JoDroid: Adding Android Support to a Static Information Flow Control Tool

Martin Mohr, Jürgen Graf, Martin Hecker

Karlsruhe Institute of Technology

Abstract

We present our ongoing work on the extension of our PDG-based information flow control tool Joana to handle Android applications. We elaborate on the challenges posed by android applications, outline what we have already done and discuss what we intend to do in the future.

1 Introduction

In today's world, smartphones are becoming more and more important and contain lots of personal data which needs to be protected from unintended dissemination. Currently, the most widespread smartphone operating system is Android [1]. Android provides a permission-based mechanism to control what apps can do, but this mechanism is not sufficient: The user can see which resources an app may access and which action it is allowed to perform, but he cannot see or control how the app uses these rights. A well-known technique called *information flow control* [2] can tackle this problem. It enables the user to specify how information is allowed to be used inside the app and in particular to specify certain unwanted information flows as forbidden. With information flow control we can guarantee the absence of such information leaks by proving the *non-interference* property for the given app.

In [3], we describe Joana, a state-of-the-art information flow control analysis tool for Java bytecode, which leverages sophisticated static program analyses to construct a *program dependence graph (PDG)* and uses *context-sensitive slicing* [4] to verify non-interference.

Currently, we are working on extending Joana to handle also Android apps. Among the challenges to be addressed to achieve this goal are (1) that although Android apps are developed in Java, they are not compiled to Java bytecode but to Android's own Dalvik bytecode, (2) that standard Java applications use a single entry point (main), but Android apps have large multitude of possible entry points which are triggered by the Android system throughout the execution of the app and (3) that Android apps employ message-passing to exchange data and start external apps' components, which requires to also analyse information flows between apps. These challenges are not specific to Android. For example, many GUI-based Java applications also have multiple entry points which handle user input. However, different frameworks require different models specifying how these entry-points are used and Joana currently has no general mechanism to specify such models. Similar considerations can be made for intents: intents are comparable to other message-passing mechanisms which are commonly found in client-server-applications but currently Joana does not provide a general mechanism which applies to a wide variety of message-passing mechanisms. We therefore consider our work on extending Joana to Android as a starting point to addressing these more general challenges.

In this work, we present the work we have already done to address the challenges we just sketched. In section 2 we give a short overview of Android apps, in section 3 we discuss some basics of Joana's underlying technology, namely PDGs and slicing, in section 4 we outline how we address the above mentioned challenges and after a discussion of the related work in section 5, we conclude in section 6 by giving an outlook on future work.

2 Overview over Android applications

In the following, we briefly discuss the architecture of Android applications. This overview is largely based on [5].

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

An Android application usually consists of multiple components which are loosely bound to each other. These components can either be *Activities*, *Broadcast Receivers*, *Services* or *Content Providers*. We now give a quick summary of what these components do and which roles they play.

- *Activities:* An activity is an application component that provides a screen with which users can interact in order to do something. Each activity is given a window in which to draw its user interface.
- *Broadcast Receivers:* A broadcast receiver responds to system-wide broadcast announcements. Many broadcasts originate from the system, but can also be initiated by arbitrary app components. Broadcast receivers do not display a user interface. Typically, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.
- *Services:* A service runs in the background to perform long-running operations or to perform work for remote processes. It does not provide a user interface. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it, using a special kind of inter-process communication.
- *Content Providers:* Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

Android components use *intents* to exchange messages with each other. In particular, intents are used to start components. Intents can be *explicit* or *implicit*. Explicit intents specify a receiver, whereas implicit intents leave it up to the Android system and/or the user to resolve their receiver. Figure 1 (taken from [5]) shows how Android processes an implicit intent.

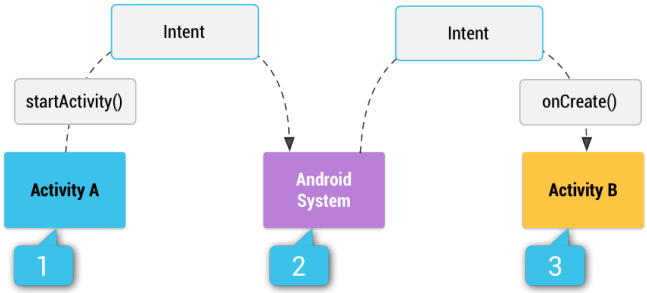

Figure 1: Illustration of how an implicit intent is delivered through the Android system to start another activity. The sending activity [1] passes an action description, the Android system [2] selects an appropriate receiver component and starts it [3].

3 Overview over Joana

In the following, we explain some fundamentals of the technologies used by Joana, namely PDGs and slicing.

PDG basics. A *Program Dependence Graph* (PDG) is a language-independent representation of a program. The nodes of a PDG represent statements and expressions, while edges model the syntactic dependencies between them. There exist different kinds of dependencies, among which the most important are *data dependencies* and *control dependencies*. Data dependencies model *explicit* information flow: they occur whenever a statement uses a value produced by another statement. Control dependencies arise when a statement or expression controls whether another statement is executed or not, and hence model *implicit* flow. As an example, consider the code snippet and its corresponding PDG snippet in figure 2: There is a data dependency between the statements in line 1 and in line 2 because the latter uses the value of `x` produced by the former. It also contains a control dependency between the `if`-statement in line 3 and the statements in lines 4 and 6 because whether line 4 or 6 is executed depends on the value of the `if`-expression in line 3.

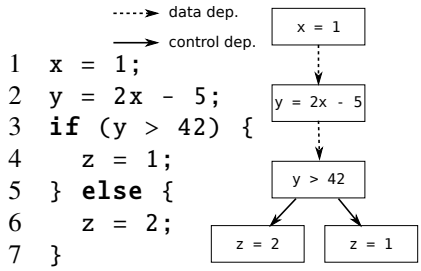

Figure 2: A code snippet and the corresponding part of its PDG

Slicing-based Information Flow Control. PDG-based information flow analysis uses *context-sensitive slicing* [6], a special form of graph reachability: given a node n of the PDG, the *backwards slice* of n contains all nodes from which n is reachable by a path in the PDG that respects calling-contexts. For sequential programs, it has been shown [7] that a node m not contained

in the backwards slice of n cannot influence n, hence PDG-based slicing on sequential programs guarantees *non-interference* [8]. It is also possible to construct and slice [9] PDGs for concurrent programs. However, in this context, additional kinds of information flows may exist, e.g. probabilistic channels [10], so mere slicing is not enough to cover *all* possible information flows between a source and a sink. A PDG- and slicing-based algorithm providing such a guarantee has recently been developed and integrated into Joana [11].

4 Approach

In this section, we explain how we address the challenges we mentioned in section 1.

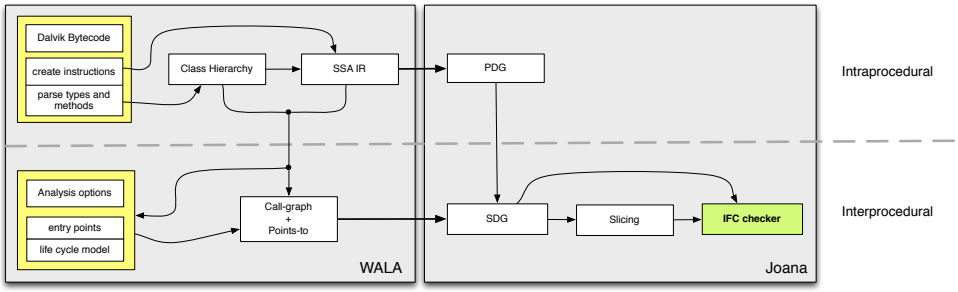

Figure 3: Overview of the architecture of Joana

Dalvik front-end Figure 3 shows the general architecture of Joana. Joana is based on WALA [12], a program analysis framework for Java bytecode. WALA provides a front-end which reads and parses the actual bytecode and basic program analyses and transformations, such as an SSA-based intermediate representation, call graph construction and points-to analysis. As can be seen in figure 3, the PDG builder of Joana only depends on WALA's analysis results and hence is decoupled from WALA's front-end. As a consequence, we only needed to adapt WALA's front end to be able to process Android apps. For this, we integrated the WALA front end code of SCanDroid [13], a security analysis tool which is also based on WALA. This was a first step to extend Joana to handle also Android apps.

Lifecycle modelling. Standard Java applications have a single entry point and every execution of the application starts with an invocation of this method. Clearly, this assumption is not met by Android apps: As we already mentioned, Android apps have multiple callbacks, which may be triggered by the user or the Android system as a reaction to certain events. However, the order and the way these callbacks are triggered is not arbitrary, but follows certain rules. More specifically, the components of an Android app are driven by their *lifecycles*. The lifecycle of an activity can be seen in figure 4.

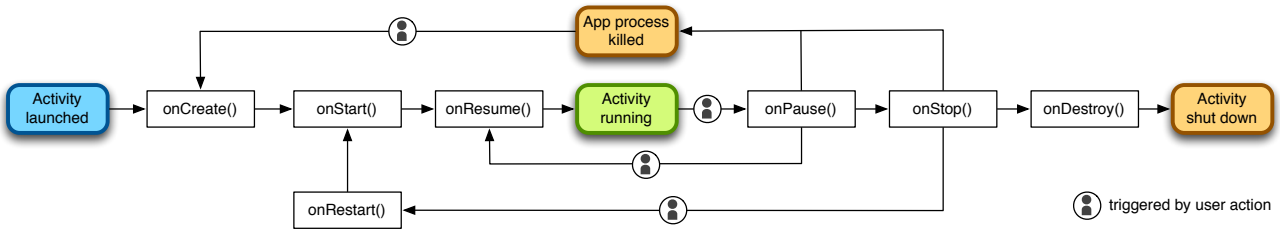

Figure 4: Lifecycle of an activity

It is not an option to run a separate analysis for each entry point, since there may be information flows which only occur if multiple callbacks are executed in sequence. Listing 1 (adapted from [14]) presents an example.

When `onCreate()` is executed, line 5 reads the IMEI of the phone and later, upon the invocation of `onStart()`, line 15 sends the IMEI to a server on the internet. However, such an information flow is not detected if `onStart()` and `onCreate()` were each analysed in isolation, since neither calls the other but both are called by the Android framework.

To also cover such flows, we synthesize an entry method which simulates the Android framework by invoking all callbacks of the given app.

In order to lose not too much precision, we take the lifecycles of the app's components into account. Consider again figure 4: When `onCreate()` is called, either the activity has just been launched, or the app's process has been destroyed and re-created. In either case, `onCreate()` is called on a fresh heap which cannot have been influenced by any other of the activity's entry points. Thus, it is safe to assume that none of the activity's entry points is called before `onCreate`. An

```
1  public class MyActivity extends Activity {
2      static String addr = "http://www.google.de/search?q=";
3      void onCreate(Bundle savedInstanceState) {
4          telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
5          imei = telephonyManager.getDeviceId();
6          addr = URL.concat(imei);
7      }
8      void onStart(){
9          super.onStart();
10         try{
11             url = new URL(addr);
12             conn = (HttpURLConnection) url.openConnection();
13             conn.setRequestMethod("GET");
14             conn.setDoInput(true);
15             conn.connect();
16         } catch(Exception ex){}
17     }
18 }
```

Listing 1: example for an information flow acress entry points

example of how this assumption can be exploited to rule out impossible information flows and thus leads to increased precision is shown in listing 2: In this variant of listing 1, the source is contained in `onStart()` and the sink is contained in `onCreate()`. Since `onCreate` is never executed after `onStart`, the sink cannot be influenced by the source.

```
1  public class MyActivity extends Activity {
2      static String URL = "http://www.google.de/search?q=";
3      void onCreate(Bundle savedInstanceState){
4          ...
5                  conn.connect();
6          ...
7      }
8      void onStart(Bundle savedInstanceState) {
9          ...
10         imei = telephonyManager.getDeviceId();
11         URL = URL.concat(imei);
12     }
13 }
```

Listing 2: An example in which there is no information flow between entry points

Intents. Our model also provides basic support for intents. In order to incorporate the intents an application may react to, the application's manifest is inspected, the possible intent targets are resolved and appropriate method calls are inserted into the artificial entry method. Similarly, our approach handles intents which may be issued during the execution of the application and whose target can be resolved to a component within the same application.

5 Related Work

From a formal description of a given web framework, the F4F system[15] generates Java code that specifies the framework-related behaviour, which allows for static taint analysis of web applications. In [16], a framework for static detection of explicit information flow in Android application using a security type system is presented. It is calling-context insensitive; the application life cycle is not modelled. In addition to explicit flow between all activities of a single Android applications, the static analysis from [13] also handles inter-application flow. A coarse model of the life cycle is used. Flow Droid[17] precisely models life cycle for a static analysis of explicit flow.

6 Conclusion and Future Work

We presented our work on extending our PDG-based information flow control tool Joana to also properly handle android applications. To achieve this goal, it was necessary to provide a front end to WALA which is able to process Dalvik bytecode. For this purpose, we integrated code from the SCanDroid project. Furthermore, we had to deal with the fact that Android apps have multiple entry points, whereas standard Java applications only have a single entry point.

We did this by synthesizing an artificial main method which calls all the app's entry points in all possible orders. To not be overly imprecise, the generated code respects the lifecycle specifications of Android components, e.g. of activities.

We now elaborate on the work that is left to do.

At the moment we do not handle callbacks of graphical user interfaces. We plan to integrate these callbacks in the future. The graphical user interfaces of Android apps are typically described in separate files and these files also contain the callbacks which are invoked on user input, e.g. when a button is pressed.

Hence, to also cover GUI callbacks, they have to be extracted from the separate files and integrated into the artificial main method appropriately.

Last but not least, we only analyse information flows inside single apps, but no information flows between different apps. This could be achieved by simply analysing all the apps simultaneously. However, such an analysis would have to be re-done

each time an app is added. Additionally, the analysis would have to be adapted in order to not assume that all the apps under analysis share a single heap (normally, different apps run in different virtual machines and hence have separate heaps).

An alternative, more modular approach is outlined in figure 5: First, the intra-app flows in each single application are analysed and summaries are generated from these analysis results. After that, a *communication graph* is built by connecting one app's summary with another app's summary if one of the former app's components may trigger one of the latter app's components by issuing an intent. The paths of such a graph represent the possible information flows between the apps.

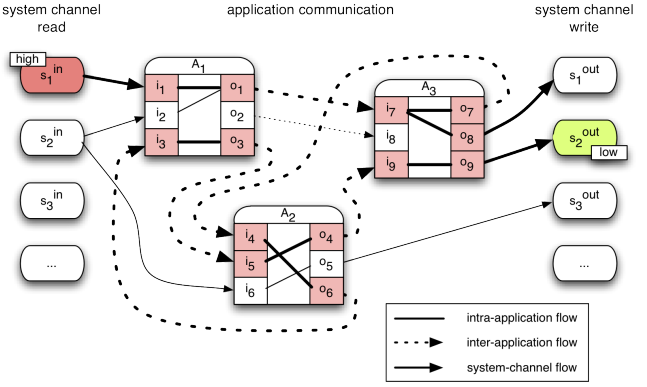

Figure 5: possible approach to capture inter-app flows

Acknowledgments.

Tobias Blaschke provided an implementation of our approach in his diploma thesis. This work was funded by the DFG under the project in the priority program RS3 (SPP 1496) and by the BMBF under the KASTEL competence center for applied IT security technology.

References

[1] International Data Corporation. Smartphone os market share, q3 2014. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed: 2014-12-22.

[2] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[3] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proc. 6th ATPS*, pages 123–138, 2013.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[5] Google. Android API Guide. `http://developer.android.com/guide/index.html`. Accessed: 2014-12-23.

[6] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 11–20, New York, 1994. ACM.

[7] D. Wasserrab and D. Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th Int. Verif. Worksh.*, pages 141–155, 2010.

[8] J.Goguen and J.Meseguer. Security policies and security models. In *IEEE Symposium on Sec. & Priv.*, pages 11–20, 1982.

[9] D. Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[10] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[11] D. Giffhorn and G. Snelting. A New Algorithm for Low-Deterministic Security. *International Journal of Information Security*, 2014.

[12] T.J.Watson Library for Analysis (WALA), `http://wala.sf.net`. http://wala.sf.net. Accessed on 2014-12-23.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. Technical Report CS-TR-4991, University of Maryland, Department of Computer Science, 2009.

[14] DroidBench. `http://sseblog.ec-spride.de/tools/droidbench/`.

[15] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *Proc. 26th OOPSLA/SPLASH (ACM SIGPLAN)*, pages 1053–1068, 2011.

[16] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1457–1462, 2012.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 49, pages 259–269, 2014.