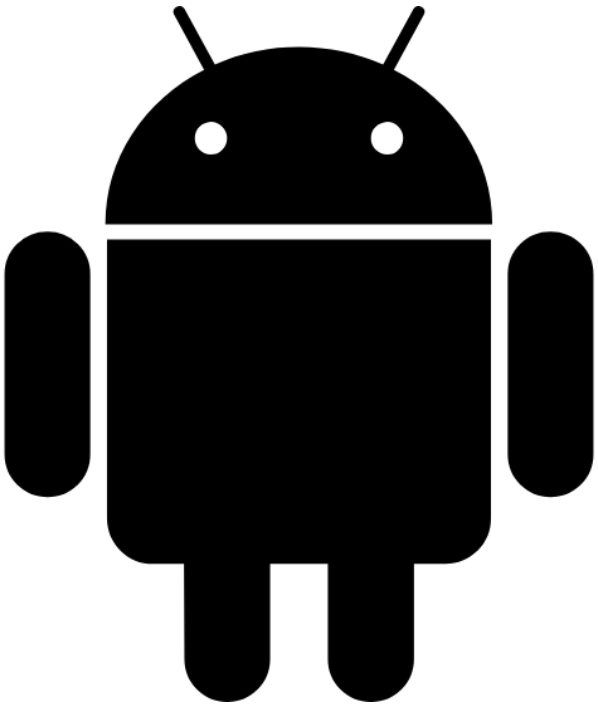
Anti Android Emulator Detection

Accurate & Effective Dynamic Android Malware Analysis



Interim Report

COMP 4801 Final Year Project
The University of Hong Kong

Student: Ho Man Hon (3035244356) Supervisor: Dr K.P. Chow

Abstract

Nowadays, mobile applications are highly demanded by increasing smartphone users. Android, which is the biggest mobile market, attracts many hackers to develop malware in order to exploit users. To encounter this situation, developing a trustworthy and fast malware analysis method is necessary. However, emulator detection in mobile has block the automated malware analysis development. Therefore, this project introduces a more flexible and feasible anti-emulator detection solution which aims to be applied in an automated analysis tool. In this project, the design will cooperate with existing automated malware analysis framework called Cuckoodroid. We will create a new module by using Frida, an open-source dynamic instrumentation toolkit. Recently, the research study has been finished. The next step will be the code implementation. Furthermore, malware analysis is a different research area in computer science hence there are many technical problems encountered. We will discuss them and attempt to explore if there exists alternative which can solve the problems.

Acknowledgment

I would like to thank my supervisor, Dr. K.P. Chow from the Department of Computer Science, for his advice and careful guidance in this project. Furthermore, I would also like to thank Jonna from Center for Applied English Studies for different suggestions when I was writing this report.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

According to the smartphone market share statistics displayed in Table 1 [1], Android is the most popular mobile operating system in the worldwide smartphone market share, which has been chosen by 84.8% of mobile device users. Regarding Android's high market popularity, it is reasonable for cybercriminals to spend long hours to expose Android's vulnerabilities. As stated in G DATA security blog in 2017, daily creation amount of Android malware was approximately 8,400 [2]. Concurrently there is no such high increment in other mobile platforms such as iOS. Hence Android is considered as a major victim of malware.

Year	2016	2017	2018
Android	84,6%	85,1%	84,8%
iOS	14,7%	14,7%	15,1%
Others	0,7%	0,2%	0,1%
TOTAL	100,0%	100,0%	100,0%

Table 1: Smartphone OS Market Share data [1]

In 2017, a well-known security firm called *Check Point* announced that a series of malware has been discovered in Google PlayStore [3]. The rationale was malware had anti-analysis techniques to bypass the security checking before release on PlayStore. Furthermore, Google allows the Android application to put on PlayStore as soon as possible. If the malware is evading from being detected by known malware analysis tools, this showed a serious problem when the security check is not performed by security specialists with sufficient investigation time.

Malware researchers are attempting to determine an efficient solution to perform security checking on mobile malware in Android for many years. One of the solution is Cuckoodroid. Cuckoodroid is an automated malware analysis tool, extended from Cuckoo Sandbox framework. The framework is responsible for managing the android emulators and generating analysis reports on the executed malware. The usage of each component within Cuckoodroid will be explained in the methodology section.

1.2 Motivation

Android is an open source framework. Many well-known mobile device manufacturers like Samsung, Sony, ASUS, HTC, LG, are the third parties developer of Android. Normally when Google announces a new security update, third parties are requested to individually develop their own update package since the phone using third party Android framework cannot direct apply the official security patches. It is not possible to rely on protection by security update to solve the Android malware problem. Prevention by malware detection is also important to tackle the problem.

An automated malware analysis tool is an ideal solution to fasten the malware detection stage. However, emulator detection will not allow the automated malware analysis function to detect the malware easily.

This project is to design an anti-emulator detection module which will be applied in the Cuckoodroid framework. To sum up the features of the project:

1- Minor fix on Cuckoodroid

> The framework itself is still not working perfectly. Bug fixing is not avoidable during the project.

2- Anti-emulator detection module

> This project provides another approach different from the original anti-emulator detection module. Reason will be explained in the methodology section

3- Report the emulator detection activity in runtime

> Let the user know the running application is performing known emulator detection methods.

1.3 Scope

Emulator detection is able to be implemented with different approaches. In accordance with a previous malware anti-emulator behavior research [5], there are 3 major emulator detection approaches:

1.3.1 Check Device Information

In Android, there are several simple GET methods defined to retrieve system information during runtime. As shown in Table 1, those API methods were able to retrieve the device information. In this scenario, emulator detection applied the knowledge of sandbox device information to identify the type of application environment. In practice, checking device information does not request root level privileges, hence, it is pretty easy to be implemented.

API method	Value	meaning
Build.ABI	armeabi	is likely emulator
Build.ABI2	unknown	is likely emulator
Build.BOARD	unknown	is emulator
Build.BRAND	generic	is emulator
Build.DEVICE	generic	is emulator
Build.FINGERPRINT	generic††	is emulator
Build.HARDWARE	goldfish	is emulator
Build.HOST	android-test††	is likely emulator
Build.ID	FRF91	is emulator
Build.MANUFACTURER	unknown	is emulator
Build.MODEL	sdk	is emulator
Build.PRODUCT	sdk	is emulator
Build.RADIO	unknown	is emulator
Build.SERIAL	null	is emulator
Build.TAGS	test-keys	is emulator
Build.USER	android-build	is emulator
TelephonyManager.getDeviceId()	All 0's	is emulator
TelephonyManager.getLine1 Number()	155552155xx†	is emulator
TelephonyManager.getNetworkCountryIso()	us	possibly emulator
TelephonyManager.getNetworkType()	3	possibly emulator (EDGE)
TelephonyManager.getNetworkOperator().substring(0,3)	310	is emulator or a USA device (MCC)‡
TelephonyManager.getNetworkOperator().substring(3)	260	is emulator or a T-Mobile USA device (MNC)
TelephonyManager.getPhoneType()	1	possibly emulator (GSM)
TelephonyManager.getSimCountryIso()	us	possibly emulator
TelephonyManager.getSimSerial Number()	89014103211118510720	is emulator OR a 2.2-based device
TelephonyManager.getSubscriberId()	310260000000000‡‡	is emulator
TelephonyManager.getVoiceMailNumber()	15552175049	is emulator

Table 2: Some API methods that can be used for emulator detection [5]

1.3.2 Runtime Performance

Emulator's computing performance is different from that of the real device. According to a computing experiment shown in Table 2, it indicated an apparent difference in computation

power between emulators and real devices. Based on this observation, emulator detection can be performed by implementing a computational checking to identify the type of application environment.

Device	Average Round Duration (Seconds)	Standard Deviation
PC (Linux)	0.153	0.012
Galaxy Nexus (4.2.2)	16.798	0.419
Samsung Charge (2.3.6)	22.647	0.398
Motorola Droid (2.2)	24.420	0.413
Emulator 2.2	62.184	7.549
Emulator 4.2.2	68.872	0.804

Table 3: PI calculation round duration of tested devices using AGM technique (16 rounds) [5]

1.3.3 Hardware Configuration

Real devices have different sensors such as accelerometer, temperature, gravity. A normal application will check for the existence of sensors before using them. However, dynamic malware analysis was not designed as a software testing model initially. In accordance with the supported sensor in Android displayed in Table 4, different mobile models were equipped with different sensors. Inside the current Android emulator, many sensors in Table 4 do not exist. Based on this observation, emulator detection can be performed by detecting and using some sensors which appear in the real device. In this situation, if the device throws an exception, it is implicitly indicated that the device is emulator.

Theoretically, the emulator detection approaches shown in 1.3.2 and 1.3.3 can be solved by designing fake sensor applications and improving hardware configuration (i.e. Emulator, with better hardware supported, has the capability to reach the same speed as the real device). However, designing every fake sensor applications suitable in emulator is time-consuming, and improving hardware configuration is not a simple computer software solution. It is hard to include them in this project.

Sensor	Android Version	Moto. Droid	Samsung Charge	HTC EVO 4G	Galaxy Nexus
Accelerometer	1.5	1	1	1	1
Ambient Temperature	4.0	-	-	-	0
Gravity	2.3	-	1	1	2
Gyroscope	1.5	0	1	0	2
Light	1.5	1	1	1	1
Linear Acceleration	2.3	-	1	1	2
Magnetic Field	1.5	1	1	1	1
Orientation	1.5	1	1	1	2
Pressure	1.5	0	0	0	1
Proximity	1.5	1	1	1	1
Relative Humidity	4.0	-	-	-	0
Rotation Vector	2.3	-	1	1	2
Temperature	1.5	1	0	0	0
Total		6	9	8	15

Table 4: Sensor Types, the earliest version of Android to support each type, and observed sensor counts on four devices. [5]

On the other hand, the approaches shown in 1.3.1 is easy to implement. In addition, according to a previous anti-emulator detection research [6], checking system information is frequently used in many emulator detection solutions implemented by software corporations. The 1.3.1 approach, therefore, can be identified as an emulator detection normal approach. Hence, this project is designed to focus on designing an anti-emulator solution based on the popular 1.3.1 approach. The 1.3.2 and 1.3.3 approaches are out of scope in this project.

1.4 Outline

This report will continue with the following sections. First, some related works about anti-emulator detection will be introduced. Next, the overall design of the project will be discussed. Then, the current status, limitations, and difficulties encountered will be reported. Finally, There will have a short conclusion and the future plan.

2. Methodology

2.1 Related works

A previous research [6] proposed an anti-emulator detection approach called API hooking. The idea was changing the API behavior during runtime, hence the system information retrieved by emulator detection were modified. As the result, the emulator detection was not able to identify the sandbox as a sandbox. In this research [6], it summarized the status of anti-emulator detection:

1- Android Layer Hooking

> Also named Java Layer Hooking, as the hooked method are come from Java libraries.

2- Linux Layer Hooking

> Linux Layer's functionality is supported by Java Native Interface (JNI). The type of hooking is focus on the native methods.

3- (Out of scope) User behavior / Emulator-based behavior detection

> This is not very effective emulator detection approach. It is not necessary to be implemented.

The primary solution proposed by this research was utilizing an Xposed Framework module - BluePill. This module includes some known emulator detection like *android.os.Build* checking, wifi / gps checking, */system/build.prop* checking, *ActivityManager.isUserAMonkey* checking, qemu-based checking, etc. When there is a new app launched (i.e. fork a zygote), BluePill will also initialize the API hooks to the app, if the app is not within the whitelist defined in BluePill. In general, BluePill has capability to automatically add defined API hooks to every new called process. However, BluePill purely focuses on Android Layer Hooking only. For JNI methods (i.e. __system_property_get), BluePill cannot to do anything on it. Instead of using Android Dynamic Binary Instrumentation Toolkit, Frida is a good candidate for implementing the anti-emulator detection module. The next section will introduce several abilities of Frida to show how it works.

2.2 Introduction to Frida

Frida is an open-source dynamic instrumentation toolkit, which implements JavaScript code injection by writing code directly into process memory, along with a powerful API that provides a lot of useful functionality, including calling and hooking native functions and injecting structured data into memory [18]. The following demonstration is running in Frida 12.2.27 under Python2 binding, along with Android 6.0-r3 x86 image.

For example there is an Android app named *com.example.one*. Within its *onCreate()* method:

```
>       if (Build.TAGS.contains("test-keys")){
>               Log.d("It is emulator!");       /* Terminate the app */
>       } else {
>               Log.d("It is real device!");    /* Do something  */
>       }
```

Build.TAGS is one of the string information extracted from *build.prop*, and normally an emulator would have a value contains string *test-keys* (i.e. Real device should be *release-keys*). To bypass this detection, Frida can solve by inject the javascript as follow - *exanple_one_bypass.js*:

```
>       'use strict';
>       Java.performNow(function(){
>       var String = Java.use('java.lang.String');
>       String.contains.implementation = function(arg0) {
>               if (arg0 == 'test-keys') return false;
>               return this.contains.call(this, arg0);
>       };
>       });
```

Frida injects *exanple_one_bypass.js* into the process of *com.example.one* using python - *example_one_inject.py*: (Precondition: connected to the emulator via adb)

```
>       import frida
>       with codecs.open("exanple_one_bypass.js", "r", encoding="utf-8") as f:
>               js_code = f.read()      # Transfer to String
```

```
>        device = frida.get_usb_device(timeout=5)    # Get the emulator connected via adb
>        pid = device.spawn(['com.example.one'])     # Start the app in suspended state
>        session = device.attach(pid)
>        script = session.create_script(js_code)     # Inject the js code
>        script.on('message', on_message)            # on_message method handles data
>                                                      sent from emulator
>        script.load()                               # Enable the js code
>        device.resume(''com.example.one'')          # Continue to run the app
>        time.sleep(1)
>        sys.stdin.read()
```

Now the example app will return *false* when it attempts to check if string contains 'test-keys'.

However, It is not easy to implement Frida as an anti-emulator detection module. First, at the beginning of the app, the injected javascript is requested to include all anti-emulator methods that will be used in the app. Therefore those methods are required to be well predefined.
Second, hooking native methods are also required when emulator methods occur. In the past, malware researchers studied native methods by direct observation since native methods (C/C++) are not standard libraries. It is hard to identify the behavior of native methods without analyzing the code. We have to consider if there is any possibility to implement static analysis to check emulator detection activities in native layer.
Third, the app launching approach in Frida is different from the original approach in Cuckoodroid. To start using Frida in Cuckoodroid, it is unavoidable to modify the logic inside the processing stage.
Finally, Android malware is possible to separate into several different processes once the app is launched. Luckily, Frida also supports another app launching approach called *SpawnGating*. It allows user to to add hooks to every new process. Furthermore, every process is allowed to send message back to the host. It is possible to affect the behaviour of Frida script at host via the context of received message. Hence, the design of message is necessary to be considered.

2.3 Introduction to Cuckoodroid

This project will utilize an existing open-source automated analysis tool called Cuckoodroid [9]. It will be further developed to fit the project objective. The architecture of CuckooDroid is shown in Figure 1. This project is designed to use VM X86 Port to run the malware.

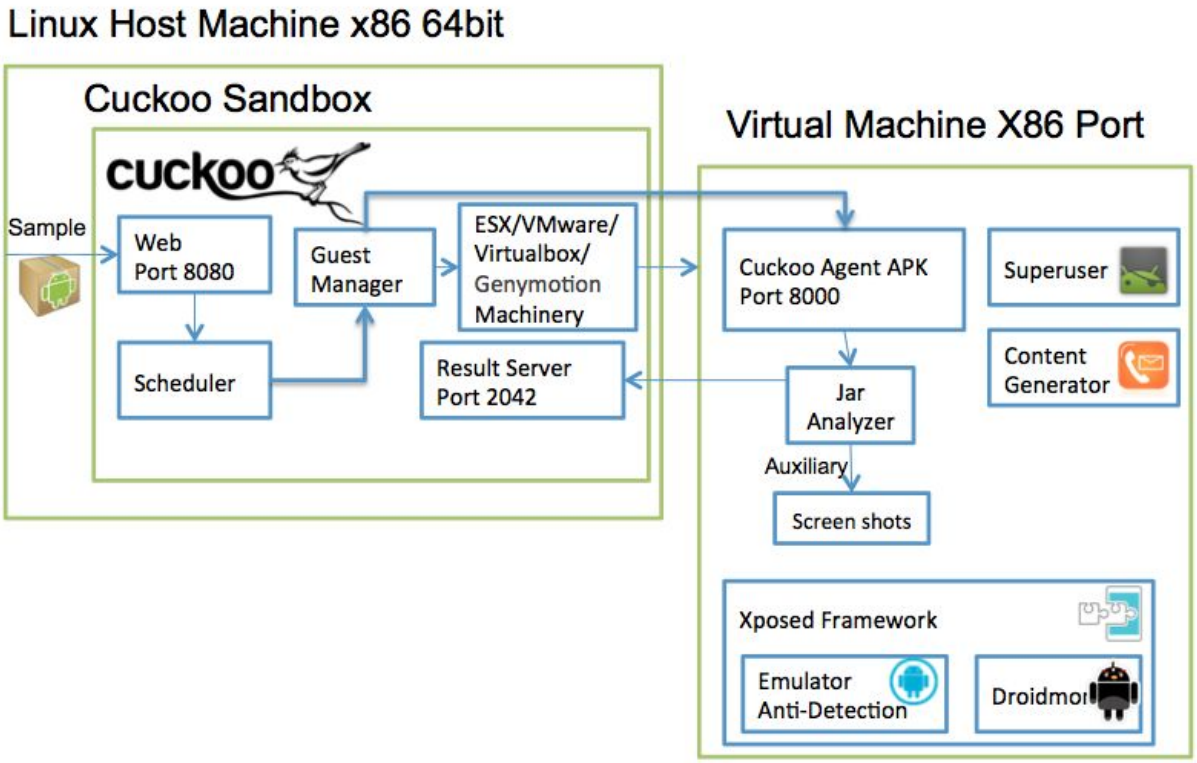


Fig 1: CuckooDroid architecture with VM X86 port [9]

In order to implement a new anti-emulator detection module, this project will plan to enhance cuckoo sandbox capability. The following is the description of this architecture:

1- Scheduler

> This is a core module in cuckoo sandbox, in order to look at the state of guest machines, and assign analysis tasks to those machines.

2 - Guest Manager

> This is a core module in cuckoo sandbox, in order to define the state of each guest machine. In

Python, each machine belongs to a GuestManager object.

3 - Machinery

> This is a sub module in cuckoo sandbox, in order to provide all necessary functionality of the guest machine based on its type. Each type of VM is a subclass of Machinery in Python.

4 - ResultServer

> This is a core module in Cuckoodroid, in order to control the analysis result returned from the emulator.

5 - Cuckoo Agent APK

> This is the apk defined set of method using to download the apk from host, install and run the apps, turn on the screenshots thread, and finally send back the droidmon log to the result server.

6 - Analyzer JAR

> For every analysis task, host will upload this core module into the emulator. This module is loaded by Cuckoo Agent APK dynamically during runtime.

7 - Droidmon

> This is a core module in Cuckoodroid, in order to log the malware events which is dangerous to the normal user. The methods interested are stored in **/data/local/tmp/hooks.json**.

8 - Emulator Anti-Detection / BluePill

> As mentioned in section 2.1, this is a sub module in Cuckoodroid, in order to bypass some known emulator detection. This project will replace this module by Frida.

9 - Superuser

> This is a core app in Cuckoodroid, in order to provide root privilege. (i.e. su binary)

10 - Contact Generator

> This is an optional app in Cuckoodroid, in order to bypass a known user behavior detection.

11- Web interface at port 8080

> This is the default Django web server for showing the analysis report. It is optional since the report can be shown as a HTML file in the */cuckoo/storage/analyses/* folder.

2.4 BluePill in Frida - Design

The following is the logic workflow of the Frida BluePill. Lines with '+' are new implementation features occurred in this project. Furthermore, the later sections will explain the main purpose of the new design and briefly describe the programming implementations.

> For Every Analysis Task:
>+ **[Section 2.4.1]** Perform static analysis of the APK
> Restore VM snapshot
>+ **[Section 2.4.2]** Connect via adb (i.e. usb connected in Frida view)
>+ **[Section 2.4.2]** Run frida-server
>+ (i.e. cmd run adb shell su -c /data/local/tmp/frida-server &)
>+ **[Section 2.4.2]** Run Frida in python binding with enabling *SpawnGating* mode
>+ **[Section 2.4.2]** Prepare injection javascript for anti-emulator detection
> [Start Cuckoodroid Original Task]
> Push, install and run the APK
> Enable screenshot thread.
>+ **[Section 2.4.3]** Monitor dynamic loaded binaries
>+ **[Section 2.4.3]** Reload the injected javascript if necessary
> [End Cuckoodroid Original Task when timeout]
> Send back the droidmon log to resultserver
> [Start analysis report generation]
> Parse the droidmon log to show malware features (Signatures Check)
>+ **[Section 2.4.4]** Show emulator detection features (New Signatures Check)
> [End analysis report generation]
> End task

2.4.1 Static analysis on Android sample

Cuckoodroid has already implemented a static analysis to the malware sample by *androguard*. However, this analysis only interests on package names, activities, permissions and receivers. In order to have a better alert on malware which will perform emulator detection. For example, the API calls stated in section 1.3.1 can be treated as emulator detection signals, since these functions are normally used to identify emulators. In order to add the static analysis on method names, androguard can do some stuff:

```
>     # d for dex file
>     ret = []
>     for c in d.get_classes():
>           ret.extend(list(map(lambda x: x.name, c.get_methods())))
>     # remove duplicate in ret, continue...
```

The *ret* contains all the methods used within the apk. This will be further used by performing anti-emulator detection by Frida.

2.4.2 Enabling Frida services

In order to use Frida, frida-server is required to be started properly. Python can use *subprocess* or *sys* to read the command, hence it is possible to implement a process to connect the active android emulator, run command via adb and start the frida-server.

Next, prepare the javascript, which contains the API hooks just like the example javascript in section 2.2. For each hooked API call, it is necessary to predefine the implementation since there is no general frida code generator to handle anti-emulator detection. The first release will include the old BluePill's hooked methods. If time is available, we will have a second release. In second release, the javascript will be designed to be generated by python script after the static analysis.

Finally, a python script (see Appendix (A)) is run for Frida to inject the javascript. Host-side tasks will be occurred at this python script.

2.4.3 Capture dynamic loading activities in runtime

Frida does not only bypass emulator detection, it also can hook some API calls related to dynamic loading. In most of the case, dynamic loading will load the code which is not initially defined in the APK. Therefore static analysis will not able to detect the problematic features at the beginning. In the worst case, emulator detection is possible to be performed after loading the code from the internet source. Hence, analyzing the dynamic loaded binaries is necessary, since the analysis is required to perform effective anti-emulator detection. In addition, the dynamic loaded binaries are attached with the analysis report for user references.

The implementation includes 2 parts. The first part is the javascript implementation. The API hook added into javascript is triggered when the app is attempting to read files from external sources (i.e. external storage and internet source). The files, then will pass back to the host.

The second part is the python implementation. For example, *on_message* method in the python script (See Appendix (A)) is the handler of message from frida-server. It is possible to trigger another process based on the receive message in the python side. Furthermore, according to the documentation of Frida, its Javascript API is allowed to perform file reading. Hence, the message handler *on_message* will capture the files in the message when the message contains bytecode in the message body.

2.4.4 Show emulator detection activities in report

In section 2.4.1, the static analysis will report the API calls used within the APK. In order to alert the malware will perform emulator detection, the analysis report should list out those related API calls. In Cuckoodroid design, this task is related to *Signatures* processing. To implement this features, we will design a new signature template (which is actually a python script). This script will check the API calls used by the application to see if those API calls is possibly performing emulator detection. Noted that this signature checking is just matching the API method names, it is possible to show false positive result (i.e. Check network status).

3. Evaluation

3.1 Current Status

The following progress was recorded before 14 January 2018:

3.1.1 Research study

Many malware analysis researches has been conducted in the past 20 years. They are good sources for learning how to conduct a regular malware analysis research. At the beginning, Android security architecture is well read and this basic knowledge will help to further understand other concepts in Android malware analysis research papers [12]. Then, a previous anti-emulator detection research is well read this project which indicates the main components are still required to discover in future malware analysis researches [5]. Next, a latest Android dynamic analysis research in Hong Kong is well read, which pointed out the missing information in current malware behavior logging function [13]. Furthermore, code documentations are well read to understand how to cooperate with those open source frameworks [9]. In addition, the implementation of API Hooks will rely on current code design, therefore the related syntax and the module implementation is necessary to be well studied [6,14,16].

3.1.2 Cuckoodroid setup

The sandbox server has been built on MacBook with OS X 10.13.2. The version of the Cuckoo framework is using v1.2. Android emulator is set as a guest machine on VirtualBox 5.1, and the Android version is Android x86 6.1 RC3. In addition, the network configuration was tested well within the home wifi condition. There are 2 network adapters is used by the emulator. Everytime the emulator snapshot is restored, the emulator is successfully connected via ADB.

3.1.3 BluePill in Frida progress

The whole program is separated into 2 PyCharm projects. First project is the cuckoodroid framework debugging, which is the python script running the sandbox server. In order to discover bugs in Cuckoodroid, different Android applications (Normal applications from PlayStore and some mobile malware from MalShare) has been tested. The analysis result seems to be normal, but there is some unexpected exception when the sample APK does not have *MainActivity* (i.e. Fail to identify the starting point of that APK). In this case, cuckoodroid will fail the analysis with following an exception. This specific case has to be further researched to see the rationale.

The second project is BluePill in Frida development, which is the python script for starting frida-server on emulator and then inject the anti-emulator detection code.

Since 2 project objectives are different, it is necessary to ensure each project is allowed to run independently. That's the rationale of the project separation.

Please note that in the final release, the second project for BluePill in Frida will be merged into the project of cuckoodroid.

3.1.4 Collection of emulator detection related API methods

Apart from the methods contained in the original BluePill APK in cuckoodroid, other related methods are extracted from the list of API provided by past malware researchers [20, 21]. These methods will be handled by the javascript and then pass to Frida to perform anti-emulator detection. One of the big challenge is handling the system property check (i.e. how Android can check system property). Based on each possible way, we have to implement a counter-script in order to let the emulator act as a real device. For native methods, they did not have very clear regular pattern when we observed the sample emulator checking JNI methods (i.e. child process property difference between real device and emulator) [22]. Therefore it is necessary to discover more the emulator detection native methods.

3.2 Limitations and Difficulties Encountered

This section will discuss critical issues and potential risks found in the current status.

3.2.1 Host machine safety

Cuckoo requires a host machine to run the sandbox server. Nevertheless, the recommended host operating system is Ubuntu 16.04. In order that the sandbox would run the network analysis, the operating system is expected to give non-root permission to the *tcpdump* service. To enable non-root permission for tcpdump, System Integrity Protection has to be disabled with *csrutil* and then run *sudo chmod +s /usr/sbin/tcpdump*. This is a potential risk of being exploited if other security settings are also affected after System Integrity Protection is disabled. However, this risk is only existing in short term as the project is still in the development stage.

3.2.2 Cuckoodroid compatibility

Cuckoodroid is only available to cooperate with Cuckoo v1.2. Compared to the latest stable Cuckoo version is v2.0.6, Cuckoo v1.2 is completely out-of-date and lots of new functions are not included in v1.2. However, anti-emulator detection has no change, compared with v.1.2 and v2.0.6. Hence the implementation of Frida anti-emulator module would not be affected in the newer cuckoo sandbox.

3.2.3 Unstable Host-Guest network

To maintain the communication between the sandbox server and android emulator, static IP addresses are requested for both core modules. Recently, the emulator and host is connected via a custom network interface *vboxnet0* created by VirtualBox. The IP addresses of host and emulator are static (i.e. 192.168.56.1 and 192.168.56.201). In addition, in order to allow network access, NAT network adapter has to be set. After testing the network, the clean snapshot of the

emulator is saved for cuckoodroid usage. However, when the snapshot is restored, the network failed. After a week of investigation, the problem seems to be related to improper resume implementation inside the x86 image. Therefore it is not possible to be fixed in short time. Accidentally, I occurred that when Android 6.0 reboot, the opened applications will be open automatically. Also, the network is stable after the boot. Hence I have tried to take the clean snapshot by setting up of the necessary application and then reboot. The snapshot has been taken just before its booting logo shows up. Although the waiting time for machine ready is being longer (about 30 seconds), this adjustment has temporarily solved the unstable network problem.

3.2.4 High CPU usage of Frida-server

While testing with Frida's *SpawnGating* approach, the whole emulator becomes very slow. The rationale is *SpawnGating* trying to interrupt every call of the new process. However, the x86 image is only allowed to use 1 vCPU as the maximum option. This means the emulator with Frida is actually slower than a real device. Hence, there is a potential risk that this emulator cannot bypass some emulator test like CPU timer [19].

3.2.5 Anti-Frida features in sample

According to OWASP, App developers are ability to implement some Anti-Frida detection methods [19]. Therefore, malware can also do emulator detection likewise action, by detecting existence of Frida before running the malicious script. This is the limitations of Frida since Frida requires to turn on its functionality by running frida-server within the emulator or real device. App developers, therefore design the anti-reversing engineering of Frida by watching this frida-server characteristic. However, Frida is still not common in the software industry, most malware is supposed not to have anti-frida features. Therefore anti-Frida is not a very immediate issue to this project.

4. Conclusion

This project attempts to implement a reverse engineering module for anti-emulator detection. This is a very challenging topics since it requires to utilize deep multi-thread concept in order to make the API hook in every process are running as expected. Furthermore, OWASP has mentioned that emulator detection in Java Native Interface is more common than that in Java Library [19]. It is unavoidable to develop another new anti-emulator detection solution which allows to cover API calls in both layers. BluePill in Frida is a good starting point to show how can malware research to be conducted with a functional anti-emulator detection module in most of the situation.

5. Future Plan

The working plan will be shown in sequence. First, the sandbox server and Android emulator will be set up before Mid December. Next, we will work out the prototype of Frida Blue Pill before late January. After that, the whole module will be implemented before 1 March. Finally, debugging and system tuning will be performed in March and April.

References

[1] IDC (n.d.). Smartphone OS Market Share [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os [Accessed: 2018, Oct 30].
[2] C. Lueg. (2017, Apr 17). 8,400 new Android malware samples every day [Online]. Available: https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day [Accessed: 2018, Oct 30].
[3] L. H. Newman. (2017, Sep 22). HOW MALWARE KEEPS SNEAKING PAST GOOGLE PLAY'S DEFENSES [Online]. Available: https://www.wired.com/story/google-play-store-malware/ [Accessed: 2018, Oct 30].
[4] F. Wei., Y. Li., S. Roy. and W. Zhou. (2017). Deep Ground Truth Analysis of Current Android Malware [Online]. Available: http://amd.arguslab.org/evolution [Accessed: 2018, Oct 30].
[5] T. Vidas and N. Christin. (2014). Evading Android Runtime Analysis via Sandbox Detection [Online]. Available: https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf [Accessed: 2018, Oct 30].
[6] W. Hu. and Z. Xiao. (2014). Guess Where I Am-Detection and Prevention of Emulator Evading on Android [Online]. Available: https://github.com/MindMac/HideAndroidEmulator [Accessed: 2018, Oct 30].
[7] ac-pm. (2016). Android Package Inspector [Online]. Available: https://github.com/ac-pm/Inspeckage [Accessed: 2018, Nov 20].
[8] C. R. Mulliner. (2015) Android Dynamic Binary Instrumentation Toolkit [Online]. Available: https://github.com/crmulliner/adbi [Accessed: 2018, Nov 20].
[9] I. Revivo. (2015). CuckooDroid - Automated Android Malware Analysis [Online]. Available: https://github.com/idanr1986/cuckoo-droid [Accessed: 2018, Oct 30].
[10] MI9. (2017). APK Downloader [Online]. Available: https://apk.support/apk-downloader [Accessed: 2018, Nov 20].
[11] MalShare. (2012). Malware Data Source [Online]. Available: https://malshare.com/ [Accessed: 2018, Oct 30].
[12] A. Dubey. and A. Misra. *ANDROID SECURITY ATTACKS AND DEFENSES*. New York: Auerbach Publications, 2013.
[13] Chenxiong Qian., Xiapu Luo., Yuru Shao. and Alvin T.S. Chan. (2017). On Tracking Information Flows through JNI in Android Applications [Online]. Available: http://www4.comp.polyu.edu.hk/~csxluo/NDroid.pdf [Accessed: 2018, Nov 20].
[14] baeldung. (2018, May 27).Guide to JNI (Java Native Interface) [Online]. Available: https://www.baeldung.com/jni [Accessed: 2018, Nov 20].

[15] Javadecompilers. (2015). Decompiler online [Online]. Available: http://www.javadecompilers.com [Accessed: 2018, Nov 20]
[16] hamzahrmalik. (2018). [TUTORIAL]Xposed module development [Online]. Available: https://forum.xda-developers.com/showthread.php?t=2709324 [Accessed: 2018, Nov 20]
[17] C. Lueg. (2017, Apr 27). 8,400 new Android malware samples every day [Online]. Available: https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day [Accessed: 2018, Nov 20]
[18] Frida. (2018). Frida • A world-class dynamic instrumentation framework | Inject JavaScript to explore native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX [Online]. Available: https://www.frida.re [Accessed: 2018, Dec 20]
[19] OWASP. (2018). OWASP Mobile Security Testing Guide [Online]. Available: https://sushi2k.gitbooks.io/the-owasp-mobile-security-testing-guide/content/0x05c-Reverse-Engineering-and-Tampering.html [Accessed: 2018, Dec 20]
[20] L. Bello and M. Pistoia. Ares: Triggering Payload of Evasive Malware for Android. 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2018) 2018.
[21] W. Rashid. (2018). Automatic Android Malware Analysis [Online]. Available: https://github.com/waqarrashid33/internship_report/blob/master/main.pdf [Accessed: 2018, Dec 20]
[22] strazzere. (2013). HitCon 2013: "Dex Education 201: Anti-Emulation [Online]. Available: https://github.com/strazzere/anti-emulator/blob/master/AntiEmulator/jni/anti.c [Accessed: 2018, Dec 20]

Appendix

(A)- Python Script for Frida to inject the javascript in *SpawnGating* mode. Referencing from the tutorial on Github [18]. This is just a code demonstration and this is slightly different to the original example in the tutorial.

```
>       pending = []
>       sessions = []
>       scripts = []
>       event = threading.Event()
>
>       def on_spawned(spawn):      # new spawn handler
>               print(spawn-added:', spawn)
>               pending.append(spawn)       # add process to pending list
>               event.set()
>
>       def on_message(spawn, message, data):     # normal message handler
>               print('on_message:', spawn, message, data)          # Need to work on
>
>       device.on('spawn-added', on_spawned)
        # Define a handler on the device. Called when new process is created.

>       device.enable_spawn_gating()        # Start Spawn Gating
>       event = threading.Event()           # Set event lock
>       print('Enabled spawn gating')

>       print('Pending:', device.enumerate_pending_spawn())
>       for spawn in device.enumerate_pending_spawn():
>               print('Resuming:', spawn)
>               device.resume(spawn.pid)
        # Prevent new process occurred before the while loop

>       while True:     # Infinite loop, never end unless stop the device.
>               while len(pending) == 0:        # This is the while loop waiting for new process.
>                       print('Waiting for data')
>                       event.wait()            # Wait for new process call event.set()
>                       event.clear()
>               spawn = pending.pop()       # Take out a new process from pending list
>               if spawn.identifier is not None:          # Process Filter will be here
>                       print('Instrumenting:', spawn)
>                       session = device.attach(spawn.pid)  # Create a session of new process
```

```
>                       session.enable_jit()    # Enable '=>' syntax in javascript
>                       jscode = " "
>                       with codecs.open("antiDetection.js", 'r', encoding='utf8') as f:
>                               jscode = f.read()
>                       script = session.create_script("""\      # Inject the javascript to device
>               (function () {
>                       rpc.exports = {
>                               init() {
>                                       Java.perform(() => {
>                                       """+jscode+""" # Add anti-emulator detection at here
>                                       });
>                               }
>                       };
>               }).call(this);""")
>                       script.on('message', lambda message, data: on_message(spawn,
>                                                                               message, data))
>                       script.load()    # Enable the javascript
>                       script.exports.init()     # Run the injection
>                       sessions.append(session)     # For future reuse, if necessary
>                       scripts.append(script)           # For future reuse, if necessary
>               else:
>                       print('Not instrumenting:', spawn)
>               device.resume(spawn.pid)
>               print('Processed:', spawn)
```