A Dynamic Taint Analysis Tool for Android App Forensics

Zhen Xu, Chen Shi, Chris Chao-Chun Cheng, Neil Zhengqiang Gong, and Yong Guan
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, 50011
Email: {xuzhen41, cshi, cccheng, neilgong, guan}@iastate.edu

***Abstract*—The plethora of mobile apps introduce critical challenges to digital forensics practitioners, due to the diversity and the large number (millions) of mobile apps available to download from Google play, Apple store, as well as hundreds of other online app stores. Law enforcement investigators often find themselves in a situation that on the seized mobile phone devices, there are many popular and less-popular apps with interface of different languages and functionalities. Investigators would not be able to have sufficient expert-knowledge about every single app, sometimes nor even a very basic understanding about what possible evidentiary data could be discoverable from these mobile devices being investigated. Existing literature in digital forensic field showed that most such investigations still rely on the investigator's manual analysis using mobile forensic toolkits like Cellebrite and Encase. The problem with such manual approaches is that there is no guarantee on the completeness of such evidence discovery. Our goal is to develop an automated mobile app analysis tool to analyze an app and discover what types of and where forensic evidentiary data that app generate and store locally on the mobile device or remotely on external 3rd-party server(s). With the app analysis tool, we will build a database of mobile apps, and for each app, we will create a list of app-generated evidence in terms of data types, locations (and/or sequence of locations) and data format/syntax. The outcome from this research will help digital forensic practitioners to reduce the complexity of their case investigations and provide a better completeness guarantee of evidence discovery, thereby deliver timely and more complete investigative results, and eventually reduce backlogs at crime labs. In this paper, we will present the main technical approaches for us to implement a dynamic Taint analysis tool for Android apps forensics. With the tool, we have analyzed 2,100 real-world Android apps. For each app, our tool produces the list of evidentiary data (e.g., GPS locations, device ID, contacts, browsing history, and some user inputs) that the app could have collected and stored on the devices' local storage in the forms of file or SQLite database. We have evaluated our tool using both benchmark apps and real-world apps. Our results demonstrated that the initial success of our tool in accurately discovering the evidentiary data.**

I. Introduction

Mobile device forensics has been a challenging problem in digital forensics domain. With the thriving of the mobile device markets and usage among all the people over the world, mobile devices have been more and more critical and helpful in generating relevant and meaningful evidences to prove the innocence of people or existence of certain criminal activities in many civil and criminal investigations. Most such evidences are generated by the apps installed on the mobile devices (implicitly or explicitly). However, the plethora of mobile apps introduce critical challenges to digital forensics practitioners, due to the diversity and the large number (millions) of mobile apps available to download from Google play, Apple store, as well as hundreds of other online app stores. Law enforcement investigators often find themselves in a situation that on the seized mobile phone devices, there are many popular and less-popular apps with interface of different languages and functionalities. Investigators would not be able to have sufficient expert-knowledge about every single app, sometimes nor even a very basic understanding about what possible evidentiary data could be discoverable from these mobile devices being investigated.

With the mobile app market having been rapidly growing in the past decade, Android has become the largest mobile application platform in terms of the number of available apps. In this study, we focus on Android apps. We propose to develop an automated Android app analysis tool to analyze apps and discover what types of and where forensic evidentiary data that apps generate and store locally on the mobile device or remotely on external 3rd-party server(s). This tool will help an investigator to quickly locate and extract evidence from a device being investigated. For example, with more and more people using their devices via all kinds of apps for web browsing, driving navigation, paying bills, etc, it is apparent that apps will generate and store evidence about geo-location, time, contacts, browsing history, and many others on the device or remotely elsewhere. Existing literature in digital forensic field showed that most such investigations still rely on the investigator's manual analysis using mobile forensic toolkits like Cellebrite and Encase. The problem with such manual approaches is that there is no guarantee on the completeness of such evidence discovery. Also, current mobile device forensic investigation practice are often very time-consuming for forensic analysts to manually examine and pinpoint the evidence from the mobile device under investigation since it may have installed many different types of apps which contains massive amount of relevant or irrelevant information on them. The outcome from this research will help digital forensic practitioners to reduce the complexity of their case investigations and provide a better completeness guarantee of evidence discovery, thereby deliver timely and more complete investigative results, and eventually reduce backlogs at crime labs.

In the literature, there have been several known dynamic

app analysis tools such as TaintDroid [1] and TaintART [2], designed for identify privacy data leakage problems. After our careful analysis and experimental evaluation, none of them works for mobile device forensics purposes, for the following reasons: Firstly, these tools were designed mainly for privacy purpose. With them, one can detect whether there exists any private data leaks or detects whether there are any sensitive data being transferred from one to another app, but the tools are unable to identify and locate all the possible evidence, except those that are not considered to be private or sensitive. This is a fundamental limitation for them to be used for forensic purposed. Secondly, these tools only support a very limited number of taint tags. For instance, TaintDroid can only analyze up to 32 types of evidence, while TaintART can analyze only 5 types of evidence, due to their special design for taint management and propagation. Thirdly, TaintDroid can only work with Dalvik Virtual Machine, which was outdated. Right now, TaintDroid does not work with the current Android Runtime platform (ART), which translates an Android app into binary code. As a result, TaintDroid is unable to analyze most current Android apps being developed for the current Android devices.

In this paper, we will present the main technical approaches for us to implement a dynamic Taint analysis tool for Android apps forensics. In our implementation, we modify the Android ART platform. Specifically, we use a 32-bit taint tag to represent type of evidence, which allows us to follow 2^{32} types of evidence possibly generated by an Android app. For each variable (possible evidence) in the app, we treat all of its taint tags as a set, and dynamically update the set according to the data flow processed by the app. Using a set to store all possible taint tags for a given variable allows us to analyze a large number of fine-grained evidences. To do so, there is a key challenge in deciding how the taint tags are stored and processed. Existing tools, for example, TaintDroid stored the taint tag next to the variable on the app's stack-frame, while TaintART stores the taint tag in a register of the mobile device. None of these methods works for our purpose, for the following reasons: First, the number of registers on a mobile device is too small to store the large set of taint tags for our purpose; Second, the set can be dynamically updated, which makes it even harder to store it next to a variable on the app's stack space. Thus, in our design, we store the set of taint tags for each variable in the app's heap space.

The innovative design and implementation of our tool can overcome the limitations of existing dynamic taint analysis approaches. Specifically, our tool can support a very fine-grained analysis of evidence, e.g., 2^{32} types of evidence. We have implemented the tool with the special design. With the prototyped tool, we have analyzed 2,100 real-world Android apps. For each app, our tool produces the list of evidentiary data (e.g., GPS locations, device ID, contacts, browsing history, and some user inputs) that the app could have collected and stored on the devices' local storage in the forms of file or SQLite database. We have evaluated our tool using both benchmark apps and real-world apps. Our results demonstrated that the initial success of our tool in accurately discovering the evidentiary data.

A. Organization of the Manuscript

We have organized the rest of this paper in the following way. We will provide a literature survey in Section II. We then proceed to the motivation and technical overview of our designs in Section III. This section gives an overview of design architecture and points out some issues to be solved in the current research. Section IV provides an in-depth description of the implementation details. Experiments regarding the effectiveness of the prototyped tool in analyzing Android apps are presented in Section V. Finally, conclusions are collected in Section VI.

II. Related Work

In this section, we will review the mobile device forensics efforts as well as some relevant work that attempted to solve a similar (but different) problem - app private data leakage.

A. Mobile Device Storage Forensics

Permanent-storage forensics for Android is still an under-developed research area. Most existing studies and tools on this topic simply leverage either manual analysis or keyword search. As a result, they can only analyze a small number of apps or construct an inaccurate App Evidence Database (AED). Our work represents the first one to perform large-scale automated permanent-storage forensic analysis for Android Apps (via program analysis).

a) Manual analysis: Some studies [3]–[6] manually analyzed apps in order to construct an AED. Specifically, they install apps on an Android device or run the apps in a sandbox environment (e.g., Android Emulator [7] and YouWave [8]). Then, they retrieve an image of the file system from the device or the sandbox environment. Specifically, the file system image can be retrieved from a device using the Android Debug Bridge. The file system image can be either logical or physical, where a physical image could also include the deleted files that are not overwritten yet. By running apps under a sandbox environment, researchers have control over the file system and main memory, so they can also retrieve images of the RAM and NAND flash memories.

After obtaining a file system image, they manually examine the files generated by the apps, e.g., analyzing the files under a directory $/data/data/ < package\ name > /files/$, where $< package\ name >$ refers to the package name of the app to be analyzed.

Since manual analysis is time-consuming, error-prone, and costly, these studies often only analyzed a very small number of apps. In particular, they often focused on instant messaging apps (e.g., WhatsApp [5], WeChat [6]) and maps navigation apps [9]. They found that instant messaging apps often save messages (i.e., a certain type of *text input*) on the local file system, while maps navigation apps collect GPS location history and post on databases. Interestingly, these studies concluded that SQLite database is the major sink of evidentiary data.

However, our results on a large amount of real-world apps indicate that SQLite database is the major sink only for visited URL. SharedPreferences is the major sink for location and time, while text file is the major sink for text input. The reason for such discrepancy is that these studies only analyzed a very small number of apps, and their statistical results are not representative.

b) Keyword search: Several commercial tools (e.g., Cellebrite UFED [10], XRY [11], and FTK [12]) are available to analyze the files on a device. Specifically, given a device, these tools first retrieve an image of the device's file system. Then, these tools provide Graphical User Interface (GUI) for forensic investigators to search files that could contain evidentiary data. However, the search is only performed by keyword matching or regular expression matching, which clearly has limitations. For instance, if a file contains GPS data but does not have the regular expressions or keywords such as GPS, latitude, or longitude, then the file will be incorrectly labeled as a file that does not contain evidentiary data. Indeed, studies [13] showed that tools based on keyword matching can only identify a small fraction of files that could store evidentiary data.

B. Dynamic Program Analysis

In this subsection, we will review some relevant work that attempted to solve a similar problem - app private data leakage. Due to that the goals set forth for these works are different from that in our research, none of these tools are directly applicable to solve our problem. But there are some similarities between these work and ours in term of technical approaches. In the following, we will provide a brief review on each of these works.

TaintDroid [1]: TaintDroid is a system-wide taint tracking tool based on Android 4.2. It aimed to monitor the apps running on the for sensitive information leakage and to minimize runtime overhead that is the amount of additional instructions needed for implementing of tracking mechanism. However, TatindDroid does not distinguish between files but label all files with one label. It can only support up to 32 different tags. TaintDroid can only work with Dalvik Virtual Machine, which was outdated. Right now, TaintDroid does not work with the current Android Runtime platform (ART).

TaintART [2]: TaintART shared similar goals with TaintDroid in that both focus on private information leakage and try to minimize runtime overhead. Unlike TaintDroid, TaintART built upon Android version 7.0 which is one of latest and most popular platform in the mobile app ecosystem. The implementation of TaintART altered the compiler to implement the dynamic taint analysis of apps. One feature of TaintART is that it uses processor registers to store the taint tags. The taint tag is 1 or 2 bits wide and represents the taint status of other processor registers. During the app compilation, TaintART reserves register 5 and treats it as the taint storage space. The register 12 is used for temporary taint propagation space. One limitation of TaintART is that it can distinguish only five types of private data. This is insufficient for the forensic purposes, which has to cover many more types of data (as possible evidence) on mobile devices.

ARTist [14]: ARTist is a compiler-based instrumentation tool. Android uses Dex2oat as compiler which takes an Android application APK as input and converts it into an oat file. The oat file contains both the byte code and binary code that is ready for execution. The dex2oat operates on objects called HGraph, which represent methods in an app. ARTist integrates its instrumentation module with the compiler and thus allows compiler-based instrumentation to track private data leakage.

III. Technical Overview

A. Overall Framework

Our proposed dynamic taint analysis tool for Android app forensics is based on a known concept - Taint Analysis. Different from traditional static program analysis methodology, it is a dynamic program analysis based on app's runtime behavior. Taint analysis itself is a program analysis approach which tries to add label (or tag. From now on, we use label and tag exchangeably.) to the variables within the application under investigation. Our app analysis tool is actually a runtime platform that allows the execution of the apps while tracking the data flows inside the app in its actual or emulated usage. During the execution of a given app, variables store data (likely useful evidence), which are passed around via the instructions of the app (like assignment in a programming language). The goal of the taint analysis is to provide a way of tracking where and how the evidentiary data of interest flows between method calls within that app such that at the end, the source (type of data) and flow (i.e., the way of its being processed within that app) of which can be revealed. Such information will be used for us to identify the type of evidence, where it went (store locally or remote elsewhere), and the syntax/format of the evidence data.

Figure 1 describes the framework for our dynamic App analysis platform. To analyze an app, we first download and install the app APK code onto either a real Android phone device or an Android emulator. We can use a modified MoneyRunner to generate touch-screen events to emulate the actual use of the app. Our modified Android ART platform taints the variables (possible evidentiary data) generated by the app and keeps track of how they are processed within the app before finally storing them into a local file or SQLite database on the phone device. Our analysis engine will use the tainted data as well as other event-generated data (via ADB Logcat) to generate the list of of evidence data (types, destination paths/SQLite DB tables, syntax/formats) as output, which will in turn written into our app database.

B. Major Concepts and Terms

a. *Taint tag*: Taint tag or tag refers to the type of information which travels with variables of interest. If a variable is tainted or has certain taint tag, then it means the variable carries certain type(s) of evidential data, e.g., GPS locations or date and time.

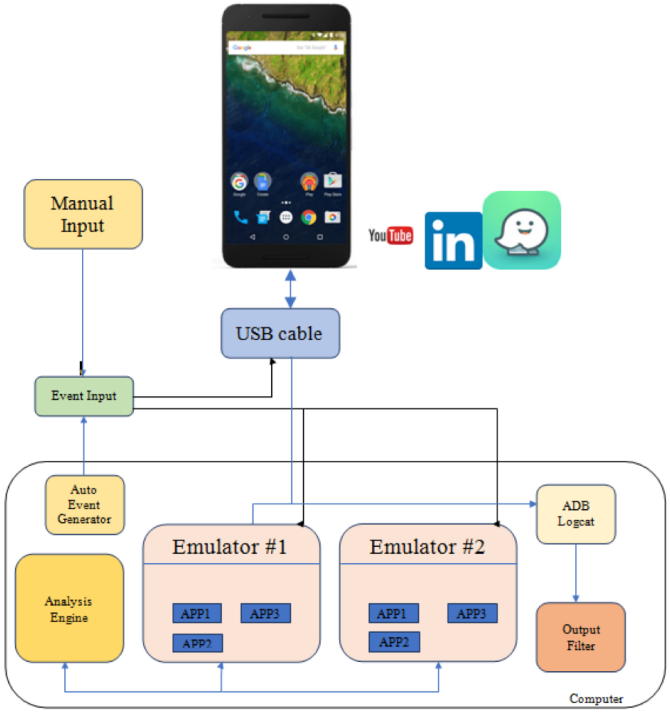

Fig. 1: Android App Analysis Framework

b. *Source*: Source refers to the starting point where a variable gets tainted. Typically, a system method can be treated as a source of variable (to represent the type of data generated by the app, and start to be processed within the app). Whenever the return value is assigned to a variable, we say that the variable become tainted.

c. *Sink*: Sink refers to the destination where the evidence data of interest went to. Methods like file write API call can be treated as a sink. Whenever a data (variable) is sent to such file write methods, the taint tag of that data will be reported as well as where it is stored (maybe a path name or SQLite database table).

d. *Evidential data*: Not all data deserves attention or could be valuable evidence to certain case work, because many of them are just temporary data or simply of no exact meaning. But some data like geo-location, device ID, browsing history, some user input, or temporary video or image objects left by the apps, for example, Snapchat, Twitter, Facebook, or Wechat. We try to keep track of most possible evidence data.

e. *Taint propagation*: Variables inside an app are often processed in different ways, for example, reassignment or appended to a string object, and so on. In such process, along with that data (variable) flows through method calls within an app, the taint tag associated with these variables should be maintained and processed appropriately such that we can follow how such data are processed inside an app. Whenever a variable is assigned to another variable, both variables should have the same taint tag. Take an addition $varA = varB + varC$ for example, the variable $varA$ should have the taint tags from both $varB$ and $varC$.

f. *Taint tag storage*: The taint tag requires some additional memory space, which is referred as taint tag storage. Depending on the total number of taint tags being maintained, the space requirement can be large. For mobile app forensics purpose, the number of possible types of evidence can be large. The approaches used by TaintDroid and TaintART would not be large enough, so would not work for our tool. In the following section, we will discuss it in details about how we do it.

g. *Propagation rules*: Propagation rules define the ways of how the taint tags shall flow and update when an instruction of the app gets executed. For example, binary addition operation should assign the resulted variable with a new taint tag, which is a union set of both operands' tags. For other operations like array element access, taint tags should be appropriate in that either an individual tag associated with the element or the tag for the whole array object, depending on the situation and the variable of interest.

IV. Implementation Details

Figure 2 shows a simplified procedure for Android runtime platform (ART) when an application gets started to run. Upon launching the application, a new process on ART is created that runs that application. The ART platform first checks if it is necessary to execute native code either because of JNI or compiled code available under normal condition. If not, it will call interpreter which loops over Dalvik bytecode and executes instructions of the app accordingly. A special situation is that if the method belongs to a core library like object class, the system will execute native code compiled

from C++ code during OS Compilation. These codes mirror the behavior of their java counterpart and is intended for better performance. In our implementation, we modify the ART platform such that we always force the Android system enter interpreter mode and bypass the checking of trusted mirror class. Doing so will guarantee that all the java-based code are executed through the interpreter. Furthermore, our modification of Android ART platform alters the interpreter instruction case handling procedure such that our own taint propagation rules can be implemented in the platform, which enable us to analyze the Android apps on our platform, as that showed in Figure 1

We have implemented two modes of operation: Bit-wise mode and Tag-id mode in our App analysis platform. Our system operations under bit-wise mode are similar to that in TaintDroid, but with the major difference that we support Android OS (7.0 or newer). Our work under bit-wise mode focuses on improved app analysis efficiency. Our Tag-id mode operations are completely different from the existing tools that (1) We define a tag to be a 4-byte long variable, which each of its 32 bits represents if a specific type of data (possibly considered to be an evidence) is carried or not. With it, our tag-id mode operations allows us to have a higher capability of distinguishing much larger types of evidence data. (2) Our system maintains a global mapping between id and tag-set. The tag in tag-set ranges from 0 to $2^{32}-1$. In comparison with Artist [14] our system under tag-id mode assigns each field in the app code a unique id (identifier) during runtime analysis, and can support up to 2^{32} types of tags. Another unique difference between the two modes of operations are in the way the tag being stored as well as how the tag union operation is carried. Under bit-wise mode, bitwise OR operation is used for both tags that are of 4-byte long size. Under tag-id mode, our system performs a set-union operation on the two tag sets.

A. Storing Taint tag on Stack of the App Address Space

Whenever a method within the app is invoked, our system will allocate a chunk of memory space for the possible taint tags on the App's stack space in the physical memory. In the App source code, the method frame is represented by the class *ShadowFrame*. The *ShadowFrame* mimics a conventional method frame used in other architecture like ARM. It has a 4-byte integer array, return value, and reference to ART method. The array acts as virtual registers and holds the values used by the instructions. For example, instruction like $v1 = v2 + v3$ refers to the addition of virtual register 2 and 3 and the result is saved into virtual register 1. Our implementation under bit-wise mode introduces an array of 4-byte integer so that each of the 32 virtual registers has a single bit (1 or 0) in the corresponding 4-byte long tag. In the meanwhile, our implementation under tag-id mode adds a vector of integer set whose content are saved on the App's stack and heap space. Figure 3 illustrates the implementation of tag-id mode on stack and heap space of the app's address space on the Android device. Under the bit-wise mode, the tag sets are replaced with the 32-bit long variable, each bit of which represents whether the data in the virtual register is tainted or not. Our implementation only need to use the stack space, similar to that in TaintDroid. Also, our implementation adds a special field for the tag space for return value(s).

B. Class Modification for Memory Alignment

When a class is loaded into the system, a memory layout is generated by the Android ART platform. Figure 4 shows an example of how a class $class\ A$ gets translated into a memory layout. The class A contains some number of variables of different sizes and types. The memory layout does not follow the same order as the one in the source code, instead follows the order given in Figure 5. The memory layout starts with the superclass and then continues with the fields in the descending-size order. Our system implementation adds the taint tags immediately after the 4-byte fields in the address space of the app on the Android device. Contrary to what we have done, TaintDroid added a tag next to every variable in the stack space. Such a choice is made due to the problem of memory alignment and memory gap. By default, an X-byte field should have an address which is divisible by the integer X. For example, a 4-byte field can have an address like $0b11111100$ but cannot have $0b11111101$, which is required by the CPU design (as a general rule). Therefore, it is best to follow this rule. However, if the rule is followed, memory gaps could be created when a taint tag is added next to a field. We can take a one-byte field as an example. The one-byte field could have an address of $0b0001$ and its taint tag has an address of $0b0100$. Such a treatment is consistent with the memory alignment, but as a result, it leaves a gap of 3-byte created between the one-byte field and its taint tag. This is not a desirable design, because of the unnecessary wasted memory space. Therefore, in our implementation, we decide to put taint tags immediately after 4-byte fields as shown in Figure 6.

C. Storing Taint tag on Heap of the App Address Space

Heap stores instances of each class. According to the modified memory layout from the previous section, the taint tags or tag-ids are added immediately after the 4-byte variable during

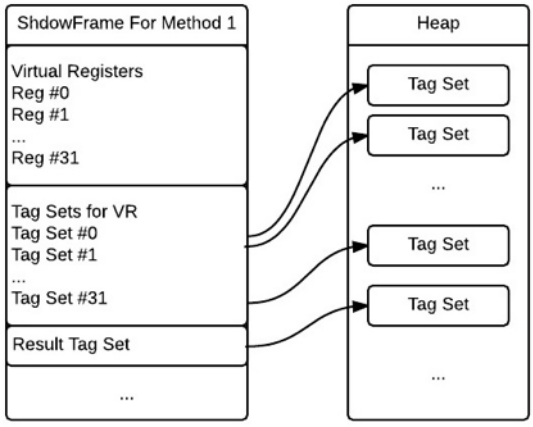

Fig. 3: Taint Storage on App's Stack and Heap Space

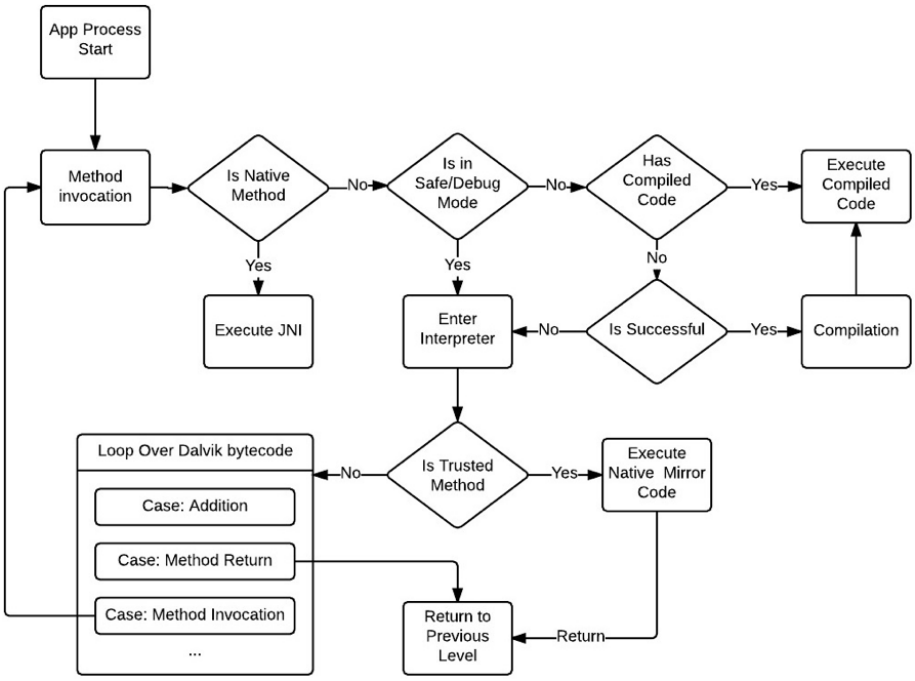

Fig. 2: A Simplified Android ART procedure

the app loading procedure. This is implemented via the following modification: (1) A method called ComputeClassSize calculates the total size an object needs. Our implementation modified this method to allow the memory expansion of an object. The additional memory space is treated as memory gap, instead of the fields from the perspective of normal execution. (2) Another method called LinkFields enumerates fields and calculates the offset of each field related to the starting point of the object. Our implementation modified this method to allow the calculation of the offset for the taint tags. A new field called *taint offset* is added to Android ART. This field stores the taint tag offset which will be used during runtime analysis in order to find the location of taint storage for a given field.

D. Taint Propagation Rules

In our implementation, we have defined four types of tain tag propagations: Local, Stack-heap, Inter-method, and memory-file propagations. Table 7 is the summary for the first two kinds.

1) The local propagation happens if a virtual register is assigned with a value. Take the algebra addition operation for example, the result register gets the taint tags from both operands. This propagation follows the propagation rules which can found in Table 7 similar to that used in TaintDroid. Our implementation modified the instruction handling cases in interpreter common.cc. For each instruction case, we have added the implementation for taint propagation procedure.
2) The stack-heap propagation happens for field-related instructions like *instance-get* and *instance-set*. Java object and its fields are all stored on Heap of the app's address space. And the data must be loaded onto stack before being processed. Our system first finds the *ArtField* instance of the field of interest. From the *ArtField*, the taint offset can be obtained. Our system then accesses the tainted data using the offset relevant to the start address of the object.
3) The Inter-method propagation happens when there is a method invocation or return. For a method invocation, our implementation modifies the argument setup procedure so that the taint tags can travel with the arguments of interest. For a method return, the current method (stack) frame will be popped and destroyed. Before the method eventually returns, the taint tag of the return value will be put in a data holder in the caller method stack frame. The instruction $move - result$ moves the return value to a virtual register, and meanwhile, our implementation also moves the taint tag accordingly.
4) The memory-file propagation happens when there is a file writing or reading. The current ART implementation only handles file to memory propagation. When a variable gets assigned value from a file, the variable also will be assigned a special taint tag that represents that file (as a type of data source).

E. Source and Sink Methods

Source methods called by the app can be used to infer the type(s) of evidence data while sink methods can be used to infer the where the evidence data goes to. Thereby, the linkage of source and methods via the data flow path inside the App can be used to generate critical information about the evidence data in terms of evidence type(s), storage location,

Operations		Propagation Rules	
		Bit-wise mode	Tag-id mode
Unary Operation: $v_1 = v_2$ op: move, move-return, …		$T(v_1) \leftarrow T(v_2)$	$T(v_1) \leftarrow T(v_2)$
Binary Operation: $v_1 = v_2$ op v_3 op: add, shift, xor, …		$T(v_1) \leftarrow T(v_2)$ or $T(v_3)$	$T(v_1) \leftarrow T(v_2)$ union $T(v_3)$
Array	aget v_1, v_2, v_3 $v_1 = v_2 [v_3]$	$T(v_1) \leftarrow T(v_2 [v_3])$ or $T(v_2)$	$T(v_1) \leftarrow T(v_2 [v3])$ union $T(v_2)$
	aput v_1, v_2, v_3 $v_2 [v_3] = v_1$	$T(v_2) \leftarrow T(v_2)$ or $T(v_3)$	$T(v_2) \leftarrow T(v_2)$ union $T(v_3)$
Static Field	sget v_1, f $v_1 = f$	$T(v_1) \leftarrow T(f)$	$T(v_1) \leftarrow T(f)$
	sput v_1, f f = v_1	$T(f) \leftarrow T(v_1)$	$T(f) \leftarrow T(v_1)$
Instance Field	iget v_1, v_2, f $v_1 = v_2$.f	$T(v_1) \leftarrow T(v_2.f)$	$T(v1) \leftarrow T(v_2.f)$
	iput v_1, v_2, f v_2.f = v_1	$T(v_2.f) \leftarrow T(v_1)$	$T(v_2.f) \leftarrow T(v_1)$

Fig. 7: Taint Propagation Rules

and evidence data syntax/formats. In our current implementation, we have included 11 sources, including the frequently evidence data types such as telephone, device ID, text input, GPS locations, visited URLs, various account information, and so on. Each of the sources is assigned with a unique taint tag. During the execution of the app, whenever a file is opened, our implementation will assign a tag for that file. These taint tags are used to distinguish different files and global mappings between tags. One or more file path(s) are maintained for reporting the storage location of the evidence data at some sink methods. Generally, the sink often includes the $LoggingPrintStream.println$ and file write operations.

V. Experimental Evaluation

We have conducted three types of experiments to evaluate our developed Android App analysis tool in this research: (1) Benchmark APK testing, (2)Real-world APK with manual

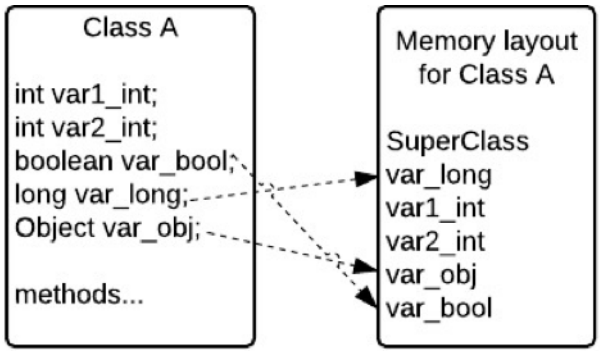

Fig. 4: Memory layout for Class A

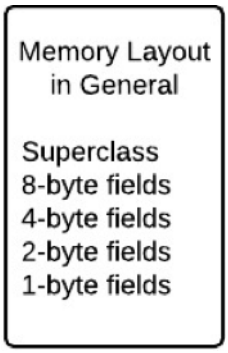

Fig. 5: General Memory Layout

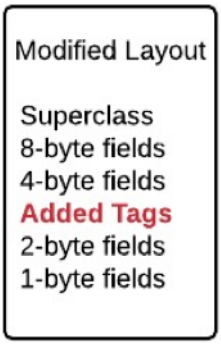

Fig. 6: Modified Memory Layout

test, (3)PlayDrone APK pool with auto test. Our system was installed on a Nexus 6P phone and has been tested under tag-set mode. Figure 8 and Figure 9 show the screenshots of our Android mobile device running the benchmark APKs and the PC-end analysis engine that monitors the device.

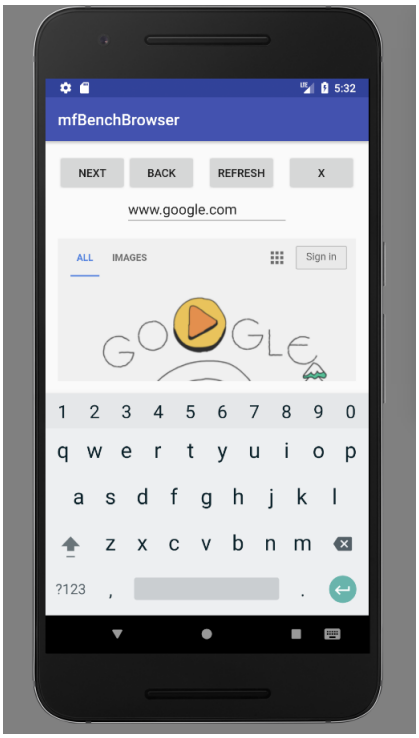

Fig. 8: Screenshot of Mobile Device Running Benchmark APK

c) Benchmark APKs: : Our system was tested with a set of testing apps (namely Benchmark Apps) in order to verify the effectiveness of our system. For each benchmark app, similar to real-world apps, some evidence data such as contacts or location data were assigned to a variable, and following that, the variable has been experienced with a sequence of simple manipulations and eventually re-assigned to another variable. Finally, the second variable is sent to a certain sink method for file write operations including system.out and file

Fig. 9: Screenshot of PC-end Analysis Engine

Fig. 10: Sample Output of Logcat

write method. Our experiment have demonstrated that the data flows containing the variables of interest have been detected, which showed that our system does work as expected. We have compared the testing results of Benchmark APKs to evaluate the accuracy of our dynamic App analysis tool. The benchmark evaluation results shown in Table I demonstrated that our proposed approach has achieved a good accuracy and coverage on the evidence data discovery from Apps.

d) Manual testing: : We have used forty-five real world applications to test with manual inputs. The 45 tested apps included Facebook, Line, Taobao [15], Sideline [16], Alipay [17], etc. We realized that our tool has a limitation that some apps cannot be run properly due to the absence of Google play service on our customized ROM. For all the applications that do not rely on the runtime support from Google Play service, our tool has detected a total of 15,000 data flows. Although there were some duplicates, most of these data flows actually happened among temporary files. Evidence data that have been detected from the experiments are shown in Table II.

One interesting case about Wechat App [18] showed that our tool has detected data flow from some configuration files to a png file, which demonstrated that it is possible to recover meaningful evidence from the image png file, though the raw evidence data may have been removed at the time of investigation. Such data flow was also evidenced by a case study showed in Figure V-0d. The results in Figure V-0d showed that there was a configuration file (labeled as 292) opened by the app. Some data from this file was latter written to a image file (with tag 306). These case studies are the very examples that apps embed potential critical evidence data into a file (like image) that is seemly irrelevant to some case investigations, thereby traditional mobile device forensic methods cannot detect.

Another interesting case about Sideline App showed that some data from Preference and Settings (configuration files) were written to a mp3 file. Both cases demonstrated the capability and usefulness of our tool. However, what exactly have been written needs further work (as our next step).

e) Application pool with auto testing:: We have written a testing script for the purpose of auto-testing with a large set of apps we have collected. As the first step of large scale experimental evaluation, we have randomly chosen 2,100 apps from over 1.1M apps in our database from PlayDrone [19]. We have created a script program that modified Monkey [20], a popular auto testing tool. Our modified Monkey script is capable of firing a large amount of events continuously to test the app. For each application, we have run a total of 5,000 random events fired with a gap of 50 milliseconds between two events. The events we fired included clicking, sliding, volume up & down, screen capture, etc. Our tool has detected nearly 1,500 data flows. For example, the FrameBlue app writed data

from a zip file to a png file.

Our experimental evaluation results showed that auto-testing approach in general achieved only 10% of what is given under manual test, given a short time period testing (mostly 5 minutes in our experiments). We analyzed the results and had the following observations: (1) The fired events may not be sufficient (compared to manual test) given the short time analysis. Some buttons or one of the execution path behind hood require a certain sequence to trigger. For example, in order to login, username and password must not be empty. As a next step, we should carefully evaluate the effectiveness of the auto-testing script to provide a stronger guarantee to fire more effective and meaningful events to achieve a better coverage about the app's data paths. (2) Applications that require login (such as Skype, Wechat, Facebook) oftentimes require valid user credentials in order to proceed. During manual testing, the tester registered an account so that the testing can proceed. (3) Among the Apps available for download from Google Play, there are still a relatively large number of APKs that cannot run properly during the auto testing, due to the fact that some apps downloaded from PlayDrone cannot run on Android 7.0+.

VI. Summary and Future Works

We have prototyped a dynamic taint analysis tool for Android app analysis. We have conducted a large scale experimental evaluation using both Benchmark Apps and real-world apps. The initial success in using the tool for forensic analysis on apps' generated evidence data on seized Android devices have attracted law enforcement and information security practitioners as well as IT industry that use apps for a variety of services. The built tool can not only help mobile device forensic investigations but also help to solve private data leakage and app correctness testing purposes.

For the future work, we are planning to address the following issues to further improve the tool we built:

- Native Code: Our implementation of the tool modified the ART Interpreter, and therefore ignored the native code. This is a common problem for the analysis tool for java-based program. For the native method in system API, there could be manual modification of the source code, which possibly helps to evade the detection of possible evidence data. For some native code from a third party like that identified in NDroid [8], there will be a need for effective solution to handle native codes.
- Implicit Data Flow: Implicit data flow is also a common problem for the app analysis work. Code like $if(varA == 1)varB = 1$; assigns $varB$ with value 1 if the $varA$ equals 1. In such cases, the variable $varB$ effectively gets the value from var, but such implicit data flow poses challenges in handling the taint tags of $varA$ to $varB$. The App of "Da vinci secret message" uses implicit data flow to embed a message from a user with a bitmap of an image. However, with such implicit flow, there will be a need for more effective methods to detect implicit data flow when such combined data are written into a file.
- Event Sequence: Without appropriate event inputs, application cannot be explored/analyzed properly, therefore there will be a need to develop a more effective event firing solution. Manual testing can generate reasonably good results, but the scalability is an issue (only feasible for a small scale app analysis). For larger scale experiments, there will be a need for a better auto testing solutions.

References

[1] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2619091

[2] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 331–342. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978343

[3] M. I. Al-Saleh and Y. A. Forihat, "Skype forensics in android devices," *International Journal of Computer Applications*, vol. 78, no. 7, 2013.

[4] T. Alyahya and F. Kausar, "Snapchat analysis to discover digital forensic artifacts on android smartphone," *Procedia Computer Science*, vol. 109, pp. 1035–1040, 2017.

[5] C. Anglano, "Forensic analysis of whatsapp messenger on android smartphones," *Digital Investigation*, vol. 11, no. 3, pp. 201–213, 2014.

[6] S. Wu, Y. Zhang, X. Wang, X. Xiong, and L. Du, "Forensic analysis of wechat on android smartphones," *Digital Investigation*, vol. 21, pp. 3–10, 2017.

[7] "Run apps on the android emulator," 2018. [Online]. Available: https://developer.android.com/studio/run/emulator.html

[8] "Youwave," 2018. [Online]. Available: https://developer.android.com/studio/run/emulator.html

[9] S. Maus, H. Höfken, and M. Schuba, "Forensic analysis of geodata in android smartphones," in *International Conference on Cybercrime, Security and Digital Forensics, http://www. schuba. fh-aachen. de/papers/11-cyberforensics. pdf*, 2011.

[10] "Cellebrite ufed ultimate," 2018. [Online]. Available: https://www.cellebrite.com/en/products/ufed-ultimate/

[11] "Xry - extract," 2018. [Online]. Available: https://www.msab.com/products/xry/

[12] "Forensic toolkit (ftk)," 2018. [Online]. Available: https://accessdata.com/products-services/forensic-toolkit-ftk

[13] H. Henseler, "Finding digital evidence in mobile devices," in *DFRWS*, 2017.

[14] M. Backes, S. Bugiel, O. Schranz, P. v. Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, April 2017, pp. 481–495.

[15] "Taobao." Mobile application software, 2017 Version 7.5.2. [Online]. Available: https://play.google.com/store/apps/details?id=com.taobao.taobao

[16] "Sideline." Mobile application software, 2017. [Online]. Available: https://play.google.com/store/apps/details?id=com.sideline.phone.number

[17] "Alipay." Mobile application software, 2017 Version 10.1.15.463. [Online]. Available: https://play.google.com/store/apps/details?id=com.eg.android.AlipayGphone

[18] "Wechat." Mobile application software, 2017. [Online]. Available: https://play.google.com/store/apps/details?id=com.tencent.mm

[19] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 221–233.

[20] "Ui/application exerciser monkey." [Online]. Available: http://developer.android.com/tools/help/ monkey.html

TABLE I: Accuracy Results of Benchmakr App Analysis

Benchmark App	Number of Evidence	Evidence detected by Dynamic Taint Analysis	Accuracy
mfBenchBrowser	6	4	67%
mfBenchGPS	4	2	50%
mfBenchChat	3	Error message (cannot run due the lack of google play)	N/A

TABLE II: Testing Result on Popular Apps

App	Type	Destination	File Format
Alipay	Media Data	/data/user/0/com.eg.android.AlipayGphone/app_plugins/multimedia-live@1.0.0.170802165137.jar	MIDI
Pinterest	Image Files	/data/user/0/com.tencent.mm/files/public/fts/res/temp/dist/f9011840880fec1d81e7db1572cca9ef.png	fts_template.zip
Sideline	GMS Preference	/storage/emulated/0/Notifications/Pinger.mp3	com.google.android.gms.measurement.prefs.xml
	Phone Number	/storage/emulated/0/Ringtones/Sideline.mp3	com.pinger.textfree.settings.xml
Wechat	Configuration	/data/user/0/com.tencent.mm/files/public/emoji/newemoji/smiley_17b.png	systemInfo.cfg
Taobao	Web Interface	/data/user/0/com.taobao.taobao/files/wvapp/apps/newuserpop/0.0.1/main.html	main.js