AUTOMATED SYSTEMS FOR TESTING ANDROID APPLICATIONS TO DETECT SENSITIVE INFORMATION LEAKAGE

by

SARKER TANVEER AHMED RUMEE

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

April 2017

To my mother Nazma Begum and my Father Md.Fazlur Rahman Sarker who set the example and who made me who I am.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Yu Lei for constantly motivating and encouraging me, and also for his invaluable advice during the course of my doctoral studies. I am indebted to late Professor Donggang Liu who worked as my supervisor for most part of my Ph.D study. He encouraged me to take up my current research topic and gave me confidence and initial background knowledge to pursue this area. I also wish to thank my academic advisors Dr. Christoph Csallner (who also worked as my supervisor for some time), Dr. David Kung and Dr. Jiang Ming for their interest in my research and for taking time to serve in my dissertation committee.

Pam McBride, Sherri Gotcher, Camille Costabile, Bito Irie, Skipper Harris and others from the CSE department's administration were always helpful since the beginning of my Ph.D program. My sincere gratitude to them who always work in background to make our life easier here at UTA.

I would also like to extend my appreciation to the graduate school of UT Arlington for providing financial support for my doctoral studies. I am grateful to all the teachers who taught me during the years I spent in school, first in Bangladesh, and finally in the Unites States. I would like to thank Dr. Nasimul Noman and Dr. Amin Ahsan Ali for encouraging and inspiring me to pursue graduate studies.

My parents, Md. Fazlur Rahman Sarker and Nazma Begum and my younger brother, Sarker Tanzir Ahmed were always there to help me. My wife Murshida supported me all the time with her care and patience. Our only child Rafia was born in 2013 and since then she has been a source of inspiration and blessing. These people are my source of strength and comfort and without their support nothing would have been possible.

Finally, I would like to thank my friends and fellow members of the Software Engineering lab and Information Security Lab at UTA who have helped me throughout my graduate studies.

April 13, 2017

ABSTRACT

AUTOMATED SYSTEMS FOR TESTING ANDROID APPLICATIONS TO DETECT SENSITIVE INFORMATION LEAKAGE

SARKER TANVEER AHMED RUMEE, Ph.D.

The University of Texas at Arlington, 2017

Supervising Professor: Yu Lei

Smart phones have become an important daily companion and often used by users to store various private data such as contacts, photos, messages, various social network accounts etc. Users can furthermore extend the functionality of their phone by downloading applications (or apps) from various developers and online application stores. However, apps may misuse the data stored on the phone or obtained from the sensors and users do not have any direct means to track that. Hence, the need for improved mechanisms to better manage the privacy of user data is very important.

There has been a lot of effort to detect and thwart unauthorized access to these private data. However, there is no consensus method which can ensure protection of user sensitive information from mobile devices and at the same time easily deployable at user side. This dissertation aims at developing methods to test Android applications for privacy leakage detection. For this, it presents a new technique: if an application is run twice and all program inputs and environment conditions are kept equal, then it should produce identical outputs. So, if a sensitive input is changed in two separate executions of the target application, and a variance is observed at output, then the output contains information from that sensitive

input. Based on this idea we developed two systems namely **DroidTest** and **MirrorDroid** to detect leakage of privacy sensitive data.

DroidTest instruments the Android framework APIs to insert security monitoring code. The instrumented APIs help to record user interactions and sensitive API values in record phase (first run of application) and restore the recorded information during replay execution (second run of the target application). Program inputs (except sensitive data) and environment conditions are kept equal in both runs and change in corresponding outputs corresponds to leakage of sensitive data. DroidTest does not require costly platform update and can be easily distributed as a modified Android SDK. On the other hand, **MirrorDroid** places the monitoring code within the Android Runtime (Dalvik Virtual Machine). It does not explicitly run an application twice like DroidTest. Rather, the instrumented Dalvik VM intercepts execution of each instruction and duplicates it before fetching next instructions, essentially running a separate execution (mirror execution) of the target program in parallel. Then the outgoing data in original and mirror execution is compared to find evidences of information leakage.

We have evaluated the proposed systems on two data sets. The first data set is taken from the Android Malware Genome Project containing 225 samples from 20 malware families. Using DroidTest and MirrorDroid to monitor information leakage, we could successfully detect leakage already reported in literature. The second data set consists of 50 top free applications from the official Android Market Place (Google Play Store). We found 36 out of this 50 applications leak some kind of information, which is very alarming considering these are very popular and highly downloaded applications. Although, the proposed systems either instruments the application framework APIs or the Dalvik Virtual Machine, they produce low runtime overhead (DroidTest 22% and MirrorDroid 8.2%). The accuracy of the proposed detection mechanisms also proves the effectiveness of our methods. DroidTest produces 22% false positives. If we ignore false warnings generated by different ordering of thread executions

in record and replay phase, the false positives rate stands at 10%. MirrorDroid does better than DroidTest and generates only 6% false positives for the applications in test data sets.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the ever increasing popularity of Android operating system, millions of third party applications have been developed in recent years. As smartphones are becoming a ubiquitous source of private and sensitive personal information, the behavior of these applications regarding user privacy is a major concern. Private content stored in the mobile devices can be misused or leaked to third party either unintentionally (by poorly programmed applications) or intentionally(malicious applications). In this dissertation we propose new methods to test Android applications for sensitive information leakage detection.

1.1 Motivation

Today's mobile devices have more computing power than the powerful computers a few years back. It allows users to perform many activities such as web browsing, sending emails, social networking, playing high end games, making video and audio call, taking high quality pictures or videos etc. As a result, smartphones have become a daily necessity to millions of people. According to a recent survey [1], over 80 percent of the US mobile users own smartphones. To meet this growing popularity, vendors like Google, Amazon etc. have put millions of applications in the online application stores. These applications cover a broad spectrum of functionality: applications for entertainment purposes, like games for children and for adults; apps for educational purposes that can be used at schools and at home; apps that render managing financial investments trivial; apps that help people manage their time and tasks; office apps, data management apps; even apps for medical purposes, facilitating decision making for doctors, or helping patients manage their treatment or daily activities

to improve quality of life.

While these variety of apps make it convenient for users to customize their smartphone systems for their needs, attackers have also become interested in them for compromising user's systems. Indeed, in recent years, Android malwares exploiting different vulnerabilities within the OS or the applications running on it. Android's security protection also has weaknesses, for example: permission re-delegation [2] and capability leaks [3]. Apart from that, the base line security mechanism of Android : *Permission based Access Control to Sensitive Data* is not enough to protect sensitive data misuse [4]. So, properly identifying which mobile app poses a threat is a major concern and requires both user side awareness and mitigation techniques/tools from the industry/research community.

Privacy protection of smartphone data is well studied in the literature. Hence, a comprehensive technique to guard against all kinds of sensitive information leakage is yet to be developed. Existing solutions include static analysis and dynamic execution monitoring. Static analysis scans byte code or source code to find paths that may leak information. However, it does not execute the program and lacks program dynamic information. Moreover, malwares are often heavily obfuscated to thwart static analysis. Furthermore, runtime information is often needed for exploit diagnosis. In this regard, dynamic analysis approaches do better in detecting actual malicious activities as they rely on analyzing program runtime behavior. However, state of the art dynamic analysis techniques (e.g. [5,6]) suffer from problems like: high runtime overhead , application breakage, limitations in native code and file handling etc.

1.1.1 Motivating Examples

State of the art TaintDroid [5] was one of the first methods to apply dynamic taint analysis to privacy monitoring of Android applications. Experimental results also proved its effectiveness in uncovering leakage of sensitive data. However, it has some limitations which

prohibit its usage in a number of scenarios. In some cases (sensitive data written to files, over tainting of program variables), the underlying taint analysis technique fails to properly assign tags to program entities. TaintDroid also produce false warnings when applications uses native code or arrays to access or propagate sensitive information.

Taint analysis can fail if a previously tainted **variable no longer carries sensitive data but the variable remains tainted**. Listing 1 shows an example code snippet to describe this. Here, sensitive data (device id) is accessed in line 3 and then in line 6 and 7, we construct an array (*converted*) by performing **exclusive-or** of each character of the device id information with itself. The resultant string *out*(calculated in line 8) should contain all zeros, a constant value. The string *out* is then passed to the sink API *sendTextMessage* in line 9. Here the variable *out* does not contain any sensitive information, but the taint analysis system will mark this as tainted and produce false warning.

Listing 1 Taint analysis fails if value of tainted variable becomes constant.

```
1 TelephonyManager tm = (TelephonyManager)
2             getSystemService(Context.TELEPHONY_SERVICE);
3 String device_id = tm.getDeviceId();
4 char converted[] = new char[device_id.length()];
5
6 for(int i=0;i<device_id.length();i++)
7   converted[i] = (char)((device_id.charAt(i))^(device_id.charAt(i)));
8 String out = new String(converted);
9 sms.sendTextMessage("X..X", out);  //sink, no leak
```

Arrays are very important and common data structure used by the developers. So, proper handling of arrays are critical for any analysis system to correctly detect leakage of sensitive data. However, in TaintDroid a single tag is assigned for each array. As a result, if a particular array index contains sensitive data the whole array becomes tainted. False warnings can be produced if non sensitive data from a tainted array is passed to sink (network, SMS etc.) as shown in Listing 2.

Listing 2 Taintdroid [5] gives false warning if non sensitive data from tainted array is used

```
1 TelephonyManager tm = (TelephonyManager)
2             getSystemService(Context.TELEPHONY_SERVICE);
3 String device_id = tm.getDeviceId();
4 String arr [] = new String[2];
5
6 arr[0] = "nonsensitive";
7 arr[1] = device_id;
8 sms.sendTextMessage("X..X", arr[0]);  //sink, no leak
```

In this example, an array named ***arr*** is declared in line 4. And later two assignments are done to this array: a non sensitive information to index 0 and the device id to index 1 (line 6 and 7 in Listing 2). TaintDroid assigns a single taint tag for the whole array *arr* indicating that it contains sensitive information. As a result, all locations within the array become tainted regardless of whether they contain sensitive data or not. In Listing 2, the *sendTextMessage* in line 8 will generate false warning even though no sensitive data is leaked. It is only sending a non sensitive data from the tainted array ***arr***.

Android applications can contain **native (C/C++) code**, which is not directly tracked by TaintDroid. However, it relies on some heuristics for propagation of taint tags from the native method's parameters and return values. For example, for native method *System.arraycopy*, it assumes the tag of the source will be copied to the tag of destination array. It serves the purpose for this particular method.

However, TaintDroid relies on heuristics developed with previous knowledge about data flow between the parameters and return value of native methods. This approach will only work for the native methods analyzed before, not for other system defined or user defined native methods. As shown in Listing 3, TaintDroid will work for the *System.arraycopy*, but will fail for the user defined native method (*ThirdPartyNativeAPI*). Here, we assumes that (*ThirdPartyNativeAPI*) takes the same parameters and performs the same operation as *System.arraycopy*. The only difference is (*ThirdPartyNativeAPI*) is user defined. In that case, TaintDroid will not be able to record the flow of sensitive data from source array to destination. This is a significant shortcoming and an attacker can clone each target native method to leak sensitive data.

Listing 3 Limited Handling of Native Code in TaintDroid

```
1  TelephonyManager tm = (TelephonyManager)
2              getSystemService(Context.TELEPHONY_SERVICE);
3  String device_id = tm.getDeviceId();
4  String arr1 [] = new String[2];
5  String arr2 [] = new String[2];
6  arr1[0] = "non sensitive";
7  arr1[1] = device_id;
8
9  /* arr2 gets tainted according to TaintDroid heuristics */
10 System.arraycopy(arr1,0,arr2,0,arr1.length);
11
12 /* TaintDroid do not work */
13 ThirdPartyNativeAPI(arr1,0,arr2,0,arr1.length);
```

Apart from the cases discussed above, taint analysis also fails to **handle the files** properly. If a file contains sensitive data, it is assigned a taint tag. But in this process, the whole file gets tainted and any read from the file (even if it does not read the sensitive content) is marked as sensitive.

In this dissertation work, our goal is to design systems which do not suffer from above mentioned limitations.

1.2 Approach

In this dissertation, we propose a novel approach to test android applications for sensitive information leakage. This also forms the basis of the proposals made in this dissertation work. The high level idea of the approach is stated below:

Given a deterministic function $f(x)$ ($x \in \mathcal{X}$), the result of this function does not leak any information about x if for all x_1 and x_2 we have $f(x_1) = f(x_2)$. 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. This basically means that if we fix all inputs to an application except those that are considered as sensitive, then the content of outgoing traffic should not change if there is no leakage of sensitive data.

To grasp the above mentioned idea, let us consider the following example shown in Figure 1.1.

In Figure 1.1(a), variable y is assigned a constant value and x accesses the sensitive operation $deviceId()$ which eventually passes through the sink method $write$. Figure. 1.1(b) shows if the code example in 1.1(a) is run again. Here, where variables x and y from 1.1(a) have corresponding versions x', y' accessing the same content.

The only difference is the call to the sensitive function $deviceId()$. As shown in Figure 1.1(b), in second execution, return value of the sensitive function is mutated by adding a random constant to it. It essentially implements the above mentioned approach of providing

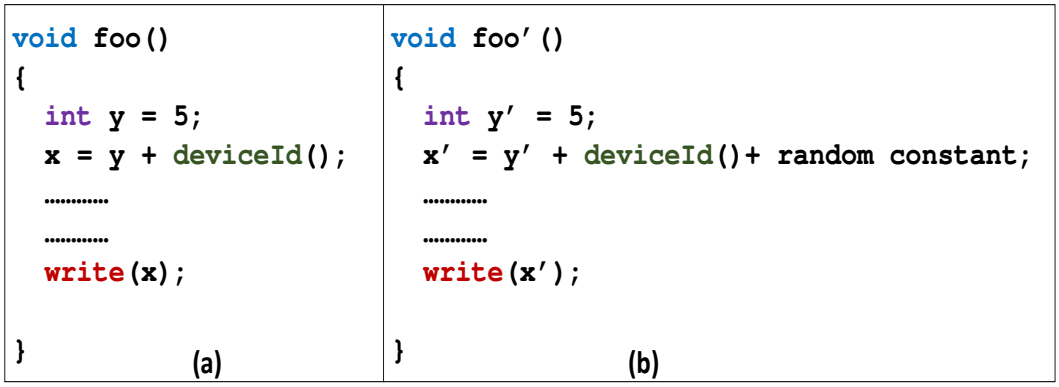

Figure 1.1: Proposed Information Leakage Detection Approach. Program run with (a) Actual Sensitive Input (b) Mutated Sensitive Input

different input to the sensitive sources for the two different run of an application. Provided that all other inputs and environment conditions equal, it is clear that the data (x and x') flowing through the output function ($write$) differ in two executions. Here, the only source of difference is the calling of sensitive function $deviceId$. With this approach in mind, we developed the proposed techniques to handle the information leakage problem which are described in the next section.

1.3 Methodology and Contributions

In this dissertation, we developed two systems: **DroidTest** and **MirrorDroid**, which can track an Android application for information leakage behavior. Both techniques employ the above mentioned approach of multiple executions of the application under test.

At first, we present **DroidTest** [7], which instruments the Android framework APIs to implements its leakage detection policy. Within this system, each application is executed twice. The program inputs and runtime environment is kept equal in these two executions, except sensitive inputs (e.g. current location, device information etc.). Then the output (data passing through network and SMS) are collected and compared. If we observe different outputs generated by a test case, then we can say that the difference is due to the difference

in the sensitive inputs. For this approach to work, we must ensure that all program inputs (except sensitive data) and environment conditions are equal during multiple executions of a program.

To achieve this, inputs coming from various non-deterministic sources (e.g. Random numbers, System time etc.) are recorded during the first execution (termed as record phase), and later restored in subsequent execution (replay phase). In the implementation level, we identify APIs which are non deterministic in nature (produce different output even with no change in input). Then, we modify the Android framework to instrument those APIs to facilitate this record and replay operation. During record operation, return values of instrumented APIs are stored in external storages and during replay, corresponding stored values are used instead of actual return values. Only the instrumented APIs are used to perform the DroidTest operation. No other component within the Android operating system or kernel needs changes.

The benefits of DroidTest are as follows. First, it does not require access to the source code. Second, the record and replay is done fully at the API level and no expensive platform modification has to be done. The system is portable and can be downloaded by users as a modified Android SDK. Then, the user just has to initiate his or her test environment with our supplied SDK (with the modified API libraries) instead of Google provided SDK. Then the user can use this custom SDK to build an emulator or real device for testing Android applications.

Next, we apply the duplicate execution technique to develop **MirrorDroid** [8]. MirrorDroid is essentially a modified Dalvik Virtual Machine. Here we also execute a target application twice, but the second execution (also called duplicate execution) runs in parallel with the original execution. From the user side, an application is run only once and the parallel execution runs transparently in the background. This is the fundamental difference with DroidTest which performs two separate executions for leakage detection.

The instrumented Dalvik VM intercepts each instruction and method calls and duplicates (re-execute) those instructions before fetching next instruction. This is repeated for all the instructions in the target application. However, these two executions differ only in sensitive input functions: methods to get *unique device id*, *contacts*, *sms*, *sd card contents* etc. Now, if the outgoing data and its mirror version differs, then MirrorDroid notifies the user about the potential sensitive data loss. Based on the notification received, the user can ignore the message or selectively close the application.

Both DroidTest and MirrorDroid were evaluated using two data sets. The first data set consists of known malware samples (206 samples from 16 malware family) and the second data set includes 50 top free applications from the official Android application store (Google Play Store [9]). The behavior of the malware samples is well studied in the literature and we also know beforehand which sensitive information is leaked by these applications. On the other hand, whether the Google play store apps leak sensitive data or not was unknown before experiment.

From our experiment, we could detect all the known leakage scenarios for the malware samples. Among the 50 applications in the second data set, DroidTest detected 36 and MirrorDroid found 34 apps to leak some kind of private data. These numbers are very high considering the fact that the applications we studied are highly popular. For DroidTest, the false positives rate was 22% for Google play store apps and 20% for malware samples. MirrorDroid bettered by this findings with a false positive rate of (6.25%) and 6% for the Android market place apps and malware respectively.

1.4 Dissertation Outline

This dissertation is organized as follows:

Chapter 2 discusses the necessary background information including the Android Framework, Android Security Model and defines the problem domain we have worked on.

DroidTest approach and implementation details were discussed in Chapter 3. This chapter also includes the details of the Android framework instrumentation process and its role in detecting information leakage.

Chapter 4 describes the design of MirrorDroid system. It includes discussions on how the modified Dalvik VM duplicates the execution of an Android app to detect information leakage. The DroidTest data sets were used to validate the proposed method. The results also show a comparison with DroidTest findings to better understand the design difference of DroidTest and MirrorDroid.

Discussion on related work to tackle this information leakage problem for Android systems can be found in Chapter 5.

Finally, in Chapter 6, we conclude the dissertation by summarizing our methods and contributions. We also highlight a number of future research directions and possible extensions to our proposed methods.

CHAPTER 2

BACKGROUND

Before going to the details of our proposed information leakage detection systems, we discuss the necessary background information about the Android operating systems and its security model in this chapter. We also briefly discuss the Android platform provided security model and its limitations in protecting users privacy sensitive data.

2.1 Android Platform Architecture

Figure 2.1 gives an high level description of The Android software stack. Green colored items are components written in native code (C/C++), while blue items are Java components executed by the Dalvik Virtual Machine. The bottom layer (red) represents the Linux kernel components. Here, we briefly discuss the components relevant to our work.

2.1.1 Android Runtime

The Android Runtime consists of the Dalvik Virtual Machine (Dalvik VM or DVM) and a set of Core Libraries. The Dalvik VM executes the applications developed primarily in Java. The core libraries are collection of various general purpose APIs.

Dalvik VM isolates the address and memory space of each application by creating a separate child VM. It gives a much greater level of security as each application runs within its own sandboxed environment. If required, inter-application communication can only happen through some well defined interfaces.

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

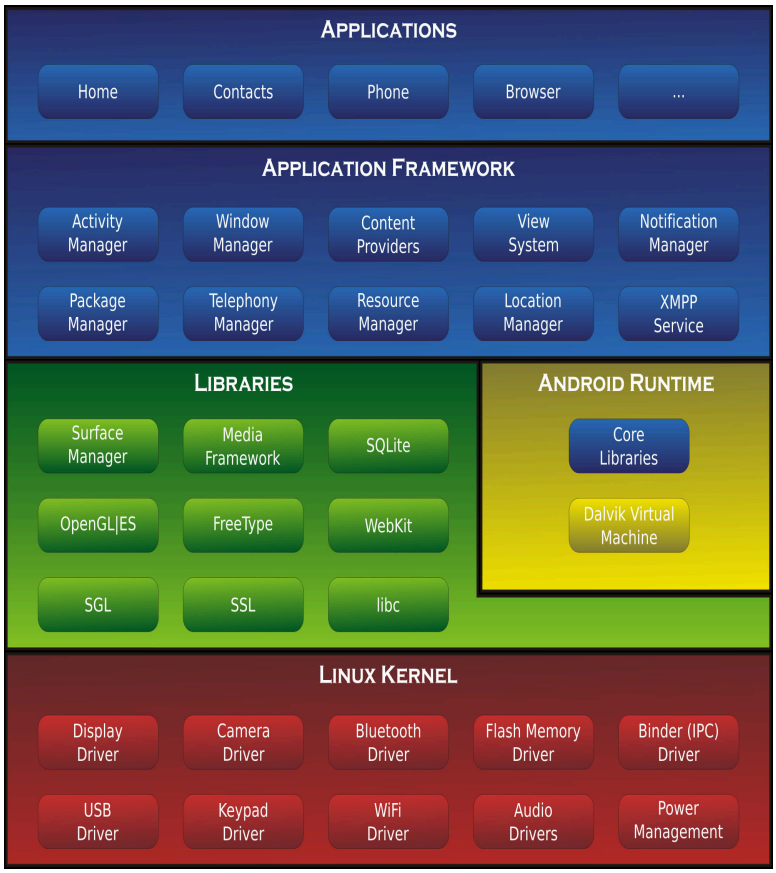

Figure 2.1: Android Software Stack

(ART) directly executes the binary code. But nothing changes for application developers. The proposed systems are 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.

2.1.2 Application Components

The Android application framework forces a structure on developers. It does not have a single *main* function or entry point for execution. Instead, Android applications have various components, each of which can be an entry point to the application. The core components are:

2.1.2.1 Activity

Any screen or window appearing in an application is actually an Activity. Apps generally contain a number of activities to perform various tasks. Let us consider the example of a music player application. One possible activity may show a list of available albums and another activity may display the currently playing item. Activities are generally independent of one another and can be invoked by other applications too.

2.1.2.2 Service

Services are activities without the user interface. It runs in the background and often used to perform long-running operations. The music application, for example, can have a service to play music in the background while the user using a different application.

2.1.2.3 Content Provider

Content providers are used to share data among multiple applications. The example music player application may use a content provider to store information e.g. the currently playing song, play lists etc. These stored data can be used by a social networking app to update a users *current listening* status.

2.1.2.4 Broadcast Receiver

This component is used to listen for system-wide events which are broadcasted by another application or service. On receiving such events, it can also optionally react with specific actions. Most of the broadcasts are initiated from the system. Some of the common broadcast events are: the system completed the boot procedure or a download process, the battery is low, or a text message was received.

2.1.2.5 Interaction between Components

Android uses a special message to communicate between application components named as intents. Intents generally define an action and may also include additional data. The music player application, for example, may send an intent to a browser to open a web page with information on the currently selected artist. Figure 2.2 shows some examples of Android interprocess communication with the aid of intents.

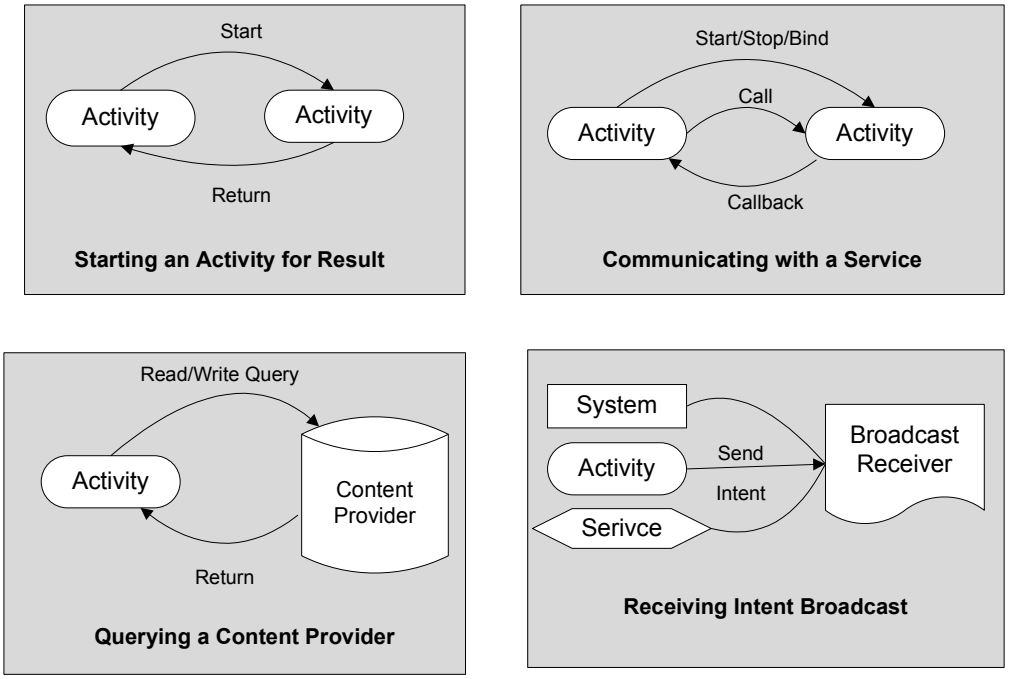

Figure 2.2: Examples of Intent Communication

2.2 Android Security Overview

Android has a number of features to ensure security of the operating system itself and the user applications and data. Here, we will focus on the underlying Linux security and the permission model to protect sensitive data.

2.2.1 Isolation through Application Sandboxing

In Android, each application is considered a separate user. So, the underlying linux kernel will assign separate user id (UID) to each running process. As a result, the application's memory space is private to itself. Android kernel ensures that no process can access the resources of another process. Applications can only communicate through well defined interprocess (known as intents) as discussed earlier in this chapter.

This setting is known as the application sandbox. Unless the kernel itself is compromised, the sandboxing technique ensures the protection of application data form authorized access by other applications or components within the Android system.

2.2.2 Android Permission Scheme

Android has a rich API to access resources on the system through its application framework. The Android sandbox allows access to some basic resources. To protect access to resources that are considered sensitive, Android governs the access such as internet, SMS, Contacts etc. through its permission mechanism.

During the applications installation process handled by the Package Manager, the user is presented with a list of permissions the app is requesting. Each permission is presented alongside a description of what an app can do with that permission granted. The user can either accept all permissions requested and install the app or abort the installation process. These are known as the system default permissions. Furthermore, an application can declare its own permissions to control access to its own resources. This way other apps can request the permission declared and the data owner app can check data requesting apps whether they have the permission or not. For example this can be used by collaborative apps, either from the same developer or not. Figure 2.3 shows the permissions requested by the popular application *Facebook* at install time.

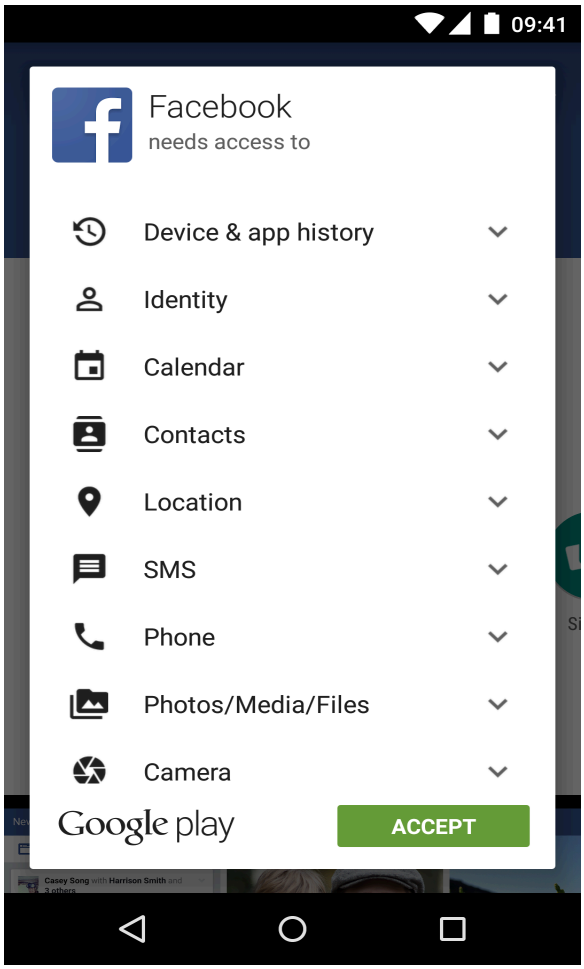

Figure 2.3: Application Install Time Permission Request

Starting form Android 6.0, permissions requests are made dynamic. Like before, users are prompted to accept the application's permission requests at install time. However, it also gives additional flexibility to the user to turn off or on certain permissions at install time. If that permissions is required, the application will prompt the user when it is necessary and user still can selectively accept or deny those permission requests. Figure 2.4 shows snapshot of the *Google+* permission request screen at install time using the new scheme. As shown here, permissions not granted initially may be requested at runtime and users have the power to either accept or deny it.

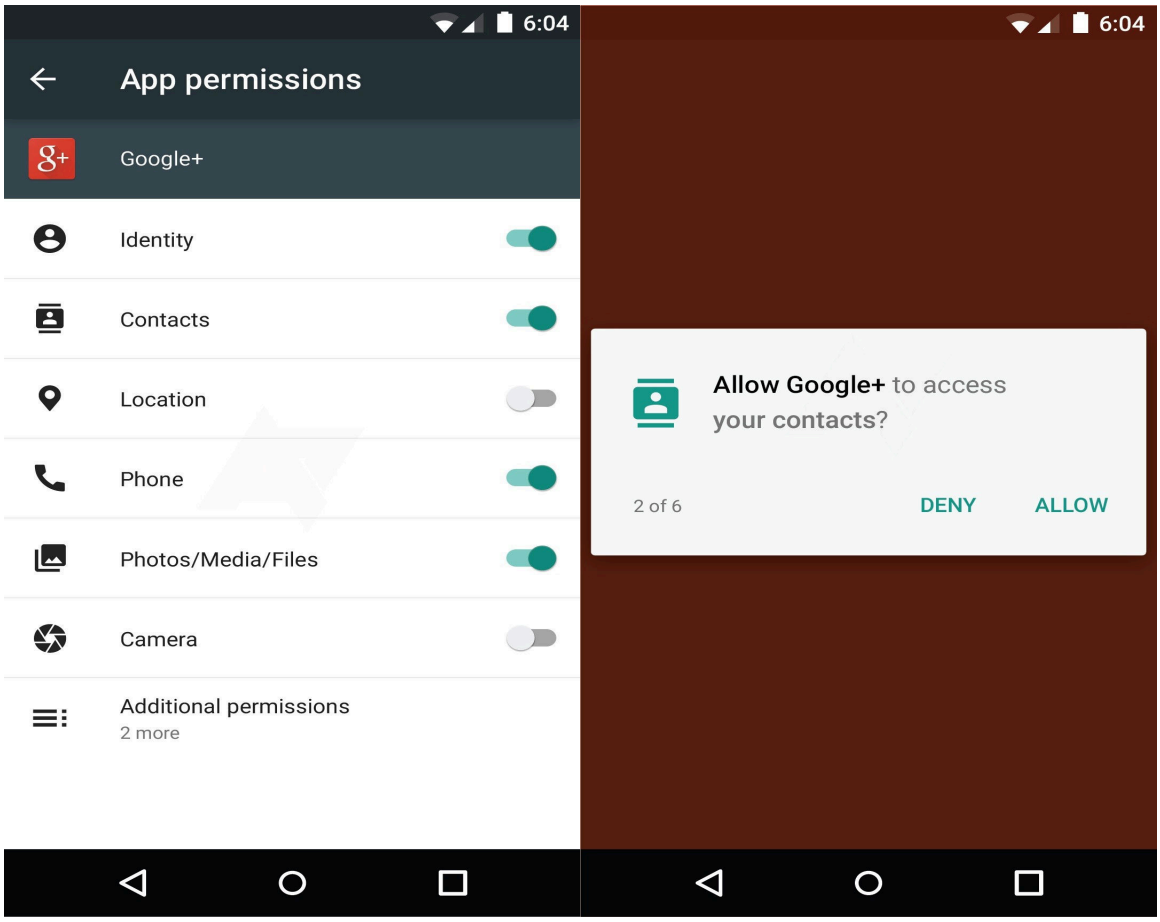

Figure 2.4: Android New Dynamic Permission Request

The first comprehensive study on Android Permissions was conducted by Felt et al [10]. However, the same authors also performed an user study [11], which reveals about user's attention and comprehension about the application permission requests in Android. Users were found to pay little to no attention to the permissions sought, rather in most cases preferred application utility over what permissions it asks from the user. As a result, it is clear that dangerous permissions are often granted to applications even if they do not require them for their intended tasks.

For example, the *Horoscope* application was found to send IMEI number of the device to the remote server. It has the permission to read various phone specific information like

IMEI, Phone Number, OS Version etc. which is simply not essential for its actual task. So, these over privileged (legitimately obtained) apps has the potential to leak privacy sensitive data without knowledge of the user. Although, with the new dynamic and runtime permission scheme, user can turn off dangerous permission at runtime, but such permissions can be sought by the application at runtime. Here comes the situation, where user need to guess about the legitimacy of the application before allowing the requests as such permissions are often asked by other well known benign applications also. Based on [11], it is understandable that users will not be able to make that decision correctly most of the time.

In summary, the scheme aims to protect sensitive resources by introducing a more flexible control over the application permission requests. But it is still not enough to protect the sensitive contents from potential misuse and additional measures must be taken to ensure that. But in practice, users pay little attention to these permission requests [11] and unnecessary permissions are often granted to applications. For example, an email application needs to send and receive emails. In Android, such an application must acquire the full internet access permission, which is more than necessary and can be exploited.

CHAPTER 3

DroidTest: Testing Android Applications for Leakage of Private Information by Duplicate Execution [1]

3.1 Introduction

Threat of malicious mobile applications are ever increasing and ensuring user's privacy is still a major challenge. The challenges increase with the fact that lot of people use mobile devices compared to other computing devices. A lot of them carry little to no knowledge about the possibility of privacy leaking applications, hence the success of a protection mechanism relies much on how easy it is for end users to adopt. One way to do this would be placing all kind of security testing and monitoring on application market places and relieve users from worrying about that. Obviously, this method is not going to protect from all kinds of mobile malware threats, but can effectively produce alarms which are easily understood by the end users and help them decide whether a particular application is safe to use.

With this in mid, we propose to test Android applications before placing them on the market. The testing can be done by the market owner or a third party other than application developers by uploading the applications to a offline server running the proposed system. Conceptually, the first step of testing is to generate test cases to explore different program paths, and the second step is to examine each program path to see if it is possible to leak any sensitive information. The first step can be done by using GUI-based test case generation [12] or concolic testing like DART and CUTE [13, 14] as long as they are ported to Android platform. Here, we assume that there exists a set of test cases produced either

[1] Copyright 2015 LNCS. Reprinted, with permission, from Sarker Ahmed Rumee, Donggang Liu DroidTest: Testing Android Applications for Leakage of Private Information (ISC), November 2013.

manually or automatically by various existing methods, and our focus is on the second step, i.e., to test if the execution under a given test input is leaking sensitive information. One method is to monitor the outgoing packet of an application and check the content to see if it is leaking something similar to a phone number, a credit card number, etc. However, when the attacker obfuscates the packet, e.g., by encryption, then this method will fail. Another natural way to solve this problem is to use taint analysis, e.g., if the outgoing packet is tainted by private data, then we can say that this application is leaking information. For example, TaintDroid [5] is built on this simple idea. However, the problem with such idea is that, the application needs to be instrumented and monitored, which requires access to the source code. While the source code can be obtained by reverse engineering, a common trend right now is that many application developers obfuscate their code to make it difficult for source code-based program analysis.

To remove the need for program source code and relieve user side from expensive monitoring, we have developed *DroidTest*. In this chapter, we detail the approach and implementation of the DroidTest. We also include the experimental analysis to validate the proposed method.

The remainder of the chapter is organized as follows. Section 3.2 introduces some key terms and assumptions made in this work. Then section 3.3 describes the high level overview of the proposed approach. Section 3.4 and 3.5 provides detailed description of the proposed method. Experiment results and findings are discussed in section 3.6. The last section concludes the chapter and discusses some possible future directions.

3.2 Definitions and Assumptions

3.2.1 Sensitive Information Source and Sink

To detect the leakage of sensitive data, the first task is to define the set of private or sensitive information sources. Some commonly regarded sources of private data are: Device id (IMEI), Subscriber Id (IMSI), Phone Number, Location and Contact information, User account information, SMS/MMS messages etc. However, the nature of application and context information are also important to decide whether a particular piece of data is sensitive or not. For example, location information sent by the popular Android application *Gasbuddy* cannot be termed as an instance of privacy violation, as it is the part of its task and user is also aware of that.

Information sink means the ways data can leave the device, which includes Network access by applications, Outgoing SMS etc.

3.2.2 Adversary Model

In this work, 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 assume that the application has the permission to read some of the sensitive information sources and access Internet. The question for us is to see if such sensitive data is leaked through any sink.

During the test, we also assume that except the application under test, all other system components, e.g, the Android OS and the Android Runtime Environment, are secure. If they are compromised, then the adversary can easily bypass our monitoring. This assumption is reasonable since the testing system is built specifically for the purpose of detecting information leakage. On the other hand, we cannot make such assumption if the detection is done at the user end. Finally, we also assume that the malicious application does not compromise the Dalvik VM or Android OS during the execution.

3.3 DroidTest Overview

The objective of DroidTest is to detect malicious activity in Android smartphone applications that send out privacy-sensitive information out through network interfaces. As mentioned in chapter 1, we apply our duplicate execution strategy where the target application is runt twice. The two runs differ only in the sensitive data and all other inputs and runtime environment are kept identical. Then a change in output corresponds to the leakage of sensitive information.

So, the main challenge is to ensure that program environment will be identical in both runs. For that, we need to identify the sources of non determinism both in inputs and the environment. Initially, our approach was to assign a constant value to these non deterministic inputs. As a result the target program will receive identical inputs in both executions. But this requires modifications to the Android API framework. From the user perspective, the full Android OS must be downloaded, patched with the proposed modifications and build to perform the tests.

Later on, we identified that, DroidTest will best serve the task of information leakage detection if it can be readily downloaded and used. So, modifying the core Android OS or the runtime environment (Dalvik VM or more recent ART) is not a feasible option in this regard. The proposed solution is to instrument the Android framework API in way that it can implement the record and replay of various program inputs in multiple execution of a target program. The benefit of this design is user need not worrying about the full Android operating system, which is kept unchanged. Rather modified framework APIs can be distributed as the modified Android SDK to be deployed readily. The proposed modification of DroidTest aims to serve the above mentioned goal.

In a summary, the application under test is run with the instrumented Android framework APIs. The APIs we instrument perform the record and replay of corresponding method calls and inputs. When an application invokes an API which was instrumented in our sys-

tem, it first checks whether it is record or replay phase. In record phase, the return value of the API is question will be stored in the external storage. If it is the replay phase, then the corresponding record for this method call is retrieved from the stored information in the record phase. Instead of the actual return value, the restored value is used. This strategy is implemented for all the non deterministic APIs we identify. As a result, we always get identical values in record and replay execution. With this setting, we check the outgoing data in two separate run and detect leakage if we see variance in the observed outputs. The details of this methodology is described in next sections.

3.4 DroidTest Desgin

Droidtest was initially developed in 2013 and we used manual process for generating test cases. We also checked the outgoing traffic manually to see if there is any instance of information leakage. Later in 2016, we modified our implementation to place all the monitoring in Android Framework APIs so that this can be distributed as a tool for end users. The next section discusses our effort to make DroidTest portable. This section discusses the preliminary(2013) design and implementation of DroidTest as mentioned in the Figure 3.1 depicting a conceptual view of the proposed system.

DroidTest has three major components: *test case generator*, *test case executor* and *kernel log collector*. The test case generator converts each existing test case into a pair of correlated test cases that only differ in the private data part; the test case executor runs the application once for each of the two correlated test cases and analyze if any private information is leaked through network interfaces; the kernel log collector basically monitors the network interface and collects the outgoing packet data. In the rest of this section, we first describe the necessary settings made in the Android operating system and then discuss the three major components of the proposed system in detail.

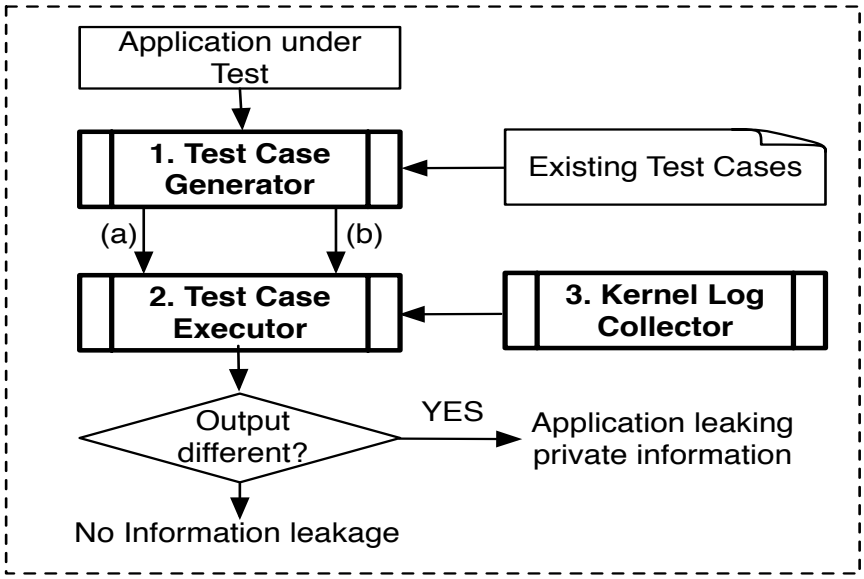

Figure 3.1: High-level view of the proposed DroidTest system.

3.4.1 Handling Non Deterministic Inputs and Sensitive Sources

At first, We locate all APIs that are required to read phone specific information, find location data, contact lists and read incoming SMS contents etc. Then custom source code is inserted in these methods to change the return values with randomly generated values. As a result, different invocation of these methods will return different values. This setting helps us to create co-related test cases with variations in the sensitive data.

For example, to get the *Device ID* or *IMEI number*, one has to call the *getDeviceId()* method of the *Android.telephony.TelePhonyManager* class. It internally uses the *getDeviceId()* method of the class *com.Android.internal.telephony.PhoneSubInfo*. We modified the *getDeviceId()* of *PhoneSubInfo* class to read a mock IMEI number (randomly generated) and return it to the application.

In addition to the modification at the information sources, we also try to fix program inputs such as system time, random number generators, and environment variables to constant values. In the current implementation, we only modified the *java.util.Random* and

java.lang.System classes. It aids to offset variation in program output for multiple runs due to the factors other than the user provided inputs.

3.4.2 Test Case Generation and Execution

As said earlier, we have manually generated test cases for the applications under test. Each application is run for some time (around 5 minutes) in the emulator. We manually exercise its basic features, e.g., try to navigate to all the screens, click the available buttons and links etc. For each such activity, we check whether the application has performed any sort of network activity or sent any text message. If so, the sequence of events that led up to this point from the application home screen (entry point) is taken as one candidate test case for this application.

In the implementation level, the co-related test case is nothing but a repeat of the above mentioned test case, where one of the sensitive input data is intentionally mutated to create variance with the original test case. Then the application is run again with this changed test case, which essentially revisits the same sequence of events while running the application.

Once the execution is done for each of the correlated test cases, we retrieve and compare the the monitoring results from the kernel logs. If they are different, then we report an alarm since some sensitive information is leaked through the internet by the application under test.

3.4.3 Information Leakage Detection

The last key component of our system is a simple Loadable Kernel Module, that intercepts the outgoing network packets. To do that, we hook the System Call Table of the Android operating system. This is done by placing a custom system call table in the same address as the original system call table.

Android operating system uses system call *sys_sendto* to send data to network. We intercept this system call and log its parameters (packet data) to the kernel logs.

3.5 Making DroidTest Portable

To make DroidTest portable and ready for use by the end users (software testers, Application Market Place Owners) we modified the earlier DroidTest design in a number of ways. We addressed following limitations of the earlier DroidTest system to make it better equipped to detect leakage of sensitive data:

- Handling of non deterministic inputs was not accurate.
- Outgoing network data were manually checked for information leakage
- Can only be used if the modified Android OS is downloaded fully.
- Built for Android version 2.2, which requires updates as newer version of Android are released with added security features.
- Replay (Re execution) of the target program was done manually, which made it difficult to exactly reproduce the execution.

The following discussions detail the main components of the proposed system.

3.5.1 APK Permission Updater

Android applications do not have permission to write to address space of other applications. It can only write to its own data folder located at (/data/data/App). However, this memory is restricted by the total memory allocated to that particular application. So, we keep the record of the APIs in external storage (sdcard) which is accessible by applications and the total space is under user control. However, for this scheme to work the target application must have necessary permission to write to external storage.

In DroidTest, we do not assume that users have access to the application source code. The application under test may not have the source code available and it is quite common

that it may not have the write permission to sdcard. To solve this, it takes help of the popular Apktool [15]. Using Apktool, we decompile the application and thereby gain access to the $AndroidManifest.xml$ file which declares all the permissions needed. We just modify this file to add write permission if it is not there. Then we reconstruct the APK using the Apktool. We do not modify the application code or any other properties. As a result, the updated APK has the permission to write to external storage.

3.5.2 API Instrumenter

This component deals with modification of stock Android API for realization of our approach. Certain APIs within the Android Framework is instrumented to insert additional code which essentially performs either the record operation (store API return values and other system input values to external storage) or the record (retrieve recorded values and feed to the APIs whose values were recorded during the earlier execution).

In DroidTest, we instrument three kinds of APIs: Non-deterministic, sensitive source and sink APIs. Based on the type, the record and replay is implemented slightly differently, which is described below.

3.5.2.1 Instrumentation of Non-deterministic API

Non deterministic inputs can cause the variance in the output for multiple execution of a program. Which essentially breaks our assumption that program outputs will be identical if inputs are kept equal. Because, the primary precondition for the proposed method is to eliminate all sources of non-determinism in the target application for multiple executions. Having achieved that, we can reliably attribute changes in the program output to information leakage when only the sensitive input is changed.

To solve this, we have to either fix those inputs to constant values (as done earlier) or record values of non deterministic inputs and retrieve them in subsequence execution of the

program. The former is easier to implement but not sound, because setting a particular non deterministic input to constant values can sometime break the application. For example, if we make system time constant ($System.currentTimeMillis$), then the network connection does not work for some applications. However, the later approach does not suffer from these problems, as we are using actual values (rather than some predetermined constants) for inputs.

For this reason, we have identified a number of sources of non-determinism generally occurring in Android applications. This non-determinism is not generally introduced by the app developer intentionally, but, instead, it is most often part of the legitimate functionality and standard network communication. In Android API library we instrumented three classes: *java.util.Calendar, android.os.SystemClock* and *java.lang.System* for recording time stamp values during the record phase. For masking effects from the randomly generated values *java.util.Random* and
java.security.SecureRandom were instrumented. Table 3.1 lists the corresponding methods in these classes whose implementation was modified to implement the DroidTest approach.

In the record phase, during each record phase we record the value returned by these methods to a file (*AllRecord.txt*) in the *sdcard* folder of the Android emulator. During replay, for each call to these methods we identify which of the recorded values in (*AllRecord.txt*) corresponds to this call. Then we replace the return value of this method with the recorded value instead of executing it again during the replay.

3.5.2.2 Instrumentation of Sensitive Source API

Our approach checks outgoing data to find variance in multiple runs. Hence, we can only change one sensitive input data at a time, perform experiment and then check with the next sensitive input in our list.

Table 3.1: Non Deterministic APIs instrumented in DroidTest

Types	Java/Android Class	Method
Date and Time	java.util.Calendar	getTimeInMillis
	android.os.SystemClock	uptimeMillis elapsedRealtime elapsedRealtimeNanos currentThreadTimeMillis currentThreadTimeMicro currentTimeMicro
	java.lang.System	currentTimeMillis
Random	java.util.Random	next(int bits) nextBoolean nextFloat nextGaussian nextInt nextInt(int n) nextLong
	java.security.SecureRandom	next(int numBits)

The sensitive sources we identify here are device state information (*android.telephony.TelephonyManager*), Location (*android.location.Location*), SMS content (*android.telephony.Sms*
model, product and OS version. (*android.os.Build*). The actual methods instrumented under these classes are listed in Table 3.2.

To test leakage from a particular source, we treat all other methods in this list just like any other library methods and do not instrument them. For the source API in question, we do not do anything special in the record phase and also do not store the values in external storage. However in the replay phase, we add some random constant to the return value of the source API. It implements our idea of providing varied input to the sensitive input in the record and replay execution. The type of the random constant added depend on the type of the return value of that particular sensitive source API.

Table 3.2: Sensitive Source APIs instrumented in DroidTest

Types	Java/Android Class	Method/Field
Phone State	android.telephony.TelephonyManager	getDeviceId
		getSubscriberId
		getLine1Number
		getDeviceSoftwareVersion
Location	android.location.Location	getTime
		getAltitude
		getLongitude
		getLatitude
		getElapsedRealtimeNanos
SMS	android.telephony.SmsMessage	getDisplayMessageBody
		getEmailBody
		getMessageBody
		getEmailFrom
Build Properties	android.os.Build	SDK
		PRODUCT
		MODEL
		SDK_INT

3.5.2.3 Instrumentation of Sink API

Sink APIs (network APIs, APIs to send SMS etc.) are those through which data can be transmitted to outside. Table 3.3 lists the classes and methods from Android framework which are instrumented in DroidTest.

The record and replay operation work slightly differently for sink APIs compared to source API. Here, we actually do not retrieve any stored data during the replay phase. Rather we record the outgoing content for the target sink API in both the record and replay execution. Outgoing data are written in text files in the sdcard and then compared for information leakage detection.

Table 3.3: Data Sink APIs instrumented in DroidTest

Types	Java/Android Class	Method/Field
Network APIs	org.apache.http.client.HttpClient	execute
	org.apache.http.impl.client.AbstractHttpClient	execute
	org.apache.http.impl.client.DefaultHttpClient	execute
	android.net.http.AndroidHttpClient	execute
SMS	android.telephony.SmsManager	sendTextMessage
	android.internal.telephony.SmsManager	sendSms
	andorid.telephony.gsm.SmsManager	sendTextMessage
Output Streams	java.io.OutputStreamWriter	write
	java.io.FileOutputStream	write
	java.io.OutputStream	write
	java.io.Writer	write
	java.io.DataOutputStream	write writeChar writeChars writeByte writeDouble writeFloat writeInt writeLong

3.5.3 Record/Replay Program using Instrumented API

As already mentioned, the proposed system modifies only the Android framework APIs with a view to detecting information leakage. The Android runtime or any other component within the Android operating system is kept intact so that the DroidTest can be usable only if the modified APIs are supplied. This section describes how it achieves this goal. Algorithm 1 defines the steps we take inside the target APIs to instrument them and thereby allow record/replay of execution.

At first, we use the *adb dumpsys* command to identify the main entry point for the application under test. This information is recorded in the sdcard folder when it is installed

Algorithm 1 Instrument API and Record/Replay Non-Deterministic Input Values from Multiple Threads

```
1: procedure InstrumentAPI
2:     curPid ← getPid()
3:     recordReplay ← /sdcard/recordReplay.txt
4:     targetPid ← /sdcard/targetProcess.txt
5:     if curPid = targetPid then
6:         template ← className of API + methodName
7:         RecordThreadInfo (AllRecord.txt, recordReplay, template)
8:     end if
9: end procedure
```

in device. At application startup the pid is recorded using that information and from that point only recorded pid is used to identify the running process. When a particular API of interest is invoked the pid of the calling process in checked. It it matches with the recorded pid then it belongs to the target process/application and instrumented using the proposed approach. The procedure *InstrumentAPI* of Algorithm 1 implement these steps.

It then call the *RecordThreadInfo* procedure to identify which thread the API belongs to and assigns custom thread ids which facilitates correct reproduction of API return values in replay phase. *RecordThreadInfo* method takes 3 parameters: *AllRecord.txt* , *recordReplay* and *template*. *AllRecord.txt* denotes the file where the recorded values will be stored, *recordReplay*; variable representing whether the phase is record or replay and *template*; the concatenated string of API class name and method name. The *template* variable is used for correctly assigning custom thread ids or retrieving already assigned custom thread id.

RecordThreadInfo procedure in Algorithm 2 aims to identify which particular API we are dealing with, the current thread identification, how many times this method was called before this invocation and by which threads etc. Android applications are inherently concurrent. Even the most basic and simplest ones. Because, in Android all GUI activity occurs in main thread, also called the UI thread. Program logic and any other operations

runs outside the main thread to enable smooth user experience. This design make it very difficult to precisely record various events and replay them in the same order if the application is executed again.

However, the problem in hand does not require reproduction of the precise ordering of events. For example, let us consider an example Android application, where a background service leak sensitive data and another thread downloads a file. Both these threads also communicate between themselves and the main thread. Now, to detect leakage by this application and to maintain other program outputs equal during the replay, we only need to correctly reproduce the events: sending sensitive data to outside and downloading the file, not their precise timing and ordering. After determining the custom thread ids *RecordThreadInfo* calls the method $RecordReplayAPIValue$, which actually implements record and replay of the instrumented API values.

Now, we discuss the details of procedure *RecordThreadInfo*. Here, apart from the main thread (thread id = 1), we assign custom thread ids to all the subsequent threads created in the application under test. This scheme has certain advantages. When a new thread is spawned form the main thread, it is not necessarily a linear thread id. As a result, if the application is executed again, the same program may assign a different thread id for the same child thread activation scenario. We solve this by assigning a linear custom thread id (with respect to the target application, not a system wide assignment) corresponding to its actual thread id both in the record and replay phase. As a result, in most cases, same piece of program code has the potential to run using the same custom thread id even in multiple runs of the application. This technique greatly simplifies our record and replay approach.

We create two files named *ThreadMap.txt* and *AllRecords.txt* in the sdcard folder of Android emulator. For each instrumented API, *ThreadMap.txt* is first consulted to check if the current thread id is already recorded. If it is there then the already assigned custom thread id is used (line 13-14 of Algorithm 2), otherwise we increase the value of last as-

Algorithm 2 Record/Replay Instrumented API Values

```
1: procedure RecordThreadInfo(AllRecord.txt, recordReplay, template)
2:     curId ← Thread.currentId
3:     if recordReplay is 0 then
4:         target_file ← /sdcard/ThreadMap.txt
5:     else
6:         target_file ← /sdcard/ThreadMapReplay.txt
7:     end if
8:     flag ← 0
9:     if target_file is empty then
10:        customId ← 1
11:        flag ← 1
12:    else
13:        if template found in target_file then
14:            customId ← recorded_customId
15:        else
16:            customId ← customId + 1
17:            flag ← 1
18:        end if
19:    end if
20:    recData ← curId + customId
21:    if flag is 1 then
22:        echo recData ≫ target_file
23:    end if
24:    if recordReplay is 0 then
25:        rValue ← API return value
26:        entryToBeRecorded ← template_customId_callCount
27:        echo entryToBeRecorded rValue ≫ AllRecord.txt
28:    else
29:        entryToBeRecorded ← template_customId_callCount
30:        rValue ← AllRecord.txt(entryToBeRecorded)
31:    end if
32: end procedure
```

signed custom id by 1 and assign to the current thread (line 16-17 of Algorithm 2). If the *ThreadMap.txt* is found to be empty then assigned custom id will be 1 for the current thread. Again, these assignments are for our record only, actual thread ids are never changed. The same set of actions are repeated for the Replay phase except that thread information are

recorded and retrieved from *ThreadMapReplay.txt* file, which is created at the start of the replay run. So, the information (*recData* in line 20 and 22 of Algorithm 2) stored in either *ThreadMap.txt* or *ThreadMapReplay.txt* consists of two tuple - *Thread id of instrumented API* and *Assigned custom ID by DroidTest*.

Then the *AllRecords.txt* is consulted to see if there is any entry for this call and calculate the *callCount* accordingly. Call count is calculated differently for same method executing in different threads. Based on that we construct the record to be written to files as shown line 26 of Algorithm 2. Finally this information is stored in file for retrieval in replay phase in line 27. During replay, the same strategy is to define the item to be restored (line 29). The corresponding item from the *AllRecord.txt* file is restored and assigned to the return value ($rValue$) of the API.

3.5.4 Information Leakage Detection

For detection of privacy violation, the proposed approach reply on comparing outgoing data recorded by the sink APIs during the record and replay. The comparison operation follows the steps listed in Algorithm 5.

As mentioned in Algorithm 5, after an application is executed twice (one fore record and one for replay), we pull all files that has an extension of $\cdot$ *data* to a local folder for comparison. The output files produced during the record phase starts with the prefix *REC*, and for the files produced during replay the prefix is *REPLAY*.

So, we take all these recorded files and compare one record file with one replay file at a time. For each record file, we get the corresponding replay file just by replacing the *REC* part of the file name with *REPLAY*. For example, if the record file is named as $REC\_AbstractHttpCLient.execute\_1\_1.data$ then the file with which its contents will be compared is $REPLAY\_AbstractHttpCLient.execute\_1\_1.data$, which corresponds to the output generated at the point in the program at replay time. If a mismatch is found a data

leakage event is reported, otherwise it is assumed to be a benign activity, which means no leakage was found in this outgoing data.

Algorithm 3 OutgoingDataEqualityCheck

```
1: procedure EqualityCheck
2:     pull all files with · data extension from /sdcard/output to local folder
3:     n ← number_of_record_files
4:     for i ← 1, n do
5:         rec_file ← recordFile_i
6:         rep_file ← replayFile_i
7:         if content of rec_file ≠ content of rep_file then
8:             LEKAGE_DETECTED
9:         end if
10:    end for
11: end procedure
```

3.5.5 Handling of File Access and Native Code

3.5.5.1 Handling Files

Taint analysis can produce false warnings if a sensitive data is written to a file and later on another part of the file is read. The same happens to TaintDroid [5] also. It assigns a single tag to file. So, even if non sensitive data is read from it, it is regarded as a sensitive operation.

DroidTest does not suffer from this problem. It only checks whether the outgoing data differs in two executions. If sensitive data is not read from a file and not sent outside, then DroidTest will not give any warnings as it is a true negative case. However, special care must be taken to handle the secondary storage files to make sure the environment remains same in two separate executions under DroidTest.

Android restricts each application to a certain area of the secondary storage. If an application's main package name is *com.example.APP*, then it can access only the location

/data/data/com.example.App. The application does not have direct access to other areas of the Android file system. However, all applications can access the external storages if there is any. Generally we can distinguish external storage access by path name starting with */sdcard/...*

Before each run (record and replay), DroidTest needs to initialize some configuration files (e.g. To indicate the current phase:record or replay, to record thread information, to record or replay API return values). To emulate identical environment regarding the files, we take a snapshot of the above mentioned two locations (for our example app: */data/data/com.example.App* and */sdcard/..*). And for each execution, we restore the snapshot by removing any update done by previous execution and restore the files which were present in those locations before the execution.

As a result, any file operation performed to these locations will be exactly same if both executions follow the same program paths and execute same set of instructions.

3.5.5.2 Handling Native Code

As discussed earlier, TaintDroid [5] relies on some heuristics to determine whether the information flow from the sensitive method parameters to the program variables appearing after the native method call. And this heuristics only cover a set of heavily used native methods defined within Android. However, attacker can purposefully use another native method which is not tracked in TaintDroid and can easily miss the leakage.

On the other hand, TaitnDroid also cannot handle user defined native methods. As shown in Listing 4, the *MyNativeFunction* was designed to perform the same operation as *System.arraycopy* and TaintDroid misses this data flow.

DroidTest can handle native code based on its design without any special heuristics. If the native code results in sensitive data transmission, DroidTest can detect that based on output difference. It dose not rely on taint flows to decide whether privacy is violated. As

Listing 4 Limited Handling of Native Code in TaintDroid

```
TelephonyManager tm = (TelephonyManager)
            getSystemService(Context.TELEPHONY_SERVICE);
String device_id = tm.getDeviceId();
String arr1 [] = new String[2];
String arr2 [] = new String[2];
arr1[0] = "nonsensitive";
arr1[1] = device_id;

/* TaintDroid can detect */
System.arraycopy(arr1,0,arr2,0,arr1.length);

/* TaintDroid cannot detect data flow to user-defined native method  */
MyNativeFunction(arr1,0,arr2,0,arr1.length);
```

a result, DroidTest will not miss by native code, not handled by TaintDroid.

3.6 Experiment and Results

3.6.1 Experiment Environment Setup

For DroidTest, we instrumented the Android 4.4.2 framework APIs and built the SDK. The modified SDK was used to fire the Android emulator to perform the tests. Although, we worked with the full Android source tree to build the SDK. Once compiled, the built SDK can be used as a stand alone tool by the testers and developers.

To record user interactions and replay, we used the tool Reran [16]. Once activated, it can record user interactions and various program events. These set of event sequences and program inputs are then replayed using the replay script of Reran which is placed inside the Android emulator. However, the record phase does not automatically generate the test inputs, user has to manually execute the application under test.

However, the task of Reran is to just record the GUI events occurring in the system and various sensor data during the test run. We have chosen Reran over GUI testing tools

(Monkey [17], Dynodroid [18] etc.). Because, the goal of this work is to reproduce user events correctly and exercise the application as much as possible so that we can uncover malicious activities. Many Android applications require logging into a valid account to proceed. It is very unlikely for the random testing tools to generate inputs that can satisfy that criteria. The same goes for other input generation techniques relying on symbolic execution. Frameworks such as Monkeyrunner [19], Robotium [20] support scripting and sending events, but scripting takes manual effort.

Hence, we executed each application for 5 minutes manually and tried to explore as much as the application functionalities as possible. Reran script was used to record the GUI interactions and instrumented APIs recorded the non-deterministic inputs. Then the recorded events from Reran and instrumented APIs were used to replay the execution.

3.6.2 Scope of Sensitive Information Source and Sink

We have discussed about the sensitive information sources earlier. For this study, we only consider the ones listed in Table 3.4. As the sink of sensitive information, we consider the *network interfaces* and *SMS*. However, Our system can be easily extended to work with additional sources and sinks of private data.

Table 3.4: Information sources considered private

IMEI number, IMSI number, Android OS version, Phone Number, Phone Model, Contacts, Location information, Incoming SMS contents

3.6.3 Data Sets

We consider two data sets. The first data set [21] includes the Android applications that are known to contain malwares. The second data set includes applications from the

Official Android Marketplace [9], whose behavior regarding privacy violation is unknown beforehand.

3.6.3.1 Data Set 1: Malicious Applications

The malware database [21] has total 1260 samples under 45 different malware families. Among them, 24 were found to leak users private data [22,23]. So, we also restrict our study to these malware samples.

From these samples, we further discarded malware samples having one or more of the following properties:

1. Perform no internet activity during its execution.
2. Conditions to activate or complete the malicious activity is no longer present. For example, the web server where the stolen information is sent is down, or the emulator need to make a call to premium rate number, which is not possible in an emulated environment etc.
3. Malware data set was based on collected malwares from the wild in 2012 and 2013. Many of the reported exploits no longer work for the newer Android versions with added operating system security features. For examples, some samples (e.g. $DroidKungFu1$,$DroidKungFu$ $GoldDream$) rely on activating a root exploit in the target device, which is very difficult with current Android versions.

As a result of these, malware data set we worked with included *210 samples from 16 malware families* taken from the malware genome project [21].

3.6.3.2 Data Set 2: Android Marketplace Applications

From the Android market place, we select top 50 (based on December 2016 statistics) free applications to perform the testing. While collecting applications, we check whether they require certain permissions: *Full Internet access* and at least one of *Read phone specific*

information, Contact data, and Location information. Applications not having such permissions are discarded from further analysis. Table 3.7 lists the name of these applications along with their leaked information.

3.6.4 Results

3.6.4.1 Experiment Results of Malware Data Set

Table 3.5 shows the result of experiment, that includes the name of the malware family, number of samples in the test suite and the leaked information type. It also includes privacy violation information about these malwares reported in existing studies. Our goal was to match the existing detection results and find additional leakage if there is any.

To calculate accuracy of the DroidTest method, we need to consider false positives rate of the above experiment. Each family has one or more samples, but some family has much more samples than some other family. Some families have only one sample. To remove bias from big families we took one sample from each family in false positives calculation. Out of 16 samples, DroidTest produced false positives for 3 applications, which is roughly 20%.

Next, we compare our findings with the state of the art TaintDroid [5]. For the comparison, we use the same set of malware samples (1 per each family). TaintDroid tracks a slightly different set of sensitive data. So, we only consider the the set of sensitive sources which are tracked both in DroidTest and TaintDroid. In Table 3.6, DroidTest leakage is shown using ✓ and TaintDroid findings are denoted by $\times$.

3.6.4.2 Experiment Results of Android Market Place Applications

Next, we apply the proposed method to test the *top 50 free applications* from the Android market place [9] (Based on December 2016 statistics). Table 3.7 lists the name of these applications alongside the type of information leaked. The value *Nil* indicates that we

Table 3.5: Android Malwares and Types of Leaked Information

Malware Samples			
Malware Family	**Number of Samples**	**Leaked Information (DroidTest)**	**Existing Knowledge**
ADRD	22	IMEI OS Version	IMEI OS Version
BeanBot	4	IMEI, IMSI OS version Phone number	IMEI, IMSI Phone number
BgServ	4	IMEI Phone number	IMEI Phone number
CruiseWin	1	IMEI	IMEI
DroidDream	12	IMEI OS version Phone model	IMEI, IMSI OS version Phone model
DroidDreamLight	35	IMEI, IMSI OS version Phone model	IMEI, IMSI OS version Phone model
DroidKungFu4	24	IMEI	IMEI Phone model
DroidKungFuSapp	3	IMEI, OS version, Phone model	IMEI Phone model
GingerMaster	4	IMEI, IMSI Phone number Phone model	IMEI, IMSI Phone number
Gone60	9	Contact	Contact
JSMSHider	16	IMEI, IMSI OS version Phone model	IMSI Phone number
Plankton	11	IMEI	IMEI
PjApps	44	IMEI	IMEI, IMSI Phone Number
RougeSP Push	9	IMEI OS version	Premium rate SMS
Walkinwat	1	IMEI	IMEI
zHash	11	IMEI, IMSI	IMEI, IMSI

Table 3.6: Malware Data Set Comparison with TaintDroid

Application	**Leaked Information**				
	IMEI	IMSI	Phone Number	Location	Contact
ADRD	✓×				
BeanBot	✓×	✓×	✓×		
BgServ	✓×		✓		
CruiseWin	✓×				
DroidDream	✓×				
DroidDreamLight	✓×	✓×		✓	
DroidKungFu4	✓×				
DroidKungFuSapp	✓×				
GingerMaster	✓×	✓×	✓×		
Gone60					✓×
JSMSHider	✓×	✓			
Plankton	✓×				
PjApps	✓×	×	×		
RougeSP Push	✓×				
Walkinwat	✓×				
zHash	✓×	✓			

Detected by : DroidTest (✓) , TaintDroid (×)

didn't find any leakage of information during the testing.

From the experiment, we see that some highly rated applications can also leak private data without user's consent. The permission requests presented by these applications had no direct mention about their sending of private information to third party. We found that, *35 out of 50 applications (70%)* send one or more sensitive data to outside, mostly to advertising servers. This number is quite alarming taking into the fact that, these applications are very much popular and widely used [9].

Among the 50 apps, DoridTest was found to produce false positives in 11 cases, making its false positive rate 22%.

Then we compare our findings with TaintDroid [5]. Like the malware data set, we ran the applications in our data set in TaintDroid system and the leakages found are reported

Table 3.7: Google Play Applications and Leaked Information

GooglePlay Apps	**Leaked Information**
DU Battery Saver, Hola Launcher, Ali Express, Yellow Pages, Hearts Free	IMEI
Coupons, Horoscope, Speedtest, Solitaire	IMEI, Location
Crackle Movie	OS version, IMSI
Brightest LED Flashlight AccuWeather Cut the Rope	Phone model, Location OS version
Dictionary, Wish Imdb, Logo Quiz, CL Mobile, NotePad, Zedge, Zillow	OS version, Phone model
Sudoku	Location, Phone model
Evernote	Location, Phone number
Walmart	OS version
DH Texas Hold Em Poker,	IMEI, OS version, Location, Phone model
Tune in Radio, News Break Zombie Frontier SPhotoEditor, ShootBubble, Power Battery Go Weather EX, Espn	OS Version,IMEI, Phone model
Antivirus Free	Location
Instagram, BBC, Google Translate, Ant Smasher, Flow Free, Credit Karma, BitMoji, Yahoo Mail, WebMD, Messenger, Alarm Clock Extreme, WikiPedia, Pandora, Paypal, Barcode Scanner	NILL

in Table 3.8.

Table 3.8: Google Play Store Data Set Comparison with TaintDroid

Application	**Leaked Information**				
	IMEI	IMSI	Phone Number	Location	Contact
DU Battery Saver, Ali Express	✓×				
Yellow Pages, Hearts Free	✓×			×	
Hola Launcher	✓				
Coupons, Horoscope, Speedtest, Solitaire DH Texas Hold Em Poker	✓×			✓×	
Paypal	×				
Crackle Movie		✓			
Brightest LED Flashlight				✓	
Sudoku				✓	
Antivirus Free		×		✓×	
Evernote			✓×	✓×	
AccuWeather, Cut the Rope				✓×	
Tune in Radio, Zombie Frontier SPhotoEditor, Power Battery Espn, News Break	✓×				
ShootBubble, Power Battery, Go Weather EX	✓				
CL Mobile, Zillow, NotePad, Zedge, Dictionary, Wish, Instagram, BBC, Imdb, Logo Quiz, Free, Credit Karma, Google Translate, Barcode Scanner, Ant Smasher, Flow Free, BitMoji, WebMD, Messenger, Yahoo Mail, Alarm Clock Extreme, Walmart, WikiPedia, Pandora					
Detected by : DroidTest (✓) , TaintDroid (×)					

3.6.5 Discussion

We tested the applications by exploring them manually. So, it may miss some instances of true positives, as the set of test cases is not comprehensive. Detection rates of malicious applications reported above are also based on the generated test cases only.

From Table 3.5 and Table 3.7, we can see the type of information leaked by various applications. The most commonly leaked information is the unique device id or the IMEI number by both type of applications. If we closely observe the results of analysis of these applications, we can easily see that, applications from the malware data set leaks IMEI, IMSI, Phone number more than the other sensitive data. On the other hand, the mostly leaked information by the Android market place applications include Android operating system version, Phone model which are definitely not as sensitive as the device id (IMEI), subscriber id (IMSI) and the phone number. So, user must avoid downloading applications from the untrusted third party market places. The chance of leaking sensitive data are much higher in case of the unofficial market place applications, because unofficial market places apply hardly any security check of the applications.

Comparison with existing knowledge about malwares in Table 3.5 reveals that in some cases, we could not detect all the reported behavior. It can happen for a number of reasons. For example, in case of *RogueSP Push*, its main exploit is to register users with the premium rate SMS service. However, DoridTest detects leakage information leakage based on observed output differences. It does not distinguish between a regular phone number and premium rate number as it cannot see difference in output for that transmission. Apart from that, some applications read device information from a system configuration file instead of using dedicated APIs. As we track sensitive data access by instrumenting corresponding APIs, DroidTest cannot detect those leakages.

Out comparisons with TaintDroid [5] findings for the same data sets reveal that DroidTest can effectively detect leakage of sensitive data. However, based on the comparisons

shown in Table 3.6 and Table 3.8, we found some leakages not reported in TaintDroid and in other cases missed some warnings raised in TaintDroid. In these tables, we only included information sensitive sources tracked both in DroidTest and TaintDroid. As DroidTest do not look at the source code for analysis, we discuss some factors which may have the reasons behind the differences in detection results.

DroidTest detect leakage if it sees difference in output. So, it does not suffer form the program of over tainting as discussed in Listing 1 and 2 in chapter 1. It also does not suffer from the over tainting problem of files in TaintDroid and have a much better handling of native methods. On the other hand, DroidTest cannot detect track flow of sensitive data between components of a two different applications. As mentioned earlier, if device specific information is read from a system configuration file, which is tainted in TaintDroid, DroidTest can miss those leakages.

Finally, we also compared our findings with the *"Application Verification Service"* introduced in Android 4.2 (JellyBean). Performance of this service in detecting malicious applications has been studied in [24]. Based on that, Table 3.9 shows the comparison of the results found by *DroidTest* and *Application Verfication Service*. It clearly shows that, the proposed system performed way better than App Verification Service. This comparison was done in 2013 (using Android 2.2) during our earlier study. So, some applications reported here later discarded as newer Android versions could not run many of those samples.

3.7 Summary

The growing popularity of smartphones has led to the rising threats from malicious mobile applications. In this chapter, we have demonstrated the need for testing Android applications before they are put into the market place. It comes from the fact that, many popular applications found in the official and unofficial Android market places leak private data to some third parties without proper user consent. To test mobile applications for data

Table 3.9: DroidTest vs Google App Verfication Service

Malware Samples and Detection Results		
Malware Family	**Detected Samples**	
	DroidTest	**App Verification Service**
BeanBot	4	0
BgServ	4	0
DroidDelux	1	0
DroidDreamLight	30	18
DroidKungFu1	16	8
DroidKungFu2	8	9
DroidKungFuSapp	3	0
DroidKungFuUpdate	1	0
GoldFream	34	6
Gone60	9	0
LoveTrap	1	0
Plankton	11	2
PjApps	47	8
RougeSP Push	9	0
SMSReplicator	1	0
SndApps	10	0
WalkinWat	1	0
YZHC	22	3
zHash	9	1
Zitmo	1	0
Total	222	56

stealing behavior, we have proposed the DroidTest system and described its architecture, operation and evaluation through the testing of two data sets. The experiment results shows its effectiveness in detecting private information leakage.

Currently our system is an offline testing tool. In future, we plan to move the monitoring to the device to notify user about information leakage dynamically. We also plan to include more information sources in the sensitive input list. This will make DroidTest better equipped to detect zero day smartphone malwares. Currently, we manually generate the test cases for the applications under test. To make our system more robust, we also plan

to generate meaningful test cases for Android applications automatically in future.

CHAPTER 4

MirrorDroid: A Framework to Detect Sensitive Information Leakage in Android by Duplicate Program Execution [1]

4.1 Introduction

Although 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 dynamic analysis techniques have limitations also such as taint explosion, high runtime overhead [5], application breakage due to mocking of sensitive data [25] etc.

To solve the above mentioned problems, this chapter 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 with slightly modified information (e.g., actual input

[1] Copyright 2017 IEEE. Reprinted, with permission, from Sarker Ahmed Rumee, Donggang Liu, Yu Lei MirrorDroid: a Framework to Detect Sensitive Information Leakage in Android by Duplicate Program Execution (CISS), March 2017.

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 are described in detail in the remainder of this chapter.

Realizing this strategy involves addressing a number of challenges: allocation and deallocation of additional (mirror) data structures in memory, selective tracking of library functions (to keep the memory overhead low), track flow of information between application components, parallel execution of native code, correctly duplicating the execution of multi-threaded applications etc. These cases are critical for the correctness and effectiveness of the proposed method and if not handled properly, the system will either crash for duplication of execution or produce outputs indicating information leakage even if there is no such cases.

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 chapter is organized as follows. Section 4.2 introduces some essential terminologies and assumptions made in this work. Section 4.3 describes the high level overview of the proposed approach. Next, section 4.4 provides detailed description of implementation. Experimental results and performance measure of the proposed system are discussed in section 4.5. The last section concludes the chapter and discusses some possible future directions.

4.2 Definitions and Assumptions

4.2.1 Definitions

Following are the few key words used repeatedly in rest of this chapter:

- *Sensitive Data* : Information source 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. Mirror variable has same type as its actual counterpart and accessed at the same time the program variable is accessed.
- *Non Deterministic Input* : Source of data (API) whose value can be different in multiple execution without any change in any program input or environment. For example, value returned by the random number generator or system time functions.

4.2.2 Threat Model

Here, 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 assume that the application has the permission to read some of the sensitive information sources and access Internet. 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 easily bypass our monitoring.

Sensitive Information can be leaked from mobile devices in many ways. Such as theft of device, leakage of automatically synced and backed up cloud data, shoulder surfing attack or malicious key loggers etc. Here MirrorDroid can only cover cases where sensitive data is accessed through a library API used in a program and later passed to a network interface or

SMS by the same application through another API. Other means for information leakage is out of scope of this work.

4.3 MirrorDroid Overview

In MirrorDroid we essentially use the same approach as presented in DroidTest (Chapter 3). Here, we move our duplicate execution of a program to Dalvik VM and make it parallel rather than two separate execution. For a brief recap, the underlying approach is stated below again:

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 $f(x_1) = f(x_2)$. 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 4.1 depicts a high level description of our system based on this approach.

It shows an example scenario of an application consisting of 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 side of Figure 4.1) 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 $send\_message()$.

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 different 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

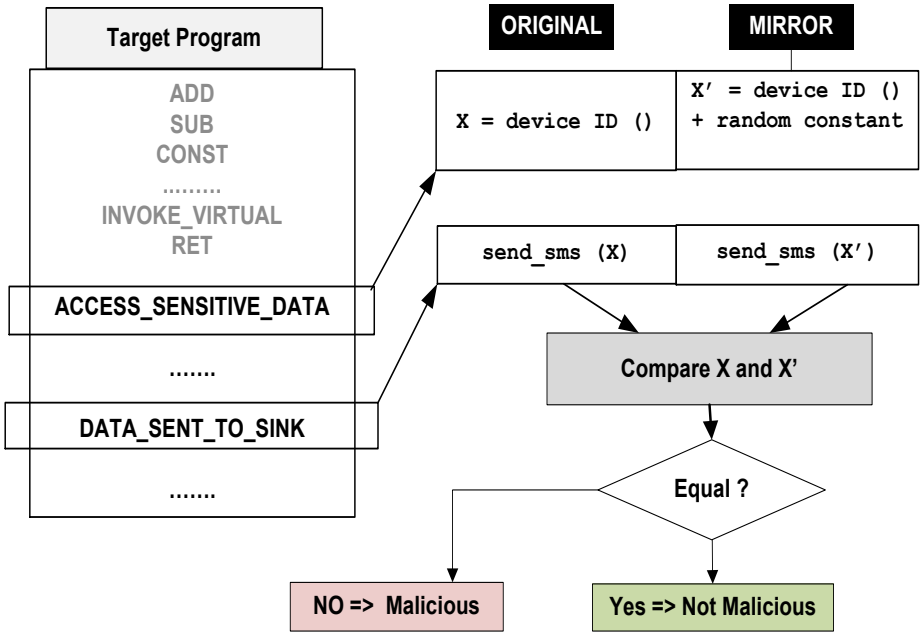

Figure 4.1: MirrorDroid Approach

outgoing data is believed to be benign.

4.4 MirrorDroid Design

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.

4.4.1 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 in the heap. As a result, in parallel to the actual program stack, we have a *mirror stack* containing the mirror frames. However, to reduce memory overhead, we only list a few member of the actual method frame structure defined in Android [26]. The mirror frame only contains *method name, class name, mirror registers, position in mirror stack*. This design of mirror method frame does not violate the goal of the proposed method and helps to reduce the overhead from the duplicate execution.

Table 4.1: Duplication of Dalvik instructions in MirrorDroid

Instruction Format	Actual Operation	Mirror Operation
$const - op\ V_a, C$	$V_a \leftarrow C$	$mV_a \leftarrow C$
$move - op\ V_a, V_b$	$V_a \leftarrow V_b$	$mV_a \leftarrow mV_b$
$unary - op\ V_a, V_b$	$V_a \leftarrow \otimes V_b$	$mV_a \leftarrow \otimes mV_b$
$binary - op\ V_a, V_b, V_c$	$V_a \leftarrow V_b \otimes V_c$	$mV_a \leftarrow mV_b \otimes mV_v$
$binary - op\ V_a, V_b$	$V_a \leftarrow V_a \otimes V_b$	$mV_a \leftarrow mV_a \otimes mV_b$
$binary - op\ V_a, V_b, C$	$V_a \leftarrow V_b \otimes C$	$mV_a \leftarrow mV_b \otimes C$
$aput - op\ V_a, V_b, V_c$	$V_b[V_c] \leftarrow V_a$	$mV_b[V_c] \leftarrow mV_a$
$aget - op\ V_a, V_b, V_c$	$V_a \leftarrow V_b[V_c]$	$mV_a \leftarrow mV_b[V_c]$
$sget - op\ V_a I_b$	$V_a \leftarrow sI_x$	$mV_a \leftarrow sI_x$
$sput - op\ V_a I_b$	$sI_x \leftarrow V_a$	NIL
$iput - op\ V_a, V_b, I_c$	$V_b.I_c \leftarrow V_a$	$mV_b.I_c \leftarrow mV_a$
$iget - op\ V_a, V_b, I_c$	$V_a \leftarrow V_b.I_c$	$mV_a \leftarrow mV_b.I_c$

Theoretically, MirrorDroid should have a 100% memory overhead as the mirror execution creates the same number of program variables (arguments and local variables) as the original execution does. However, we apply a heuristic to reduce this memory requirement significantly.

From our experiment, we found that depth of a certain method call sequence before making a return hardly crosses 30. So, at the application startup, we actually created a linked list of 20 mirror method frames with dummy data. Each method frame had a storage

for 65,536 register locations which is the highest possible allowed per method. Later, when a particular method is invoked its actual variable contents replaced the dummy data. Upon return from the method, it just changed the current access pointer to the previous frame and the current frame becomes a dummy method frame again.

In case, the number of live methods in a method call sequence become more than 30, we resort to the traditional way of dynamically allocating a mirror frame alongside the actual method frame. So, this heuristic is both sound and complete in its design and works in every cases.

4.4.2 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. 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. 4.1 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 assign 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.

```
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
```

Listing 4.1: Handling OP_CONST in MirrorDroid

4.4.2.1 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 4.1 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 4. 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 4 describes various strategies 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

Algorithm 4 Handle Method Call Instruction

```
1: procedure INVOKE foo(x_1, x_2, ..., x_n)
2:     flag ← 0
3:     for i ← 1, n do
4:         if x_i ≠ x_i' then
5:             flag ← 1
6:         end if
7:     end for
8:     if isSensitiveSource(foo) = 1 then
9:         Invoke foo(x_1, x_2, ..., x_n)
10:        mRetval ← retval + random_constant
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
14:    else if isNative(foo) = 1 then
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 ← retval
21:        end if
22:    else if isNoTrackMethod(foo) = 1 then
23:        Invoke foo(x_1, x_2, ..., x_n)
24:        mRetval ← retval
25:    else if flag = 0 and islibMethod(foo) = 1 then
26:        Invoke foo(x_1, x_2, ..., x_n)
27:        mRetval ← retval
28:    else
29:        Invoke foo(x_1, x_2, ..., x_n, x_1', x_2', ...., x_n')
30:    end if
31: end procedure
```

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 4), duplication is not necessary . The detail of this comparison for equality check is described in Algorithm 5.

The comparison of actual and mirror method parameters is not straight forward. Because, often the parameters are complex objects which contains reference types and those reference variables can also contain references and so on. So, we developed a special program named *EqualityChecker*, which compares two object variables recursively. As our monitoring scheme is within the virtual machine, it cannot see at runtime the type of an object. So, every object is cast to the *Object* type, which is the base super class of all objects. Algorithm 2 highlights our approach for object equality comparison. If any of the arguments and its mirror version are found to be different, it will return 1. If all arguments in the original and duplicate execution are equal, then it will return 0.

For primitive types, a direct comparison is done between the actual and mirror arguments. If it is of non primitive type (*assumed to be of Object type*), then the *EqualityChecker* is invoked repeatedly for all the non primitive arguments and their corresponding mirror variables.

Algorithm 5 Equality Comparison of Actual and Mirror Method Arguments

```
1: procedure EqualityCheck(x, x`)                    ▷ x` = mirror value of x
2:     flag ← 0
3:     n ← Numberoffields in x
4:     for i ← 1, n do
5:         if x_i is Primitive then
6:             if x_i ≠ x`_i then
7:                 flag ← 1
8:                 return flag
9:             end if
10:        else
11:            flag ← EqualityCheck(x_i, x`_i)
12:            if flag ≠ 0 then
13:                return flag
14:            end if
15:        end if
16:    end forreturn flag
17: end procedure
```

After that, 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 4) with actual and mirror method parameters respectively. If satisfied, based on the type of method being invoked MirrorDroid performs various tasks as discussed below.

4.4.2.2 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.

As discussed earlier, TaintDroid [5] relies on some heuristics to determine whether the information flow from the sensitive method parameters to the program variables appearing after the native method call, hence there is a potential of producing false negatives for the conditions they miss. However, we execute the native methods twice, with the actual and the mirror variable. As a result, any changes made by the native function will be visible in both the executions.

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 be copied to the corresponding mirror execution (line $14-21$ of Algorithm 4). Let us consider the example shown in Listing 5 to better understand the proposed policy.

Listing 5 Handling of Native Code in MirrorDroid

```
1  TelephonyManager tm = (TelephonyManager)
2              getSystemService(Context.TELEPHONY_SERVICE);
3  String device_id = tm.getDeviceId();
4  String arr1 [] = new String[2];
5  String arr2 [] = new String[2];
6  arr1[0] = "nonsensitive";
7  arr1[1] = device_id;
8
9  /* arr2 gets tainted according to TaintDroid heuristics */
10 System.arraycopy(arr1,0,arr2,0,arr1.length);
11
12 /* Done in mirror execution, not visible to user */
13 System.arraycopy(mirror_arr1,0,mirror_arr2,0,arr1.length);
```

Here, the native method *System.arraycopy* is used to copy contents of *arr1* to *arr2* (line 10 of Listing 5). For this native call, in mirror execution we invoke the same native function again before fetching the next instruction. However, the second call involves the mirror version of the parameters used in original invocation (line 13 of 5). As a result, the effects of native method code are visible to both the actual and mirror parameters.

4.4.2.3 Selective Tracing of Android Library API

Android applications call hundreds of library APIs as part of its operation. 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.

Some APIs are required to draw various GUI elements on the screen, refreshing and resuming the GUI components, thread scheduling, exception handling etc. As the mirror execution follows the same program path as the original execution, the duplicate execution of such methods is unnecessary.

For this, we form a list of such methods named $no-track-list$ (Table 4.2) 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 4).

Table 4.2: 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*

Apart from the $no-track-list$ methods , we can also avoid executing a method twice if it is a library method and the method parameters and their mirror versions are identical ($flag = 0$ in line 25 of Algorithm 4). In that case also, no sensitive data is involved and we can safely copy the return value of the method to the mirror execution without executing it again (line 25-27 of Algorithm 4). Otherwise, we cannot ignore tracing the method and handle those like any other program methods.

4.4.2.4 Handling InterComponent Communication

Android intercomponent communication is controlled by a special system process called the *Binder IPC*, which is not tracked by the MirrorDroid. However mirror version of the program arguments passed between components must be retrieved for correctness.

MirrorDroid records the following set of data items for each intra-process communication between two components: (*process_id,thread_id, validity,*arg_1*,* arg_2*,...,*arg_n). When a new component starts execution, we check this record to see if the current component registers are listed and valid (validity field is true). In that case, we copy the mirror variables

from this record and mark the validity field as false. To prevent unnecessary growth of this record and to make searching faster, invalid entries (validity = false) are periodically checked and removed.

If the newly initiated component is not found in the above mentioned record, we initiate the local variables and its mirror versions as identical for this component. This will ensure the goal of MirrorDroid design to keep a data item and its mirror equal if no information leakage is involved.

4.4.2.5 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 the non-deterministic APIs we track in MirrorDroid, please refer to Table 3.1 of section 3.5 in Chapter 3.

For example, the *nextInt(n)* method of *Random* class in Android returns a pseudo random number 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. Essentially this employs the same approach as mentioned for selective tracing of library APIs.

The actual and mirror executions always follow the same program path, even in cases such: exceptions, thread switching, secondary storage access etc. So, we omit executing the non-deterministic methods twice. It ensures that these APIs create no difference in actual and mirror execution as seen in DroidTest, where applications were executed separately. As a result, handling of non-deterministic methods is straightforward in MirrorDroid.

4.4.2.6 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 4). For the source APIs we track in MirrorDroid please refer to Table 3.2 of section 3.5 in Chapter 3.

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 (Line 11-13 of Algorithm 4). For the sink APIs we track in MirrorDroid, please refer to Table 3.3 of section 3.5 in Chapter 3.

The information leakage events detected are recorded in the Android System Logs. The record contains the process name, the destination address for a particular event of privacy leakage. MirrorDroid is accompanied with a service program *MirrorDroidNotifier*, which reads Android log buffer to find MirrorDroid specific messages corresponding to information leakage and sends pop up notification to user describing the event.

4.5 Experiment and Results

4.5.1 Experiment Environment Setup

Testing of applications were done on Android emulator running OS version 4.4.2 (kitkat). The base Android source code was first downloaded and later patched with proposed modifications to build the modified operating system, namely MirrorDroid. The idea can be easily applied to other Android versions as well.

Table 4.3: Android Malwares and Types of Leaked Information

Malware Samples			
Malware Family	**Number of Samples**	**Leaked Information (MirrorDroid)**	**Existing Knowledge**
ADRD	22	IMEI OS Version	IMEI OS Version
BeanBot	4	IMEI, IMSI OS version Phone number	IMEI, IMSI Phone number
BgServ	4	IMEI Phone number	IMEI Phone number
CruiseWin	1	IMEI	IMEI
DroidDream	12	IMEI OS version Phone model	IMEI, IMSI OS version Phone model
DroidDreamLight	35	IMEI, IMSI OS version Phone model	IMEI, IMSI OS version Phone model
DroidKungFu4	24	IMEI	IMEI Phone model
DroidKungFuSapp	3	IMEI, OS version, Phone model	IMEI Phone model
GingerMaster	4	IMEI, IMSI Phone number Phone model	IMEI, IMSI Phone number
Gone60	9	Contact	Contact
JSMSHider	16	IMEI, IMSI OS version Phone model	IMSI Phone number
Plankton	11	IMEI	IMEI
PjApps	44	IMEI	IMEI, IMSI Phone Number
RougeSP Push	9	IMEI OS version	Premium rate SMS
Walkinwat	1	IMEI	IMEI
zHash	11	IMEI, IMSI	IMEI, IMSI

Currently we consider: *IMEI (Device Id) number*, *IMSI (Sim Serial) number*, *Contacts*, *Location*, *SMS*, *OS version*, *Phone number* and *model* as the *source* of sensitive data. And for the *sink*, we consider *outgoing network data* and *SMS*. Both IMEI and IMSI are unique to a device and if leaked, can be used to replicate the user's identity through attacker controlled devices. Apart from this, in most cases, users also do not want their phone numbers and contact list to be released to unintended parties. We listed the Phone model and OS Version has sensitive data here, because it can also be used to launch attacks exploiting application environment specific properties. However, 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.

To track an application in MirrorDroid, user has to select the target application before running it. For that, MirrorDroid comes with a helper application which lists the currently installed applications in the system. Users must select an application from that list, which allows the modified Dalvik VM of MirrorDroid to identify which application to execute in parallel with the mirror execution.

4.5.2 Data Sets and 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.

4.5.2.1 Malware Data Set

The first data set [21], has total 1260 samples under 45 different malware families. Samples in this set, perform various malicious activities. However, not all of them leak sensitive data [22,23]. Apart from that, some malware families in this collection (*e.g.* DroidKungFu1, DroidKungFu2, GoldDream, DroidDelux) can no longer be run in operating system version

higher than 2.3, hence discarded from our study. Taking these into considerations, our experiment includes 210 samples of 16 different malware families.

Table 4.3 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 4.3 match with the known behavior of these malwares from the literature [22, 23] in most cases. But in some cases, it could not reproduce known leakage. For example the *RougeSP Push* malware is known to register users with premium rate SMS service. However, the MirrorDroid design attempts to find if the outgoing data contains sensitive information or not. But, it cannot differentiate between whether a target number represents a premium rate service or not as there will be no difference in actual and mirror execution for that transmission. Similarly, some high privileged applications read device information from a system configuration file instead of using dedicated APIs. In that case also, MirrorDroid cannot detect leakage as it relies on tracking access to sensitive source APIs. Last of all, MirrorDroid cannot track inter-application communications at present which may also contribute to some of the leakages missed as reported in Table 4.3.

4.5.2.2 Google Play Store Applications

The second set of applications includes 50 popular Android applications selected randomly from the top free 500 applications of Google Play Store [9]. Table 4.4 lists these applications alongside the type of information leaked found in our study. Like Table 4.3, it also groups applications leaking similar type of data together. Those who did not send any private data (only those we track), are shown in the last row and their behavior is listed as *NIL*.

Here, we can see that out of 50 applications, 35 leak some kind of data. We later extracted the source code of these applications for verification purposes and found that all the instances reported here actually sensitive data flow from the source to sink, which proves the effectiveness of our method. It is to be noted that we only monitored the flow of certain sensitive information (described earlier in Experiment Setup) to sink (internet, SMS). So, if an application leak other kind of information or send data to another application via IPC channel and leak from that application, MirrorDroid cannot track those particular flows.

4.5.3 Discussion

For the malware data set and Google play store applications, we found the same leakage as we did in DroidTest. Hence, comparisons with TaintDroid findings are not shown here to avoid repeatation. Please refer to Results section (Section 4.5) of chapter 4 to find TaintDroid detection results for the same data sets.

In MirrorDroid, the non deterministic functions and its duplicate versions run in parallel and handle program variables with identical values. The non determinism from thread scheduling is also not a factor here, because the original and the mirror execution both follow the same order of thread executions in a multi threaded execution environment. As a result, the chances of false positives are much lower in MirrorDroid compared to DroidTest systems. On data sets of 16 malware samples and 50 Google play store apps, MirrorDroid produces 6.25% and 6% false positives respectively.

MirrorDroid cannot track whether sensitive data is leaked from a file. As the parallel (or mirror) execution follows exactly same program paths and execute same instructions, we omit any file write operation from the duplication process. Such operations are performed only once and results are used in both the actual and mirror execution. It also cannot handle flow of sensitive information passed to or coming from another application. So, if a leakage occurs from these sources MirrorDroid cannot detect that and thereby produce false

Table 4.4: Google Play Applications and Leaked Information

GooglePlay Apps	**Leaked Information**
DU Battery Saver, Hola Launcher, Ali Express, Yellow Pages, Hearts Free	IMEI
Coupons, Horoscope, Speedtest, Solitaire	IMEI, Location
Crackle Movie	OS version, IMSI
Brightest LED Flashlight AccuWeather Cut the Rope	Phone model, Location OS version
Dictionary, Wish Imdb, Logo Quiz, CL Mobile, NotePad, Zedge, Zillow	OS version, Phone model
Sudoku	Location, Phone model
Evernote	Location, Phone number
Walmart	OS version
DH Texas Hold Em Poker,	IMEI, OS version, Location, Phone model
Tune in Radio, News Break Zombie Frontier SPhotoEditor, ShootBubble, Power Battery Go Weather EX, Espn	OS Version,IMEI, Phone model
Antivirus Free	Location
Instagram, BBC, Google Translate, Ant Smasher, Flow Free, Credit Karma, BitMoji, Yahoo Mail, WebMD, Messenger, Alarm Clock Extreme, WikiPedia, Pandora, Paypal, Barcode Scanner	NILL

negatives.

MirrorDroid also has the potential to produce false negatives in the case of sensitive data being flown to another application. We do not track such data flow at present. As a result, a proper estimate of such cases was not possible during the testing.

4.5.4 Performance Evaluation

The memory overhead of MirrorDroid is discussed in detail earlier in the Implementation section. Here, we focus on the runtime overhead of MirrorDroid's duplicate execution strategy. 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. Unlike TaintDroid [5], we only track any application which is installed by the user and omit tracking stock applications and system processes. As a result, MirrorDroid does not create extra overhead for the operations like making phone calls, taking pictures, adding new entry in the contacts etc, unless performed by the target application.

The standard CaffeineMark 3.0 for Android [27] was used for overhead calculation. State of the art TaintDroid [5] also uses the same benchmark application to measure runtime overhead. CaffeineMark scoring is based on the number of Java instructions executed per second. The overall result is a cumulative score across individual benchmarks. The experiment results was shown in Figure4.2. Here the x-axis represents the bench mark program within CaffeineMark and the y-axis represents score (the higher the better).

The unmodified Android (*version* 4.4.2) system had an average score of 585.0 in our experiment setting (using portable interpreter). MirrorDroid, was measured at 541.0 overall, resulting in a 8.2% overhead with respect to the unmodified system, compared to 14.0% overhead produced by TaintDroid. The benchmark program was run for 30 times each for the Android and MirrorDroid system and the error bars in the Figure4.2 indicates the 95%

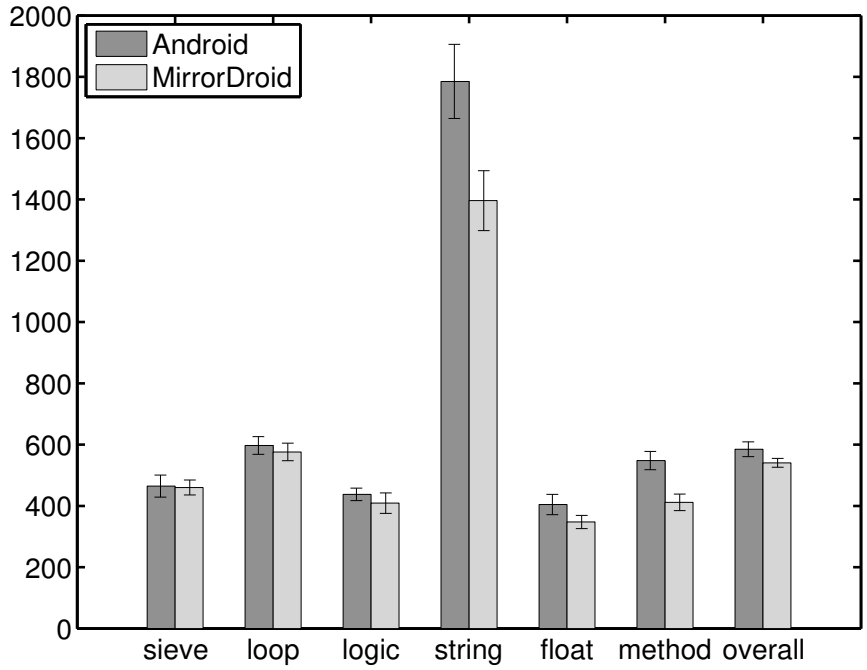

Figure 4.2: Performance of MirrorDroid compared to Stock Android

confidence intervals.

4.6 Summary

In this chapter, we discussed 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.

At present MirrorDroid cannot track sensitive data flow to and from the secondary storage or files. The duplicate execution model of MirrorDroid makes it difficult to parallelize file access operations (read, write and append). In Future, we plan to apply taint tracking for access to files only to solve this issue and combine with the proposed duplicate execution strategy. We also plan to work on adapting the proposed idea to the new Android Runtime (ART). We also plan to add the following features to the MirrorDroid system : track intercomponent communication between two separate processes and include more sen-

sitive information sources and sinks as part of the MirrorDroid tracking system to make it better equipped to prevent sensitive data disclosure.

CHAPTER 5

Related Work

Since its adoption in 2008, Android security has got a lot of attention. Researchers have come up with many different solutions aiming at securing both the Android Operating System and the third party applications. In this chapter we discuss the related work to detect privacy violation in Android systems. We also discuss approaches that can record user interactions and inputs and replay them in subsequent executions, which is a part of the proposed *DroidTest* system.

5.1 Privacy Leakage Detection

The analysis techniques aiming at protecting user privacy can be broadly classified to Static and Dynamic analysis. Some combine both and commonly termed as hybrid approaches. Apart from them there are other techniques like machine learning based approaches, formal methods and symbolic execution, fingerprint matching etc. The proposed systems in this thesis involve runtime monitoring of the Android application, so our discussions will put more emphasis on related work using dynamic analysis.

5.1.1 Static Analysis

Static analysis involves looking at the application source code, bytecode (either Android bytecode or modified bytecode like smali, jimple etc.) and finding program paths that perform a malicious or undesired activity. It aims at examining all possible paths an application can take during its execution. As a result, static analysis based approaches have the potential to find more vulnerabilities. But the downside is, applications are not run for

analysis and often misses many runtime conditions producing a lot of false positives. For example, a particular program path may not be executed at all during execution yet produce false warnings.

Quite a number of recent work has been proved to be very successful in finding vulnerabilities within Android applications with high precision. Existing static analysis based solutions can be differentiated based on the analysis techniques they adopt: taint analysis, analyzing inter-component flow and call graphs, etc. In reality, most of the work employs a combination of these techniques.

There is a wide body of work to track flow of information whose basis is static taint analysis [28], [29], [30], [31], [32], [33], [31], [32], [34], [34], [35], [36] etc.

AndroidLeaks [28] tracks data flow at the object-level instead of tainting their individual fields. FlowDroid [29] identifies possible sources of private information and determines how their values flow from sensitive sources to sinks(e.g. SMS,network). However, it performs taint analysis within single components of Android applications and misses intercomponent communication.

On the other hand, Appintent [37] uses symbolic execution to find a set of execution paths involving the flow of sensitive information. Appintent generates a sequence of GUI events for the sequence of events corresponding to a data transmission. It helps the market place owners and analysts with valuable information to decide if the data transmission is user intended or not. But, this technique needs a lot of user intervention for detection of information leakage.

Approaches like EdgeMiner [9], LeakMiner [30] etc., address the implicit control flows, and Amandroid [38], IccTA [39], handle inter-component instead of just intra-component information leaks. MorphDroid [40] tracks individual data units within sensitive data instead of the complete information (i.e., longitude and latitude instead of the location as a whole).

CHEX [41] aims to detect component hijacking vulnerabilities in Android applications. It at first, finds all code sections reachable from an entry point and then permutes the identified code segments to find intercomponent data flows.

Epicc [42] developed a basis for connecting multiple components in case of static analysis approach by following some heuristics. But it also has the same limitation as CHEX and often over-approximates the connections.

5.1.2 Dynamic Analysis

Dynamic analysis involves tracking or monitoring an application while it is running. The analysis depends on collecting runtime information and analyzing those information to determine application behavior. Following are some of the well known techniques applied by the researchers to dynamically monitor Android applications for privacy violation.

5.1.2.1 Taint Analysis

TaintDroid [5] is a system wide taint tracking mechanism. Like *MirrorDroid*, it also modifies the Dalvik Virutal Machine (VM) for security monitoring. It associates a tag with each variable in memory. If a variable contains sensitive data (either by accessing sensitive data directly or by anther variable already tainted), then certain bits of the tag are turned on to give a taint to it. By this way, it tracks flow of certain sensitive data within the program and if such tainted variables are passed to sink then the information leakage is reported. It is the first of the many to follow to implement taint tracking in Android systems. Although being a state of the art technique, it suffers from problems such as taint explosion and false positives from over approximation in certain taint tags assignment. In Android, IMSI number (a single numeric String) consists of a Mobile Country Code (MCC), Mobile Network Code (MNC), and Mobile Station Identifier Number (MSIN). As a result they are tainted together. When Android uses MCC or MNC (generally uses it heavily), all information

in a parcel become tainted resulting in an explosion of tainted information. Our proposed tool MirrorDroid closely resembles TaintDroid in principle that both techniques use dynamic information flow tracking. However, results of our experiments suggest that *MirrorDroid* do not suffer from high false positives. It also has a better solution to track native code which is not directly tracked by both systems. Apart from that, DroidTest does not suffer from the TaintDroid limitations due to the single taint tag per file.

Many other taint analysis systems like VetDroid [43], AppFence [6], MockDroid [25] use TaintDroid [5] as a base framework.

MockDroid [25] allows users to mock the access from an untrusted application to particular resources at runtime. Similarly, TISSA [44] gives users detailed control over an application's access to a few selected private data (phone identity, location, contacts, and the call log) by letting the user decide whether the application can see the true data or some mock data. AppFence [6] replaces sensitive information with mock data. While these methods stop the leakage of data, they also impact legitimate applications that are allowed to access these data. Experiments showed that blindly replacing all sensitive inputs with mock data can cause applications crash also.

5.1.2.2 System Call Pattern Analysis

Systems such as Crowdroid [45], Paranoid Android [46] and [47, 48] detect malicious apps through monitoring and analysis of system calls. [49] also traces system calls as part of its analysis. Having collected the system call traces, these techniques either apply machine learning or various heuristics to predict malicious behavior of the application.

FlaskDroid [50], Patronus [51], ARTDroid [52],and ASM [53] are some other notable work to analyze system calls traces. A fundamental shortcoming of these approaches is the gap between the system calls and specific behaviors (e.g., it is exceedingly difficult to know whether an app sends an SMS to a premium number by analyzing a sequence of Android

kernel-level system calls).

5.1.2.3 Native Code Tracking

One of the limitations of VM bases systems is they cannot track data flow in native code. NDroid [54] is one of the first to solve this problem. It provides a complementary mechanism for taint-tracking information flows through Java Native Interface(JNI). It relies on TaintDroid [5] on the Dalvik VM side and, in the native context, maintains taint storage using shadow registers and memory maps.

Similarly, DroidScope and CopperDroid can inspect both the application code and the native code as part of their analysis. The recent TaintART [55] extends the work of TaintDroid [5] to the newly designed Android Runtime (ART), which generates binary code at application installation time. This design also helped TaintART to track flow in native code, which was not possible directly in Dalvik based analysis systems.

5.1.2.4 Privilege Escalation Detection

This is related to preventing the well known confused deputy attack in Android, where an application with less privilege can trick a higher privileged application to gain access to sensitive data. A great number of previous studies aim to mitigate this inter-application or inter-component permission leaks by either checking IPC call chains or by monitoring the run-time communication between application components [2, 56–59] etc.

QUIRE [56] tracks the IPC call sequence chain. IPC requests are denied if the requesting application does not have the corresponding permission. XManDroid [57] restricts potential dangerous communication (data flow that leak information) at runtime.

Felt et al. also discuss *permission re-delegation* attacks against confused deputies on the Android system [2] and mitigation techniques.

5.1.2.5 Automatic Test Input Generation

Dynamic analysis based approaches rely on test input generation techniques to guide the exploration of application of the application under test. The proposed solutions in this dissertation also depend on external test input generation method. There have been a significant effort to systematically explore Android GUI or input generation to trigger a target sequence of events. These techniques can be broadly classified to random and guided exploration.

Random exploration or fuzz testing techniques feed the application with events generated in pure random fashion. Sometimes malformed and unexpected input data are also given to stress test the application. Though no systematic exploration is performed, this approach often obtains better code coverage compared to guided or policy based exploration techniques. Well known random testing tools are Monkey [17], Null intent fuzzer [60], Intent Fuzzer [61] etc.

Though random exploration is found to perform well, in some cases, specific program conditions can only be triggered if the target program executes that particular path. For this reason, application testing methods often employs symbolic execution, evolutionary algorithms etc.

A3E-Targeted [62] provides an exploration strategy based on taint analysis. It builds the Static Activity Transition Graph of the app. Then the graph is traversed in DFS manner to exercise the application. But their approach mainly explores the application GUI and does not take into account the concrete inputs. To generate meaningful concrete values for various GUI elements such as text boxes, text area etc., symbolic execution techniques are used.

Mirzaei et al. [63] was among the first ones to use symbolic execution for Android applications. They use static analysis to deduce the set of feasible event-sequences. The deduced event sequences are then analyzed through symbolic execution. Several recent

efforts [64–66] have also applied symbolic execution. However, application source code may not be available always and technique to decompile the Android application binary is not mature enough. So, in the proposed systems we avoid using symbolic execution based systems for test case generation.

Other form of guided approach such as evolutionary computation is also used by some techniques. For example, EvoDroid [67] uses evolutionary algorithms to find inputs relevant to the application under test. For that, it represents individuals as sequences of test inputs and fitness function is designed to maximize code coverage.

5.1.3 Hybrid Techniques

These techniques employ both static and dynamic analysis in their design. In addition, they also apply additional measures such as symbolic execution, machine learning, data flow analysis to perform security analysis of Android system and applications.

ContentScope [68] finds vulnerabilities in the components of an application. It determines the vulnerable paths first and then executes those paths dynamically. But, the analysis is limited to content providers. The program paths in content provides are much less complicated than the real life Android malware application components.

Cassandra [69] enables users of mobile devices to check whether Android applications follow their privacy requirements before installing. They actually propose a secured app store for Android, which can guarantee prior security analysis on the applications before they can be used. Kynoid [70] enhances Taintdorid and introduces fine grained security permission for individual data items. Woodpecker [3] experiment with eight highly popular Android smartphones to find out that the stock operating systems did not not properly enforce the permission based security model.

Machine learning based approaches also rely underlying static or dynamic analysis based systems. Crowdroid [45], DroidMat [71], MAST [72], and DroidAPIMiner [73] are

few of the prominent work that apply machine learning techniques to find vulnerabilities in Android apps.

RiskRanker [49] detects malicious apps based on the knowledge of known Android system vulnerabilities. Similarly, DroidRanger detects malicious Android apps by statically matching against pre-determined signatures (permissions and Android Framework API calls) of well-known malware families. It also includes a heuristic-based approach to detect malicious applications from unknown families.

5.2 Record and Replay Execution

Android applications are GUI based and repeating an execution primarily requires replaying the same set of GUI activities. Apart from them there are other challenges : Android apps have several concurrent events that have to recorded and replayed precisely. The major concern here is the timing of events. To keep the overhead low, some existing methods only replay the GUI events and ignore non determinism in program inputs and thread schedules. They also do not record various real time sensors like audio, location etc. which can create non determinism between subsequent execution of the program. However, for simply replaying GUI events they work quite well and efficiently.

Reran [16] can record GUI events and replay them in correct order. It consists of three steps. First, events are recorded with the Android SDK's getevent tool. Then the output is sent into Reran's Translate program. The output from the Translate program is then sent into Reran's Replay program. The Replay program sends the events back into the event stream of the phone. It performs quite well in replaying the application events, it has some limitations. The replay program is not aware of which application it is replaying. The record phase just record the co-ordinates of the GUI element on screen which are exercised and the replay will be only successful if the same elements are present in the same location of the screen. As a result, before replay the user has to navigate to the same screen where

recording started. If any new notification comes during replay or any advertisement pops up in the screen which was not present during the record phase then the replay phase may not follow the same path as the record phase. We used *Reran* in our experiment mainly because of its simplicity and low overhead. The Reran limitations were resolved through manual intervention when needed.

Like *Reran*, Mosaic [74] also records events by reading */dev/input/event* files. It extends Reran with support for device-independent replay of GUI events. However it suffers from the same limitations as Reran. Mosaic does not support network, camera, microphone, or GPS, which are required by many applications. It also does not support record and replay of Android API calls or event schedules.

MobiPlay [75] works by running the replay program in parallel with the record phase in an online server. It has the benefits of recording user events from the application level without modifying the Android OS. However, as argued by the authors, success depends on creating similar environment and highly reliable communication. Although, the proposed idea is nice but not directly suitable for record and replaying program from large scale real life applications.

Basic GUI based replay methods do not take into consideration the concurrency of events and some hardware sensors like audio, camera activity etc., which are also important to successfully reproduce application behavior. For Android, Valera [76] is the first method to address these issues. It is built on top of Reran and adds additional record/replay functionalities.

It tries to remove non-determinism in program by recording and replaying non-deterministic APIs (e.g. Random numbers, Time, Location etc.) and inputs from various sensors used in Android applications. It also has the ability to correctly record the concurrency among various runtime events and replay those events in the same order. These two features make Valera a very promising tool for testing Android applications.

Valera [76] is closest to what we propose in the extension of DroidTest. It requires the full modified Android OS to be downloaded on the user side to set it up. On the other hand, we only modify the Android framework APIs and our system does not require to download the Android OS. Rather, we release our system as the modified Android SDK, which makes it more portable compared to Valera. We also did not implement the detail thread schedule replay of Valera. Instead, we applied some heuristics to record method invocations in different threads, which was proved to be successful in most of the cases.

CHAPTER 6

Conclusions and Future Work

In this thesis, we proposed a novel approach to determine an application's behavior by comparing its output in multiple executions while keeping the inputs same except the sensitive inputs. Based on the proposed approach, we also presented two different systems: DroidTest and MirrorDroid.

In DroidTest, we have shown that record and replay Android application execution for privacy monitoring can be done without expensive platform modification. The proposed system works by only instrumenting the Android framework APIs and can be easily distributed as a modified Android SDK. The experiment results showed that DroidTest can detect information leakage with high accuracy and produce only 11% false positives.

We also presented MirrorDroid, a virtual machine based dynamic information flow tracking system. It re-executes applications using a novel method by running a duplicate execution in parallel with the original execution within the Android (Dalvik) virtual machine. Though, it requires Android platform modifications, it produces less false positives (6%) compared to DroidTest as it does not have handle issues like non deterministic inputs, concurrency in the Android applications etc.

The proposed systems found many leakages in applications in the test data set. However, Android malwares are evolving and there are limitations in our systems which can be used by malware developers to evade analysis.

- Both DroidTest and MirrorDroid rely on manual exploration of the application under test, which may not reveal potential program paths during the test session. Android GUI testing tools combined with symbolic execution based methods can be used to

uncover more paths which are not always possible through manual or GUI testing methods.

- Our methods track only the application under test. As a result, they may not work if the adversary passes sensitive data to another application which is not under test. This limitation can be solved if we accommodate inter component communication between applications, which we plan to address next.
- DroidTest modification was performed on Android framework API version 4.4.2. However, some applications are solely developed using newer Android framework (5.0 or higher) and DroidTest needs to be updated to cover that. We resorted to version 4.4.2 mainly because of the instability in the compilation of newer Android SDK from the source code. Once the build process of newer Android versions is stable, our tool can be easily made to work for the latest Android versions with little to no modifications at all.
- MirrorDroid currently cannot track information leakage through files, because duplicate execution of file access is not realistic to implement. For this reason, we plan to add this feature using some heuristics such as partially using taint tags for files.

REFERENCES

[1] "Survey on smartphone users," http://www.engadget.com/2012/05/07/.

[2] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX Security Symposium*, vol. 30, 2011.

[3] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012, p. 19.

[4] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proc. of the USENIX Conference on Web Application Development*, 2011.

[5] 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*, 2010, pp. 1–6.

[6] 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*. ACM, 2011, pp. 639–652.

[7] S. T. A. Rumee and D. Liu, "Droidtest: Testing android applications for leakage of private information," in *Information Security*. Springer, 2015, pp. 341–353.

[8] S. T. A. Rumee, D. Liu, and Y. Lei, "An adaptive bacterial foraging algorithm for color image enhancement," in *Information Science and Systems (CISS), 2017 Annual Conference on*. IEEE, 2017.

[9] "Official android marketplace: Google play," https://play.google.com/.

[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*. ACM, 2011, pp. 3–14.

[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*. ACM, 2012, p. 3.

[12] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 77–83.

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[14] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*. ACM, 2005, vol. 30, no. 5.

[15] "Apktool," https://ibotpeaches.github.io/Apktool/.

[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 72–81.

[17] "Android monkey," https://developer.android.com/studio/test/monkey.html.

[18] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[19] "Monkey runner," https://developer.android.com/studio/test/monkeyrunner/index.html.

[20] "Robotium," http://code.google.com/p/robotium/.

[21] "Malware data set," http://www.malgenomeproject.org/policy.html.

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

[23] "Contagio mobile malware mini dump," http://contagiominidump.blogspot.com/.

[24] "Analysis of appverification tool from google," http://www.csc.ncsu.edu/faculty/jiang/appverify/.

[25] 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*. ACM, 2011, pp. 49–54.

[26] "Android method frame structure," http://androidxref.com/4.4.2_r1/xref/dalvik/vm/oo/Object.h.

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

[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," *Trust and Trustworthy Computing*, pp. 291–307, 2012.

[29] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[30] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[31] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014, pp. 1–6.

[32] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[33] X. Chen and S. Zhu, "Droidjust: Automated functionality-aware privacy leakage analysis for android applications," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 5.

[34] X. Cui, J. Wang, L. C. Hui, Z. Xie, T. Zeng, and S.-M. Yiu, "Wechecker: Efficient and precise detection of privilege escalation vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 25.

[35] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.

[36] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models," *computers & security*, vol. 59, pp. 92–117, 2016.

[37] 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*. ACM, 2013, pp. 1043–1054.

[38] F. Wei, S. Roy, X. Ou, *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[39] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[40] P. Ferrara, O. Tripp, and M. Pistoia, "Morphdroid: fine-grained privacy verification," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 371–380.

[41] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[42] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 543–558.

[43] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 611–622.

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

[45] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[46] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 347–356.

[47] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *Communications, 2009. ICC'09. IEEE International Conference on*. IEEE, 2009, pp. 1–5.

[48] A.-D. Schmidt, H.-G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak, "Enhancing security of linux-based android devices," in *in Proceedings of 15th International Linux Kongress. Lehmann*, 2008.

[49] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services.* ACM, 2012, pp. 281–294.

[50] G. Abirami and R. Venkataraman, "Access control policy on mobile operating system frameworks–a survey," *Indian Journal of Science and Technology*, vol. 9, no. 48, 2016.

[51] L. Zhang, Y. Guo, and X. Chen, "Patronus: Augmented privacy protection for resource publication in online social networks," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on.* IEEE, 2013, pp. 578–583.

[52] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime," *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)*, pp. 24–32, 2016.

[53] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending android security." in *USENIX Security*, vol. 14, 2014, pp. 1005–1109.

[54] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 2014, pp. 180–191.

[55] M. Sun, T. Wei, and J. 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.* ACM, 2016, pp. 331–342.

[56] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, vol. 31, 2011.

[57] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[58] J. Zhong, J. Huang, and B. Liang, "Android permission re-delegation detection and test case generation," in *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE, 2012, pp. 871–874.

[59] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing androids permission system," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 1–18.

[60] "Null intent fuzzer," https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer/.

[61] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 2014, pp. 1–5.

[62] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[63] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[64] J. Jeon, K. K. Micinski, and J. S. Foster, "Symdroid: Symbolic execution for dalvik bytecode," Tech. Rep., 2012.

[65] S. Anand, M. Naik, H. Yang, and M. J. Harrold, "Automated concolic testing of smartphone apps," *No. GIT-CERCS-12-02*, 2012.

[66] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software*

Testing and Analysis. ACM, 2013, pp. 67–77.

[67] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[68] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[69] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "Cassandra: Towards a certifying app store for android," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014, pp. 93–104.

[70] D. Schreckling, J. Köstler, and M. Schaff, "Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," *information security technical report*, vol. 17, no. 3, pp. 71–80, 2013.

[71] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[72] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. ACM, 2013, pp. 13–24.

[73] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 86–103.

[74] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 215–224.

[75] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 571–582.

[76] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 349–366.

BIOGRAPHICAL STATEMENT

Sarker T. Ahmed Rumee was born in Netrokona, Bangladesh, in 1983. He received his B.Sc. degree from University of Dhaka, Bangladesh, in 2007 and his M.S. degree from the same university in 2009, respectively, all in Computer Science and Engineering. From 2009 to 2010, he was with the Department of Computer Science and Engineering, University of Dhaka as Lecturer. Before that, he also served the Department of Computer Science and Engineering of State University of Bangladesh as Lecturer from 2008 to 2009. His current research interest is in the area of security and privacy for smartphone operating systems, software security and program analysis.