2 How does DroidRA compare with its earlier version for resolving reflective call targets in Android apps?
RQ3 How does DroidRA compare with state-of-the-art approaches for resolving reflective call targets in Android apps?
RQ4 Is the customization function of DroidRA useful for improving the performance of reflection analysis of Android App?
RQ5 Does DroidRA support existing static analyzers to build sounder call graphs of Android apps?
RQ6 Does DroidRA support existing static analyzers to yield reflection-aware results?

5.1 RQ1: Coverage of Reflective Calls

The goal of DroidRA's app reflection analysis is to provide the necessary information for analysts, other approaches, to better determine how reflections are used by an Android app. Thus, instead of considering all reflection-related

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

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 used in our common pattern in Figure 1 and are critical as they can be used by malware e.g. to exchange sensitive 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.

To investigate DroidRA's coverage of an app's reflection calls, we randomly select 1,000 apps from Google Play to set up our experiment. All 1,000 apps were released after 2018 (i.e., based on their last modified date). Instead of reusing the original corpus of 500 apps that are selected in the earlier conference version of this paper, we use Google Play to form the new dataset because we want to evaluate our approach based on the latest apps. Our original 500 apps were quite old and selected in 2015 when we were working on the first version of DroidRA. From each app with reflective calls, we extract two key items of information:

(1) **Reached:** The total number of reflective calls that are identified by our RAM reflection analysis module; and
(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 illustrated in Figure 4, which shows the performance of DroidRA in reaching reflective calls from the dummy main, and in resolving their targets. Unfortunately, as a static analyzer, DroidRA shares the same limitation of any other static analysis approaches – it cannot finish the analysis within limited time and hardware resources. Indeed, as experimentally demonstrated by Avdiienko et al. [16], their approach sometimes cannot finish the analysis of a single app in 24 hours on a computer server with 730 GB of RAM and 64 Intel Xeon CPU cores. In our experiment, we launch our experiment on a rather small server (with 32 GB memory and 28 CPUs), and a short timeout (with just 10 minutes).

Among the randomly selected 1,000 apps, 152 of them cannot be successfully analyzed by DroidRA. Some representative failing reasons include (1) exceptions of DroidRA due to malformed Android apps (e.g., no DEX file included), (2) exceptions of Soot and COAL, the underlying tools leveraged by DroidRA, (3) timeout errors, for which the analysis cannot finish in 10 minutes, etc. Therefore, in this experiment, we report the experimental results based on the remaining 848 apps. Overall, among the 848 successfully analyzed apps, we extract 13,073 reflection calls, among which the number of resolved reflective calls is 11,646, giving a resolution rate of 89%. This high rate experimentally shows the effectiveness of DroidRA in resolving reflective calls for Android apps.

The missing resolutions 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; and 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.

Regarding the accuracy of our approach, we go one step further to calculate the accuracy of our approach in pinpointing reflective calls in Android apps. Unfortunately, there is no known ground truth available for evaluating the usage of reflective calls in Android apps. We have to resort to a manual process to calculate the accuracy. In particular, among the 848 successfully analyzed apps, we randomly select 10 of them and manually look into their disassembled code to check whether the reported reflective calls are actually leveraged or not. Table 3 enumerates the selected apps (package name and version code), the analysis results of DroidRA, and the confirmed results of our manual analysis. In total, DroidRA can statically reach 437 reflective calls, among which 369 of them are successfully resolved while 68 of them remain to be "*" (i.e., fails to be resolved by the conference version of DroidRA and also fails to be optimized by the

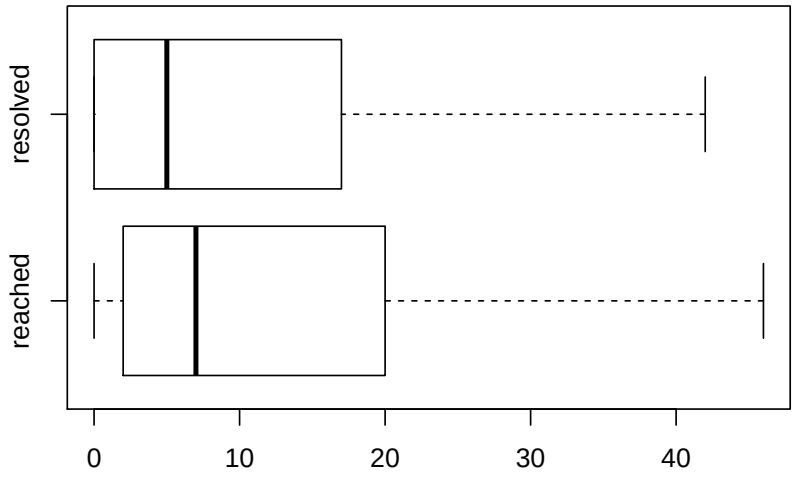


Fig. 4. Results of the coverage of reflection methods.

extended version). Among the resolved calls, our manual analysis reveals that 24 of them are inaccurate results, giving an accuracy of 93.4% (i.e., 345/369). We consider a result as inaccurate if (1) it contains more targets than it actually represents (i.e., in addition to the correct results, there are also false-positive results[8]), or (2) the resolved targets are incorrect (i.e., false-positive results). The latter case is mainly relevant to reflection-based field accesses. DroidRA fails to correctly pinpoint the reflectively accessed fields because Java fields do not provide additional information (such as parameter numbers and types, return types in method calls) to help in inferring their possible values. Overall, apart from a few inaccurate results, our approach (along with the newly introduced improvements) is useful for resolving reflection targets in Android apps.

Table 3. The experimental results of the manually checked apps.

App Name	Version	#. Reached Reflection Calls	#. Resolved Reflection Calls	#. Resolved (Accurate)	Unresolved (i.e., '*')
bhakti.sagar.aroma.clock.lwp	1.4	60	48	41	12
com.magzter.lichfieldgazette	4.0	41	39	37	2
com.jb.gokeyboard.theme.tmegreenkeyboardskkin	4.172.54.79	22	21	19	1
bhakti.sagar.aroma.clock.lwp	1.0	40	27	24	13
au.get.freshops	1.2.11.2	57	50	48	7
com.icontrolenergy.app	2.1.15	69	61	57	8
com.hhgregg.endlessblitz	15	33	29	29	4
com.andromo.dev456237.app434261	2.0	48	35	32	13
info.rguide.zzmtr	6.5.4	29	29	28	0
com.justinleingang.khw	1.0.0	38	30	30	8
	Total	437	369	345	68

[8]False-positive usually refers to a result that indicates a given condition exists when it does not. In this work, it refers to such results that are reported by our approach as such but are not presented in the corresponding app.

Table 4. The comparison results between the current version and the conference version of DroidRA.

Tool Version	# Analyzed Apps	# Reached Reflective Calls	# Resolved Reflective Targets	Resolved Rate	Common Apps	# Reached Reflective Calls	# Resolved Reflective Targets	Resolved Rate
DroidRA (current)	848/1,000	13,073	11,646	89.1%	742	12,481	11,349	90.9%
DroidRA (earlier)	819/1,000	11,952	9,375	78.4%	742	10,611	8,451	79.6%

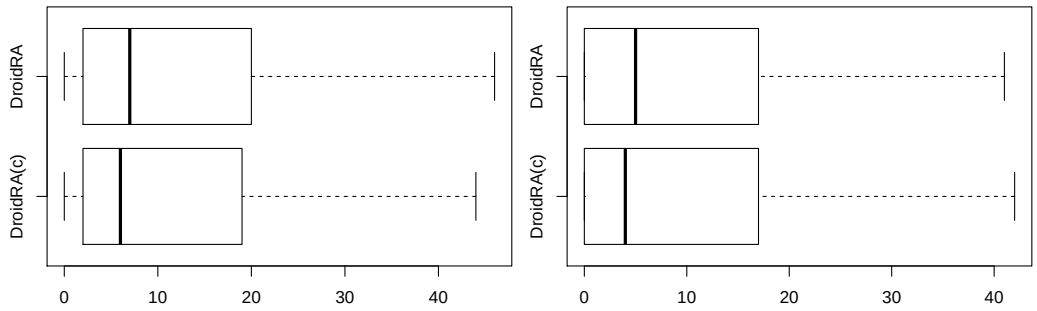
5.2 RQ2: Comparison with the earlier version of DroidRA

As revealed in the previous subsection, compared to the original version of DroidRA (as reported in the conference paper), which achieves only 81.2% of resolving rate, the new version of DroidRA has exceeded the original version by 7.88%. This evidence demonstrates the effectiveness of our improvements newly contributed to DroidRA. In the second research question, we now give more details about the performance growth.

In this extended version, we have improved DroidRA from three angles: (1) approximating possible values for such targets that cannot be resolved originally; (2) taking Fragments into consideration for reflection analysis; and (3) providing a means to customize the code to be analyzed. The first two improvements aim at enhancing the static analysis capability of DroidRA: the first improvement attempts to increase the resolving rate of reflection targets, while the second improvement aims at expanding the coverage of reflective calls that can be reached by DroidRA. In this research question, we will mainly evaluate the effectiveness of the first two improvements. The last improvement, which mainly looks at improving the performance of DroidRA in terms of time and memory usages, will be evaluated later in an independent research question.

To set up the experiments for comparison, we launch the earlier version of DroidRA on the same 1,000 apps selected in answering the RQ1. The experiments are executed under the same environment, i.e., the same server and the same timeout. Table 4 summarizes the experimental results. As shown in the second column, a similar number of apps are successfully analyzed by the two versions of DroidRA. There are 29 apps that are more analyzed by our extended version, compared to that of the conference version. This result is expected, as we have introduced into DroidRA various improvements. Some changes have been made to reduce the time and memory complexities, which allow DroidRA to analyze more apps under the same timeout, significant improvements are made to improve the analysis capabilities (e.g., covering more code). Although this increases the time and memory complexities, our extended version of DroidRA achieve better performance due to the fact that more number of apps can be successfully analyzed. The number of reached and resolved reflective calls (i.e., 13,073 and 11,646, respectively (or 89.1% resolving rate)) is larger than that of the conference version, which is respectively 11,952 and 9,375 (or 78.4% resolving rate). This result experimentally demonstrates the effectiveness of our improvements for DroidRA towards resolving reflective calls in Android apps.

If we only consider the common apps (742 apps as shown in the fifth column) that are successfully analyzed by both versions, the reached and resolved reflective calls of the current version are significantly larger than that yielded by the earlier version (as shown in the sixth and seventh columns). Interestingly, as far as reflective calls concerned, the numbers of reached reflective calls collected from the new dataset (cf. Fig. 5 (a)) are also much larger than that obtained from the original 500 apps. This result suggests that the latest Android apps may leverage more reflective calls that older Android apps. As illustrated in Fig. 5, the distribution of the number of reached and resolved reflective calls in each app yielded by the earlier version is also significantly less than that of the current version. This significance is confirmed by Mann-Whitney-Wilcoxon (MWW) tests, for which the resulting *p-value*s are both less than $\alpha = 0.001$. Given a significance level $\alpha = 0.001$, if $p - value < \alpha$, there is one chance in a thousand that the difference between the two datasets is due to a coincidence.



(a) Reached Calls. (b) Resolved Calls.

Fig. 5. Distribution of the number of reached and resolved reflection calls between the current version (i.e., DroidRA) and the conference version (i.e., DroidRA(c)) of DroidRA.

We now go one step further to break down the experimental results brought by the first two tool improvements. Using the newly introduced approximation model i.e., the first improvement, 1,068 reached reflective targets – which could not be resolved by the earlier version of DroidRA – are now resolved by DroidRA. This improvement allows DroidRA to additionally resolve 10% of its reached reflective calls (over 90%), which experimentally demonstrates that the approximation model is effective for DroidRA to resolve the possible targets of reflective calls.

Second, the inclusion of Fragments i.e. the second improvement, enables DroidRA to additionally discover 1,870 reflective calls, among which 1,815 (or 97.0%) of them are further resolved by the current version of DroidRA. This evidence further empirically shows that the second improvement we integrate into DroidRA is useful and practical. Note that the number of additionally reached reflective calls brought by Fragments is quite small compared to the total number of reached reflective calls. Aiming at understanding the possible reasons behind this, we check how are Fragments used by Android apps and how often do they access reflective calls. Among the 742 common apps, a scan of their code reveals that 419 of them have leveraged Fragments, suggesting that Fragment has been frequently leveraged by Android apps and thereby should not be ignored when statically analyzing Android apps. Moreover, among the 419 apps, 107 of them have actually accessed reflective calls in Fragments. This evidence further shows that there is a large portion of apps accessing reflective calls over Fragments, which demonstrates the necessity of taking Fragments into consideration for reflection analysis of Android apps.

5.3 RQ3: Comparison with the state-of-the-art

We compare our approach with some state-of-the-art works targeting the problem of resolving reflective targets. Several other proposed approaches include: Smaragdakis et al. [64], SOLAR [51], Elf [50], EdgeMiner [23], Ripple [72], DL2 [68], StaDynA [73], DINA [10], the Checker framework [17]. Unfortunately, Smaragdakis et al., SOLAR, and Elf are designed for analyzing Java applications and hence cannot be directly applied to analyze Android apps. Indeed, as replied by the authors of SOLAR to our request about launching their approach to analyze Android apps, SOLAR, at the moment, only theoretically works for Android apps. Additional non-trivial extensions are needed to apply it for this purpose. Therefore, we cannot compare our approach with these Java-focused analyzers.

State-of-the-art tools EdgeMiner and Ripple, although being proposed for Android apps, do not specifically focus on resolving reflective calls in Android apps. Indeed, the objective of EdgeMiner is to mine the Android framework for

pinpointing implicit control flow transitions. It does not consider the whole Java reflection mechanism when conducting the analysis. Ripple only attempts to resolve reflection for Android apps in incomplete information environments. We believe it is not fair to compare DroidRA against these two approaches. The authors of Ripple have endorsed our decision after we communicated with them in this regard.

Another three tools proposed by our fellow researchers, namely DL2, and StaDynA, and DINA, combine both static and dynamic analyses to analyze Android apps. These three tools do not directly focus on the analysis of reflective calls. All of them are proposed for detecting security issues, such as privacy leaks hidden by dynamically loaded code. Nonetheless, since dynamically loaded classes have to be accessed via reflection, we consider these approaches are relevant for comparison with our approach. Unfortunately, both StaDynA and DL are not made publicly available and we can only compare our approach with DINA in this work.

Finally, the approach proposed by Barros et al. [17] is another closely related work to ours. Their work presents an approach, referred to as *Checker*, to address reflection in the Information Checker Framework (IFC) [25]. This tool has been made publicly available in the community. We also compare Checker with our DroidRA approach.

The Checker framework has been evaluated based on a dataset consisting of 10 real-world apps crawled from the F-Droid open-source apps repository [5]. Checker has been evaluated by providing statistics on methods and constructors related to reflective invocations. We thus consider the same settings for the comparison. Table 5 lists the 10 apps and provides comparative results for Checker, DINA, and DroidRA. It is worth noting that, by comparing with Checker, 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. When comparing with DINA, our approach is purely static, while DINA is a hybrid approach combining advantages of both static and dynamic analyses.

Table 5. The comparison results of Checker, DINA, and DroidRA.

App	Checker		DINA		DroidRA	
	methods	constructors	methods	constructors	methods	constructors
AbstractArt	1	0	1	0	1	0
arXiv	14	0	13	0	14	0
Bluez IME	4	2	4	2	4	2
ComicsReader	6	0	8^{γ}	0	7	0
MultiPicture	1	0	0	0	1	0
PrimitiveFTP	2	0	1	4	2	7
RemoteKeyboard	1^{α}	0	1^{γ}	0	0	3
SuperGenPass	1	0	0	0	1	0
VimTouch	3^{β}	0	1	0	2	0
VLCRemote	1	0	1	0	1	0

$^{\alpha}$ Reached but not resolved.
$^{\beta}$ One from dead code.
$^{\gamma}$ Contain a reflective call that is overlooked by DroidRA.

Overall, as shown in Table 5, on this dataset DroidRA resolves 9 more methods/constructors than Checker. For the app *RemoteKeyboard*, DroidRA missed one method and Checker reports that it is not able to resolve it either. On further investigation, we observe that it is impossible for static approaches to resolve the reflective call in this case as the reflection target is read from configuration files at runtime. Indeed, as illustrated in Listing 8, the class name of the

```
1  public class RemoteKeyboardService extends InputMethodService implements
       OnKeyboardActionListener {
2   public void onCreate() {
3    InputStream inputStream = assetManager.open("telnetd.properties");
4    props.load(inputStream);
5    if (telnetServer == null) {
6     telnetServer = TelnetD.createTelnetD(props);
7   }}
8   public static TelnetD createTelnetD(Properties main) throws BootException {
9    TelnetD td = new TelnetD();
10   td.prepareShellManager(main);
11 }}
12 public class ShellManager {
13  //key is read from property configuration files
14  public Shell getShell(String key) {
15   Object obj = shells.get(key);
16   if(obj instanceof Class){
17   Class shclass = (Class) obj;
18   Method factory = shclass.getMethod("createShell", null);
19   myShell = (Shell) factory.invoke(shclass, null);
20 }}
```

Listing 8. Simplified source code extracted from app RemoteKeyboard.

reflective method is loaded from configuration file “telnetd.properties". 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. It thus 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 positive of DroidRA. However, with *ComicsReader*, DroidRA missed one method[9], 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 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.

When comparing DroidRA to DINA on the same dataset, DroidRA is able to resolve nine more reflective methods or constructors. This is expected as DINA is a hybrid approach, which relies on dynamic testing to resolve reflective calls. More specifically, the dynamic module of DINA is realized via monkey [6], the default dynamic testing tool provided by Google. It is known that the monkey tool has code coverage limitations, where certain codes in the app may not be able to be explored, resulting in less reflective calls resolved. Nevertheless, dynamic analysis techniques also come with advantages that cannot be simply achieved by static analysis. Indeed, as shown in Table 5, DINA has additionally resolved two reflective calls. One is related to the reflective call lying in the RemoteKeyboard app we discussed previously. This reflective call involves external configuration files that cannot be simply parsed statically. The other case is related to the ComicsReader app. DINA resolves a reflective call that has been overlooked by both DroidRA and Checker. This reflective call, as shown in Listing 9, is achieved in two steps where the reflectively accessed method is stored as a class attribute (i.e., *mSetSystemUiVisibility*) and then invoked in another class (line 11). This case is non-trivial to be handled by static analysis approaches in general as it requires to model the invocation sequence of the

[9] *View.setSystemUiVisibility()*.

```
1  public class FullImageView extends View {
2   private Method mSetSystemUiVisibility;
3   public void initFullImageView() {
4    try {
5     mSetSystemUiVisibility = getClass().getMethod("setSystemUiVisibility",
        Integer.TYPE);
6    } catch (NoSuchMethodException e) {}
7   }
8   public boolean setFullScreen(boolean fullscreen) {
9    if (mSetSystemUiVisibility == null)
10    return false;
11   mSetSystemUiVisibility.invoke(this, newVis);
12  }
13 }
```

Listing 9. Simplified source code extracted from app ComicsReader.

code. This experimental evidence shows that dynamic analysis can indeed be useful to supplement the capabilities of static analyses. As of future work, we plan to also integrate dynamic analysis into DroidRA to improve its performance in resolving reflective calls in Android apps.

5.4 RQ4: Usefulness of the customized reflection analysis

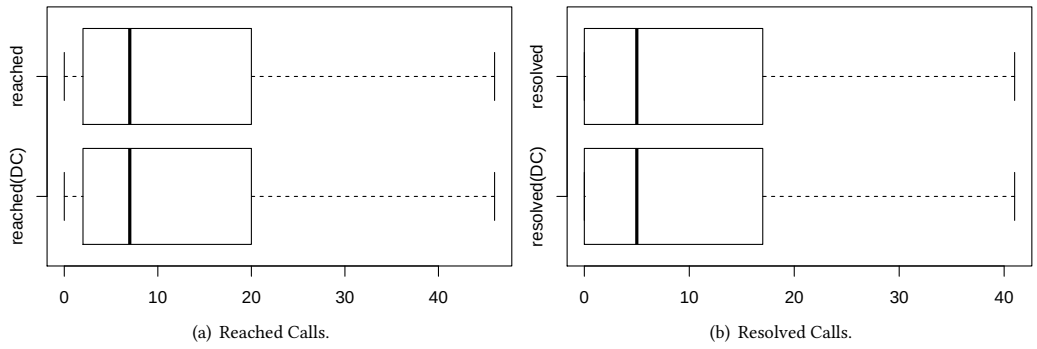
We introduced three main improvements to DroidRA. The second research question has empirically demonstrated the effectiveness of the first two improvements (the inclusion of Fragments and the approximation of unresolved targets). We now empirically evaluate the third improvement, which enables customized reflection analysis for Android apps, aiming at improving the time and memory complexities of the analysis of DroidRA.

For the sake of simplicity to demonstrate the usefulness of the customization capability, we introduce a simple customization strategy to DroidRA. Since the objective of DroidRA in this work is to resolve reflection targets in Android apps, our customization enforces DroidRA to only analyze components that actually use reflective calls. In other words, if a component is known to not involve reflection, it will not be included for analysis. To implement this, we have implemented a static analyzer to check whether a component accesses reflection or not. This checking nonetheless is not straightforward as components may not directly access reflective calls but access other classes or libraries that may further access reflective calls. We make sure that our static analyzer also checks such indirect reflective calls through an inter-procedural static analysis, implemented on top of Soot.

By applying the aforementioned simple customization strategy, we aim at evaluating the customization capability of DroidRA in two aspects:

- (1) reducing the time and memory usage while achieving the same results in terms of resolving reflective calls in Android apps.
- (2) being able to successfully analyze Android apps that cannot be done otherwise. In this aspect, we apply our customization strategy to the 152 apps that cannot initially be analyzed by DroidRA as reported in RQ1 experiments.

For the first aspect, we apply our customization strategy to the 848 apps that have already been successfully analyzed by DroidRA, on the same server with the same 10 minutes timeout. We then compare the obtained results with the original results yielded by DroidRA. Fig. 6 illustrates these reflection analysis results. The distribution of the number of



(a) Reached Calls. (b) Resolved Calls.

Fig. 6. Distribution of the number of reached and resolved reflection calls between the current version (i.e., DroidRA) of DroidRA and the version with dedicated customization (i.e., DroidRA(DC)).

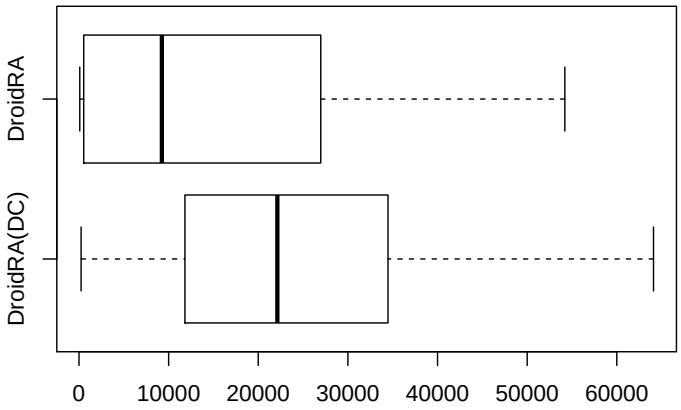


Fig. 7. Distribution of the size of call graph between DroidRA (without involving customization strategy) and the version with dedicated customization (i.e., DroidRA(DC)).

reached and resolved reflection calls is generally the same, suggesting that our customization strategy does not impact the underlying reflection analysis of DroidRA.

Fig. 7 shows that while keeping the same reflection analysis results, the call graph size of the current version without using a customization strategy is significantly larger than that of the version with customization, confirmed by a MWW test. The call graph size is critical to the time and memory complexities of the reflection analysis as all the methods reached on the call graph need to be statically visited and explored to uncover potential reflective calls. This empirical evidence confirms the effectiveness and usability of the customization capability towards reducing the time and memory complexities while achieving more or less the same reflection analysis results.

Regarding the second aspect, among the 152 apps that could not initially be analyzed by DroidRA, by employing the customization strategy to DroidRA, under the same execution environment (i.e., same server and timeout), 57 of them can now be successfully analyzed. This leads to 1,368 newly reached reflective calls, among which 1,280 of them are further successfully resolved. This experimental evidence further demonstrates the usefulness of the customization capability we have introduced, as the third improvement, to DroidRA.

5.5 RQ5: Call Graph Construction

An essential step of performing precise and sound static analysis is to build at a complete method call graph (CG) for an app. This will be used by static analyzers to visit all the reachable code, and thus perform a sound analysis. Indeed, methods not included in the CG will 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 [36]. We use both the CHA [24] algorithm, which is the default algorithm for CG construction in Soot, and the more recent Spark [37] 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.

In our study dataset of 500 apps, on average 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 6. The CG edges added (Diff column) vary between apps. We have further analyzed the added edges to check whether they reach sensitive API methods[10] (`ActivityManager.getRunningTasks(int)`) which are protected by a system permission (`GET_TASKS`). The recorded number of such newly reachable APIs (Perm column) 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.

Table 6. The call graph results of three apps we highlight for our evaluation. Permission column means the number of call graph edges that are actually empowered by DroidRA and are accessing permission protected APIs.

Package	Algorithm	Original	DroidRA	Difference	Permission
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

Consider the example of app *org.bl.cadone* which further highlights the improvement in CG construction by DroidRA. 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). 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.

Figure 8 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 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 [1]. This further suggests that the corresponding reflective calls were meant to hide dangerous actions.

[10]The list of sensitive API methods are collected from PScout [15].

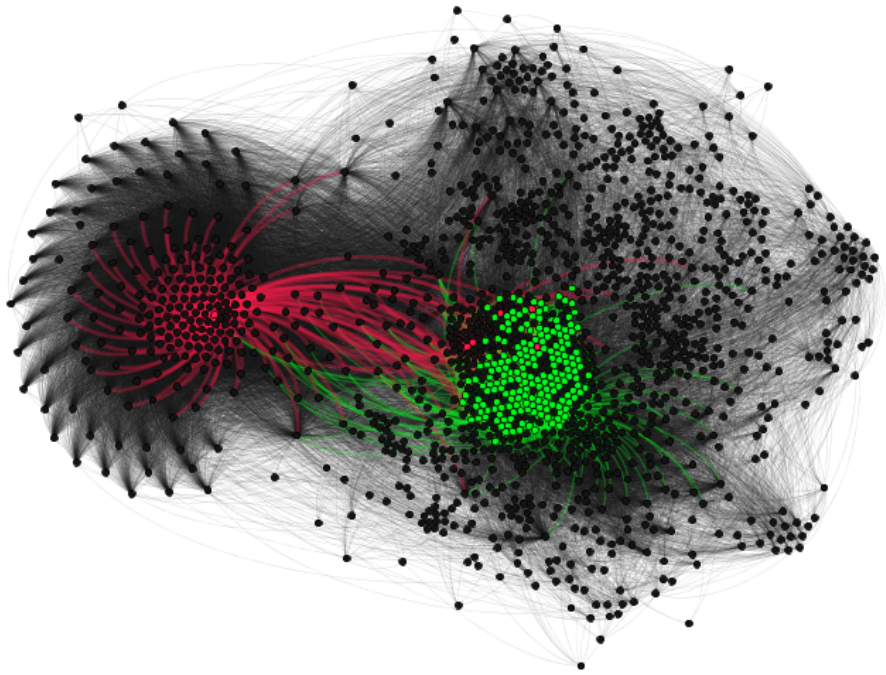


Fig. 8. The class dependency graph (simplified call graph) of app *org.bl.cadone* (based on CHA algorithm). **Black** color shows the originally edges/nodes, green color shows edges/nodes that are introduced by DroidRA while red color shows new edges that are further protected by permissions.

5.6 RQ6: Improvement of Static Analysis Results by DroidRA

We use 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. On the real-word apps, we check whether the runtime performance of DroidRA will not prevent its use in complementing 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 that include other reflective call patterns (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 7 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. This is 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 of the other 12 test cases. This was expected since IccTA is not a reflection-aware approach. When reflection is resolved 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 our current DroidRA implementation of constant propagation cannot

Table 7. 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()	✗	✗

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, as discovered by IccTA. Compared to using IccTA on the 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.

We further investigate the time overhead brought by the instrumentation of BOM to check whether it will be an obstacle to support practical usage in complement with state-of-the-art static analyzers. On the previous set of apps, we observe that the overheads are always within a minute. This value is reasonable (and neglectable) compared to the execution time of static analysis tools such as IccTA or FlowDroid, which can run for several minutes and even hours on a given app.

6 DISCUSSION AND LIMITATIONS

Observant readers may have noted that we have set 10 minutes as timeout for all the reflection analyses we have conducted in this work. In some circumstances this timeout may not be adequate for DroidRA to completely analyze an Android app. Indeed, among the 1,000 randomly selected apps, 152 of them cannot be successfully analyzed by DroidRA within that time, as demonstrated in Section 5.1. We would like to stress that the unsuccessful cases may not necessarily be related to timeout (e.g., could be tool-chain exceptions). Since time performance is known to be a common issue for static code analyzers, like many other static Android app analysis approaches, we prioritize precision and accuracy over computational cost. We believe that enlarging the timeout may enable DroidRA to successfully analyze more apps, but this may not impact the tool's capabilities of locating and resolving reflective calls in Android apps. Nevertheless, to evaluate this hypothesis, we check how this timeout period could impact on our DroidRA reflection analysis approach. To this end, we re-ran DroidRA on the selected apps with different timeouts: 1, 5, 10, 20, 30, and 50 minutes. For the sake of time efficiency, instead of considering the whole 1,000 apps, we take into account only 304 apps that are formed as (A) 152 apps randomly selected from the set of apps that are successfully analyzed in 10 minutes, and (B) 152 apps that cannot be successfully analyzed by DroidRA with 10 minutes timeout.

Table 8 presents these experimental results. With 1 minute as timeout, as expected, DroidRA cannot finish its analysis on any of the selected apps. This situation is dramatically changed when the timeout is set to 5 minutes: around two-thirds of the apps in Group-A can now be successfully analyzed, although the number of analyzed apps in Group-B remains to be zero. With 10 minutes as timeout, we observe the same experimental results as what we have presented previously: all the apps in group A are now successfully analyzed while all the apps in Group-B are not. This result is not changed, even when we increase the timeout to 20 minutes. Only when we set the timeout to be 30 and 50 minutes, the apps in Group-B start to be successfully analyzed. In our experiments, there are 11, and 14 apps (out of the 152 failed apps) are successfully analyzed when 30 and 50 minutes are given as timeout, respectively. This experimental evidence shows that 10 minutes is a suitable timeout for our study as the majority of apps can be successfully analyzed within this time (cf. Group-A) and such apps that cannot be analyzed within 10 minutes are unlikely to be analyzed even when the timeout is significantly increased.

Table 8. The numbers of successfully analyzed apps with different timeouts.

Group	1 minute	5 minute	10 minute	20 minute	30 minute	50 minute
Group-A (152 successfully analyzed apps)	0	105	152	152	152	152
Group-B (152 unsuccessfully analyzed apps)	0	0	0	0	11	14

Limitations. The main threats to validity of our evaluations of DroidRA are carried over 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. The conservative setting where a string is represented by a regular expression ("*") if COAL cannot statically infer its value can be taken as everything and thus may also introduce false positives. We have applied a probabilistic model attempting to mitigate this threat by leveraging domain knowledge in developing Android apps [56] [72]. However, for some targets, it still remains to be unresolved (i.e., "*"). In some other cases, although resolved successfully by the probabilistic model, these may include false-positive results. Indeed, the probabilistic model may yield multiple sets of results, among which only one of them is correct. Users of DroidRA hence need to put additional effort (such as taking type and data-flow information into consideration) to exclude such false-positive results. Fortunately, as shown in our experimental results, the fact that only a small portion of results are inaccurate, shows that our approach is practical in taming reflection for supporting the whole-program analysis of Android apps.

The current implementation of our approach may also confront false-negative results.[11] The single entry-point method (i.e., 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 (and thereby result in false-negative results). 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, which would again introduce false-negative results to our approach. However, our objective in this paper was not to solve the DCL problem. Other approaches [58] [73] can be used to complement our work. As of our future work, we plan to continuously improve our approach to address the aforementioned challenges so as to reduce false-negative results as much as possible.

[11] False-negative usually refers to a result that wrongly indicates that a given condition does not hold. In this work, it refers to such results that are missed by our approach, although they do exist in the corresponding app.

Since our experiments are conducted based on 1000 randomly selected Google Play apps, our results may overly influenced by these apps and hence may not generalize to other apps. Nevertheless, compared to the results obtained over the previously randomly selected 500 apps, the fact that DroidRA achieves good results on the newly selected 1000 apps suggests that this threat might be limited.

The code dependency analysis we applied to identify "reflection-free" code in Android apps may not be fully accurate. Our approach at the moment cannot guarantee that there is absolutely no dependency between the retained and excluded code. Moreover, the call graph built by RAM leverages the implementation of the Spark algorithm in Soot, which also comes with specific limitations [8]. Nevertheless, as shown in Section 5.4, this feature has been demonstrated to be effective in practice. This feature does provide an option for users to achieve a trade-off between the soundness of the analysis results and the time performance consumed by successfully analyzing an app.

The release date we leveraged to select apps is based on the last modification date of the app, which, unfortunately, is known to be not reliable. Indeed, as shown in our previous work [43], according to the last modification date, some apps may access APIs that do not yet exist at that time i.e., such APIs are introduced posterior to the last modification date. Nonetheless, the time information is not critical to our approach, and hence we believe its impact on our results is limited.

Finally, DroidRA handles neither native code, Javascript code, nor multi-threaded code. These are challenges that most current Android static analysis approaches ignore, and are for now out of the scope of this work.

7 RELATED WORK

Research on static analysis of Android apps presents strong limitations related to reflection handling [19, 29, 42, 69]. Authors of recent approaches explicitly acknowledge such limitations, indicating that they ignore reflection in their approaches [41, 57, 67] or failing to state whether reflective calls are handled [66] in their approach.

The closest work to ours was concurrently proposed by Barros et al. [17] within their Checker framework. Their work differs from ours in several ways: first, the design of their approach focuses on helping developers to check 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 an 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 whose performance can be boosted by our augmented app code with reflection resolutions.

Reflection, by itself, has been investigated in several works for Java applications. Most notably, Bodden et al. [20] have presented TamiFlex for aiding static analysis in the presence of reflections in Java programs. Similar 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 [73].

Recently, Li et al. [52] present a comprehensive understanding of Java reflection through examining its underlying concepts and its real-world usages. Building on this, the authors further introduce a static approach to resolve Java reflection in practice. This static approach has been integrated into SOLAR, a soundness-guided reflection analysis tool for Java [51]. The authors also introduced tools called Elf [50] and Ripple [72] for reflection analysis of Java and

Android apps. Elf infers the targets of reflective calls using the API semantics, such as class name, method name, method arguments, etc. Ripple takes into account incomplete information environments to improve the coverage of potential reflective calls. Smaragdakis et al. [64] present another static reflection handling technique to resolve reflective targets for Java programs. Some of the incomplete information environments discussed by these approaches could be leveraged to further refine our approximation model so as to help DroidRA resolve more reflective targets.

Another work that tackles reflection for Java has been done by Livshits et al. [55], in which points-to analysis is leveraged to approximate the targets of reflection calls. Unlike our approach, their approach needs users to provide a per-app specification in order to resolve reflections. This is difficult to apply for large scale analysis. Similarly, Braux et al. [21] 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. [58] have proposed a systematic review on how and why Android apps load additional code dynamically. In their work, they use 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 [73] 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 presents a coverage issue in dynamic execution. Our approach, DroidRA, provides a better solution for reflective method calls and 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 [13, 39]. For example, IccTA [41], a state-of-the-art ICC leaks analyzer, instruments apps to bridge ICC gaps and eventually enables inter-component static analysis. AppSealer [70] instruments Android apps for generating vulnerability-specific patches, which prevent component hijacking attacks at runtime. Other approaches [63, 71] apply the same idea, which injects shadow code into Android apps, to perform privacy leaks prevention.

In addition to statically infer the target values for reflective calls, various approaches to resolve the reflective targets dynamically have been proposed [35]. Hirzel et al [31] proposed an approach to handle dynamic features in Java at runtime through online pointer analyses. When new code is introduced by reflection, their approach will gradually take them into consideration and will incrementally update the points-to information. More recently, Rasthofer et al. [59, 60] introduce an approach for automatically extracting runtime values from Android apps. This approach can be leveraged to identify the reflective calls, which are frequently leveraged by malware samples to hide sensitive behaviors. All of these dynamic approaches can be leveraged to supplement our work towards statically taming the reflective calls in Android apps.

8 CONCLUSION

A long time challenge has been how to effectively perform a complete and sound reflection-aware static analysis of Android apps. We have presented an open source tool DroidRA to perform such reflection analysis. DroidRA models the identification of reflective calls as a composite constant propagation problem via the COAL declarative language. It leverages the COAL solver to automatically infer reflection-based values. DroidRA then uses a booster module, which is based on the previously inferred results to augment apps with traditional Java calls, augmenting app code with resolved reflective calls. This augmented app code provides a non-invasive way of supporting existing static analyzers in

performing highly sound and complete reflection-aware analysis, without any modification or configuration. Through various evaluations we have demonstrated the benefits and performance of DroidRA.

9 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments to the conference version of this extension. This work was partly supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020, by the Luxembourg National Research Fund (FNR) (under project CHARACTERIZE C17/IS/11693861), by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830892.

REFERENCES

[1] Android permission element. http://developer.android.com/guide/topics/manifest/permission-element.html. Last updated: 2019-12-27.
[2] Backward compatibility for android applications. http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html.
[3] Droidbench. http://sseblog.ec-spride.de/tools/droidbench/. Accessed: 2015-08-22.
[4] Droidra. https://github.com/MobileSE/DroidRA. Last updated: 2020-05-07.
[5] F-droid. https://f-droid.org. Accessed: 2020-03-16.
[6] Google. android monkey. http://developer.android.com/tools/help/monkey.html.
[7] Google play developer program policies. https://play.google.com/about/developer-content-policy.html. Accessed: 2015-07-22.
[8] Missing call edges (for spark, not cha). https://www.marc.info/?l=soot-list&m=142350513016832. Accessed: 2015-02-09.
[9] Varargs. http://docs.oracle.com/javase/7/docs/technotes/guides/language/varargs.html. Accessed: 2018.
[10] Alhanahnah, M., Yan, Q., Bagheri, H., Zhou, H., Tsutano, Y., Srisa-An, W., and Luo, X. Dina: Detecting hidden android inter-app communication in dynamic loaded code. *IEEE Transactions on Information Forensics and Security 15* (2020), 2782–2797.
[11] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *The 13th International Conference on Mining Software Repositories, Data Showcase track* (2016).
[12] Apvrille, A., and Nigam, R. Obfuscation in android malware, and how to fight back. *Virus Bulletin* (2014). https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation.
[13] Arzt, S., Rasthofer, S., and Bodden, E. Instrumenting android and java applications as easy as abc. In *Runtime Verification* (2013), Springer, pp. 364–381.
[14] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. 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).
[15] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 217–228.
[16] Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., and Bodden, E. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, IEEE, pp. 426–436.
[17] Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Armorim, M., and Ernst, M. D. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska, 2015), ASE.
[18] Bartel, A., Klein, J., Monperrus, M., and Le Traon, Y. 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).
[19] Batyuk, L., Herpich, M., Camtepe, S. A., Raddatz, K., Schmidt, A.-D., and Albayrak, S. 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* (2011), IEEE, pp. 66–72.
[20] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE* (2011), ACM, pp. 241–250.
[21] Braux, M., and Noyé, J. Towards partially evaluating reflection in java. *ACM SIGPLAN Notices 34*, 11 (1999), 2–11.
[22] Cai, H., Zhang, Z., Li, L., and Fu, X. A large-scale study of application incompatibilities in android. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)* (2019).
[23] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS* (2015).

[24] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (London, UK, UK, 1995), ECOOP '95, Springer-Verlag, pp. 77–101.

[25] Ernst, M. D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P., Bhoraskar, R., Han, S., Vines, P., and Wu, E. X. Collaborative verification of information flow for a high-assurance app store. In *CCS* (Scottsdale, AZ, USA, November 4–6, 2014), pp. 1092–1104.

[26] Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE* (2014).

[27] Forman, I. R., and Forman, N. *Java Reflection in Action (In Action Series).* Manning Publications Co., Greenwich, CT, USA, 2004.

[28] Gao, J., Li, L., Kong, P., Bissyandé, T. F., and Klein, J. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability (TRel) 70* (2019), 212–230.

[29] Gibler, C., Crussell, J., Erickson, J., and Chen, H. 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* (Berlin, Heidelberg, 2012), TRUST'12, Springer-Verlag, pp. 291–307.

[30] Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. Information-flow analysis of android applications in droidsafe.

[31] Hirzel, M., Dincklage, D. V., Diwan, A., and Hind, M. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS) 29*, 2 (2007), 11–es.

[32] Hu, Y., Wang, H., Zhou, Y., Guo, Y., Li, L., Luo, B., and Xu, F. Dating with scambots: Understanding the ecosystem of fraudulent dating applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2019).

[33] Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B. 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).

[34] Kazdagli, M., Huang, L., Reddi, V., and Tiwari, M. Morpheus: Benchmarking computational diversity in mobile malware. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2014), HASP '14, ACM, pp. 3:1–3:8.

[35] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., and Klein, J. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* (2018).

[36] Lam, P., Bodden, E., Lhoták, O., and Hendren, L. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)* (2011).

[37] Lhoták, O., and Hendren, L. Scaling java points-to analysis using spark. In *Compiler Construction* (2003), Springer, pp. 153–169.

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

[39] Li, L. Mining androzoo: A retrospect. In *The Doctoral Symposium of 33rd International Conference on Software Maintenance and Evolution (ICSME-DS 2017)* (2017).

[40] Li, L., Allix, K., Li, D., Bartel, A., Bissyandé, T. F., and Klein, J. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *QRS* (2015).

[41] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and Mcdaniel, P. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE* (2015).

[42] Li, L., Bartel, A., Klein, J., and Le Traon, Y. 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)* (2014), IEEE.

[43] Li, L., Bissyandé, T. F., and Klein, J. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)* (2018).

[44] Li, L., Bissyandé, T. F., and Klein, J. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering (TSE)* (2019).

[45] Li, L., Bissyandé, T. F., Klein, J., and Le Traon, Y. 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).

[46] Li, L., Bissyandé, T. F., Le Traon, Y., and Klein, J. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)* (2016).

[47] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Le Traon, Y. Static analysis of android apps: A systematic literature review. *Information and Software Technology* (2017).

[48] Li, L., Bissyandé, T. F., Wang, H., and Klein, J. Cid: Automating the detection of api-related compatibility issues in android apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)* (2018).

[49] Li, L., Li, D., Bissyandé, T. F., Klein, J., Le Traon, Y., Lo, D., and Cavallaro, L. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)* (2017).

[50] Li, Y., Tan, T., Sui, Y., and Xue, J. Self-inferencing reflection resolution for java. In *European Conference on Object-Oriented Programming* (2014), Springer, pp. 27–53.

[51] Li, Y., Tan, T., and Xue, J. Effective soundness-guided reflection analysis. In *International Static Analysis Symposium* (2015), Springer, pp. 162–180.

[52] Li, Y., Tan, T., and Xue, J. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM) 28*, 2 (2019), 1–50.

[53] Lindorfer, M., Neugschw, M., Weichselbaum, L., Fratantonio, Y., Veen, V. V. D., and Platzer, C. Andrubis- 1,000,000 apps later: A view on current android malware behaviors.

[54] Liu, T., Wang, H., Li, L., Luo, X., Dong, F., Guo, Y., Wang, L., Bissyandé, T. F., and Klein, J. Maddroid: Characterising and detecting devious ad content for android apps. In *The Web Conference 2020 (WWW 2020)* (2020).

[55] Livshits, B., Whaley, J., and Lam, M. S. Reflection analysis for java. In *Programming Languages and Systems*. Springer, 2005, pp. 139–160.

[56] Octeau, D., Jha, S., Dering, M., Mcdaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. 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).

[57] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)* (2015).

[58] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. 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).

[59] Rasthofer, S., Arzt, S., Miltenberger, M., and Bodden, E. Harvesting runtime values in android applications that feature anti-analysis techniques.

[60] Rasthofer, S., Arzt, S., Triller, S., and Pradel, M. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 300–311.

[61] Rastogi, V., Chen, Y., and Jiang, X. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 329–334.

[62] Ravitch, T., Creswick, E. R., Tomb, A., Foltzer, A., Elliott, T., and Casburn, L. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop* (2014), ACM, p. 4.

[63] Schutte, J., Titze, D., and De Fuentes, J. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *TrustCom* (2014), IEEE, pp. 370–379.

[64] Smaragdakis, Y., Balatsouras, G., Kastrinis, G., and Bravenboer, M. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems* (2015), Springer, pp. 485–503.

[65] Statista. Number of new mobile malware variants observed on android platform by 360 security software monthly in china in 1st quarter 2019. Tech. rep., June 2019.

[66] Yang, S., Yan, D., Wu, H., Wang, Y., and Rountev, A. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)* (2015).

[67] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. AppContext: Differentiating Malicious and Benign Mobile App Behavior Under Contexts. In *International Conference on Software Engineering (ICSE)* (2015).

[68] Yang, Y., Luo, W., Pei, Y., Pan, M., and Zhang, T. Execution enhanced static detection of android privacy leakage hidden by dynamic class loading. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)* (2019), vol. 1, IEEE, pp. 149–158.

[69] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1043–1054.

[70] Zhang, M., and Yin, H. 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).

[71] Zhang, M., and Yin, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *AsiaCCS* (2014).

[72] Zhang, Y., Li, Y., Tan, T., and Xue, J. Ripple: Reflection analysis for android apps in incomplete information environments. *Software: Practice and Experience 48*, 8 (2018), 1419–1437.

[73] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., and Massacci, F. 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* (2015), ACM, pp. 37–48.