MirrorDroid: A Framework to Detect Sensitive Information Leakage in Android by Duplicate Program Execution

Sarker T. Ahmed Rumee, Donggang Liu (Deceased) and Yu Lei
Department of Computer Science and Engineering
University of Texas at Arlington, Texas, USA
sarker.ahmedrumee@mavs.uta.edu, donggang.liu@yahoo.com, ylei@cse@uta.edu

***Abstract*—Android applications can leak user's private information and now a days is a major concern. This paper presents *MirrorDroid*, a novel dynamic analysis based test framework to detect the leakage of private data by Android applications. Our method modifies the Dalvik Virtual Machine to follow execution of the target application with a parallel (mirror) version, which only differs in the sensitive input data. If all other inputs are kept equal, a variance between program output and its parallel version corresponds to an event of sensitive data disclosure. MirrorDroid was evaluated on an Android malware data set and 50 popular applications from the official Android Application Store. The result shows that the proposed system can detect leakage of sensitive data effectively and efficiently.**

I. Introduction

According to a recent survey [4], Android has the largest market share, which is now approximately 82.8%. Due to its popularity and open source nature, Android has become the prime target of malware developers to launch a variety of attacks. Among the dangers of malicious android applications, leakage of user's private information is one of the major concern [10].

Privacy protection of smartphone data is well studied in the literature. Existing solutions can be broadly classified to static analysis and dynamic analysis techniques. Static analysis methods scan byte code or source code to find paths that may leak information. However, they do not execute the program and lacks program dynamic information. As a result, suffer from high false positive rate. Dynamic analysis techniques rely on inserting various monitors in the program or operating system level to track an application while it is running and do not suffer from most of the problems of static analysis methods. But state of the art the dynamic analysis techniques have limitations also such as taint explosion, high runtime overhead [9], application breakage due to mocking of sensitive data [7] etc.

To solve the above mentioned problems, this paper presents MirrorDroid, a novel framework that dynamically detects information leakage on Android. It depends on the following idea - if we can execute a program deterministically, then multiple executions of the program should always produce the same outputs if inputs are not changed. With this in mind, to monitor an Android application for information leakage, MirrorDroid executes it twice with identical inputs. But for sensitive inputs (such as IMEI number, Contacts etc.), the duplicate(mirror) execution is fed slightly modified information (e.g., actual input value with some added noise). As a result, if there is any difference in the outputs of the two executions, we can attribute this difference to the changes in input. It is because, we ensure that the changed sensitive input is the only difference between the two executions and all other inputs and program condition remain identical. This is achieved by running the mirror execution in parallel with the actual program execution, rather than running it separately after the original execution is completed. The details of this parallel execution approach is described in detail in the remainder of this paper.

MirrorDroid was evaluated using two data sets. The first one includes applications that are known to contain malicious code leaking information; the second data set includes popular applications from the official Android Application Store. The experiment results show that we can successfully detect the information leakage from the applications in both data sets.

The rest of the paper is organized as follows. Section II introduces some essential terminologies and a brief background on the Android. Section III describes the high level overview of the proposed approach. Next, section IV provides detailed description of implementation. Experimental results and performance measure of the proposed system are discussed in section V. Related research to tackle this problem are briefly discussed in section VI. The last section concludes the paper and discusses some possible future directions.

II. Background and Assumptions

A. Definitions

Following are the few terms used repeatedly in rest of the paper:

- *Sensitive Data* : Data considered private by the user.
- *Source* : A program input or API which returns sensitive data
- *Sink* : An output API through which data can be sent outside the mobile device.
- *Mirror Execution* : The duplicate execution of the target Android application which runs in parallel with the original execution.
- *Mirror Variable* : The corresponding duplicate variable of a program variable in mirror execution.

978-1-5090-4780-2/17/$31.00 ©2017 IEEE

- *Non Deterministic Input* : Source of data (API) whose value can be different in multiple execution without any change in any program input or environment.

B. Dalvik Virtual Machine

Android is a linux based mobile operating system. Applications are written in Java and compiled to a custom byte-code known as the Dalvik Executable (DEX) format. However, in contrast to traditional Java Virtual Machine(JVM), it runs each application within a separate instance of the VM (called Dalvik VM). It gives a much greater level of security as applications cannot interact with outside world directly.

The core of Dalvik VM is the Interpreter, which follows the traditional java like fetch, decode and execute cycle for each instruction in an application. However, starting from Android 5.0, applications are no longer interpreted by the virtual machine. Rather they are compiled to binary at install time and the new Android Runtime (ART) directly executes the binary code. But nothing changes for application developers. The proposed system is a testing framework and not meant to replace the user side Android Operating System. So, we can safely use the Dalvik based interpreter without sacrificing correctness and usability.

C. Android Permission Scheme

Access to sensitive information sources are controlled by means of a set of permissions, which an application acquires during install time. Even with the improved permission scheme introduced by Android 6.0, the permission based scheme to protect sensitive information is not enough. It is found that, users pay little attention to these permission requests and unnecessary permissions are often granted to applications making them over privileged [11].

D. Threat Model

In this paper, we assume that the attacker fully controls the development of the application under test. In other words, he can embed any code he wants. We also assume that except the application under test, all other system components, e.g, the Android OS and the Dalvik VM, are secure. If they are compromised, then the adversary can bypass our system.

III. Approach Overview

Generally, when a program executes multiple times, if the inputs remain same, the outputs are also expected to be same. Given a deterministic function $f(x)$ ($x \in \mathcal{X}$), the result of this function does not carry information about x if for all x_1 and x_2 we have $f(x_1) = f(x_2)$. In other words, given $f(x)$, the probability distribution of x is a uniform distribution over $\mathcal{X}$. As a result, if we test $f(\cdot)$ using two inputs $\{x_1, x_2\}$ and find that $f(x_1) \neq f(x_2)$, then it is for sure that function $f(x)$ is leaking some information about the input x. Figure. 1 depicts a high level description of our system based on this approach.

It shows an example scenario of an application consisting a number of instructions. In Mirror executions, each instruction is duplicated and executed on the corresponding duplicate (mirror) variables maintained in parallel with the original program variables. Among the instructions, two types are of particular interest (shown in detail on the right) and are handled differently. One is used to access sensitive data (here using source API $deviceId()$) and another instruction sends data to outside (here using the sink API $send\_message()$.

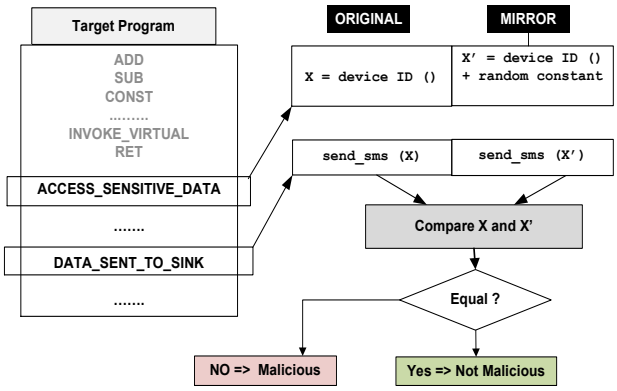

Figure 1. MirrorDroid Approach

The return value (X) of source API is mutated using a random constant in the mirror execution (mirror version X' becomes different from X), which basically implements our idea of giving varying sensitive input data in the original and mirror execution. When the sink API is called, the parameters and corresponding mirror versions (X and X') are compared. A difference in output corresponds to an event of data leakage, otherwise, this particular outgoing data is believed to be benign.

IV. Implementation

This section describes the inner working of MirrorDroid in detail . It first describes the necessary modifications introduced to the Android operating system and then discuss how these changes can actually detect leakage of sensitive data.

A. Mirror Data Structures

For each instruction in a program, MirrorDroid intervenes the interpreter operation to add additional operation which actually repeats (duplicates) that instruction and the results of the operation are reflected on a set of identical program variables (*mirror variables*). MirrorDroid dynamically creates these extra storages in Android heap memory and uses some heuristics for easy allocation and deallocation of these memory spaces.

For every method frame in actual program stack, a duplicate method frame (mirror frame) is created. However, to reduce memory overhead, we only list a few member of the actual method frame structure defined in Android [1]. The mirror frame only contains - *method name, class name, mirror registers, position in mirror stack*. This design does not violate the goal of the proposed method and helps to reduce the overhead from the duplicate execution.

B. Modified Dalvik Virtual Machine

Here, we worked on the portable version (*C++*) of the Dalvik interpreter for simplicity of development. However, the same technique can be easily applied to other versions as well.

Table I. **HANDLING DALVIK INSTRUCTIONS FOR MIRROR EXECUTION WITHIN MIRRORDROID.**

Instruction Format	Actual Operation	Mirror Operation	Description
$const - op\ V_a, C$	$V_a \leftarrow C$	$mV_a \leftarrow C$	Assignment of constant to program variable
$move - op\ V_a, V_b$	$V_a \leftarrow V_b$	$mV_a \leftarrow mV_b$	Assign contents of V_b to V_a
$unary - op\ V_a, V_b$	$V_a \leftarrow \otimes V_b$	$mV_a \leftarrow \otimes mV_b$	Handing unary operation
$binary - op\ V_a, V_b, V_c$	$V_a \leftarrow V_b \otimes V_c$	$mV_a \leftarrow mV_b \otimes mV_v$	Binary operation involving three variables
$binary - op\ V_a, V_b$	$V_a \leftarrow V_a \otimes V_b$	$mV_a \leftarrow mV_a \otimes mV_b$	Binary operation involving two variables
$binary - op\ V_a, V_b, C$	$V_a \leftarrow V_b \otimes C$	$mV_a \leftarrow mV_b \otimes C$	Binary operation involving two variables and a constant
$aput - op\ V_a, V_b, V_c$	$V_b[V_c] \leftarrow V_a$	$mV_b[V_c] \leftarrow mV_a$	Write Array element, index remain same in both executions
$aget - op\ V_a, V_b, V_c$	$V_a \leftarrow V_b[V_c]$	$mV_a \leftarrow mV_b[V_c]$	Read Array element, index remain same in both executions
$sget - op\ V_a I_b$	$V_a \leftarrow sI_x$	$mV_a \leftarrow sI_x$	Class Field Value remain same in original and mirror program
$sput - op\ V_a I_b$	$sI_x \leftarrow V_a$	NIL	Static Class Field only assigned once, not repeated in mirror program
$iput - op\ V_a, V_b, I_c$	$V_b.I_c \leftarrow V_a$	$mV_b.I_c \leftarrow mV_a$	Write to Class field I_c of V_a
$iget - op\ V_a, V_b, I_c$	$V_a \leftarrow V_b.I_c$	$mV_a \leftarrow mV_b.I_c$	Read from Class field I_c of V_b

For each instruction (opcode) in the program, interpreter executes a handler routine to take necessary action. MirrorDroid modifies these handlers to add some additional code, which actually repeats the handler operations (the mirror execution). The Listing. 1 shows the handler routine for the opcode OP_CONST, which depicts what happens if a constant is assigned to a program variable. Here, Line 7 to 9 were added as part of the MirrorDroid implementation and essentially duplicates the assignment operation. The difference is - these lines assigns the constant to a mirror variable. Similarly, handler routines of all the opcodes (256 different types defined in Android) are modified and described in next section.

Listing 1. Handling OP_CONST in MirrorDroid

```
1  HANDLE_OPCODE(OP_CONST){
2    u4 tmp;
3    vdst = INST_AA(inst);
4    tmp = FETCH(1);
5    tmp |= (u4)FETCH(2) << 16;
6    SET_REGISTER(vdst, tmp);
7    ret = isTarget(curMethod);
8    if(ret == 1)
9     MSET_REGISTER(vdst, tmp);
10 }
11 FINISH(3);
12 OP_END
```

1) Parallel Execution of Interpreted Code: As discussed above, parallel execution of instructions of an application is achieved by repeating each instruction's execution before the next instruction is fetched by the interpreter. Table. I summarizes this process for the various types of instructions (256 in total) defined in Android . It describes the underlying logic behind duplicating (mirroring) of each such instruction type apart from method call instructions which are discussed separately in Algorithm 1. The instructions listed here is an abstract representation of the actual byte-code instruction formats specified in DEX (Android byte code format).

Here, the local and argument variables (V_x) correspond to virtual registers. Apart from the static fields and class field variables, mirror of V_x is denoted by the mV_x. Constant values are represented by C. Ix represents class field variables of type x. Instance fields are denoted $V_y.I_x$, where V_y (mV_y is the mirror) is the instance object reference. Static fields are denoted as sI_x, where x is the type of that field. Finally, $V_y[.]$ ($mV_y[.]$ corresponds to the mirror) represents an array, where V_y is an array object reference variable.

Algorithm 1 describes various strategy to handle method call instructions based on the state of the program and the type of method being called. Within the Dalvik VM, the return value of a method is always copied to a special system variable $retval$. To hold the return value of duplicate method execution, we define $mRetval$, which has the same structure and type as *retval*. If the arguments and their mirror versions are equal ($flag = 0$ in line 3-7 of Algorithm 1), duplication is not necessary . However, MirrorDroid then checks if any of the conditions specified between line $8 - 27$ is satisfied or not. If not satisfied and $flag = 1$, we must execute the target method twice (line 29 of Algorithm 1) with actual and mirror method parameters respectively. If satisfied, based on the type of method being invoked MirrorDroid performs various tasks as discussed below.

2) Handling Native Code: Although, Android applications are primarily Java based, they can include native code (C/C++) to request operations which can be only performed by the higher privileged system processes. The native code is not interpreted by the interpreter, so cannot be directly monitored by MirrorDroid. So, we employ the following strategy to retain the effects of native operation in actual and mirror execution.

Whenever a native method is invoked, MirrorDroid saves the actual and mirror method parameters and waits for its completion. When completed, the same native method is called again with the mirror parameters and the return value is stored in the variable *mRetval*. However, If the method arguments and the mirror counterparts are equal then there is no need to call the native function twice, it is called once and the return value will copied to the corresponding mirror execution(line $14 - 21$ of Algorithm 1).

3) Selective Tracing of Android Library API: Android applications call hundreds of library APIs as part of its operation. Some APIs are required to draw various GUI elements on the screen, refreshing and resuming the GUI components, thread scheduling, exception handling etc. In most cases, these functions have nothing to do with the sensitive operation. Blindly tracing and duplicating each program methods in mirror execution will incur a huge runtime and memory overhead, which with careful steps can be avoided.

For this, we form a list of such methods named $no - track{-}list$ (Table. II) and omit them from duplicate execution. These methods are executed only once and the return value is used in verbatim in mirror execution (line 22- 24 of Algorithm 1).

Apart from the $no - track - list$ methods , we can also avoid executing a method twice if it is a library method and

Algorithm 1 Handle Method Call Instruction

1: **procedure** INVOKE ***foo***$(x_1, x_2, ..., x_n)$
2: $flag \leftarrow 0$
3: **for** $i \leftarrow 1, n$ **do**
4: **if** $x_i \neq x_i^{'}$ **then** ▷ $x_i^{'}$ = mirror value of x_i
5: $flag \leftarrow 1$
6: **end if**
7: **end for**
8: **if** $isSensitiveSource(foo) = 1$ **then**
9: Invoke *foo*$(x_1, x_2, ..., x_n)$
10: $mRetval \leftarrow retval + random\_constant$ ▷ *mRetval* = mirror return value,*retval* = Actual return value
11: **else if** $isSink(foo) = 1$ and $flag = 1$ **then**
12: Invoke *foo*$(x_1, x_2, ..., x_n)$
13: Alert Information Leakage to user ▷ Sink method is executed once, mirror execution avoided
14: **else if** $isNative(foo) = 1$ **then** ▷ $isNative$=1 for native,otherwise 0
15: **if** $flag = 1$ **then**
16: Invoke *foo*$(x_1, x_2, ..., x_n)$
17: Invoke *foo*$(x_1^{'}, x_2^{'}, ..., x_n^{'})$
18: **else**
19: Invoke *foo*$(x_1, x_2, ..., x_n)$
20: $mRetval \leftarrow retval$ ▷ Return value of actual execution if copied to mirror return value
21: **end if**
22: **else if** $isNoTrackMethod(foo) = 1$ **then** ▷ $isNoTrackMethod$ = 1 for $no-track-list$ methods,otherwise 0
23: Invoke *foo*$(x_1, x_2, ..., x_n)$
24: $mRetval \leftarrow retval$ ▷ retval copied to *mRetval*
25: **else if** $flag = 0$ and $islibMethod(foo) = 1$ **then** ▷ $islibMethod$ = 1 for traced methods,otherwise 0
26: Invoke *foo*$(x_1, x_2, ..., x_n)$
27: $mRetval \leftarrow retval$ ▷ retval copied to *mRetval*
28: **else**
29: Invoke *foo*$(x_1, x_2, ..., x_n, x_1^{'}, x_2^{'}, ..., x_n^{'})$ ▷ Start parallel execution of method foo
30: **end if**
31: **end procedure**

Table II. **ANDROID APIS EXCLUDED FROM DUPLICATE EXECUTION**

Package Name of $no-track-list$ Methods
$android.app$, $android.widget$, $android.transition$, $android.accessibilityservice$, $javax.microedition$, $android.print$ $org.apache.commons$, $android.animation$, $org.apache.commons$, $java.util.logging$, $android.test$, $android.graphics$ $android.annotation$, $android.bluetooth$, $android.gesture$,$android.support.multidex$, $android.animation$, $android.nfc$ $android.hardware$, $android.mtp$, $anroid.opengl$, $android.preference$, $android.hardware$, $android.inputmethodservice$

the method parameters and their mirror versions are identical ($flag = 0$ in line 25 of Algorithm 1).

4) Handling Non Deterministic Input: Some program inputs are non deterministic by nature, $e.g$ random number, system time, current location information etc. We must ensure that, these input data will be identical in two executions of the target program. For example, the *nextInt(n)* method of *Random* class in Android returns a pseudo random uniformly distributed in the range $[0, n)$. Calling it twice will return two different value resulting in a difference in mirror execution without actually accessing any sensitive data. To solve that the *nextInt(n)* method is executed once and the return value is used both in the actual and mirror execution. This is repeated for all such non deterministic APIs.

5) Detection of Information leakage: MirrorDroid detects sensitive data leakage disclosure when it finds different output data in actual and mirror execution. For this detection scheme to work, source and sink apis are handled as discussed below. If the called method is a source API, we avoid executing it twice. Return value($retval$) of original method call is changed by adding some random constant and assigned to $mRetval$ (line 8-10 of Algorithm 1). Like source API, MirrorDroid places its privacy hooks in sink APIs (Network and SMS). When a sink API is encountered, the actual and mirror parameters are compared. If a difference is found, it is assumed that it is a result of earlier private data access and a push notification is sent to user describing the event (Line 11-13 of Algorithm 1).

V. APPLICATION STUDY

A. Experiment Setup

Testing of applications were done on an Android emulator running OS version 4.4.2 (kitkat). The base Android source source code was first downloaded and later patched with proposed modifications to build the modified system, named as MirrorDroid.

Currently we consider: *IMEI(Device Id) number*, *IMSI (Sim Serial) number*, *Contacts*, *Location*, *SMS*, *Android OS Version*, *Phone number* and *model* as the *source* of sensitive data. And

Table III. **ANDROID MALWARES AND LEAKED INFORMATION**

Malware Family (Number of Samples)	Leaked Information
ADRD (22)	IMEI, IMSI, OS Version
BeanBot (8), BgServ (9), RougeSP Push (9), DroidKungFuUpdate (1)	IMEI, Phone Number, OS Version
CruiseWin (2), DroidKungFu4 (96), JSMSHider (16), Plankton (11)	IMEI
DroidDreamLight (46), ZHash (11)	IMEI, IMSI
DroidKungFuSapp (3)	IMEI, OS version, Phone model
Geinimi (69)	Location, IMEI, IMSI
Ginger Master (4)	IMEI, IMSI, Phone Number
Gone60 (9), HippoSMS (4), Kmin (52)	Contacts, SMS
JiFake (1)	Contacts
LoveTrap (1)	IMSI, Location
PjApps (58), SndApps (10)	IMEI, Phone Number
SMSReplicator (1), ZSone (12)	SMS
WalkinWat (1)	IMEI, Phone Number, Phone Model, OS Version
Zitmo (1)	IMEI, SMS

Total 449 samples from 25 Family

Table IV. **GOOGLE PLAY APPLICATIONS AND LEAKED INFORMATION**

GooglePlay Apps	Leaked Information
BBC, DU Battery Saver, Hola Launcher, Dictionary	IMEI
Coupons, Horoscope, Power Battery, SpeedTest, Yellow Pages	IMEI, Location
Antivirus Free, Solitaire	IMEI, IMSI
Crackle Movie	IMEI, IMSI, Android Version
Brightest LED Flashlight	Android Version, Phone Model, Location
Imdb, iHeartRadio, DH Texas Poker, Espn Score Center, Castle Defence, Mind Game	IMEI, Android Version, Phone Model
AccuWeather, 4 Pics in 1 Word, Amazon Mp3, Logo Quiz, Craiglist Mobile, Inkpad, MeetMe	Android Version, Phone Model
Hulu, Logo Quiz	Android Version
Kik	Contacts
Evernote	Contacts, Location, Phone Model
Zedge	IMEI, Android Version
Wikipedia, Walmart, Flixter, WebMD, TuneInRadio, Paypal, Wish	Location
Zillo, Instagram, AskFM, Drag Racing, FruitNinja, Ant Smasher, Google Translate Pandora, Messenger, Power Cleaner, Yahoo Mail, OfferUp, Yelp	NIL

for the *sink*, we consider *outgoing network data* and *SMS*. This list is easily modifiable as it is determined using a configuration file, which can be loaded to the running emulator with a single command.

B. Data Set and Experiment Results

We consider two data sets - a collection of known malware samples and a set of popular Android Market Place(Google PlayStore) applications whose information leaking behavior is unknown before experiment.

1) Malware Data Set: The first data set [5], has total 1260 samples under 45 different malware families. However, not all of them leak sensitive data [3], [16]. Apart from that, some malwares families in this collection(For example, DroidKungFu1, DroidKungFu2,GoldDream, DroidDelux) can no longer be run in operating system version higher than 2.2, hence discarded from our study. Taking these into considerations, our experiment includes 449 samples of 25 different malware families taken from this data set.

Table. III shows the result of experiment on malware data set, that includes - the name of the malware family and the leaked information type. The number of samples taken from each family is shown in the brackets. For brevity, if more than one malware families leak similar kind of private information, they are grouped together. The findings in Table. III matches with the known behavior of these malwares from the literature [3], [16].

In our study we found zero false positives for malware data set. The reason behind this high accuracy level is for each type of sensitive data, we checked whether a particular application leaks that data or not. We did not change the input of other sensitive inputs in the duplicate run, so the observed result obviously corresponds to the leakage of the particular data item we were tracking. This process is repeated for all the sensitive data in the test.

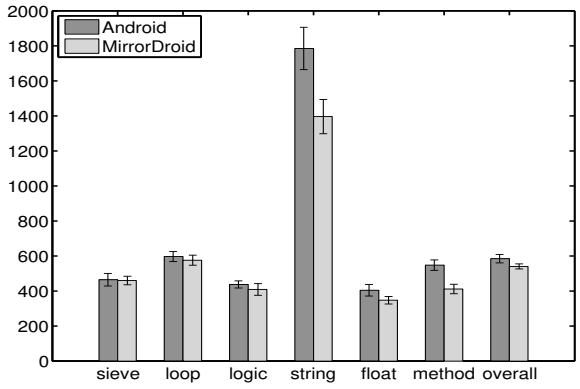

Figure 2. **Performance of MirrorDroid compared to Stock Android**

2) Google Play Store Applications: The second set of applications include 50 popular Android applications selected randomly from the top free 500 applications of Google Play Store [6]. Table. IV shows findings in our study(*NIL* indicates no sensitive data leakage detected). Out of 50 applications, 34 leak some kind of data. We later extracted the source code of these applications for verification purposes and found that all the reported instances are true positives, which proves the

effectiveness of our method.

C. Performance Evaluation

Theoretically, MirrorDroid should have 100% runtime overhead. However, we could avoid duplicating a large part of an application without sacrificing the correctness. This allows to reduce its runtime cost to a great extent. Here, the standard CaffeineMark 3.0 for Android [2] was used for overhead calculation. CaffeineMark uses an internal scoring metric that roughly corresponds to the number of java instructions executed per second. The overall results indicate a cumulative score across individual benchmarks.

The results was shown in Figure. 2. The unmodified Android ($version$ $4.4.2$) had an average score of 585.0. MirrorDroid, was measured at 541.0 overall, resulting in a 8.2% overhead with respect to the unmodified system. The benchmark program was run for 30 times each for the Android and MirrorDroid system and the error bars indicates the 95% confidence intervals.

VI. Related Work

In recent years, there have been a significant number of work aimed at securing Android applications. Here we restrict our attention to some of the closely related work employing various runtime monitoring schemes.

TaintDroid [9] is one of the first to dynamically monitor the flow of sensitive information to unintended third party destinations. It taints data from private information sources, and then checks the data leaving the system via network interface. DroidScope [14] provides APIs to facilitate custom analysis at different levels: hardware, OS, and application level. CopperDroid [13] dynamically observes interactions between Android components and the Linux system to reconstruct high-level behavior to find malicious activities.

Appintent [15] combines static and dynamic analysis to find out whether a flow of sensitive data to sink is user intended or not. However, this method rely much on the various heuristics and success depends on underlying natural language processing scheme. TISSA [17] gives users detailed control over an application's access to a few selected private data by letting the user decide whether the application can see the true data or some mock data. MockDroid [7] and AppFence [12] replaces sensitive information with shadow data. While these methods stop the leakage of data, they also impact legitimate applications that are expected to access these data. A recent work [8] combine dynamic analysis and machine learning techniques to detect android malware based on system call invocations.

VII. Conclusion

In this paper, we proposed MirrorDroid, an efficient runtime monitoring system to detect sensitive information leakage from Android based mobile devices. We also described its architecture, operation and evaluation through experiment on two data sets.

Currently, we are working on adapting the proposed idea to the new Android Runtime (ART). In future, we also plan to add the following features to the MirrorDroid system - track intercomponent communication between two separate processes and include more sensitive information sources and sinks as part of the MirrorDroid tracking system to make it better equipped to prevent sensitive data disclosure.

VIII. Acknowledgments

The authors would like to thank Professor Xuxian Jiang and his research group from North Carolina State University for sharing us with the android malware data set.

References

[1] Android method frame structure. http://androidxref.com/4.4.2_r1/xref/dalvik/vm/oo/Object.h#486.

[2] Caffainemark 3.0 for android. https://play.google.com/store/apps/details?id=com.android.cm3&hl=en.

[3] Contagio mobile malware mini dump. http://contagiominidump.blogspot.com/.

[4] I. d. corporation. worldwide quarterly mobile phone tracker 2015 q2. www.idc.com/prodserv/smartphone-os-market-share.jsp.

[5] Malware data set. http://www.malgenomeproject.org/policy.html.

[6] Official android marketplace: Google play. https://play.google.com/.

[7] A. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.

[8] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016.

[9] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[10] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[11] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[12] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[13] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[14] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

[15] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[16] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[17] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.