Digital Investigation 26 (2018) S59–S66

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

Automated forensic analysis of mobile applications on Android devices

Xiaodong Lin [a], Ting Chen [b, *], Tong Zhu [c], Kun Yang [b], Fengguo Wei [d]

[a] Wilfrid Laurier University, Waterloo, Canada
[b] Center for Cyber Security, University of Electronic Science and Technology of China, Chengdu, China
[c] School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China
[d] University of South Florida, Florida, USA

Keywords:
Automated forensic analysis
Android applications
Inter-component static analysis
Taint analysis

A B S T R A C T

It is not uncommon that mobile phones are involved in criminal activities, e.g., the surreptitious collection of credit card information. Forensic analysis of mobile applications plays a crucial part in order to gather evidences against criminals. However, traditional forensic approaches, which are based on manual investigation, are not scalable to the large number of mobile applications. On the other hand, dynamic analysis is hard to automate due to the burden of setting up the proper runtime environment to accommodate OS differences and dependent libraries and activate all feasible program paths. We propose a *fully automated* tool, `Fordroid` for the forensic analysis of mobile applications on Android. `Fordroid` conducts inter-component static analysis on Android APKs and builds control flow and data dependency graphs. Furthermore, `Fordroid` identifies what and where information written in local storage with taint analysis. Data is located by traversing the graphs. This addresses several technique challenges, which include inter-component string propagation, string operations (e.g., append) and API invocations. Also, `Fordroid` identifies how the information is stored by parsing SQL commands, i.e., the structure of database tables. Finally, we selected 100 random Android applications consisting of 2841 components from four categories for evaluation. Analysis of all apps took 64 h. Fordroid discovered 469 paths in 36 applications that wrote sensitive information (e.g., GPS) to local storage. Furthermore, `Fordroid` successfully located where the information was written for 458 (98%) paths and identified the structure of all (22) database tables.

1. Introduction

Mobile phones have become essential parts of our lives. Previously, mobile phones were used solely for communication purposes only. Today, their capabilities have extended to include a myriad of uses including gaming, social media, online banking and stock trading. Accompanying the proliferation of mobile devices is the presence of these devices in crime. In some instances, malicious developers can collect sensitive information without user knowledge. Mobile applications can also be used as tools to perpetrate criminal activity or be on the person of those involved in untoward or criminal behavior. Increasingly mobile devices are seen as key evidence in many cases. An example being the iPhone of the attackers in the 2015 San Bernadino attack (Wikipedia, 2015) and mobile devices in Adnan Syeds murder trial (Sali).

Mobile applications process a significant amount of user information. A large amount of sensitive information is stored locally on smartphones (Scrivens and Lin, 2017). Therefore, acquiring and analyzing artifacts generated by mobile applications is a crucial and necessary step in the forensic analysis of mobile devices.

Digital forensics on mobile devices is a complicated affair. Data acquisition and analysis in mobile phone forensics involve the extraction of information from mobile phones followed by identifying and concluding whether evidence is pertinent to the ongoing investigation. Conducting a digital forensic investigation often entails complete image extraction, however, it may be appropriate at times to only extract and examine particular mobile applications. In these cases, digital evidence on mobile devices are generated by specific applications and being stored locally.

* Corresponding author.
E-mail addresses: xlin@wlu.ca (X. Lin), brokendragon@uestc.edu.cn (T. Chen), tong.zh@foxmail.com (T. Zhu), 1481978708@qq.com (K. Yang), fwei@mail.usf.edu (F. Wei).

https://doi.org/10.1016/j.diin.2018.04.012

The digital forensic of local storage on mobile devices needs to answer the following three questions: *what* is the information stored (e.g., GPS); *where* is the information stored (e.g., file path); and *how* the information is stored (e.g., the structure of a database table). Extensive studies have been conducted in the past to identify and analyze the artifacts generated by various applications (Scrivens and Lin, 2017; Anglano, 2014). *Dynamic analysis* is the most common practice. More specifically, applications are installed on test phones or simulation environments. The application is played manually over a period of time to generate forensic traces. Unfortunately, this approach has several drawbacks.

First, it is hard to trigger all interesting program paths. Consequently, criminal behaviors may not be discovered by dynamic analysis. Moreover, it is nontrivial to identify what information is stored and how it is stored. For example, a file generated by a mobile application whose content is encoded or whose format is unknown needs considerable efforts to analyze. An alternative approach is *manual reverse engineering*. However, manually parsing documents for relevant artifacts was an arduous and long task. This approach is time consuming and requires rich technical expertise.

Hence, the aforementioned issues have motivated us to pursue an automated approach to address mobile application forensic analysis. It is hard to automate dynamic analysis given a large number of applications due to the differences in runtime environments (e.g., operation systems, dependent libraries).

This work proposes to automate forensic analysis on Android applications via static analysis. Our approach overcomes the shortcomings of manual analysis and dynamic analysis. Particularly, our approach is scalable for a large number of applications because no human intervention is required. Additionally, our approach does not need to set up a test environment and can cover all application codes.

We implement our method in `Fordroid`, an inter-component analysis tool which is able to identify what, where and how the information is stored in local storage. Technically speaking, `Fordroid` takes in an Android APK (without source code), then builds control flow and data dependency graphs after decompiling the APK. Next, `Fordroid` identifies the types of sensitive information written in local storage through taint analysis. `Fordroid` then reveals the place or file path where information is stored by traversing the graphs. Through our approach, we have overcome several of the technical challenges resulting from inter-component string propagation, string operations (e.g., append) and API invocations. Finally, `Fordroid` identifies the structure of database tables by parsing SQL commands extracted from applications.

We randomly selected 100 practical applications which belong to four categories from a popular Chinese Android application market, AppChina.[1] `Fordroid` analyzed all applications consisting of 2841 components in 3860 min (38 min per application). Results show that there are 469 paths in more that one-third (36 out of 100) of applications which wrote sensitive information to local storage. `Fordroid` successfully locates where sensitive information was written for 458 (98%) paths. Moreover, the structure of all (22) database tables which contain sensitive information was successfully identified.

In summary, our work makes the following contributions.

(1) We design and implement `Fordroid`, an inter-component static forensic tool for Android applications which automatically identifies what, where and how sensitive information is stored in local storage.

(2) We conduct experiments on 100 Android applications. `Fordroid` discovers that approximately one-third of them write sensitive information to local storage. Moreover, `Fordroid` successfully locates the places sensitive information is written for 98% paths and identifies the structure of all database tables.

The remainder of this paper is organized as follows. Section 2 gives a motivating example. Section 3 describes the design and implementation of `Fordroid`. Experimental results are given in Section 4. The limitations of our approach are discussed in Section 5. We briefly introduce related studies in Section 6 and conclude this paper in Section 7.

2. Motivating example

In this section, we provide an example of mobile application forensic analysis. Such examples are commonplace and motivated our development of `Fordroid`. We use a practical Android application, agilebuddy[2] to illustrate the difficulty of manually reverse engineering and dynamic analysis to locate sensitive information. agilebuddy is a game application with 703 KB large. It has 13 packages, 7 components, 80 classes and 559 functions. For ease of presentation, we decompile[3] this APK and illustrate its source in Fig. 1.

Line 138 in function c(), class h, package com.uucunadsks.b (Fig. 1(a)) produces a string v0_1 by calling the function a(). Line 139 creates a File object v1 using v0_1 as the file name. Line 140 creates another File object v4 which takes in two parameters, v1 and a string, v0_1. Finally, sensitive information is written into this file in Line 171. *It is difficult to reverse engineer this app to locate the critical four lines of code*.

We failed to create the file using dynamic analysis which prompted us to investigate the reason. To begin, in order to trigger the code in Line 138, several conditions should be satisfied. First, the function c() should be called and then the Boolean arg6 (Line 124) should be false. Besides, h.g.length() should be no smaller than h.f (Line 126) which is 8192 (Line 26). Moreover, h.e should not be equal to null (Line 128). Additionally, an sdcard should be mounted (Line 134). The last condition can be satisfied by preparing an Android phone with sdcard mounted. We found the condition in Line 128 to be easily satisfied through code inspection. Particularly, h.e is a *context* object which is the *this* pointer of a component.

However, it is hard to meet the condition in Line 126. h.g is a string, so this condition indicates that the length of this string should be no shorter than 8 K bytes. h.g is used to log exception information, as shown in Fig. 1(c) which invokes h.a() to generate exception information. Please note that the code snippet in Fig. 1(c) resides in another package, com.uucunadsdk.c which further increases the difficulty of analysis. h.a() (Fig. 1(b)) appends a flag (i.e., arg6, Line 72), date, class name, method name, line number and exception type (i.e., arg8, Line 73) into h.g. Hence, the space required for logging one exception cannot be longer than 100 bytes. Consequently, dynamic analysis must trigger at least 80 exceptions before it creates a file.

Therefore, *it is difficult for dynamic analysis to discover the file due to the difficulty of triggering the program path to the critical code*. The limitations of manual reverse engineering and dynamic analysis motivate us to develop an automated static approach. We will demonstrate how `Fordroid` processes this APK in Section 3.

[1] http://www.appchina.com/.

[2] http://www.appchina.com/app/com.app.kg.agilebuddy.

[3] Decompiled by JEB, https://www.pnfsoftware.com/jeb2/.