Reflection-Aware Static Analysis of Android Apps

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

ABSTRACT

We demonstrate the benefits of DroidRA, a tool for taming reflection in Android apps. DroidRA first statically extracts reflection-related object values from a given Android app. Then, it leverages the extracted values to boost the app in a way that reflective calls are no longer a challenge for existing static analyzers. This is achieved through a bytecode instrumentation approach, where reflective calls are supplemented with explicit traditional Java method calls which can be followed by state-of-the-art analyzers which do not handle reflection. Instrumented apps can thus be completely analyzed by existing static analyzers, which are no longer required to be modified to support reflection-aware analysis. The video demo of DroidRA can be found at https://youtu.be/-HW0V68aAWc

CCS Concepts

•Software and its engineering → Software notations and tools;

Keywords

Android; Static Analysis; Reflection; DroidRA

1. INTRODUCTION

Many static programming languages support the use of reflection to allow a program to introspect its own behaviors at runtime. As an example, reflection is frequently leveraged by Java testers to access private fields and methods in order to ensure a high coverage rate. Actually, reflection has been taken as an advanced feature, because it enables a program to perform sophisticated operations, which would otherwise be impossible. Android apps, which are essentially written through Java, have also extended the ability of leveraging reflection mechanism and many of them indeed leverage this feature to achieve advanced functionality, e.g., to access inaccessible APIs (we will show more legitimate usages in the next Section).

ASE'16, September 3–7, 2016, Singapore, Singapore
ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970277

Unfortunately, reflection has also been leveraged by "bad" guys to perform malicious behaviors, in an attempt to bypass the detection of static analyzers. Indeed, many static analyzers of Android apps have already been proposed in the community such as call graph building [1, 2, 3] and program slicing [4], which are not yet reflection-aware and thus will inherently yield false negatives. Furthermore, existing reflection-aware Java-based static analyzers cannot be simply adapted to tackle Android apps. As an example, TamiFlex [5], a well-known reflection taming tool that is designed and implemented for Java apps, cannot be applied to Android apps.

To the best of our knowledge, except our previous work [6, 7], there is only one work presented by Barros et al. [8], who proposes a solution for reflection analysis of Android apps. However, that approach analyzes the source code of Android apps only which is difficult to obtain for most market apps, and requires additional developer efforts to annotate certain code properties. Last but not the least, it does not provide a way to directly benefit existing static analyzers regarding its extracted reflection results.

Our objective in this work is thus to bridge the gap between exploiting the capabilities of state-of-the-art analysis tools and allowing them to perform whole-program analysis on apps with reflective calls (in a non-invasive means). This is done by taming reflection in Android apps in which we present a tool called DroidRA to 1) statically retrieve all reflection-relevant values to resolve reflective calls and 2) add instrumentation code with corresponding Java standard method calls.

DroidRA is dedicated to support researchers and analysts to perform reflection-aware static analysis of Android apps. It is fully automated and is written in Java, which takes as input an Android app and outputs two artifacts: 1) A summary of reflection usages (json format) and 2) A new version of Android app where its reflective calls have been tamed for static analysis.

As indicated in our research track paper [7], we have made available online our full implementation as an open source project, along with the benchmarks and scripts we leveraged at:

https://github.com/serval-snt-uni-lu/DroidRA

The rest of this paper is organized as follows: Section 2 introduces some background information on reflection usages in Android apps. Then in Section 3 we detail the design and implementation of our tool DroidRA, along with a summary of its evaluations. After that, we show some related works in Section 4 and then conclude this paper in Section 5.

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

Listing 1: Reflection usage in real Android apps.

2. REFLECTION IN ANDROID APPS

Reflection usages. Essentially, Android apps are written in Java, for which the reflection mechanism introduced by Java language is also extended. Our previous work [7] has shown that reflection is frequently used by Android developers to 1) provide genericity, e.g., using reflection to implement generic functions. Example (1) in Listing 1 illustrates a real example on how reflection is leveraged to achieve this kind of functionality, where a Collection object is initialized reflectively; 2) maintain backward compatibility, e.g., ensuring apps that are developed with latest features to be still executable on old devices, where the latest features are not yet available. Example (2) in Listing 1 illustrates a real example, which depends on reflection to check whether the running Android device supports text-to-speech feature or not; 3) reinforce app security, e.g., separating the core functionality of a given app to an independent library, which can then be loaded dynamically (through reflection) when a specific rule is reached; and 4) access inaccessible APIs. In the Android framework base, there are two types of APIs that are inaccessible by third-party apps, including internal and hidden APIs. Those inaccessible APIs will not be released to the Android SDK, namely *android.jar* that will be used by developers to implement Android apps. However, those APIs will be shipped into running Android devices, through *framework.jar* that will be used to replace *android.jar* at runtime to support the execution of Android apps. Within this release model, inaccessible APIs can be accessed by third-party apps through reflection, although they are not available in the developing environment, the final app will work fine in Android devices. Example (3) in Listing 1 illustrates a real example showing how inaccessible APIs are leveraged.

Patterns of reflective calls. As a means to build a clear picture on how reflection is used by developers, so as to tame reflection for static analysis approaches, we have investigated the sequences of reflective method calls on randomly selected 500 apps and summarized their common usage patterns. Figure 1 illustrates the final patterns on the usage of reflective method calls we build, which is capable to model the most typical usages of reflective calls. This pattern shows how the reflection mechanism allows to access methods/fields dynamically. Reflective methods or fields can be leveraged directly if they are statically declared (solid arrows). Otherwise, they have to used after an initialization of their classes (dotted arrows).

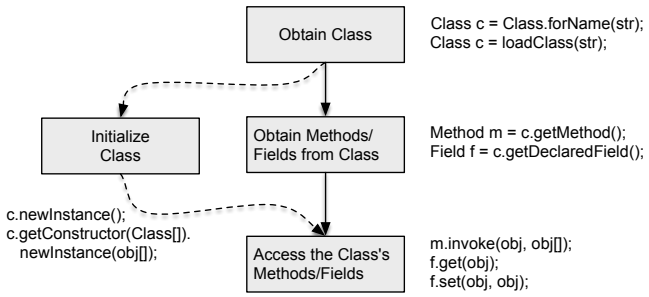

Figure 1: Abstract pattern of reflection usage and some possible examples.

Reflection in malicious vs. benign apps. Given the potential of using reflection to hide malicious operations, which could not be statically detected, we investigate whether reflection usage can constitute in discriminating between malicious and benign apps. We build a basic machine learning classification scheme with reflective calls as the unique feature set (e.g., *invoke()* is considered as a feature). For the experiments, we use RandomForest ensemble learning algorithm and build our ground truth by leveraging VirusTotal scanning results[1] on the randomly selected 500 apps. 10-Fold cross validation experiments[2] yield a performance of 97.5% for F-Measure, the harmonic mean of precision and recall. Although such a high performance can be favored by our experimental settings, it nonetheless suggests that reflective calls may constitute a discriminate feature between malware and goodware.

3. DROIDRA

In this section, we detail the design and implementation of our tool DroidRA. Figure 2 overviews the working process of DroidRA, which contains three phases: 1) Jimple Preprocessing Phase; 2) Reflection Analysis Phase; and 3) Booster Phase. In the following of this section, we detail the implementation of these three phases respectively.

3.1 Phase 1

As the first phase, DroidRA transforms a given Android app to appropriate Jimple code, in order to support further reflection analysis. Jimple is selected, because it is presented as a 3-address representation that has been designed to simplify analysis and transformation of Java bytecode [10].

At beginning of this phase, DroidRA extracts DEX files (i.e., the actual app code) from Android apps (cf. step (1.1)).

[1]https://www.virustotal.com: we consider an app a to be malicious if any of the anti-virus products from VirusTotal has flagged it as malicious. Otherwise a is a benign app.

[2]To address the imbalance dataset problem (only 20 out of the 500 apps are malicious), we perform oversampling using SMOTE [9]

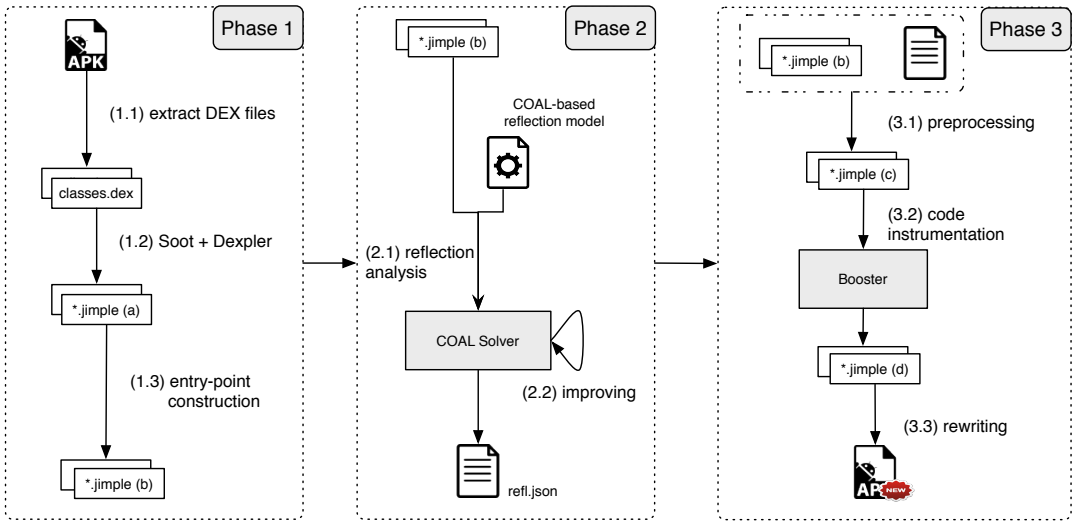

Figure 2: Overview of DroidRA.

```
MD5: b0eea79daa92d90d8d3dd1a644b7edaa
com.miralcestudios.cjsiebl-26.apk/
----AndroidManifest.xml
----classes.dex     // the default DEX file
----META-INF/
----res/
----assets/
    ----obmhwzogztvvvhcbd.dat    //zip archive
        ----META-INF/
        ----classes.dex     //to be dynamically loaded DEX file
    ----dousn    //zip archive
        ----META-INF/
        ----classes.dex     //to be dynamically loaded DEX file
```

Figure 3: A real-world example showing the distribution of DEX files in an Android app.

There are two types of DEX files existing in Android apps. Figure 3 illustrates an example containing both of these two types. The first type is the default DEX file, namely *classes.dex*, which lives in the top-level of a given Android app[3] and applies to every Android app.

The first type is the to-be dynamically loaded DEX file (also named as *classes.dex*), as shown in Figure 3, which is usually hidden in zip files. The name of those zip files will likely be further obfuscated in order to bypass the detection of potential analyzers. As an example, there is a file named *obmhwzogztvvvhcbd.dat* under the *assets* directory of app *com.miralcestudios.cjsiebl*, which is actually a carrier of a *classes.dex* file.

The second type of DEX files could be eventually accessed through the well-known dynamic code loading (DCL) mechanism. Thus, it is essential to take them into consideration in order to support whole-program analysis. However, DCL is yet another challenge for static analysis [11], as some would-be loaded classes may not exist statically at all, e.g., they can be downloaded from a server at runtime. DroidRA focuses on dynamically loaded code that is included in the apk file and that can be accessed statically.

[3]The format of an Android app is actually a compressed Zip archive.

After extracting all the DEX files, in step (1.2), DroidRA transforms all of them into Jimple code through a tool we released previously called Dexpler [12], which now has been integrated into Soot [13]. Soot is a framework for analyzing and transforming Java/Android apps, which has already been widely used to analyze, instrument, optimize and visualize Java/Android apps.

Finally, similar to other static approaches, DroidRA also needs to start its analysis from a single *entry-point* method. Unfortunately, Android apps do not have a well-defined entry-point method like *main()* in traditional Java apps. But instead, they have multiple entry points, where each component (as an essential unit that Android apps are made up of) could be a potential entry point. To mitigate this, DroidRA constructs a single *entry-point* method based on all the aforementioned Jimple code, following the successful experiences that we have learned from our previous work [1, 14, 15, 7]. Given an Android app, DroidRA artificially assembles a *dummy main* method, taking into account all components and their lifecycle methods (e.g., *onStart()* and callback methods (e.g., onClick()). This dummy main method will eventually support static analysis approaches to build an inter-component control-flow graph and consequently to support whole-program analysis of Android apps.

3.2 Phase 2

We now detail the second phase of DroidRA. Our reflection analysis is implemented on top of COAL and COAL solver [16]. COAL is a composite COnstant propAgation Language that is dedicated to specify constant propagation problems. COAL solver is an inter-procedural, context-sensitive, flow-sensitive analyzing engine that performs composite constant propagation to solve COAL-modeled constant propagation problems. As an example, ParamHarver [15] leverages COAL and COAL solver to harvest parameter values (propagated in an inter-procedural manner) of a given set of Android APIs.

In Phase 2, we first build a COAL-based reflection model, and then send it to an improved COAL solver along with the analyzed Jimple code. As the output, the reflection analysis result will be presented in a json file named *refl.json*, which

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

then can be leveraged by other static analysis approaches to support reflection-aware analysis.

COAL-based reflection model. Listing 2 illustrates a part of our COAL-based reflection analysis model, where class *Method* is modeled. For class *Method*, we define two string fields for it. Field *declaringClass_method* is used to represent the class type that a given reflective call belongs to. Filed *name_method* is used to represent the name of the reflective call. Based on the method name and its declaring class name, we are able to uniquely identify the reflectively called method.

Improvements to the COAL solver. Although COAL is design to be as generic as possible, it cannot be directly applied to model and tackle the reflection problem that DroidRA intends to solve. We thus contribute several improvements to the COAL solver to enable the analysis of reflective calls. Our first improvement is on the COAL language itself, we slightly extend it to be able to represent the query of values of objects on which instance method calls are made. The second improvement is on the propagating of values of array objects. Given an array object, we mark it as containing all element values. With this limited improvement, our result may not be precise in theory. However, for our targeted method *getConstructors()*, that we consider typically only have a few elements, our improvement is able to ensure that the propagation of constructors is precise enough in practice. Finally, we have contributed various optimizations to improve the overall performance of the COAL solver.

3.3 Phase 3

Now, in the last phase, we detail how DroidRA instruments Jimple code towards supporting reflection-aware static analysis, where reflective calls have been conservatively[4] augmented with appropriate standard Java calls.

Let us take Listing 3 as an example to better explain the idea of Phase 3, where the code snippet is extracted from an app of DroidBench named *Reflection3.apk* [1]. DroidBench is a set of hand crafted Android apps for which all leaks are known in advance. These leaks are used as ground truth to evaluate how well static and dynamic security tools such as IccTA [14] and CopperDroid [17] detect data leaks. The Reflection3 benchmark app has used reflective calls, which is known to be problematic for many tools, including FlowDroid [1] and IccTA [14]. In this example, class `ReflectiveClass` is first retrieved (line 3) and initialized (line 4). Then, method *setImei()* is reflectively retrieved and invoked

[4]We keep the reflective calls to conserve their initial runtime behaviors, while standard calls are injected in the call graph to allow static exploration.

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

Listing 5: The boosting results of our motivating example. '+' indicates the line it appears is newly injected.

to store the device ID into field *imei* (lines 5-6). After that, another reflective method called *getImei()* is retrieved and invoked to obtain back the device ID into the current context (lines 7-8) so that it can be eventually sent outside the device via SMS to a hard-coded phone number (line 10).

Phase 3 takes as input the output of Phase 2, Listing 4 shows the results reported by DroidRA in Phase 2, which contain reflection-related target values for two reflective methods (i.e., *invoke()*) that we are interested in and that we have explicitly configured in the COAL model (cf. the *query* keyword at line 13 in Listing 2).

Listing 5 shows the instrumentation results of our example snippet. Given a reflectively accessed class, DroidRA explicitly initializes it by calling its construct method (lines 3-4), which will also be retrieved by our reflection analysis. For reflective method calls, DroidRA will inject code to simulate them in a traditional manner. As an example, the statement "m.invoke(o, imei)" (line 5) has been augmented with "o.setImei(imei)", where *o* is an instance of *ReflectiveClass* that DroidRA has initialized previously. We note the readers that we have introduced a method called *check()* whose implementation will be excluded for static analysis. However, for runtime execution, *check()* will always return false, keeping the newly injected code from being executed, while for sound static analysis, it allows the path to be explored.

3.4 Summary of Evaluation

We have evaluated DroidRA in four aspects. At first, we compute the coverage of reflection calls that DroidRA identifies and inspects. Experiments on 500 randomly selected apps show that DroidRA is able to correctly resolve 81.2%[5] of the targets. Second, we compare DroidRA with a state-of-the-art approach proposed by Barros et al. [8] (hereinafter we refer to it as the checker framework) on 10 open-sourced apps, which are actually used by Barros previously in their experiments. The results show that DroidRA is able to resolve more reflective methods/constructors (e.g., 10 more constructors on 10 F-Droid apps that are previously used by Barros), though it does not need the source code of analyzed apps and additional developers efforts as what checker framework needs. Third, we check whether DroidRA is capable to support existing static analyzers to build sounder call graphs of Android apps. To this end, we select Soot static analyzer to perform our experiments. In the same 500 apps that we have leveraged in our first evaluation aspect, for each app,

[5]The unresolved targets are mainly due to limitations of static analysis where runtime values cannot be resolved and limitations of the COAL solver where arrays of objects cannot be fully propagate at the moment. Other approaches like primo [18] can be leveraged to facilitate the resolution of those unresolved targets.

```
1  class java.lang.reflect.Method {
2    Class declaringClass_method;
3    String name_method;
4
5    mod gen <java.lang.Class: java.lang.reflect.Method getDeclaredMethod(java.lang.String,java.lang.Class[])> {
6      -1: replace declaringClass_method;
7      0: replace name_method;
8    }
9    mod gen <java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.Class[])> {
10     -1: replace declaringClass_method;
11     0: replace name_method;
12   }
13   query <java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[])> {
14     -1: type java.lang.reflect.Method;
15   }
16 }
```

Listing 2: The COAL-based reflection analysis model for class *java.lang.reflect.Method*. Similar specifications apply for all other reflection classes. *mod* keyword describes how the method modifies the state of the program. *gen* keyword depicts that the method will generate a new object of type Method. *query* keyword specifies the point that we would like to obtain the values. Indexes such as *-1* and *0* indicate the reference to the instance on which the method call is made. As an example, statement `-1: replace declaringClass_method` at line 6 indicates that the name of the Class object on which the method is called is used as the field `declaringClass_method` of the generated object.

```
1  de.ecspride.MainActivity/void onCreate(android.os.Bundle) : $r8 = virtualinvoke
       $r6.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[])>(r19, $r7)
2      -1 : Value: 1 path values
3    declaringClass_method=de.ecspride.ReflectiveClass, name_method=getImei,
4
5  de.ecspride.MainActivity/void onCreate(android.os.Bundle) : virtualinvoke r21.<java.lang.reflect.Method:
       java.lang.Object invoke(java.lang.Object,java.lang.Object[])>(r19, r1)
6      -1 : Value: 1 path values
7    declaringClass_method=de.ecspride.ReflectiveClass, name_method=setImei
```

Listing 4: The reflection analysis results for the example code shown in Listing 3.

DroidRA improves by 3.8% and 0.6% the number of edges in the constructed call graph with Spark [19] and CHA [13] respectively. In the last evaluation aspect, we evaluate whether existing static analyzers can yield reflection-aware results or not, based on the instrumented app generated by DroidRA. To this end, we select IccTA [14] static analyzer to conduct our experiments. We launch IccTA on both benchmark apps where the reflective calls are known in advance and on real-world apps. Our experimental results show that in both cases IccTA is able to report reflection-aware results, thanks to DroidRA which would otherwise be impossible.

4. RELATED WORK

To the best of our knowledge, we are the first one that automatically solve the reflection analysis of Android apps for boosting static analysis approaches. Research on static analysis of Android apps have shown strong limitations on taming reflections [20, 21, 22, 2], many of them have even explicitly acknowledge such limitations [14, 23, 16]

The closest work to ours was concurrently proposed by Barros et al. [8] within their Checker framework [24]. However, their work differs from ours in several ways: 1) their approach focuses on checking information flows from developer's perspective, who has to add annotations in the analyzed source code. 2) their approach performs intra-procedural type inferences while ours is inter-procedural and context-sensitive. 3) While both of our approaches perform reflection analysis, our approach goes one step further to also boost Android apps, which can directly benefit other static analysis approaches in an non-invasive manner.

Reflection, by itself, has been investigated by several works for Java applications. For example, Livshits et al. [25] leverages points-to analysis to estimate the target value of reflective calls. Bodden [5] et al. present a dynamic approach, namely TamiFlex, to support static analysis in the presence of reflections in Java programs. Although Android apps are written in Java, TamiFlex cannot be simply adapted to analyze Android apps [26].

Using Instrumentation to strengthen the static analysis of Java/Android apps is not new [27, 6]. As examples, Stoller et al. [28] and Artho et al. [29] merge multi-process apps into a single one through instrumentation for easing other approaches like software model checking. IccTA [14] instruments Android apps to bridge the ICC gaps and thus to enable inter-component static analysis. AppSealer [30] instruments Android apps to detect and inject vulnerability-specific patches, so as to prevent at runtime the component hijacking attacks.

5. CONCLUSION

To handle reflection for static analysis is an essential step towards whole-program analysis. Unfortunately, reflection in Android apps has not been fully investigated by the community. To bridge this gap, we thus designed and implemented a tool called DroidRA, which statically extracts the target object values of reflective methods (e.g., the name of a reflective call or the declaring class name of a reflectively accessed field), in an attempt to have a better understanding on how reflection is leveraged in a given app. We then present a booster module, which leverages the previously extracted values to instrument a given app by representing reflective calls with traditional Java calls. The instrumented app results

in a non-invasive approach, where existing static analysis approaches, without any modification, can now be directly benefited to yield reflection-aware results.

6. ACKNOWLEDGMENTS

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

7. REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[2] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *TrustCom*, 2014.

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

[4] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM, 2013.

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

[6] Li Li. Boosting static analysis of android apps through code instrumentation. In *The Doctoral Symposium of 38th International Conference on Software Engineering (ICSE-DS 2016)*, 2016.

[7] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

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

[9] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.

[10] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[11] Li Li, Tegawendé François D Assise Bissyande, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. Technical report, SnT, 2016.

[12] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*, 2012.

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

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

[15] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *SANER*, 2016.

[16] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *ICSE*, 2015.

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

[18] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *POPL*, 2016.

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

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

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

[22] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, pages 1043–1054. ACM, 2013.

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

[24] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1092–1104, Scottsdale, AZ, USA, November 4–6, 2014.

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

[26] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *CODASPY*. ACM, 2015.

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

[28] Scott D Stoller and Yanhong A Liu. Transformations for model checking distributed java programs. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 192–199, 2001.

[29] Cyrille Artho and Pierre-Loic Garoche. Accurate centralization for applying model checking on networked applications. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 177–188. IEEE, 2006.

[30] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.