Boosting Static Analysis of Android Apps through Code Instrumentation

Li Li*
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
li.li@uni.lu
http://lilicoding.github.io

ABSTRACT

Static analysis has been applied to dissect Android apps for many years. The main advantage of using static analysis is its efficiency and entire code coverage characteristics. However, the community has not yet produced complete tools to perform in-depth static analysis, putting users at risk to malicious apps. Because of the diverse challenges caused by Android apps, it is hard for a single tool to efficiently address all of them. Thus, in this work, we propose to boost static analysis of Android apps through code instrumentation, in which the knotty code can be reduced or simplified into an equivalent but analyzable code. Consequently, existing static analyzers, without any modification, can be leveraged to perform extensive analysis, although originally they cannot.

Previously, we have successfully applied instrumentation for two challenges of static analysis of Android apps: Inter-Component Communication (ICC) and Reflection. However, these two case studies are implemented separately and the implementation is not reusable, letting some functionality, that could be reused from one to another, be reinvented and thus lots of resources are wasted. To this end, in this work, we aim at providing a generic and non-invasive approach for existing static analyzers, enabling them to perform more broad analysis.

1. INTRODUCTION

Android devices have become pervasive in our daily life with over 1.5 million devices activated everyday. As of July 2015, there are 1.6 million apps available for users to choose. Those apps have infiltrated into nearly every activity of users, including social networking, schedule planning, financial resource managing, etc. Undoubtedly, users' quality of life has been improved because of those apps. However, these phenomena, on the other hand, also threaten users' daily life, as the quality of Android apps are uneven, e.g., some of them may be malware [11, 13].

Static analysis has been taken as a common means to dissect Android apps for years. Popular implementation techniques behind static analysis include *symbolic execution*, which generates possible inputs in order to reach a program point at runtime, and *taint analysis*, which propagates sensitive data from a program point (known as *source*) to another program point (known as *sink*) in order to guard the data flow.

Although static analysis techniques have been used for decades, it still remains some challenges for specific analyzers to tackle. For example, a well-known problem for statically analyzing Android apps is to take into account their ICC mechanism[1]. Advanced static analysis approach like FlowDroid [3], at the moment, is unable to perform ICC analysis and consequently limits itself to only perform intra-component analysis, missing the chance to go beyond the border of components.

To fill this gap, recent approaches [8, 12, 19] have solved the ICC problem for static analysis. However, none of them are generic enough so that their solutions are difficult to be applied to solve other challenges (e.g., reflective method calls). Besides, existing static analyzers cannot directly benefit from these solutions, which are actually became invasive.

Our goal in this work is to implement a **generic** and **non-invasive** approach for boosting existing static analyzers. In this work, "generic" means our approach can be easily leveraged to address different challenges while "non-invasive" means that we assist existing static analyzers without modifying them. To that end, we propose an instrumentation-based approach, which manipulates app code through injecting/deleting/updating statements, leading to a simplified app version for analysis. More specifically, a static analysis problem (e.g., for ICC mechanism) would be modeled, through a Soot-based Instrumentation Language (SIL), as an instrumentation problem, which further can be resolved by a SIL solver.

2. BACKGROUND AND MOTIVATION

To ease the understanding of the challenges of static analysis of Android apps, we give more details of the concept of Android apps in section 2.1. Then, we motivate our work through an illustrative example in section 2.2.

*Li is supervised by Prof. Yves Le Traon and advised by Dr. Jacques Klein and Dr. Tegawendé F. Bissyandé. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4205-6/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889258

[1]ICC stands for Inter-Component Communication, more details will be given in Section 2.1.

2.1 Android

Android apps are made up of components (the basic units). There are four types of components, including *Activity*, which represents the visible parts of an app (e.g., UI interface); *Service*, which executes some (computation-intensive) tasks in the background; *Broadcast Receiver*, which listens to incoming events and *Content Provider*, which plays as a standard means for structural data access.

These components can communicate with each other, where the communication mechanism is known as ICC. In the previous section, we have illustrated that the ICC mechanism makes troubles for static analysis approaches. The reason is that some specific methods (known as ICC methods) can perform the cross-component communication at the system level. In other words, there is no direct code connection in the app level, which would make static analyzers unaware of the ICC mechanism.

2.2 Motivating Example

We now motivate our work through a concrete example (Listing 1), to highlight the imperativeness of addressing challenges of static analysis of Android apps.

The snippets are made of two Android components: `MainActivity` and `InFlowActivity`. In `MainActivity`, the device id is first obtained (line 3) and then stored into an explicit Intent (line 5) and finally sent to `InFlowActivity` through ICC method *startActivity()*. In `InFlowActivity`, in line 10, the device id is extracted from the incoming Intent. Then, it is used as a parameter of method *setImei()* reflectively (line 13). Later, the device id is called back through reflective method *getImei()* (line 16). Finally, it is sent outside of the app through SMS (line 18).

In this example, there are two obstacles making static analysis inactive. The first one is that, due to the Android ICC, as shown in Listing 1, the two components are not actually connected by code, meaning that basic static analysis approaches (e.g., Soot) cannot directly build an inter-component control-flow graph (CFG) and consequently become ICC unaware. The second challenge is that, due to the reflection mechanism extended from Java, like Android ICC, reflective calls break the CFG as well, as there are no directly code connected between the "normal" code and reflectively called code.

3. RESEARCH PROPOSAL

```
 1 // class MainActivity
 2 TelephonyManager telephonyManager = default;
 3 String imei = telephonyManager.getDeviceId();
 4 Intent i = new Intent(this, InFlowActivity.class);
 5 i.putExtra("DroidBench", imei);
 6 this.startActivity(i);
 7
 8 //class InFlowActivity
 9 Intent i = this.getIntent();
10 String imei = i.getStringExtra("DroidBench");
11 Class c =
        Class.forName("de.ecspride.ReflectiveClass");
12 Object o = c.newInstance();
13 Method m = c.getMethod("setIme"+"i",String.class);
14 m.invoke(o, imei);
15 Method m2 = c.getMethod("getImei");
16 String s = (String) m2.invoke(o);
17 SmsManager sms = SmsManager.getDefault();
18 sms.sendTextMessage("+352 001",null,s,null,null);
```

Listing 1: Snippets extracted from a running app, showing an example of sensitive data leak with using ICC and Reflection characteristics.

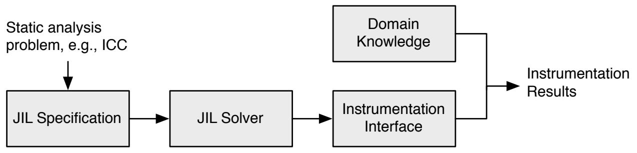

Figure 1: Research Proposal Overview.

It has been demonstrated by us [12] and others [6] through experiments that instrumentation is a good means to solve the aforementioned challenges of static analysis. Thus, our goal in this work is to provide a generic and non-invasive approach to instrument Android apps and therefore to boost existing static analyzers to perform more sound analysis.

Figure 1 illustrates an overview of our research proposal. It works as follows:

- Static analysis problems, such as reflective call resolution, are specified using a declarative language called JIL (Jimple-based Instrumentation Language). Jimple is a 3-address intermediate representation of Soot, which is designed to simplify Java (or Java-based apps such as Android) bytecode for analysis and transformation [9].

- JIL specifications are then forwarded to a JIL solver, which parses the specification and prepares an instrumentation skeleton along with some instrumentation interfaces. These interfaces will be used later for complementing the instrumentation process. The reason why we provide such interfaces, is related to the diversity of static analysis problems and the domain knowledge which is unknown beforehand. Ideally, we would like to offer a generic approach for instrumentation. Nevertheless, because of the diversified requirements, we have to compromise a trade-off, which automates the instrumentation as much as possible and also provide a way for developers to work on the remaining parts.

- In the last step, instrumentation interfaces are implemented with some domain knowledge. As an example, suppose we want to instrument an ICC method *startActivity(intent)*[2] of an app. It is hard for us to know beforehand which component *intent* points to, stopping us from automating the instrumentation process fully. In this example, the target component of *intent* is the so-called domain knowledge. To automatically infer such domain knowledge is out of scope of this work. Existing approaches (e.g., Epicc [18] and IC3 [17] for ICC values) can be leveraged to complement our approach.

4. RESEARCH PROGRESS

Before making our approach generic, we have to investigate whether instrumentation is a reliable solution, i.e., whether instrumentation is able to support existing static analyzers to perform extensive analysis? To this end, we have introduced instrumentation to solve two challenges of

[2]This method switches the current UI interface to a new one.

```
19 // class MainActivity
20   this.startActivity(i);
21 + if (Opaque.false) {
22 +   InFlowActivity a = new InFlowActivity(i);
23 +   a.dummyMain();
24 + }
25
26 //class InFlowActivity
27 + Intent i = null;
28 + public InFlowActivity() {
29 +   this.i = i;
30 + }
31 + public Intent getIntent() {
32 +   return this.i;
33 + }
34 + public void dummyMain() {
35 +   //lifecycle/callback methods
36 + }
```

Listing 2: Instrumentation results for Android ICC.

static analysis of Android apps: `ICC` and `Reflection`, for which we are going to detail in section 4.1. Later, we summarize the remaining work that we still need to do in section 4.2.

4.1 Finished Work

Now, we show two successfully applied case studies (`ICC` in Section 4.1.1 and `Reflection` in Section 4.1.2) on instrumenting Android apps. In this work, "successfully" means that the instrumentation we do for Android apps is indeed able to support existing static analyzers to perform extensive analysis.

4.1.1 IccTA: Android ICC

In the first case study, we intend to solve ICC challenge for static analysis. To this end, we have implemented a tool called IccTA (Icc Taint Analysis) to support the instrumentation. The working process of IccTA is as follows: 1) IccTA leverages Dexpler [5] to transform Android Dalvik bytecode into *Jimple*, a Soot's intermediate representation [9]. 2) IccTA extracts ICC links from analyzed Android apps, this step is performed on top of Epicc [18], which is designed to statically infer ICC values. 3) Based on the ICC links we extracted before, IccTA modifies the Jimple representation to directly connect components, enabling static analyzers to build an inter-component control-flow graph (ICCFG) and thereby to perform data-flow analysis across components.

Listing 2 illustrates the instrumentation results of IccTA, on the example code shown in Listing 1. The original ICC method call (*startActivity()*) is covered by a block of code (line 21-24), in which the ICC method is imitated by directly initializing the target component (*InFlowActivity*) and calling its entry methods (which has been modeled inside a dummy main method). The parameter (intent i) is also imitated through a new generated construct method (line 28-30). In order to appropriately deliver intent i, method *getIntent()* is overwritten to return the Intent received through the constructing method.

```
37 //class InFlowActivity
38 Object o = c.newInstance();
39 + if (Opaque.false)
40 +   o = new ReflectiveClass();
41 m.invoke(o, imei);
42 + if (Opaque.false)
43 +   o.setImei(imei);
44 String s = (String) m2.invoke(o);
45 + if (Opaque.false)
46 +   s = o.getImei();
```

Listing 3: Instrumentation results for Reflection.

Thanks to the instrumentation, Listing 2 no longer contains ICC challenge. Consequently, existing static analyzers (e.g., FlowDroid), even without being ICC-aware, are now able to perform ICC-aware analysis. Indeed, we have evaluate this work on 15,000 Google Play apps and 1023 MalGenome apps, from which we detect 534 ICC leaks in 108 apps from MalGenome and 2,395 ICC leaks in 337 apps from the Google Play apps.

4.1.2 DroidRA: Reflection

In the second case study, we solve Android reflection challenge for static analyzers. Particularly, we have proposed a tool called DroidRA [14] (anDroid Reflection Analysis), which aims at 1) resolving reflective call targets for the purpose of exposing all program behaviors; and 2) unbreaking app control-flow in the presence of reflective calls for the purpose of allowing static analyzers to produce extensive results.

The second aim of DroidRA is actually instrumenting Android apps to achieve its functionality. Let us take Listing 1 again as an example to show how DroidRA's instrumentation works for solving reflective calls. Listing 3 illustrates the instrumentation results of DroidRA for reflective calls. For *newInstance()* reflective call, DroidRA explicitly *new*s the corresponding class (line 40). While for *invoke()* reflective call, DroidRA explicitly calls the real method (e.g., *setImei()* in line 43 for *invoke()* at line 41). At the end, all the reflective calls have been represented by their real method calls, which is no longer a problem of static analysis.

We have evaluated DroidRA on 500 real-world apps, for which DroidRA improves by 3.8% and 0.6% the number of edges in the call-graph constructed with Spark [10] and CHA algorithm respectively. We have also sent our instrumented version of apps on both benchmark apps and in-the-wild apps to IccTA. IccTA successfully reports all the leaks on the four DroidBench apps, while originally IccTA cannot report any of such. IccTA is also able to impact 16% of in-the-wild apps by yielding more results.

Thanks to DroidRA, IccTA actually is also able to report a privacy leak for Listing 1. The instrumentation has enabled IccTA to become reflection-aware while enabled FlowDroid to become ICC-aware. Consequently, Instrumentation enables FlowDroid, which is ICC-unaware and reflection-unaware, to be able to yield ICC-aware and reflection-aware results. This example indeed shows that instrumentation is reliable to be leveraged to solve tough static analysis challenges, and consequently support existing static analyzers without any modification to perform extensive analysis.

4.2 Remaining Work

So far, we have finished the aforementioned two case studies, which have shown that instrumentation is a good approach for providing non-invasive supporting for existing static analyzers. Furthermore, we have implemented a generic framework called Apkpler [15], which integrates a plugin system to reduce the analysis complexity of Android apps. With the help of Apkpler, FlowDroid can actually benefit from IccTA's output to detect ICC-aware sensitive data leaks.

However, the instrumentation part of IccTA (for Android ICC) and DroidRA (for reflection) are not generic, which are designed and implemented separately. We have summed up some common parts that are used in both approaches. For example, both approaches need to define a set of instrumentation points beforehand and then localize them in the code before instrumentation. Besides, both approaches have to resolve/create local variables so as to satisfy the instrumentation. We believe that the common parts of instrumentation such as the aforementioned processes can be presented and solved in a generic way. Thus, in future work, we will thoroughly summarize the common parts between our two previous case studies and then implement the JIL solver to ease the reuse of other instrumentation tasks.

5. RELATED WORK

Instrumenting Android apps can be used not only for security purposes, but also for code analysis and optimization in general, such as *time-bomb elimination*. In addition to manual instrumentation, the most well-known tool that supports automatic instrumentation is Soot. For example, AppSealer [20] instruments vulnerability-specific patches for keeping Android components from hijacking attacks. Other tools like *abc* (the AspectBench Compiler [2]), WALA [1] can also be used to instrument Android apps.

Our previous work called IccTA, as introduced before, leverages FlowDroid and Epicc to perform inter-component static taint analysis. Other tools like DidFail [8] and AmanDroid [19] are also proposed to support ICC-aware analysis. Neither DidFail nor AmanDroid uses instrumentation to solve the ICC problem, leading to non-reusable approaches. Interestingly, DidFail also leverages instrumentation technique in its implementation, which inserts a unique ID into Intents, so as to appropriately match Intents with Intent Filters afterwords.

Reflection, by itself, has been studied by several works for Java apps. For example, Livshits et al. [16] leverage points-to analysis to approximate the targets of reflection calls. Braux et al. [7] optimize reflective calls to increase time performance. Most notably, Bodden et al. [6] have presented TamiFlex for boosting static analysis in the presence of reflections. This work, along with our previous work named DroidRA, are actually inspired by TamiFlex.

Most recently, Barros et al. [4] propose to tackle Android ICC and reflection challenges at the same time within their Checker framework. They use annotations on top of source code to help developers checking information flows in their own apps. Unlike our approach, which, thanks to instrumentation, can directly benefit other approaches, their approach cannot be easily reused and thus presents an invasive approach (i.e., the beneficiaries have to be modified).

6. CONCLUSION

In this paper, we have demonstrated that code instrumentation is a good means to boost static analysis of Android apps. More specifically, we have shown two successful case studies that resolve ICC and reflection challenges respectively through instrumentation. We have also shown that there is a need to provide a generic approach to automate instrumentation and thus to implement a non-invasive approach for existing static analyzers, enabling them to perform extensive analysis.

7. REFERENCES

[1] T. j. watson libraries for analysis, Aug. 2014. http://wala.sourceforge.net.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Bruno Dufour, Christopher Goard, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, et al. abc: the aspectbench compiler for aspectj. In *OOPSLA*. ACM, 2005.

[3] 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.

[4] 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 *ASE*, 2015.

[5] 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.

[6] 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.

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

[8] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *SOAP*, pages 1–6. ACM, 2014.

[9] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.

[10] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *CC*, 2003.

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

[12] 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.

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

[14] Li Li, Tegawendé Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. 2015.

[15] Li Li, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Towards a generic framework for automating extensive analysis of android applications. In *SAC*, 2016.

[16] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *APLAS*. 2005.

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

[18] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.

[19] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.

[20] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*, 2014.