Poster: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps

Fengguo Wei (Student), Sankardas Roy (Research Associate), Xinming Ou (Professor), Robby (Professor)
Dept. of Computing and Information Sciences, Kansas State University, Manhattan, Kansas 66506
Email: fgwei, sroy, xou, robby@ksu.edu

I. Introduction

Most of the prior works [1], [2], [3], [4] on applying static analysis to address security problems in Android applications focused on specific problems and built specialized tools for the problems. We observe that the analysis of a large variety of security issues in Android apps (malicious or vulnerable) share an underlying core problem, which is capturing semantic behaviors of the apps such as determining object points-to information precisely.

We have thus designed a new approach to conducting static analysis for vetting Android apps, and built a general framework called Amandroid. Amandroid can determine points-to information for *all* objects in an Android app in a flow and context-sensitive way across Android apps components. We show that: (a) this type of comprehensive analysis is completely feasible in terms of computing resources needed with modern hardware, (b) one can easily leverage the results from this general analysis to build various types of specialized security analyses – in many cases the amount of additional coding needed is less than 150 lines of code (LOC), and (c) the result of those specialized analyses leveraging Amandroid is at least on par and often exceeds prior works designed for the specific problems. Detailed comparison of Amandroid's results with those of prior works (whenever we can obtain those tools) will be made available in a technical report.

Since Amandroid's analysis directly handles inter-component communication (ICC), it can be used to address security problems that result from interactions among multiple components from either the same or different apps. Amandroid's analysis is sound in that it can provide assurance of the *absence* of the specified security problems in an app with well-specified and reasonable assumptions on Android runtime system and its library.

II. The Amandroid Approach

Figure 1 illustrates the pipeline of Amandroid's main steps: (1) Amandroid converts an app's Dalvik bytecode to an intermediate representation (*IR*) amiable to static analysis; (2) it generates an environment model that emulates the interactions of the Android System with the app. Because of the event-based nature of Android system, we need such a model to capture the control and data flows stemming from these interactions; (3) Amandroid builds a flow- and context-sensitive inter-component data flow graph ($IDFG$) of the whole app; $IDFG$ includes the control flow graph spanning over all the reachable components of the app, and it also tracks objects flowing to any particular program point; the points-to information is extremely useful for analyzing a number of security problems that have been discussed in prior works using customized methods; (4) Amandroid builds the data dependence graph (DDG) on top of the $IDFG$, which implies explicit information flow; (5) Amandroid then can be applied in various types of security analysis using the information presented in $IDFG$ and DDG. For example, one can use DDG to find whether there is any information leakage from a sensitive source to a critical sink by querying whether there is a data dependence chain from source to sink.

III. Experimentation and Evaluation

On top of the Amandroid framework, one can design and implement plugins for the specific analysis. We extensively experimented with Amandroid in multiple types of security analyses. We used two sets of apps in the experiments: 350 popular apps (kindly shared by the Epicc group [4]) from Google Play, and a sample malware set (containing 100 apps) from a third-party security company. For brevity, we call these data sets GPlay, and MAL, respectively. We perform our experiments on a machine with 2×2.26 GHz Quad-Core Xeon and 32GB of RAM.

A. Performance and Scalability

The most computational intensive step in Amandroid is building the $IDFG$. Once the $IDFG$ is built, the running time of the subsequent analyses is negligible. We measured the time taken by Amandroid to construct $IDFG$ for 450 apps. This measurement is done on the datasets GPlay + MAL, which are all real-world apps. The median is 22 seconds; min is 5 seconds; and max is 1629 seconds. We limited the processing time of one component of an app to 10 minutes. Amandroid raised this timeout on 27 of the 450 apps.

B. Application to Security Analysis

We can map multiple security problems to queries on $IDFG$ and DDG. Below we report particular experiments addressing data leak, data injection, and misuse of APIs. All the experiments were performed on the GPlay and MAL datasets.

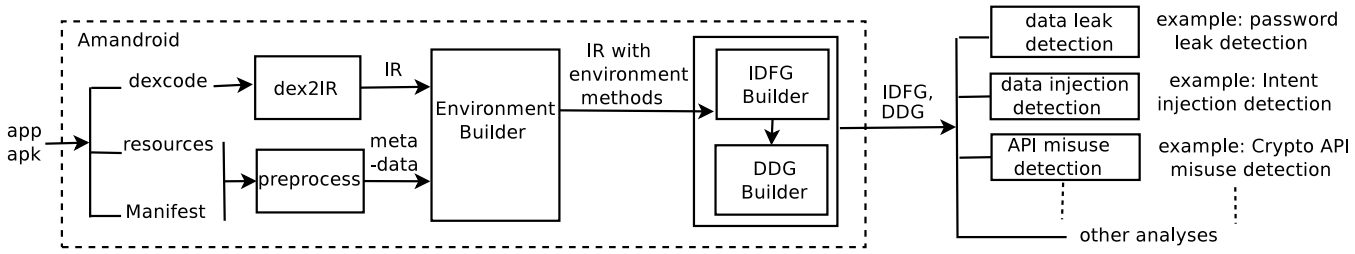

Fig. 1: The Amandroid Analysis Pipeline

TABLE I: Password Leakage Case Study

App name (app source)	App behavior
Case 1: com.datpiff.mobile.xxx (GPlay)	Get user password, encode it, then write it into log.
Case 2: com.toystorymusic-.musicapp.xxx (MAL)	Send password to server via http.
Case 3: com.snappii.angel_investing-__news_v10.xxx (GPlay)	Write user password into SharedPreference.

TABLE II: Intent Injection Case Study

App name (app source)	App behavior
Case 1: com.app.amberalertgp-stracker.xxx (GPlay)	Retrieve string from an incoming intent and display it on an Activity screen
Case 2: com.kamagames.note-pad.xxx (GPlay)	Start an activity by using the *mData* of the incoming intent

TABLE III: Crypto API Usage Case Study

App name (app source)	App behavior
Case 1: hu.sanomabp.citromail-.xxx (GPlay)	Encrypt oath token using AES default mode, then store it in SharedPreference.
Case 2: diesel.peko.ninkyodobro-wser.xxx (GPlay)	Encrypt the password using AES default mode.

TABLE IV: Plugin Summary

Plugin Name	Plugin Size	Total Analysis Time (Avg.)
Tracking password flow plugin	120 LOC	23s
Intent Injection plugin	70 LOC	71s
Crypto API check plugin	140 LOC	18s

1) Data Leak: Examples of sensitive data include user-login credentials (*e.g.*, password), location information, and so on. The leak detection can be performed through standard data dependence analysis using the DDG.

Password Leak: Amandroid can be used to check whether the input app obeys a password protection policy. In particular, Amandroid can track the path of the password object, *e.g.*, from the UI screen to the network or a storage such as *Internal Storage*, *External Storage*, or *SharedPreference*. The only "to-do" task for this analysis is to identify which variables in the app's code corresponds to a password object, which we achieve by examining the layout mapping. We prepare the list of sink APIs by considering the relevant I/O operations (*e.g.*, *URL.openconnection*). The rest of the analysis is the straightforward application of DDG. We found several examples of password leakage some of which are shown in Table I.

2) Data Injection: An app can have a vulnerability which allows an attacker to inject data into some internal data structures, leading to security problems.

Intent Injection: This is an example [3] of the data injection attack. For instance, the attacker can control the destination of an *intent*, which can lead to serious security problems. The intent injection problem can be handled via a special analysis on the DDG. We found a variety of intent injection problem in our experiment. Table II shows part of the results. We observe a couple of interesting patterns: (a) The attacker controls the content of a message (*e.g.*, Case 1 in Table II). (b) The destination of an ICC depends on an incoming intent controlled by the attacker. Furthermore, Amandroid was able to rediscover the *Next Intent* [5] problem in the Dropbox app.

3) API Misuse: Another critical part of security vetting is to find if the developer has used a library API in an improper way. We can detect these problems by querying the $IDFG$ for the possible values of each parameter object. Prior work [2] has applied static analysis techniques to identify misuse of cryptographic APIs. The main idea is to detect if the app contains improper usage of the APIs. Table III shows part of the results. These example apps encrypt the user credential using the AES cipher in the default (*i.e.*, ECB) mode. It is not clear though whether this will be a security problem, since authentication credentials (*e.g.*, passwords) may not contain more data than one block of the cipher.

4) What it Takes to Build a New Analysis: The advantage of Amandroid's approach is that the general framework (which is built with about 30000 LOC) provides a means for building a variety of further security analyses in a very straightforward and easy way. Each special analysis built on top of Amandroid involves developing a "plugin" that leverages the $IDFG$ and DDG from Amandroid's analysis. We present the summary of the plugins used in the above applications in Table IV, which shows the sizes of the plugin in LOC, as well as the average *total running times* (involving both the framework and the plugin) for each analysis.

REFERENCES

[1] ARZT, S., AND OTHERS. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI* (2014).

[2] EGELE, M., AND OTHERS. An empirical study of cryptographic misuse in Android applications. In *CCS* (2013).

[3] LU, L., AND OTHERS. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS* (2012).

[4] OCTEAU, D., AND OTHERS. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium* (2013).

[5] WANG, R., AND OTHERS. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS* (2013).