DFRWS
DIGITAL FORENSIC RESEARCH CONFERENCE

Automated Forensic Analysis of Mobile Applications on Android Devices

By

Xiaodong Lin, Ting Chen, Tong Zhu, Kun Yang, Fengguo Wei

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2018 USA

Providence, RI (July 15th - 18th)

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/.

```
26   h.f = 8192;
124 if(!arg6){
125     try{
126       if(h.g.length() < h.f){
127       }
128       else if(h.e != null){
129         goto label_11;
130       }
131       goto label_9;
133     label_11:
134       if(!Environment.getExternalStorageState().equals(``mounted``)){
135         goto label_9;
…
138       v0_1 = h.a(new Date(), ``yyyy-MM-dd`` + ``.log``;
139       File v1 = new File(Environment.getExternalStorageDirectory(),
            h.e.getPackageName() + ``/logs/``;
140       v4 = new File(v1, v0_1);
…
162       v2 = new FileWriter(v4, true);
…
171       v2.write(v0_1);
```

(a) Code snippet in function c(), class h, package com.uucunadsdk.h

```
72  v0.append(arg6).append("|").append(h.a(new Date(),
      "yyyy-MM-dd HH:mm:ss")).append("|").append(v1[v5].
      getClassName()).append("|").append(v1[v5].getMethodName()).
      append("|").append(v1[v5].getLineNumber());
73  h.g.append(v0.toString()).append("|").append(arg8).append("\n");
```

(b) Code snippet in function a(), class h, package com.uucunadsdk.h

```
166 catch(UnknownHostException v0) {
…
169     try{
170     label_86:
171       h.a(this.a, v0_toString());
```

(c) Code snippet in function b(), class m, package com.uucunadsdk.c

Fig. 1. How agilebuddy writes data to a file.

3. Approach

3.1. Overview

The current implementation of `Fordroid` is based on Amandroid (Wei et al., 2014), an inter-component static analysis platform for Android applications. We will briefly introduce Amandroid here. Amandroid determines points-to information (a core underlying problem in almost all static analyses for Android applications) in a flow and context-sensitive way. It first decompiles an APK into an IR representation and then builds an abstract syntax tree (AST) from it. Amandroid builds inter-procedural control flow graph (ICFG) of the whole application and treats inter-component communication (ICC) just like method calls. Subsequently, it builds inter-component data flow graph (IDFG) which associates ICFG with reaching facts and then derives data dependence graph (DDG) from IDFG. Moreover, Amandroid provides a rich set of APIs for developing analysis applications.

Fig. 2 presents the architecture of `Fordroid` which takes in an Android APK and produces a report containing what, where and how sensitive information is stored in local storage. `Fordroid` takes advantages of Amandroid to decompile APKs, build AST, ICFG, IDFG, and DDG, and use one of its sample applications, taint analyzer to find the types of sensitive information written to local storage. Technically speaking, taint analysis (Wikipedia and Taint checking) consists of taint source, taint propagation and taint sink. In the context of tracking information leakage, a taint source is the place where sensitive information enters an application (e.g., get the location by invoking getLastKnownLocation()). Taint propagation tracks the propagation of sensitive information along the execution of program instructions. A taint sink is the place where sensitive information leaks outside of the application (e.g., write data into a local file by invoking Write()).

Android applications typically obtain and leak sensitive information through invoking APIs, so `Fordroid` monitors the related APIs. Amandroid already supports taint analysis, so `Fordroid` extends the list of taint sinks with five more APIs. Amandroid reports the paths of information leakage, each of which consists of a taint source, the propagation path and a taint sink. Hence, from Amandroid, we discover the type of sensitive information and the code locations of APIs which acquire and leak information, respectively. `Fordroid` has 970 lines of Scala code.

To identify where the sensitive information is stored, `Fordroid` traverses the graphs backward from the code location of the taint sink until a string (e.g., file path) indicating the storage location is reached. `Fordroid` implements three modules (inside the dotted box in Fig. 2) to overcome three technical challenges which are API invocations, string operations and inter-component string propagation (ICSP) during graph traversing, respectively. Finally, `Fordroid` unveils the structure of database tables by parsing SQL commands which create new tables.

We will go into detail on the approach `Fordroid` takes to find the location where sensitive information is stored and the structure of database tables in more detail in the following sections.

3.2. Data storage modes

Android applications typically have three modes to store information in local storage, i.e., stored in SharedPreferences, database or to file. `Fordroid` handles these three modes differently because they require different APIs and code patterns. By checking taint sinks, `Fordroid` can identify which mode a writing operation follows. For example, an application invokes Editor.putString(), SQLiteDatabase.insert() and FileWriter.write() to record data in SharedPreferences, database and file, respectively.

3.2.1. Handling SharedPreferences

We propose Algorithm 1 to find the path of SharedPreferences whose content contains sensitive information. This algorithm takes in a taint sink, *sk* indicating the code location where sensitive information is written into SharedPreferences, AST, CFG and DDG. Firstly, `Fordroid` finds the caller (i.e., editor) of the taint sink, sk (Line 1) and then searches backward in CFG and DDG for the definition location (i.e., ed_def) of the caller (Line 2). Note that the use-definition (Wikipedia) relationship is available after building IDFG (Inter-component Data Flow Graph, Fig. 1).

Then, `Fordroid` searches AST for the SharedPreferences object, sp (Line 3). Afterwards, `Fordroid` searches the graphs backward to find the definition location of sp. There are three approaches to define a SharedPreferences object, which are invoking getDefaultSharedPreferences(), getPreferences(), and getSharedPreferences(). The first two APIs produce SharedPreferences in a default path, so `Fordroid` returns a default string (e.g., "defaultSharedPreferences") (Line 8). If getSharedPreferences() is used (Line 6), its first parameter is the file path (Line 7).

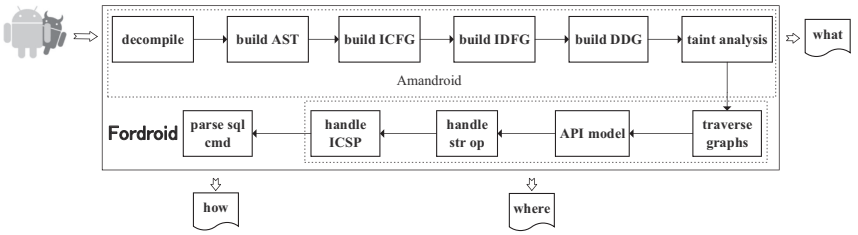

Fig. 2. Architecture of `Fordroid`.

```
423 SharedPreferences v2 = arg10.getSharedPreferences
        ("message_prefs", 0);
…
441 SharedPreferences$Editor v0_1 = v2.edit();
442 v0_1.remove(v3);
443 v0_1.putString(v3, arg11.a());
```

Fig. 3. A code snippet in function b(), class a, package com.kuguo.pushads of gobang.

We use an example (Fig. 3) to explain Algorithm 1 in plain words. The analyzed APK, gobang[4] is a game and the code snippet locates in function b(), class a of package com.kuguo.pushads. Sensitive information is written to SharedPreferences by invoking putString() at Line 443. The caller is v0_1 and its definition site is Line 441. Afterwards, `Fordroid` finds its corresponding SharedPreferences object, v2 from the AST and then finds v2's definition site, Line 423. Finally, `Fordroid` discovers the path in the first parameter of getSharedPreferences().

Algorithm 1. Find the path of SharedPreferences.

```
Input: sk, ast, icfg, ddg;
Ouput: path;
1  editor = getCaller(sk, ast);
2  ed_def = defsite(editor, icfg, ddg);
3  sp = getCaller(ed_def, ast);
4  sp_def = defsite(sp, icfg, ddg);
5  switch sp_def{
6     case SharedPreferences:
7        path = getPara(sp_def, 1, ast); break;
8     default: path = defaultPath();}
9  return path;
```

3.2.2. Handling databases

If the application stores data in a database, we need to know both the name of database and the name of table. We propose Algorithm 2 to find both names, which takes in a taint sink, *sk*, AST, CFG and DDG. It is not difficult to obtain the table name because it is the first parameter of *sk* (Line 1). Then, `Fordroid` finds the caller, db, (i.e., SQLiteDatabase) of sk (Line 2). Afterwards, `Fordroid` traverses the graphs to find the definition site of db (Line 3). After searching the AST, `Fordroid` finds an object, helper, which extends the class SQLiteOpenhelper (Line 4). Then, `Fordroid` searches for the constructor, cons of helper in the AST (Line 5). `Fordroid` then searches for the invocation of the constructor (i.e., super, Line 6) of helper's superclass (i.e., SQLiteOpenhelper). Finally, `Fordroid` finds the database name in the second parameter of super (Line 7).

Algorithm 2. Find database name and table name.

```
Input: sk, ast, icfg, ddg;
Ouput: <db_name, tb_name>;
1  tb_name = getPara(sk, 1, ast);
2  db = getCaller(sk, ast);
3  db_def = defsite(db, icfg, ddg);
4  helper = getCaller (db_def, ast);
5  cons = constructor(helper, ast);
6  super = superFunc(cons, ast);
7  db_name = getPara(supper, 2, ast);
8  return <db_name, tb_name>;
```

We also use gobang to explain Algorithm 2. This application writes sensitive information into a database table at Line 45, Fig. 4(a) by invoking insert(). The table name (i.e., downloads) is the first parameter of insert(). `Fordroid` finds the caller of insert(), which is v0, and then finds v0s definition site, Line 38. After searching AST, `Fordroid` gets the object this.a, whose constructor a() is shown in Fig. 4(b) a() invokes the constructor of its superclass at Line 9. Finally, `Fordroid` finds the database name (i.e., downloads) in the second parameter.

Algorithm 3. Find file path.

```
Input: sk, ast, icfg, ddg;
Ouput: path;
1  fw = getCaller(sk, ast);
2  fw_def = defsite(fw, icfg, ddg);
3  para = getPara(fw_def, 1, ast);
4  if(type(para) == str)
5     path = para;
6  else if(type(para == File))
7     para_def = defsite(para, ast);
8     path = getPathFromFile(para_def, ast, icfg, ddg);
9  return path;

10 str getPathFromFile(f, ast, icfg, ddg){
11    num = numPara(f, ast);
12    para1 = getPara(f, 1, ast);
13    if(num == 1) return para1;
14    else if(num == 2){
15       para2 = getPara(f, 2, ast);
16       if(type(para1) == File){
17          para1_def = defsite(para1, icfg, ddg);
18          return getPathFromFile(para1_def, ast,
               icfg, ddg) + para2;
19       else if(type(para1) == str)
20          return para1 + para2;}}}
```

[4] http://www.appchina.com/app/com.yj.gobang.

```
38  SQLiteDatabase v0 = this.a.getWritableDatabase();
39  ContentValues v1 = new ContentValues();
40  v1.put("url", arg5.a);
41  v1.put("file", arg5.b);
42  v1.put("size", Integer.valueOf(arg5.c));
43  v1.put("total_size", Integer.valueOf(arg5.d));
44  v1.put("state", Integer.valueOf(arg5.e));
45  v0.insert("downloads", null, v1);
```

(a) Code snippet in function a(), class b, package com.kuguo.a

```
7  class a extends SQLiteOpenHelper {
8    a(Context arg4) {
9      super(arg4, "downloads", null, 1);
10   }
```

(b) Code snippet in class a, package com.kuguo.a

Fig. 4. Code snippets in package com.kuguo.a of gobang.

3.2.3. Handling files

Android applications can use FileWriter, BufferedWriter and FileOutputStream to write files. Fordroid handles all of these cases. For ease of presentation, this paper describes writing files with FileWriter because the three cases share the similar programming pattern. The procedure is shown in Algorithm 3, which is an iterative process because the first parameter of new File() can be another File object.

We explain this algorithm using the example shown in Fig. 1(a) for ease of understanding. Taint analysis reveals that in Line 171, Fig. 1(a) writes sensitive information to a file by invoking write(). Fordroid gets the caller of write(), fw at Line 1, Algorithm 3. Then, it searches the graphs for the definition site fw_def of fw (Line 2, Algorithm 3). fw_def locates at Line 162, Fig. 1(a). Fordroid, then, gets the first parameter, v4 at Line 162, Fig. 1(a). FileWriter has two constructors. The first accepts a string, which is the file path. In this case, Fordroid simply gets the parameter (Line 5, Algorithm 3).

The second constructor takes in two parameters and the first is a File object, as shown in Line 162, Fig. 1(a). In this case, Fordroid searches for its definition site, Line 140, Fig. 1(a). Then, Fordroid invokes getPathFromFile() (Line 8, Algorithm 3) iteratively. In this routine, Fordroid gets the parameter number and the first parameter (Line 11 and 12, Algorithm 3). If new file() accepts only one parameter, the parameter should be the file path (Line 13, Algorithm 3). If new File() takes in two parameters, the second one should be the latter part of file path (Line 15, Algorithm 3).

If the first parameter is a string, it should be the first part of file path (Line 19, Algorithm 3). If the first parameter is a File object, as shown in this example, Fig. 1(a), Fordroid searches for the definition site of the file object (Line 17, Algorithm 3). In this example, the definition site is Line 139, Fig. 1(a). Afterwards, Fordroid invokes getPathFromFile() again (Line 18, Algorithm 3). Particularly, Fordroid finds that new file() at Line 139, Fig. 1(a) accepts two strings (refer to str1 and str2) and then returns the string str1/str2 as the file path. Consequently, the file path in this example is str1/str2/logs/str3.log, where str1, str2 and str3 refer to the path of external storage, package name and date, respectively.

3.3. Handling inter-component string propagation

We are particularly interested in strings since they can represent the paths of SharedPreferences, database names, table names, file paths and SQL commands. Fordroid needs to track string propagation across components because a string can be sent from one component to another. Fordroid is based on Amandroid (Wei et al., 2014), and allows us to find connection between components. In other words, Fordroid is aware of the target components of intents (i.e., messages) sent by a component. The remaining task for Fordroid is understanding how strings are stored in and retrieved from the intents.

To achieve this, Fordroid models two APIs, putStringExtra() for packing a string into an intent and getStringExtra() for extracting a string from an intent. Please note that putStringExtra() is invoked by the component which sends the intent and getStringExtra() is invoked by the component which receives the intent. Hence, by modeling them, Fordroid maps the connections of strings across components.

3.4. String operations

String operations (e.g., append, substring, index) are commonly used in Android applications to build various strings. Although string analysis (Li et al., 2015a) is powerful to infer the structure of strings, it results in high overhead. Moreover, although Java allows various string operations, *append* (e.g., h.e.getPackageName() + "/logs/", Line 139, Fig. 1(a)) is the most widely-used string operation involving the construction of the path of SharedPreferences, database name, table name and file path. Therefore, Fordroid models the API StringBuilder.append() to avoid the high overhead of string analysis.

3.5. API invocations

File paths often contain substrings created via API calls. Recall the example in Fig. 1(a), the application invokes Date() (Line 138), getExternalStorageDirectory() (Line 139) and getPackageName() (Line 139) to retrieve the current date, the path of external storage and the package name, respectively. All of these strings constitute the file path. To overcome the challenge, Fordroid models common APIs whose return values can be parts of the path of SharedPreferences, database names, table names and file paths.

3.6. Identifying the structure of database table

To reveal how sensitive information is stored in local storage, Fordroid identifies the structure of database tables from APKs. Technically, Fordroid monitors the API, execSQL() for executing SQL commands and then extracts SQL commands from its parameters. Then, Fordroid finds the SQL command for creating database table by looking for the keyword "CREATE TABLE". Finally, Fordroid parses the SQL command to retrieve the table name and the name and type of each column.

Fig. 5 shows an example from gobang. After parsing the SQL command at Line 13, Fordroid is aware that the table is named "downloads" and contains six columns. As well, the name and type of each column are also obtained, e.g., the first column is _id which is an integer and the primary key of this table.

4. Evaluation

To evaluate our approach, we randomly select 100 practical applications belonging to four categories from AppChina. Fordroid is

```
12  public void onCreate(SQLiteDatabase arg2) {
13    arg2.execSQL("CREATE TABLE downloads(
        _id INTEGER PRIMARY KEY, url TEXT, file TEXT,
        size INTEGER, total_size INTEGER, state INTEGER);");}
```

Fig. 5. A code snippet from class a, package com.kuguo.a.

Table 1
Analysis results of 100 Android applications.

category	#APKs	# comp.	time (min)	# paths	# paths to storage			where		# APKs with paths	# APKs write storage
					sp	db	file	suc.	fa.		
comm.	26	978	1827	310	27	0	7	33	1	11	6
enter. & game	26	278	207	422	196	18	8	221	1	11	8
news & info.	24	715	902	360	30	4	10	38	6	16	10
tool	24	870	924	1221	163	0	6	166	3	18	12
total	100	2841	3860	2313	416	22	31	458	11	56	36
ave.	/	28.4	38.6	23.1	41.6	2.2	3.1	45.8	1.1	/	/

installed in a desktop equipped with 12 Intel E5-2640 CPUs and 20 GB main memory.

4.1. Results

`Fordroid` completes the analysis of 100 applications in about 64 h and the results are shown in Table 1. The first four columns provide the category, number of APKs, number of components, and the time for analysis, respectively. Column 5 reports the number of paths discovered by taint analysis, which considers all taint sinks defined in Amandroid. Hence, column 5 includes other venues to leak information, e.g., SMS message, Internet. Columns six to eight provide the number of paths writing sensitive information to SharedPreferences, database and file, respectively.

Columns nine and ten present the number of paths `Fordroid` succeeds and fails to find where sensitive information is written to. Column 11 provides the number of APKs leaking sensitive information and column 12 shows the number of APKs writing sensitive information to local storage. Rows three to six present the statistics of four categories (i.e., communication, entertainment & game, news & information, and tool). Row seven denotes the total number and the last row presents the averages per application.

We have made several observations from the experimental results. Firstly, more than a half (56%) of applications leak sensitive information and more than one-third (36%) of applications write sensitive information to local storage. Hence, *information leakage is prevalent* even if all those applications are not malware. Another insight is that it is also interesting to investigate information leakage via other venues because 20 applications leaked sensitive information without writing local storage. Secondly, *sensitive information is more likely to be written into SharedPreferences* (416), compared to database (22) and file (31). This observation is not surprising because Android recommends developers to use SharedPreferences, which is a light-weight file.

Moreover, `Fordroid` is efficient because it needs just 38 min to analyze one application on average. Furthermore, `Fordroid` is effective in locating where sensitive information is stored, as shown in Table 1 where `Fordroid` finds 458 out of 469 (98%). We will investigate the 11 paths `Fordroid` fails to identify locations in Section 4.2. Furthermore, `Fordroid` successfully reveals the structure of all (i.e., 22) database tables which contain sensitive information.

4.2. Investigation of failed cases

`Fordroid` failed to identify the sensitive data locations for 11 paths (2%). We investigated the 11 paths manually and concluded this to be due to two reasons which are string operations (3 paths) and input dependency (8 paths).

Fig. 6 gives code snippets from ttwindow[5] which contains string operations resulting in unidentifiable file path by `Fordroid`. Line 95,

```
89  public static final boolean a(byte[] arg4, String arg5){
90      File v1;
91      try{
92          v1 = new File(arg5);
93          if(!v1.exist()){
94              FileOutputStream v2 = new FileOutputStream(v1);
95              v2.write(arg4);}}
```

(a) Write file in class b, package com.yulong.d

```
393 String v2 = com.yulong.d.b.a(this.a, ((b)v1).d);
…
397 else if(com.yulong.d.b.a(((b)v1).e.a, v2)){
```

(b) A code snippet in function onCreate(), class ImageDetailActivity, package com.yulong.ttwindow

```
12  public static final String a(Context arg10, String arg11) {
…
21      String v1 = …
…
33      String v2 = String.valueOf(arg11.hashCode()) + "." +
            arg11.substring(0, v5);
…
35      v6 = v1 + v2;
…
42      return v6;}
```

(c) Generate file path in in class b, package com.yulong.d

Fig. 6. String operations that cannot be handled by `Fordroid`.

Fig. 6(a) writes sensitive information to a file which is created at Line 92. The file path is passed as a parameter, arg5 to function a(), which is invoked at Line 397, Fig. 6(b). By searching for the definition site of v2 (at Line 397), `Fordroid` locates Line 393 and is aware of the file path generated by invoking com.yulong.d.b.a(), whose code is shown in Fig. 6(c). The string v2 (Line 33) is a result of the hashCode() operation and a substring() operation of the string arg11. However, the current implementation of `Fordroid` does not support such two string operations. We plan to extend `Fordroid` with the capability of handling more string operations in future work.

Fig. 7 presents code snippets from radish[6] which contains a file path depending on its input. Line 118, Fig. 7(a) writes sensitive information into a file and the OutputStream object is passed as a parameter by invoking the copy() function at Line 106. `Fordroid` backtracks the File object, arg3 and finds the corresponding object, this.imagefile at Line 51, Fig. 7(a). Next, we found that this.imagefile depends on v3 and v3 results from this.imageUrl which comes from a parameter of the function doInBackground() (Line 36). This

[5] http://www.appchina.com/app/com.yulong.standalone.ttwindow.

[6] http://www.appchina.com/app/com.mofang.radish.

```
105 public static long copy(InputStream arg2, File arg3)
        throws IOException {
106     return BitmapUtil.copy(arg2, new FileOutputStream(arg3));}
…
109 public static long copy(InputStream arg6, OutputStream arg7)
        throws IOException {
…
118     arg7.write(v0, 0, v3);}
```

(a) Write file in class BitmapUtil, package com.mofang.radish.utils

```
34  protected Bitmap doInBackground(String[] arg15) {
35      Bitmap v11;
36      this.imageUrl = arg15[0];
…
44      String v3 = StringUtil.encodeByMD5(this.imageUrl);
45      File v5 = new File(…);
46      this.imagefile = new File(v5, v3);
…
51      BitmapUtil.copy(v9, this.imagefile);}
```

(b) Create file in class LruImageAsyncTaskForAcitivity, package com.mofang.radish.utils

Fig. 7. A file path depends on input.

function is a callback function for executing asynchronous tasks which receive inputs. To identify input data, `Fordroid` should work in tandem with dynamic analysis.

4.3. Case study of doudizhugm

doudizhugm[7] is a card game with 2837 KB, which consists of 21 packages, 12 components, 259 classes and 2055 functions. `Fordroid` found 123 paths by which sensitive information was leaked. Among them, 78 paths wrote sensitive data into local storage (i.e., 70 to SharedPreferences and 8 to database). All the eight paths to the database write data into a table named “downloads” of a database “downloads”. doudizhigm has the same behaviors of obtaining access to database as gobang (Section 3.2.2) because they incorporate the same package com.kuguo. We provide a code snippet (Fig. 8) which leaks information to SharedPreferences.

`Fordroid` discovers two taint paths (refer to p1 and p2) after taint analysis. The taint source of p1 locates in Line 35, class SetPreferences, package com.airpush.android which collects the longitude when the location changes. The taint sink is located in Line 273. Similarly, p2 gets the latitude at Line 36 and writes it into SharedPreferences at Line 274. `Fordroid` firstly finds the caller v1 and then locates its definition site at Line 260. Finally, it extracts the path of SharedPreferences from the first parameter of getSharedPreferences(), which is “dataPrefs”.

5. Discussion

This section discusses several limitations of our work. The implementation of `Fordroid` is based on Amandroid, and hence `Fordroid` shares similar shortcomings with Amandroid. For example, Amandroid is path-insensitive, so it may take infeasible paths into consideration. Besides, as a static analysis tool, it is not easy for Amandroid to analyze highly obfuscated applications.

[7] http://www.appchina.com/app/com.supergame.game.doudizhugm.

```
33  public void onLocationChanged(Location arg3) {
34      try {
35          SetPreferences.lon = String.valueOf(arg3.getLongitude());
36          SetPreferences.lat = String.valueOf(arg3.getLatitude());
…
257 protected void setSharedPreferences() {
…
260     SharedPreferences$Editor v1 = SetPreferences.ctx.
            getSharedPreferences("dataPrefs", 2).edit();
…
273     v1.putString("longitude", SetPreferences.lon);
274     v1.putString("latitude", SetPreferences.lat);
```

Fig. 8. doudizhugm writes location information in SharedPreferences.

Additionally, some features of Java language, e.g., reflection will increase the difficulty of static analysis.

Although `Fordroid` is built on top of Amandroid, there are little technical obstacles to adapt it to other tools because `Fordroid` needs AST, control flow and data flow information which are supported by most Android static analysis tools. `Fordroid` reuses the taint analyzer of Amandroid. A recent survey shows that taint analysis is the most applied technique in Android static analysis tools (Li et al., 2017).

`Fordroid` identifies the structure of database tables, rather than SharedPreferences and files. To reverse engineer the format of files written by the application, advanced techniques (e.g., Dispatcher (Caballero et al., 2009) infers command format sent by the application) should be incorporated, which will be our future work. Moreover, Android applications sometimes include native code, however, `Fordroid` only processes Dalvik bytecode. Additionally, we plan to extend `Fordroid` to support more string operations.

In summary, `Fordroid` is suitable for analyzing normal applications (e.g., those in application markets as shown in Section 4). However, to analyze Android malware, `Fordroid` should cooperate with other techniques (e.g. deobfuscation, unpacking, binary analysis, dynamic analysis) because malware is likely to adopt sophisticated obfuscation, packing and native code to protect itself.

6. Related work

To the best of our knowledge, `Fordroid` is the first work conducting digital forensics of Android applications to unveil what, where and how sensitive information is stored in local storage in a fully automatic way. Therefore, `Fordroid` can be used to forensically analyze any Android applications. Currently, a variety of commercial mobile phone forensic systems have become available, such as MPE+, DC4500, FT, Fanaldata. However, they only support a limited number of mobile applications. Extensive studies have been conducted in the past to identify and analyze the artifacts generated by various mobile applications. Particularly, the popular applications, including WhatsAPP, Weixin, Facebook Messenger and Google Hangouts (Scrivens and Lin, 2017; Anglano, 2014).

Unfortunately, more obscure or less popular applications require more intensive studies since these applications also generate forensically rich data. For example, Pokémon GO, a popular location-based virtual-reality game developed by Niantic, keeps track of the players' location in the background (Hafner, 2016). However, due to a lack of studies on the applications, it is hard to find where all the important information is stored. Furthermore, the common practice of existing works is to utilize manual tests, more specifically, applications that are installed on test phones that

will be used in simulated environments. The application is then played for a period of time to generate forensic traces. Unfortunately, this approach is problematic as discussed early. It has become obvious manual approaches will not meet today's requirements for mobile application forensic analysis.

Next, we briefly introduce static techniques for analyzing Android applications because `Fordroid` belongs to such category.

Flowdroid (Arzt et al., 2014) proposes a precise model of Androids lifecycle that allows the analysis to properly handle callbacks invoked by the Android framework. Flowdroid is context, flow, field and object-sensitive. This allows the analysis to reduce the number of false alarms. However, Flowdroid over-approximates ICC. EPICC (Octeau et al., 2013) reduces the discovery of ICC to an instance of the inter-procedural distributive environment (IDE) problem, and hence it scales to large numbers of applications. IC3 (Octeau et al., 2015) defines ICC analysis as the problem of multi-valued composite (MVC) constant propagation and designs a COAL solver to infer ICC values.

Iccta (Li et al., 2015b) integrates Flowdroid, EPICC and IC3, and takes advantages of their strengths. DroidSafe (Gordon et al., 2015) combines a comprehensive, accurate, and precise model of the Android runtime with static analysis design decisions to achieve high analysis precision. Edgeminer (Cao et al., 2015) addresses the challenge of Android callbacks by statically analyzing the entire Android framework to automatically generate API summaries that describe implicit control flow transitions.

Yang et al.'s work (Yang et al., 2015) reduces control flow analysis problem to modeling of the possible sequences of callbacks and then proposes to represent an application by a callback control–flow graph (CCFG). HornDroid (Calzavara et al., 2016) abstracts the semantics of Android applications as Horn clauses and formulates security properties as a set of proof obligations, which are solved by invoking off-the-shelf satisfiability modulo theories (SMT) solvers. Amandroid (Wei et al., 2014) performs data flow and data dependence analysis for each component of the analyzed application. Amandroid also tracks the inter-component communication activities, and hence it can be used to address security problems that result from interactions among multiple components.

7. Conclusion

Manual digital forensics for Android applications is time-consuming and hard to scale to large number of applications. We propose and implement a fully automated approach, `Fordroid` to unveil what, where and how sensitive information is stored in local storage. We overcome the challenges resulting from inter-component string propagation, string operations and API invocations. `Fordroid` completes the analysis of 100 randomly selected Android applications in about 64 h. Results demonstrate that more than 1/3 out of them leaked sensitive information to local storage. Moreover, `Fordroid` successfully locates where sensitive information is written to for 98% paths and identifies the structure of all database tables which contain sensitive data.

In the future, we plan to enhance `Fordroid` with stronger string analysis abilities and incorporate other techniques in order to reverse engineering file formats.

Acknowledgement

This work is supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada), Canada, and National Key Research and Development Plan (2017YFB0802900), Project 2117H14243A and Sichuan Province Research and Technology Supporting Plan, China.

References

Anglano, C., 2014. Forensic Analysis of Whatsapp Messenger on Android Smartphones, pp. 201–213.

Arzt, S., Rasthofer, S., Fritza, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proc. PLDI.

Calzavara, S., Grishchenko, I., Maffei, M., 2016. Horndroid: practical and sound static analysis of android applications by smt solving. In: Proc. EuroS&P.

Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y., 2015. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In: Proc. NDSS.

Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C., 2015. Information flow analysis of android applications in droidsafe. In: Proc. NDSS.

Hafner, J., 2016. While you track Pokémon, Pokémon Go tracks you. https://eu.usatoday.com/story/tech/nation-now/2016/07/11/while-you-track-pokmon-pokmon-go-tracks-you/86955092/.

Juan Caballero, C.K., Poosankam, P., Song, D., 2009. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proc. CCS.

Li, D., Lyu, Y., Wan, M., Halfond, W.G., 2015. String analysis for java and android applications. In: Proc. FSE.

Li, L., Bartel, A., Bissyande, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015. Iccta: detecting inter-component privacy leaks in android apps. In: Proc. ICSE.

Li, L., Bissyande, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, Y.L., 2017. Static analysis of android apps: a systematic literature review. Inf. Software Technol. 88.

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Traon, Y.L., 2013. Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In: Proc. USENIX Security.

Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P., 2015. Composite constant propagation: application to android inter-component communication analysis. In: Proc. ICSE.

Sali, K. The Cell Phone Evidence in Adnan Syeds Case Illustrates a Depressingly Common Problem. URL https://www.huffingtonpost.com/kevin-sali/the-cell-phone-evidence-in-adnan-syeds-case_b_9202422.html.

Scrivens, N., Lin, X., 2017. Android digital forensics: data, extraction and analysis. In: Proc. ACM Turing 50th Celebration Conference-China.

Wei, F., Roy, S., Ou, X., 2014. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proc. CCS.

Wikipedia, 2015. San Bernardino Attack. https://en.wikipedia.org/wiki/2015_San_Bernardino_attack.

Wikipedia, Taint checking. URL https://en.wikipedia.org/wiki/Taint_checking.

Wikipedia. URL https://en.wikipedia.org/wiki/Use-define_chain.

Yang, S., Yan, D., Wu, H., Wang, Y., Rountev, A., 2015. Static control-flow analysis of user-driven callbacks in android applications. In: Proc. ICSE.