Artificial User Emulator to Detect Intelligent Malware on Android

Shirish Singh, Saket Singh, Bhairavi Mishra
The LNM Institute of Information Technology
Jaipur, India

Abstract

Given that most emulator based analysis environment have many static and dynamic differences from a real device used by a user, such environments can be easily tricked by an intelligent malware which may prey upon specific details like IMEI number, button press, accelerometer, GPS coordinates etc. to know whether it is in an emulator thereafter it can act benignly and pass the test undetected. This, and other vulnerabilities have been recognized by this paper and to enhance the detection capability of dynamic analysis environments, we present a framework with a twofold objective, to emulate artificial user behavior and, to help unravel malware's true behavior. Furthermore, our framework is divided into two major categories based on dynamic and static properties of a Smartphone. The framework is tested with an open-source sandbox environment and an existing emulator detection application.

1. Introduction

Android is a popular open source smart-phone and tablet operating system developed by Android, Inc. which was later bought by Google. According to IDC's 2014 report, 1.3 billion Android based mobile phones were shipped during the third quarter of 2014, and as a result, it captured over 84% of smart-phone market. Android was originally based on Linux Kernel 2.6 but to bring innovation and value to customers, Android applications are modified or extended extensively to use sophisticated hardware and software through the platform. On one hand, this popularity of Android has encouraged its developers to develop more applications, but on the other, it has also attracted malicious application developers, due to which not all applications can be trusted. Security of the user being a primary concern, security community has developed static and dynamic analysis tools to identify the malicious applications. Their objective was to identify the malware before they can be installed on user's device. Many researchers have developed similar systems [3] to monitor the behavior of the application while running on an emulator. Moreover, Google Bouncer tests all applications uploaded on Google Play Store using a QEMU-based emulator. As a result, Google Bouncer has helped to drop the number of malware downloads in Google Play by 40% in the year 2011. Unfortunately, these techniques worked when they first came to existence, but gradually the malware developers have found loopholes to bypass detection on such analysis tools.

Say for example, static analysis is not effective enough if the application uses encryption [16]. Emulator based dynamic analysis can be bypassed if the application is able to identify whether it is running on an emulator or not. If it is running on an emulator then it's is highly likely that it is being run on a sandbox. Once emulator is identified, the malware acts benign like a normal application should and clears the analysis with flying colors. Emulator identification is a crucial step for any malware and there has been many researches to distinguish the properties of a real device and an emulator [18], [20], [19]. Our framework addresses the issue of QEMU emulator detection by malware, by providing the necessary support, such that it simulates the properties of a real device. Primary focus of our framework is on fake user behavior. Our paper is organized into eight sections. In section 2, related work in the field has been discussed followed by section 3, which discusses differences between emulator and real device. Section 4 briefly describes types of properties. Section 5 elaborates our framework. Section 6 presents results of the framework with an existing open-source dynamic analysis tool. Finally Section 7, gives the limitations followed by conclusion and future work.

2. Related Work

There has been significant work in the field of emulator detection and bypassing dynamic analysis. This section covers some of the important work related to our proposed framework. Yiming et. al. [18] identified more than 10000 heuristics to detect android emulator and ranked top 30 artifacts of detection heuristics with their respective accuracy. These heuristics were further subdivided into file, API and Property of emulator. The study was done for QEMU-based and Virtualbox-based emulators. They had also developed an android application1 to detect whether the device is a real device or an emulator, using the artifacts. Furthermore, the paper discusses the discrepancies in Android emulators and

potential countermeasures. Timothy V. and Nicolas C. [20] conducted a similar study to detect Android malware analysis environments. They have detected systems based on differences in behavior, performance, hardware, software components and system design choices. They have focused on CPU and GPU performance differences among other hardware. Thanasis Petsas et al. [19] presented anti-analysis techniques based on static properties, dynamic sensor information, and VM-related properties of the Android emulator. Static heuristics include unique device identifiers, such as IMEI, IMSI, SIM Number, Build Properties, etc. Dynamic heuristics included sensor specific readings. Maier et. al. [15] fingerprinted ten different sandboxes. They demonstrated that dynamic code loading can bypass Google Bouncer. The study suggests that malicious applications use dynamic code loading more frequently as compared to the benign applications. They concluded that neither the static nor the dynamic analysis can provide comprehensive security from malware if the attacker designs the application to behave differently is different scenarios of real and emulated environment. The paper addresses the shortcomings of the QEMU-based emulators, taking these studies as a base for eradicating the differences between a real device and a virtual device, and hence consolidating dynamic analysis environments, making them difficult to get detected.

3. Emulator VS real Device

Android emulator is a virtual device, similar to a real android device which runs on a PC so as to support development and testing of applications by developers without having to use a real device. Although there are subtle differences in the working of an emulator and a real device, the functions are similar. This section lists the important properties, which differentiate an emulator from a real device.

3.1. Build.prop File

It is a text file residing in \system directory of Android system image. It contains the information about the current build of Android. It also defines the properties of the operating system and thereby affects its functioning [1]. It tells the application, the specifications of the device, so that apps can change their interface or functions according to the device. Say for example, some of the applications available on Play store does not work with all the devices because they have specific system requirements to run, which are defined in this particular file. Changing the build.prop file can change the properties of the system such as the Build Number, Manufacturer, etc. Also, these properties are fetched by the android.os.Build class in java API. This file can be manipulated to reflect the properties of a real device.

3.2. Sensor Data

By default, all sensor values are predefined on an emulator. These values can be used as a criterion for emulator detection. For example, no application needs special permission to access the accelerometer. On a default emulator, the accelerometer reading remains constant. Judgment of whether the device is emulator or not can be made by just observing these readings. Android Debug Bridge (ADB) gives options to simulate specific sensors on an emulator. Table 1 summarizes the initial values of sensors on an emulator

Table 1. Sensor Valued Derived from an Emulator

Sensors	Values
Accelerometer	0 : 9.77622 : 0.813417
Magnetic-Field	0:0:0
Orientation	0:0:0
Temperature	0:0:0
Proximity	0:0:0
GPS	0.0,0.0

3.3. User Behavior

One of the most differentiating aspect of real device is that it gets constant user input. This lacks on an average sandbox environment. User input can be a major factor by which it can be decided if the malware is running on an emulator or not. Although, Android provides tools like monkeyrunner to simulate user input events, they are not effective enough if it is not applied properly. A number of the analysis environments, available online, use random inputs of button press events, key press events, touch drag events, etc.

5. Framework

Our framework is an add-on to existing dynamic analysis environments. It is divided into four different parts dealing with separate things based on above properties. First, we perform static analysis of the application package (apk file) through the Analyzer. Analyzer returns a set of permissions, activities, services, etc. of the application. Sensor simulator reads the permissions needed by the application and starts simulation of the necessary sensors based on real open source data-sets in timely manner. Event simulator works in parallel with the sensor simulator. It utilizes uiautomator tool inbuilt in Android, to fetch the user interface coordinates and the elements in XML format. It then reads the

file and triggers the touch/drag event in the identified coordinates on the Android Emulator‘s screen. Figure 1 gives an overview of the framework

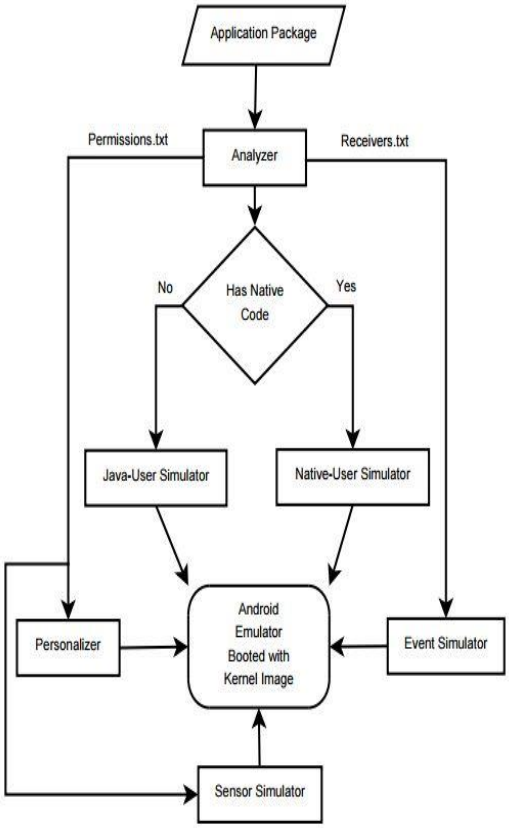

Figure 1: Overview of the framework: Analyzer is a part of static analysis, whereas Personalizer, Sensor Simulator, Event Simulator and User Simulator work dynamically

5.1 Static Components

1. Kernal Image: There exist some discrepancies at file level in emulator as compared to a real device [18]. To overcome these differences, the authors have compiled the kernel image, which bypasses all the file and directory lookups made by the application process or its sub-processes running in the same userspace. The framework utilizes android-goldfish 2.6.29 kernel [6]. For all lookup operations done for android file system, the stat64 system call returns ‘0’ if the files or directory also reside in an ideal real device, meaning that the queried file or directory exists. Based on Morpheus [18], our kernel is able to falsify the existence of the files and directories which does and does not exist in an emulator, creating a real device like environment for the malware**.**

2. Build Properties: Every device has pre-defined build properties which will remain consistent through-out the device’s lifetime, if the device is not rooted. The framework modifies these critical properties of the system to make the emulator look similar to a real device, by modifying the properties of build.prop file residing in \system directory of the device.

3. Analyzer: Androguard is a static analysis tool-box for reverse engineering android applications. It consists of various tools for specific purposes of analysis. Analyzer uses Androguard’s Androapkinfo tool to extract the Application Manifest file from the Android Application Package (APK) under consideration for analysis. It then extracts the Activities, Permissions, Services, Providers, Receivers and existence of native code from the apk file and stores it in separate files, made for each category. Similar permissions have been grouped together for convenience, such as ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION have been grouped together as they both require GPS Sensor data. Table 2 summarizes the permission groups used in the framework.

Table 2. Permission Categories

Permission Type	**Description**
Location	All Location related permissions
External Storage	Permission to read/write on external storage
Network State	Permissions to check network state
Phone	Permissions related to telephony services to view call log, make phone call, etc.
Contact	Permissions related to contacts on device
Calendar	Permissions to read and write calendar
SMS	Permissions related to SMS
Hardware	Hardware Sensor requirements

5.2 Dynamic Components

1. Personalizer: The majority of real devices consist of contacts, phone logs and SMS history. In order to portray emulator as a real device, it should contain these characteristics too. Personalizer reads the Permissions.txt file and if the application needs permission of “Phone”, “Contact” or “SMS” category, it call either the phone script, contact script or the SMS script to populate the lists with bogus data. Each script utilizes an open-source data-set so that the details are legitimate. Although the phone-numbers are bogus and randomly generated, the

names are genuine. Personalizer uses telnet and adb(Android Debug Bridge) to perform the above tasks. If an application also requests permission to access the memory, either read or write, Personalizer populates the sdcard with bogus data files containing random texts, images, songs, etc. In case the apk file is encrypted and 'Permissions.txt' cannot be retrieved, all the above mentioned actions will take place by default.

2. Sensor Simulator: Sensor simulator accesses the Permissions.txt file created by the Analyzer. Reading each category of the permissions and hardware sensors required by the application, sensor simulator uses real opensource data-sets [9], [14], [17] to simulate sensor with particular time delays, fetched from the data-set itself. It has been found that periodically updated values can lead to emulator detection [19], hence we use time stamps from the data-set itself. All applications are free to access accelerometer without any permission; hence it can be used as a major detection strategy. To mitigate the detection by accelerometer, the emulator is fed with data according to human behavior such as walking, running, sitting, driving etc. Similarly all other sensors emulate real data open source collected from various anonymous contributors such as [10]. Network, as a special case, will be emulated based on the technology speed, chosen from [8].

3. Event Simulator: Many applications listen to certain events happening in a device. The framework extracts the events that are needed by the application and then simulates them to trigger their behavior upon receiving the events. Event simulator uses the Receivers.txt to read the Broadcast intents that the application has registered to. Each event requires very particular action to be performed. Event simulator has script pertaining to each broadcast intent. Say for example, one of the most distinguishing features of emulator is the battery state and battery level. Battery level in emulator is consistent to be 50% all the time. Malware can easily identify if the device is emulator by monitoring the battery level. The framework updates the battery level in timely manner while also changing its status based on real dataset [10], either reducing the battery while its state is disconnected and increasing the level while it is in charging state.

4. User Simulator: User simulator's prime objective is to simulate a fake user, who performs operations on the UI of the application. Application UI can be categorized into two variants, based on whether it was created using JAVA or Native code. Java based applications are used for general purpose and Native code is used to run the CPU-Intensive applications such as game engines and physis simulations, etc. Java code runs on Dalvik Virtual Machine or Android Runtime, while the Native code runs outside the virtual machine. There are tools like UI Automator and Hierarchy Viewer [11], [5] available to extract the user interface data from the device/emulator, but it is observed that they do not return the UI elements if the UI is written in Native code. To tackle this issue, we have divided the User Simulator into two parts, separately dealing with different UI types. Following sections discuss the implementation details of User Simulator in two different scenarios.

Table 3. UI Automator Attributes of elements [13] used in the framework

File Name	**Function**
Class	Class property for a widget
Package	Package name of the application that contains the widget
Checked	Widgets that are currently checked (usually for checkboxes)
Clickable	Widgets that are clickable
Focusable	Widgets that are focusable
Focused	Widgets that have focus
Scrollable	Widgets that are scrollable
long-clickable	Widgets that are long-clickable
Selected	Widgets that are currently selected
Bounds	Location of the Widget on User Interface in terms of pixels

JAVA Base-User Simulator: JAVA based UI are quite easy to fetch through UI Automator [11]. The retrieved UI dump file contains the elements of UI in XML Format. Table 3 summarizes the attributes of data retrieved through UI Automator. The User Simulator contains a parser. Once the dump is retrieved for an activity, each element of the UI is segregated according to its clickable, focusable and long-clickable attributes with the help of parser. These are then fed to Monkeyrunner [7] scripts to simulate user behavior. The framework takes a random approach to select the element to perform a task upon. However, the number of actions that can be performed on an UI element is limited, based on its properties. Such as, 'scrollable' elements will be simulated with inputs like 'touch' and 'drag'. Elements which are 'long-clickable', will be simulated with 'touch-hold', followed by 'drag' event. Similarly, relevant functions will be applied

on 'checked' and 'clickable' elements. The 'bound' attribute of the element will be taken as a input of the simulator, because it identifies the location of the element on the screen, in the format: [X,Y][X_Length, Y_Length]. The event will be triggered in the middle of the button for maximum accuracy.

Native Code Base-User Simulator: Most of the applications written in Native code are either Games or Graphic Simulations, because native code improves the performance of CPU intensive applications. Since we do not possess a definite location of elements on UI, we have taken a probabilistic approach. We have calculated the probability distribution of elements present in the UI screen. We have divided the screen into cells of 48dp x 48dp3 matrix. 48dp is considered the minimum size of a button [10]. Following formula is used to convert display independent pixels to absolute pixels:

$$Pixels = dp * (Display\ Metrics/Def\ ault\ Density)$$

Display Metrics is the screen density of the height or width of a device, Default Density is 160 (standard reference density) [2]. The above equation gives us the absolute pixels which are used to create the screen matrix. To calculate the probability of a button or any interactive element in a cell (either partial or full) of screen matrix, we have considered top 50 free gaming applications available on Google Play Store and calculated the probability of existence of a button in each matrix cell. The probabilities were also calculated for cells where simple touch, drag and touch & hold events can be performed. While dynamic analysis of the application, the cells of screen matrix is sorted in descending order corresponding to the Probability values. The cell coordinates are then fed to the monkeyrunner script to simulate user input with 1 second latency for the android activity to respond. Following equations calculate the probability:

$$Q(i,j,k) = \begin{cases} 1, if\ button\ exists\ in\ (i,j) \\ 0, Otherwise \end{cases} \quad (1)$$

$$P(i,j) = \sum_{k=0}^{50} \frac{(Q(i,j,k))}{50} \quad (2)$$

Where, Q(i,j,k) represents the probability of existence of a button in kth activity, P(i,j) is the actual probability of button existing in the cell of ith row and jth column in screen matrix. Rather than giving pseudo-random user inputs by monkey [12], the framework can give input based on probability, which is more systematic and reduces the unnecessary computations of giving random inputs.

6. Result

Our framework has been fully tested with Droidbox [3]. We have considered Morpheus as a testing base as it detects emulator based on its detection heuristics. It has qualified the tests of Morpheus [18] and has been identified as a real device. Figure 2 and 3 shows the results of Morpheus Application on android emulator running without and with the framework respectively.

Figure 2: Android emulator: Morpheus running in Droidbox without the framework

Application shows results consisting of 3 rows with 10 columns in each. Each element of the result is a binary value of either '1' or '0' representing either true or false respectively. Morpheus paradigm is that if more than 50 % detection heuristics based on the artifacts return true, the device will be considered an emulator, otherwise a real device. First row represents file heuristics, second row represents API heuristics and the third row represents property heuristics. Morpheus considers procfs (proc filesystem) and sysfs as a base for file heuristics. Both are RAM based file system that help in exchanging data between kernel objects to userspace programs. Procfs contains all kinds of data about processes, CPU information, memory information, etc. while sysfs contains information about the devices, drivers, etc. of a system. For property heuristics, Morpheus analyzes system configurations and status through /system/bin/getprop binary [18].

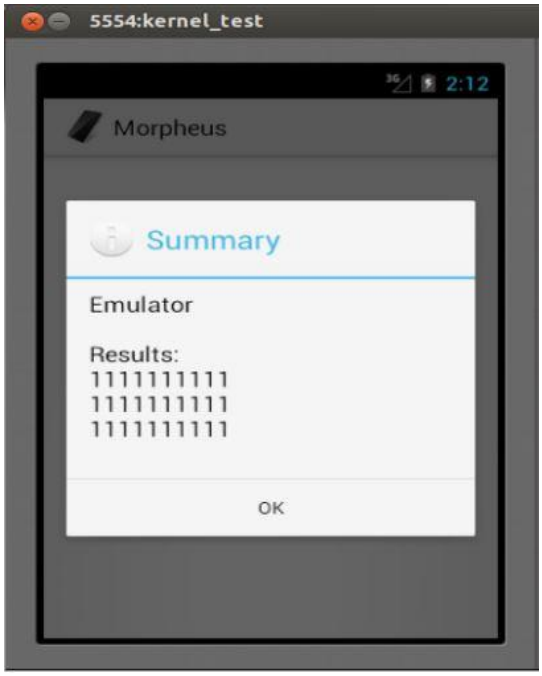

Figure 3: Android emulator: Morpheus running in Droidbox with the framework

The framework has bypassed all the file heuristics and more than 50% of the property heuristics of Morpheus, implying that the emulator can be safely categorized as a real device. Moreover our framework has bypassed the file heuristics which had the accuracy of 100% [18]. Table 5 lists the heuristics and their accuracy bypassed by the framework.

7. Limitations

The framework doesn't deal with the internal JAVA APIs as it is assumed that the dynamic analysis tools will incorporate the necessary changes in it while compiling the android emulator image. Although detection through JAVA APIs can be eliminated if the emulator is compiled from scratch with relevant changes in the API calls such as, Droidbox's getDeviceId() returns non-zero IMEI number ("357242043237511") however, if it is called via binder, it will return a "000000000000000" [18]. If the emulator image is compiled carefully taking into consideration, what a real device would return, it would definitely enhance the system. User input function is not comprehensive at this stage and will have to be updated as the application UI trend changes in future. Screen matrix will have to be re-evaluated periodically for updated probabilities. Some hardware, such as Headphones and Bluetooth are not emulated by the emulator [4]. This can be a setback if the malware is detecting the device using Bluetooth API calls. However, the emulator image can be compiled to give desirable results for particular API calls made by the application.

Table 4. Morpheus Artifacts and their accuracy, bypassed by the Framework [18]

Artifact	**Type**	**Accuracy (%)**
/proc/misc	file	98.5
/proc/ioports	file	100.0
/proc/uid stat	file	94.9
/sys/devices/virtual/misc/ cpu dma latency/uevent	file	97.1
/sys/devices/virtual/ppp	file	99.3
/sys/devices/virtual/switch	file	99.3
/sys/module/alarm/parameters	file	94.9
/sys/devices/system/cpu/ cpu0/cpufreq	file	100.0
/sys/devices/virtual/misc/ android adb	file	100.0
/proc/sys/net/ipv4/ tcp syncookies	file	94.2
ro.build.description	property	86.9
ro.build.fingerprint	property	83.9
rild.libpath	property	98.5
gsm.version.baseband	property	89.8
ro.build.tags	property	92.0
ro.build.display.id	property	99.3

Implementation of real device like hypervisor heuristics [19] has not been taken into consideration as it is a wide domain.

8. Conclusion

In this paper we have proposed a emulator anti-detection framework for QEMU-based emulators, which works with dynamic analysis environments. Our results show that the system's static attributes can be exploited to make it similar to a real device. Also the dynamic attributes can be simulated from real data-sets to make it look realistic. Adding the framework to any dynamic analysis environment will considerably improve its performance in terms of detection. However, despite the authors' effort, there are certain loopholes which still need to be fixed, such as Bluetooth emulation. This field is ever changing and will require constant modification in the framework as soon as changes are made in emulator functioning. In our future work, we plan to work on VirtualBox based emulators and develop a

mechanism for Hardware emulation, which doesn't exist in current emulators. Also, we'll work on hardware based detection heuristics to make the emulator resilient towards hypervisor based detection.

9. References

[1] Build: Class overview. http://developer.android.com/reference/android/os/Build.html, Accessed: September 2015.

[2] Display metrics.http://developer.android.com/reference/android/util/DisplayMetrics.html, Accessed: September 2015.

[3] Droidbox: An android application sandbox for dynamic analysis.https://code.google.com/p/droidbo x/, Accessed: September 2015.

[4] Emulator limitations, http://developer.android.com/tools/devices/emulator. html#starting, Accessed: September 2015.

[5] Hierarchy viewer, http://developer.android.com/tools/help/hierarchyviewer.html, Accessed: Septem-ber 2015.

[6] Kernel/goldfish, https://android.googlesource.com/kernel/goldfish/+/androidgoldfish-2.6.29, Access date: September 2015.

[7] Monkeyrunner.http://developer.android.com/tools/help/monkeyrunner concepts.html, Access date: September 2015.

[8] Network speed emulation, http://developer.android.com/tools/devices/emulator.html#netspeed, Access date: September 2015.

[9] Public gps traces.http://www.openstreetmap.org/traces, Access date: September 2015.

[10]Rotterdam open data store, http://rotterdamopendata.nl/dataset, Access date: September 2015.

[11] Ui automator.https://developer.android.com/tools/testing-supportlibrary/index.html, Access date: September 2015.

[12]Ui/application exerciser monkey, http://developer.android.com/tools/help/monkey.html, Access date: September 2015.

[13] Uiselector. Android Git Repository Access date: September 2015.

[14] Wisdm: Wireless sensor data mining. http://www.cis.fordham.edu/wisdm/dataset.php, Accessed September 2015, Access date: September 2015.

[15] Maier, D., Protsenko, M., Müller, T. (2015). A game of droid and mouse: The threat of split-personality malware on android. http://dx.doi.org/10.1016/j.cose.2015.05.001.

[16] C. Ionescu. Obfuscating embedded malware on android. http://www.symantec.com/connect/blogs/obfuscating-embeddedmalware-android, Access date: September 2015.

[17] M. Jain, A. P. Singh, S. Bali, and S. Kaul. CRAWDAD data set jiit/accelerometer (v. 2012-11-03). Downloaded from http://crawdad.org/jiit/accelerometer/, Nov. 2012.

[18] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: Automatically generating heuristics to detect android emulators. In Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, pages 216–225, New York, NY, USA, 2014. ACM.

[19] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In Proceedings of the Seventh European Workshop on System Security, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[20] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

