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 byte-code 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).

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.
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).

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 Random Forest 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].
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 app⁵ 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 `obmhwzogztvvhcbd.dat` under the `assets` directory of app `com.miralcestudios.csiebl`, 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.

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 `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 COntant 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...
Then, method `setImei()` which is known to be problematic for many tools, including Reflection3 apk from DroidBench’s exploration.

Given an array object, we mark it as `ReflectiveClass` containing all element values. With this limited improvement, we have explicitly configured in the COAL model (cf. the query keyword at line 13 in Listing 2).

### Improvements to the COAL solver

Although COAL is designed 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 propagation of constructors 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.

### 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 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 detect data leaks. 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 to store the device ID into field `imei` (lines 5-6).

```
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 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.

### 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% 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, 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.

---

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.

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.
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. To this end, we select IccTA [14] static analyzer to conduct our experiments. We launch IccTA on both benchmark apps respectively. In the last evaluation aspect, we evaluate whether 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.

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.

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 a 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 et al. [5] 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


