Anti Android Emulator Detection

Accurate & Effective Dynamic Android Malware Analysis

Final 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 because of increasing smartphone users. Android, which is the biggest mobile market, attracts many hackers to develop malware to exploit users. To combat this situation, developing a trustworthy and fast malware analysis method is necessary. On the other hand, emulator detection in mobile allows malware to bypass automated malware analysis easily. If there is no solution to tackle emulator detection in malware, automated malware analysis will never be accurate since the malware is just ignoring the malware researcher.

Cuckoodroid, which is an open source automated malware analysis framework, is very famous to many security companies and malware researchers. However, this framework development on Github is inactive, and according to latest update, the author was still improving the performance of multi-tasking. Inside this framework, anti-emulation solution was using Xposed framework. According to a previous research [21], the original anti-emulator solution is not enough to bypass newer malware. Another problem is the device has to be restarted to apply all the changes in the Xposed code. Therefore it is not convenient when users need to frequently update the anti-emulator module.

This project built a more flexible and feasible anti-emulator module using Frida, in order to introduce an automated malware analysis environment, supported with a fast development environment for editing anti-emulator solution. This project had implemented some well-known anti-emulator detection methods by Frida to bypass the emulator checking in the malware.

This project gained a new feature in Cuckoodroid, so users could make use of Frida to implement anti-emulator detection code, and run the dynamic analysis as usual, with additional logs and notification to show how the anti-emulator detection takes places during the dynamic analysis.

Some root checking application available in Github, and real malware obtained from malware database were well-tested by the new Cuckoodroid. The result showed that the functionality of Cuckoodroid was not affected by the new implementation. Furthermore, it showed what emulator detection events which happened during the analysis in the output report. It is a useful information for researchers to know.

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

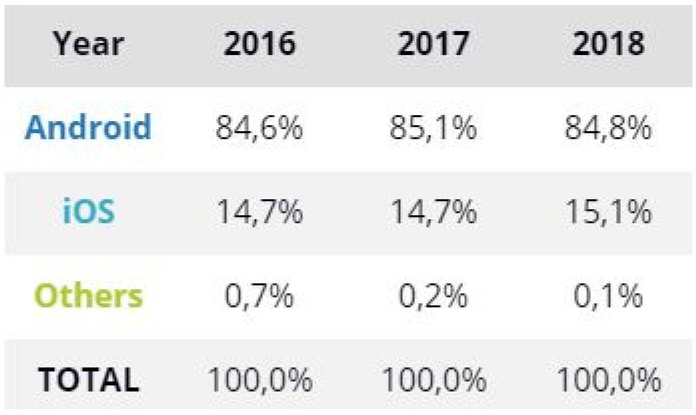
List of Figures

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, the 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 the Google Play Store [3]. The rationale was malware had anti-analysis techniques to bypass the security checking before releasing 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 solutions 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 directly 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

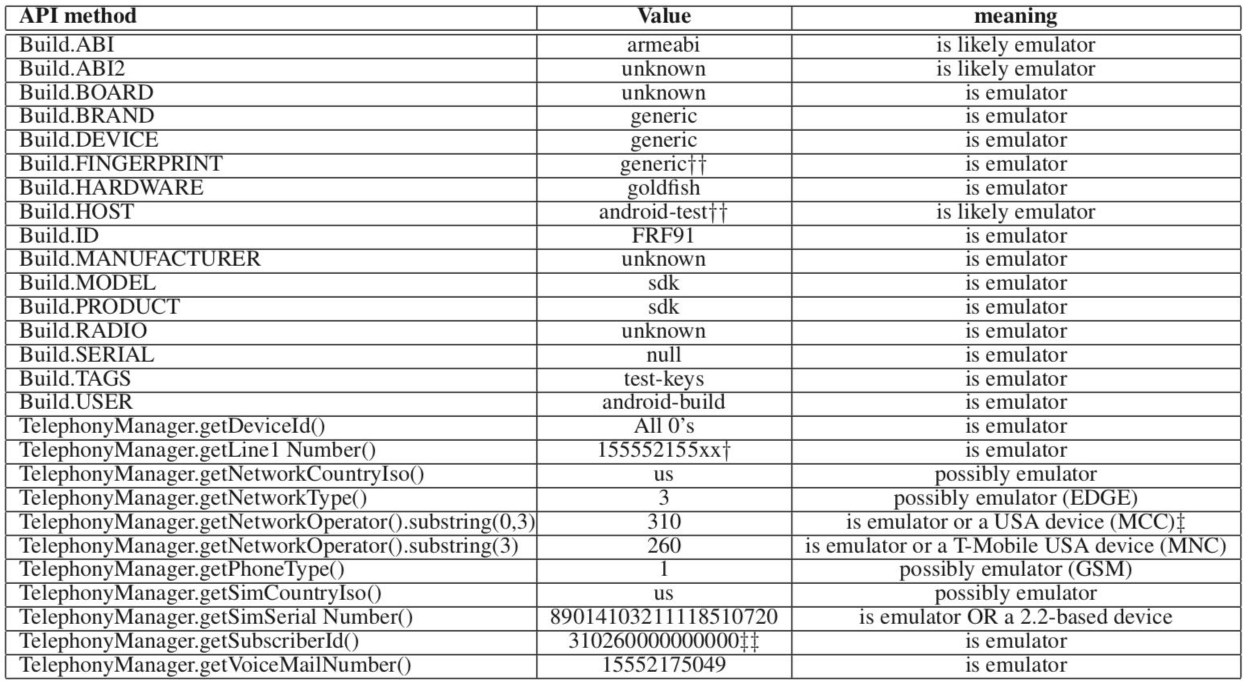
Let the user know the running application is performing known emulator detection methods.

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.

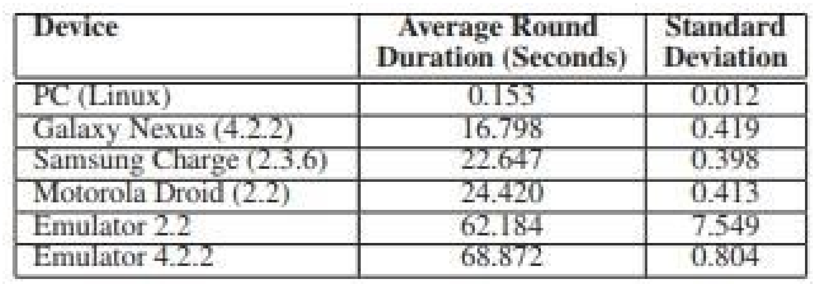


API method	Value	meaning
Build.ABI	armeabi	is likely emulator
Build.ABI2	unknown	is likely emulator
Build.BOARD	unknown	is emulator
Build.BRAND	generic	is emulator
Build.DEVICE	generic	is emulator
Build.FINGERPRINT	generic††	is emulator
Build.HARDWARE	goldfish	is emulator
Build.HOST	android-test††	is 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.



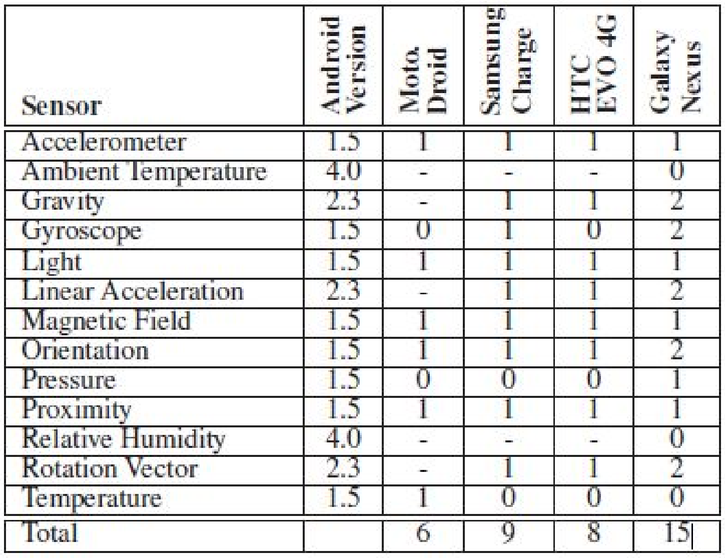
Device	Average Round Duration (Seconds)	Standard Deviation
PC (Linux)	0.153	0.012
Galaxy Nexus (4.2.2)	16.798	0.419
Samsung Charge (2.3.6)	22.647	0.398
Motorola Droid (2.2)	24.420	0.413
Emulator 2.2	62.184	7.549
Emulator 4.2.2	68.872	0.804

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 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. Related works

2.1 Previous Anti-emulator research

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 a 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

It is also named Java Layer Hooking, as the hooked methods come from Java libraries.

2- Linux Layer Hooking

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

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

This is not a 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 the 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 do anything on it. Instead of using Android Dynamic Binary Instrumentation Toolkit, Frida is a better candidate for implementing the anti-emulator detection module.

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].

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 it by injecting 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
>           pid = device.spawn(['com.example.one'])  # Start the app
>           session = device.attach(pid)
>           script = session.create_script(js_code)  # Inject the js code
>           script.on('message', on_message)        # on_message method
>           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 users to add hooks to every new process. Furthermore, every process is allowed to send messages back to the host. It is possible to affect the behavior of Frida script at host via the context of the received message. Hence, the design of the message format 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.

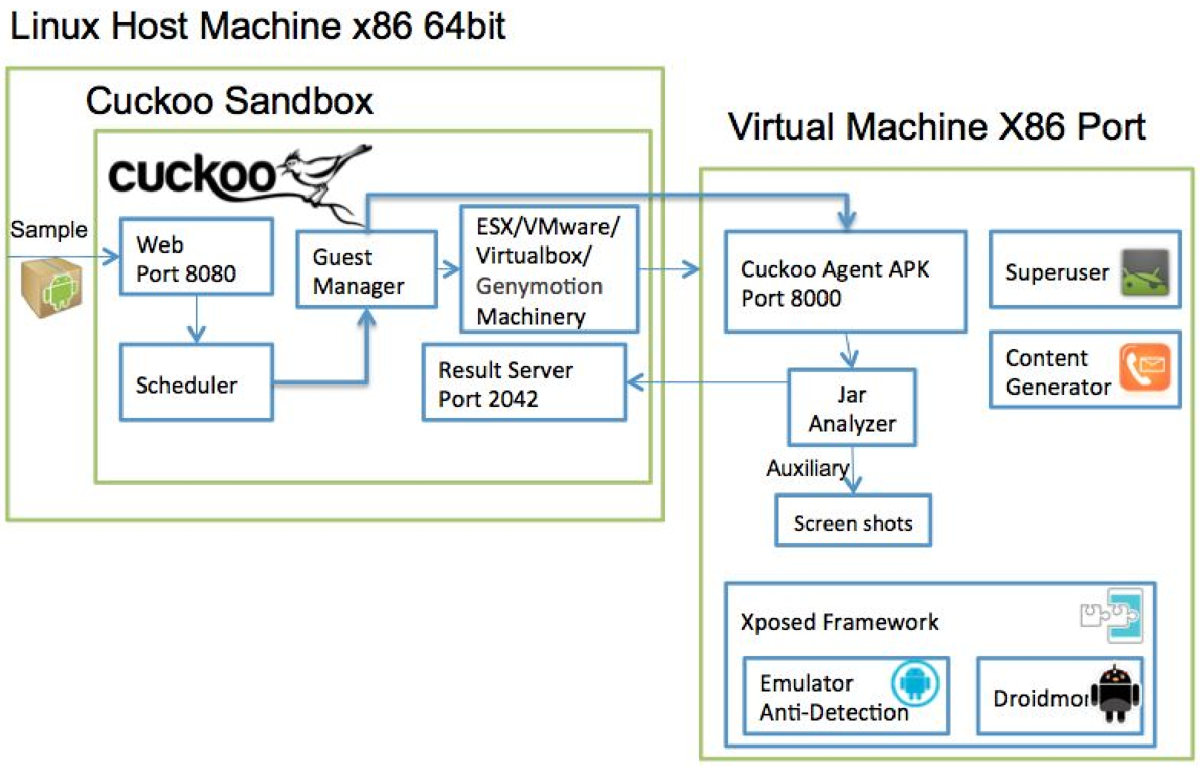


Fig 1: CuckooDroid architecture with VM X86 port [9]

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 submodule 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 methods using to download the apk from the 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, the 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 submodule 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 an HTML file in the */cuckoo/storage/analyses/* folder.

3. Methodology

This section will briefly introduce how the overall system works.

3.1 Environment Configuration Stage

This project is based on the existing Cuckoodroid framework, so in order to start using this tool, it is important to ensure the configuration of host and guest have to be set correctly. In this section, it will briefly describe the information of the development device

(I) Sandbox Server

The Cuckoo sandbox server has been built on MacBook with OS X Sierra 10.13.2, with i7 CPU core and 8GB RAM. The version of the Cuckoo framework is using v1.2.

(II) Android Virtual Machine

Android emulator is set as a guest machine on VirtualBox 5.1, and the Android versions available are **Android x86 6.1 RC3** and **Android x86 4.4 RC2**. Those VMs are downloaded from the official website of Android x86 open project website.

For each VM, the first network adapter is set to be a Host-only adapter. Then, in order to provide the capability to access the Internet, some network routing rules have to be changed in OSX by running the following commands in the host [9]:

```
> sudo sysctl -w net.inet.ip.forwarding=1
> rules="nat on en0 from vboxnet0:network to any -> (en0)
pass inet proto icmp all
pass in on vboxnet0 proto udp from any to any port domain keep state
pass quick on en0 proto udp from any to any port domain keep state"
> echo "$rules" > ./pfrules
> sudo pfctl -e -f ./pfrules
> rm -f ./pfrules
```

Where `vboxnet0` is your host-only network interface and `en0` is your main network interface, the name will depend on how did you configure your networks.

Next, in order to fix the IP address of the VM, It is necessary to open the Terminal app inside the VM and run the following commands [9]:

```
> su
> vi /etc/init.sh

    #add inside the file
    ifconfig eth0 192.168.56.10 netmask 255.255.255.0 up
    route add default gw 192.168.56.1 dev eth0
    ndc resolver setifdns eth0 8.8.8.8 8.8.4.4
    ndc resolver setdefaultif eth0

> reboot
```

Where `192.168.56.10` is the IP address you will be used to communicate with the host, and `192.168.56.1` is the IP address of your virtual network interface.

(III) Frida-server and its Python binding

This project is using Frida 12.2.27 (or higher version).

First, frida-server binary has to be pushed into the virtual machine in order to allow inject the scripts to modify the API performance when the dynamic analysis begins. For ease of management, all additional files for anti-emulator detection or API monitoring purposes will be put into `/data/local/tmp/` folder in the Android VM.

Second, Frida's python binding has to be installed. The fastest way to install is via pip.

Please note that the version of frida-server in Android VM and that of the Frida binding in the host are equivalent, otherwise the connection of guest and host will never be successful.

After the environment configuration, it is ready for use in the Cuckoodroid framework.

3.2 Sandbox Server Implementation Stage

Originally, Cuckoodroid framework did not contain Frida. In order to allow Frida to assist the dynamic analysis, the original code has been changed. In this section, it will briefly describe how to configure the Frida module within Cuckoodroid.

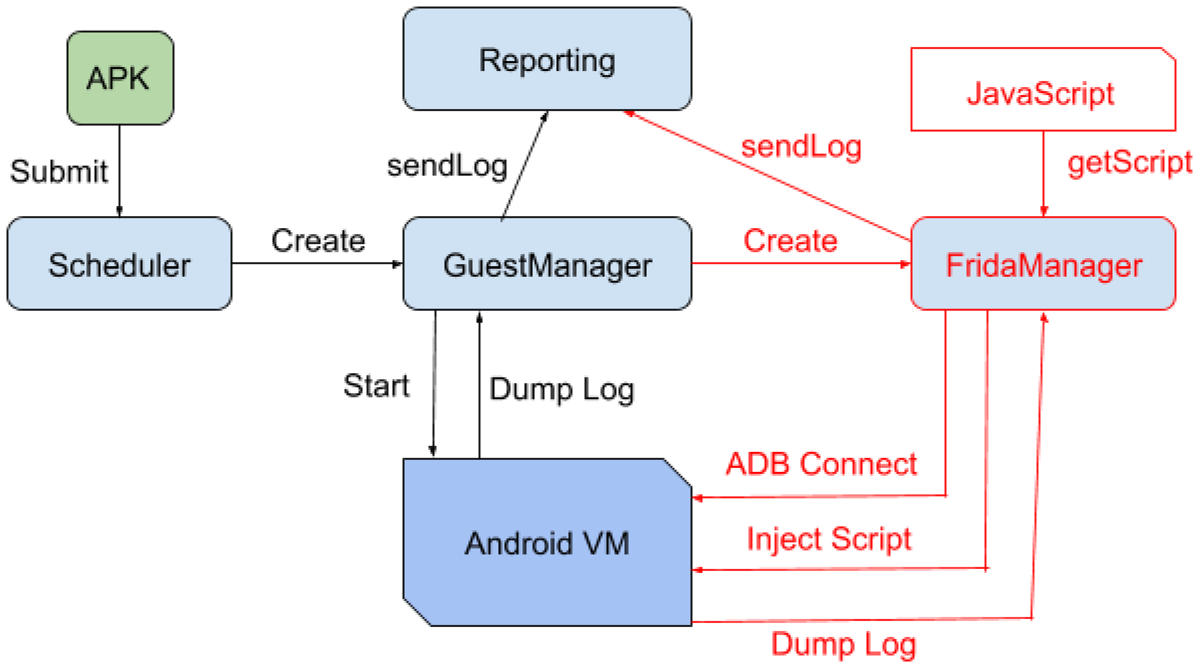


Fig.2 The current implementation of FridaManager

In the above figure, the items in red color are the new features in this project. They are used to support Frida.

3.2.1 FridaManager

This module is implemented by python within the cuckoodroid framework. It is a subclass of Thread class. In summary, Frida Manager can perform the following functions:

(i) Connect the Android VM via adb.

(ii) Get JavaScript (in Frida defined syntax) which is using to change the process behavior in the Android VM. (Further discussion in the next section.)

(iii) Turn on frida-server in the Android VM.

(iv) Inject the JavaScript into the running process.

(vi) Handle the message back from the Android VM, dump them into the log if those are related to emulator detection activities. The logs are used to generate a report to show how Frida handle those activities.

Every time when there is a new APK sample submitted, the Frida Manager will be created by the GuestManager, and then initialize the connection and then inject the JavaScript into the target (the sample APK).
When the analysis timeout is reached, GuestManager will turn off the Android VM. At the same time, GuestManager will also trigger the destroy function in the FridaManager, then FridaManager will stop working and destroy itself (i.e. Prevent from deadlock state). It is important to stop Frida Manager thread after every analysis task since Cuckoodroid itself is designed for performing consecutive dynamic analysis with many malware samples automatically.

3.2.2 Frida JavaScript

This is the most important component of the whole project. When there are new emulator detection methods occurred, Frida allows users to implement their own JavaScript in order to bypass the emulator detection. Based on the type of emulator detection, the following will list out what anti-emulator detection that this project has implemented.

(I) Device Information based emulator detection

Table 2 showed some of the API calls which are also effective emulator detection methods. For each API calls, relevant scripts are added in order to change the return value like the following code snapshots:

```
> var TelephonyManager = Java.use("android.telephony.TelephonyManager");
> TelephonyManager.getDeviceId.overload().implementation = function () {
>                  //Change return value
>};
```

According to the original design of BluePill, A hashmap of SystemProperties is created in the scripts, in order to identify which value should be returned.

```
var SystemProperties = Java.use('android.os.SystemProperties');
SystemProperties.get.overload('java.lang.String').implementation = function (key) {
   if (key in build_prop) {
       return build_prop[key];
   }
   return this.get.call(this, key);
};
```

Moreover, system properties are not only can be requested by the Java layer, and also the Native layer. Hence, a similar API hook is also required to be implemented in order to fully bypass this kind of emulator detection.

```
Interceptor.attach(Module.findExportByName("libc.so", "__system_property_get"), {
   onEnter: function(args) {
       var prop = Memory.readCString(args[0]);
       if (prop in build_prop) {
           Memory.writeUtf8String(args[1], build_prop[prop]);
       }
   },
   onLeave: function(retval) { }
});
```

Note that the above 2 functions work identically.

This project has implemented the API hook for type to the following API:

- android.os.Build.TAGS
- android.os.Build.BOARD
- android.os.Build.BRAND
- android.os.Build.DEVICE
- android.os.Build.FINGERPRINT
- android.os.Build.HARDWARE
- android.os.Build.ID
- android.os.Build.MANUFACTURER
- android.os.Build.MODEL
- android.os.Build.PRODUCT
- android.os.Build.SERIAL

- android.os.Build.USER
- android.location.LocationManager.getLastKnownLocation()
- android.location.Location.getLatitude()
- android.location.Location.getLongitude()
- android.os.SystemProperties.get() || libc.so → __system_property_get(&k, &v)
- android.telephony.TelephonyManager.getDeviceId()
- android.telephony.TelephonyManager.getLine1Number()
- android.telephony.TelephonyManager.getNetworkOperatorName()
- android.telephony.TelephonyManager.getNetworkOperator()
- android.telephony.TelephonyManager.getSimSerialNumber()
- android.telephony.TelephonyManager.getNetworkCountryIso()
- android.telephony.TelephonyManager.getSubscriberId()
- android.telephony.TelephonyManager.getNetworkType()
- android.telephony.TelephonyManager.getPhoneType()
- android.telephony.TelephonyManager.getVoiceMailNumber()
- java.io.BufferedReader.readLine()
- java.lang.String.contains()

(II) Debugging detection based emulator detection

This project has implemented the API hook for type to the following API:

- Debug.isDebuggerConnected()
- android.app.ActivityManager.isUserMonkey()

(III) File existence based emulator detection

This project has implemented the API hook for type to the following API:

- libcore.io.IoBridge.open(path, flag)
- java.io.File.exists() | libc.so → fopen
- libc.so → system
- java.lang.ProcessBuilder.start()

- java.lang.ProcessManager$ProcessImpl.getInputStream()
- android.app.ActivityManager.getRunningServices() [21]
- android.app.ApplicationPackageManager.getPackageInfo() [21]

(IV) Network information based emulator detection

This project has implemented the API hook for the type to the following API:

- android.net.wifi.WifiInfo.getMacAddress()
- java.net.NetworkInterface.getName()

(V) Shell command based emulator detection

This project has implemented the API hook for type to the following API:

- java.lang.ProcessManager.exec() (Note: Have multiple overload methods) [29]
- libc.so → system
- java.lang.ProcessBuilder.start()

3.3 Reporting module

When some known emulator detection was found, the dynamic analysis report is supposed to include those happened API activities in order to have a full picture of the purpose of the testing malware sample. Hence, the original reporting module has to be modified in order to process the additional log given by the Frida JavaScript, extract necessary information from the log, and finally parse into the analysis report. In this section, it will explain how to achieve this goal by adding and changing the Cuckoodroid framework.

3.3.1 Frida JavaScript and message handler

Section 2.2 has explained that Frida allows sending the text back to the python binding. In practice, this feature is not only limited to send back the plaintext, treat the content as a JSON object would provide lower the data processing complexity in the python side.

```
function send_msg(msg) {
    send({"type" : "msg", "Process" : Process.id, "message" :  msg});
};
```

In the above code snapshot, it shows the `send_msg(msg)` function in the Frida JavaScript, which is implement the original `send(msg, [,data])` function in Frida JavaScript API, in order to send the required information back to the python binding for every time other than just do `send(msg)`. For type, it is used to identify what type of message, so different message data handling is possible. For Process, it is using to store the id number of the process which triggers the send() to send the message. Process id of the message is important information that the FridaManager has to keep track with since there is a possible situation that the malware sample is trying to spawn a new process that actually performing malware behavior. In this case, the Frida Manager will receive messages from 2 or more different processes. In order to keep track with the whole picture, separate the process id of the message will be more clear.
Note that, this send_msg() is used to in the debug mode of cuckoodroid only. Next, the droidmon_log function is based on the idea of this send_msg() function in order to go further.

```
function droidmon_log(msg, data) {
    send({"type" : "droidmon", "Process" : Process.id,  "message" :
msg, "data": data});
};
```

Function body of droidmon_log()

```
    var date = new Date(); var timestamp = date.getTime();
            droidmon_log("Bypass isDebuggerConnected",{
                "class":"android.os.Debug",
                "method":"isDebuggerConnected",
                "timestamp": timestamp,
                "type":"content",
                "args": "",
                "result": this.isDebuggerConnected.call(this),
                "hook_result": false
            });
```

Example use case

```
pid = payload.get("Process")
if pid:
   pid = spawns.get(pid, pid)
t = payload.get("type")
msg = payload.get("message")
data = payload.get("data")
log.debug('on_message: {} {} {}'.format(pid, msg, data))
write_to_file(log_dir, "emulatorDetect.log", data, json_enabled=True)
```

Python side message handler for type == droidmon

The format of JSON content is according to the droidmon module in the original cuckoodroid framework. In order to maintain a similar objective as this emulator detection log is also an API log to the processes. In addition, Frida should be better as it can print out the child process generated from the malware sample, but droidmon is not likely to take care of the child process. But the project time is limited so droidmon is not to be replaced.

For every defined anti-emulator detection method, a droidmon message will send back to the binding. Next, it will check for the PID if it exists in the hashmap of process id which has been injected the Frida script before. Afterward, it will extract the data part of the message and append the data into the emulatorDetect.log, which will be used to process the result into the written analysis report.

3.3.2 Additional codes in droidmon.py

In the original Cuckoodroid reporting module, all the logs generated by the droidmon module during the dynamic analysis will be processed in droidmon.py. In this project, lines of code have to be added in order to dump the emulator log into the report.

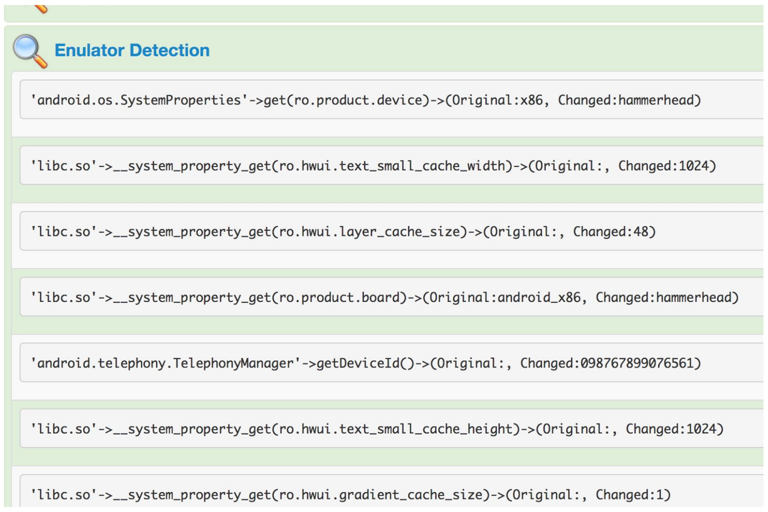
```
log_path = self.logs_path + "/emulatorDetect.log"
if not os.path.exists(log_path):
   pass
```

```
else:
   def checker(n):
       if isinstance(n, six.string_types):
          return n.encode('utf-8')
       return n
   for line in open(log_path, "rb"):
       try:
          data_t = json.loads(line)

          c = data_t.get('class').encode('utf-8')
          m = data_t.get('method').encode('utf-8')
          args = data_t.get('args')
          args = list(map(checker, args))
          original_r = data_t.get('result')
          if isinstance(original_r, six.string_types):
             original_r = original_r.encode('utf-8')
          hooked_r = data_t.get('hook_result')
          if c and m:
            msg_t = "'{}'->{}({})->(Original:{},.........
            Self.droidmon["emulator_detection"].add(msg_t)
```

Python script for processing the emulatorDetect.log

Also, the web interface of cuckoodroid is using Django. This project has made slight changes in the layout in order to present the additional content of emulator detection log.


Enulator Detection

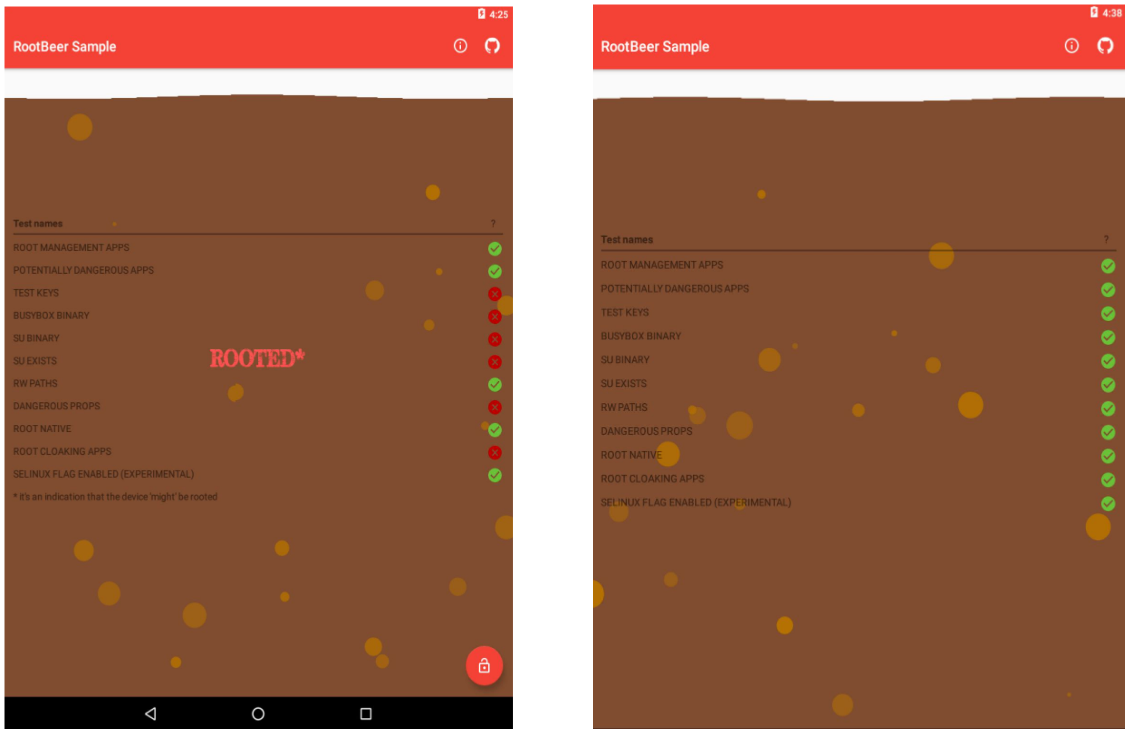
```
'android.os.SystemProperties'->get(ro.product.device)->(Original:x86, Changed:hammerhead)
'libc.so'->__system_property_get(ro.hwui.text_small_cache_width)->(Original:, Changed:1024)
'libc.so'->__system_property_get(ro.hwui.layer_cache_size)->(Original:, Changed:48)
'libc.so'->__system_property_get(ro.product.board)->(Original:android_x86, Changed:hammerhead)
'android.telephony.TelephonyManager'->getDeviceId()->(Original:, Changed:098767899076561)
'libc.so'->__system_property_get(ro.hwui.text_small_cache_height)->(Original:, Changed:1024)
'libc.so'->__system_property_get(ro.hwui.gradient_cache_size)->(Original:, Changed:1)
```

Fig.3 Report View

4. Results

4.1 Test with Root Checking Application

Before testing with the malware sample, testing with some root detection applications like RootBear, RootInspector, or other existing android root detection application from Github to see the basic function of anti-emulator detection is working as expected performance.
This test is important since if the malware really wants to bypass analysis, a rooted device is supposed to be performing malware analysis.



Test names	(Left)	(Right)
ROOT MANAGEMENT APPS	✓	✓
POTENTIALLY DANGEROUS APPS	✓	✓
TEST KEYS	✗	✓
BUSYBOX BINARY	✗	✓
SU BINARY	✗	✓
SU EXISTS	✗	✓
RW PATHS	✓	✓
DANGEROUS PROPS	✗	✓
ROOT NATIVE	✓	✓
ROOT CLOAKING APPS	✗	✓
SELINUX FLAG ENABLED (EXPERIMENTAL)	✓	✓

ROOTED*

* it's an indication that the device 'might' be rooted

Figure 4,5: (Left)Testing with RootBeer without Frida. (Right)Testing with RootBeer with Frida.

As shown in Figure 4 and 5, they are the screenshots of running RootBeer in Android 6.0 VM (Same result in Android 4.4). Originally, the VM cannot pass the test-keys test, su-binary test, busybox-binary test, su-exists test, props check and root apps test since those are not avoidable to install in the VM in order to make sure the original Android function to be available. After using Frida, those root features are hidden and then RootBeer is no longer to be able to find out the VM is rooted now.

Other passed application: (Which means the detection said the Android VM is not rooted)

- Android Hostile Environment Detection [23] (Make changes in Gradle as it fails to build APK using ndk-build)
- Android Emulator Detect [24] (Slightly modified part of the code as deprecated API is used)

As the first testing stage, the overall performance of anti-emulator detection is working correctly. It gives a positive result to show that the Frida JavaScript works in an expected manner.

4.2 Test with Malware Samples

Samples are using APKs from androzoo [25] to do the testing:

- es.pfleon.retateline.apk
- kr.mintech.btreader_survey_us.apk
- com.barnamefa.epistle_google_inapp
- com.novelmisterikisahnyata.abdulmudah.apk
- com.KoreaGirlFashion.Secelup.apk
- com.MotorTraiL.odieapps.apk
- com.TartCakeDesignCollection.heidismith.apk
- com.WeddingBouquetIdea.heidismith.apk
- Etc. Total around 50 malware samples

For most malware running with Chromium, they are launched and run as expected. In addition, the emulator detection activities are also detected by this project's Frida module (i.e. com.MotorTraiL.odieapps.apk will check the running device to see if `Superuser.apk` exists, even though the Android VM in use does not contain this apk. The following line is the emulation detection shown in the analysis report).

```
'java.io.File'->exists((this : Superuser.apk))->(Original:False, Changed:False)
```

Also, during this test, some malware was only suggested to be using by maximum Android API level equal to 19 (i.e. com.KoreaGirlFashion.Secelup.apk). In this case, using Android 6.0 VM might not be suitable (as Android 6.0 (Marshmallow) 's API level is 23). In this situation, the test VM was changed to use Android 4.4 VM (as Android 4.4 (KitKat) 's API level is 19). After that, the same apk was tested again, and in this time the malware finally could run in the correct manner with the appropriate API level.

Some of the malware cannot be tested if their target device is only for ARM architecture. The error message was shown as the following: (Sample: com.abcc.runner.apk)

```
Error: java.lang.UnsatisfiedLinkError: dlopen failed:
"/data/app/com.abcc.runner-1/lib/arm/libandroidgl20.so" has
unexpected e_machine: 40
```

It seems that some malware from the database was created very long time ago. Nowadays most of the new Android smartphone are built based on x86 architecture so this old malware might not be runnable anymore. On the other hand, before this malware crashed, the Frida script captured some suspicious emulator detection API activities like checking the properties of TelephonyManager. It seems that emulator detection is common in the android application and malware.

5 Limitations and Difficulties Encountered

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

5.1 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 the short term as the project is still in the development stage.

5.2 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.

5.3 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 are 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 a 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 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 will temporarily solve the unstable network problem.

5.4 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].

5.5 Potential Anti-Frida features in the sample

According to OWASP, App developers are the ability to implement some Anti-Frida detection methods [19]. Therefore, malware can also do emulator detection likewise action, by detecting the 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.

5.6 Android 4.4 VM Soft-reboot issue

This issue was found when the project is firstly designed to use Android 4.4 VM iso as Cuckoodroid framework also use this version of Android VM iso. Before working on the cuckoodroid implementation, a separate pretest was done to see if frida-server could run in the 4.4 VM. However, when the frida-server was triggered to start, the whole Android OS becomes unstable and then it did soft reboot very soon. This caused an issue when it went to soft reboot: the cuckoodroid agent in the VM was not running, and if the agent was unable to work, the dynamic analysis will never to get started!

Therefore, in the first half of the project time, the focus was going to Android 6.0 iso as it is the only one iso that supports Frida and Xposed Framework at the same time. But when it comes to the real sample test, the fact showed that many malware samples contain maximum SDK version (i.e API level) of 19, which is lower than the Android 6.0 API level. In order to test the malware in a correct SDK version, it is unavoidable to use Android 4.4 iso. Therefore this project has injected the following codes in guest.py to handle the startup issue when using 4.4 iso:

```
if self._fm and check_ping(self.ip):
   if not self._fm.process_on:
      run_cmd_with_timeout("killall adb", 4)
      run_cmd_with_timeout("adb root", 4)
      run_cmd_with_timeout("adb connect {}".format(self.ip), 4)
      self._fm.set_frida_server()
      time.sleep(2)
      run_cmd_with_timeout("adb shell am restart", 4)
      self._fm.process_on = True
      time.sleep(10)
   run_cmd_with_timeout("adb shell am start -n
com.cuckoo.agent/com.cuckoo.agent.MainActivity -a
android.intent.action.MAIN,android.intent.action.BOOT_COMPLETED -c
android.intent.category.LAUNCHER", 4)
   time.sleep(2)
   run_cmd_with_timeout("adb shell input keyevent KEYCODE_BACK", 4)
   time.sleep(1)
```

After that, running the malware sample in Android 4.4 is available, and the analysis report can also be generated successfully. However, a minor problem was some unnecessary HTTP requests were captured accidentally since the soft-reboot restarted the google services. In order to read the report, it is necessary to ignore those HTTP requests generated by the VM itself, and only study the HTTP requests which are likely generated by the testing sample.

5.7 Python library issue

There were some problems in the original web interface in cuckoodroid. Initially, the sample upload function was unable to work as expected. Since the original framework did not implement a debugger to show what was the actual problem, a code review was done in order to investigate which part of the code was not working. Finally, the reason was found - the sqlalchemy library was installed with a newer version, where the code was using depreciated code. After I changed back into the correct version which contains that depreciated method, the web interface can be started after the fix.

6. Conclusion

This project aims to provide a better automated and more accurate dynamic malware analysis solution when the malware samples contain emulator detection methods to bypass the analysis. Using Frida of modifying the Frida JavaScript is much faster than that of modifying the Java code in the previous Xposed Framework solution.
In dynamic analysis, Frida can take an important role, to utilize more anti-emulator detection methods than the previous Xposed Framework solution. Furthermore, when users make changes to the Frida JavaScript, they do not need to restart the VM and reproduce a new VM snapshot. Users only need to save the scripts and then inject them during the dynamic analysis. They do not need to cost more time on the preparation of environment configuration, which results in faster development speed. Therefore, the overall user experience is supposed to be better when anti-emulator detection is one of the main concerns of the malware researcher.

Lastly, this project not only uses Frida to perform anti-emulator detection, but it also provides an alternative development possibility. In recent years, more and more reversing engineering projects have started to use Frida to finish their objectives, because of Frida's attractive characteristics and high potential to build an automated solution for malware analysis. This project implemented the module with a basic support of Frida, so in the future it is possible to import another Frida module to gain more functionalities. For Cuckoodroid, if those new JavaScript exists and are compatible with the objective of Cuckoodroid, that will be very attractive development topics in the malware analysis software product industry.

7. Future works

At the result, this project only works on the anti-emulator detection features using Frida. Generally speaking, Frida can also perform memory dump analysis [26], Dynamic loaded API analysis [27], child-process analysis, etc. The time limit of this project is impossible to include those analysis implementations inside Cuckoodroid. If time is allowed, this project will be continued to the new features.

Furthermore, the current project can be further improved if it can be implemented in the following ways instead of the current Frida Manager implementation shown in Figure 2:

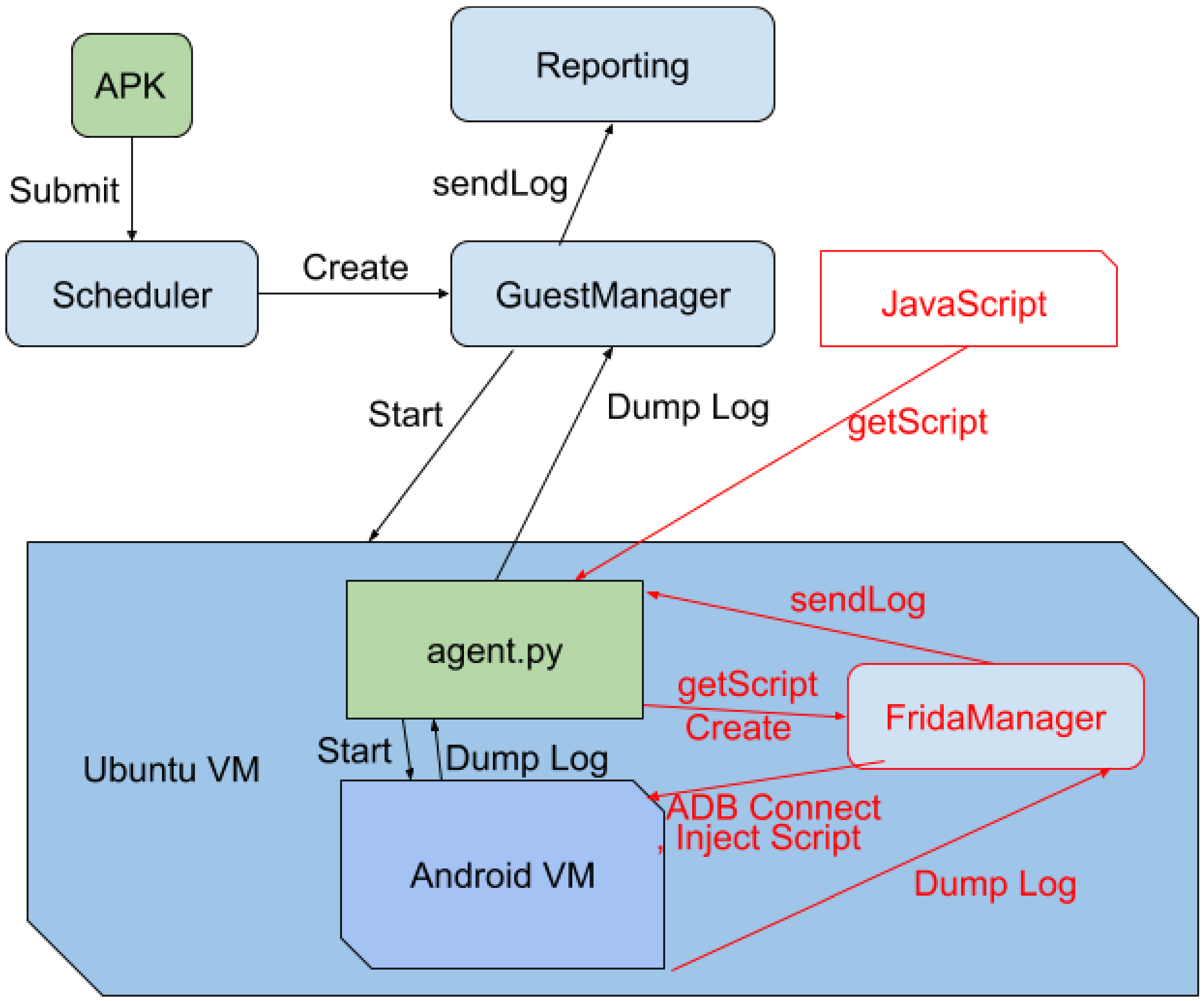


Fig. 6 The original plan of the FridaManager Implementation

However, the additional networking in the Ubuntu VM is always unstable and it is also very hard to do the debugging. Since this is a new topic of implementing new modules in an existing open source project, it should be built in simple structure first. Therefore, a simpler version shown in Figure 2 was chosen to start this project. In production, the design shown in Figure 6 should be more useful as it can totally decrease the network traffic between host guest (i.e. Frida Manager and VM are communicating inside the Ubuntu VM), and also allow to access the Internet.

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-ever y-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-Engi neering-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]

[23] Fuzion24. (2016). Android Hostile Environment Detection [Online]. Available: https://github.com/Fuzion24/AndroidHostileEnvironmentDetection [Accessed: 2019, Jan 21]

[24] CalebFenton. (2017). Android Emulator Detect [Online]. Available: https://github.com/CalebFenton/AndroidEmulatorDetect [Accessed: 2019, Jan 21]

[25] Androzoo. (2019). Available: https://androzoo.uni.lu/ [Accessed: 2019, Mar 23]

[26] Nightbringer21. (2018). Fridump. Available: https://github.com/Nightbringer21/fridump [Accessed: 2019, Apr 8]

[27] ghostmazeW. (2018). [原创]进阶Frida--Android逆向之动态加载dex Hook（三）（下篇) [Online]. Available: https://bbs.pediy.com/thread-229657.htm [Accessed: 2019, Apr 8]

[28] Frida codeshare. (2018). Available: https://codeshare.frida.re/ [Accessed: 2019, Jan 21]

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 from 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

>         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: # 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)
>                             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
                                                 });
                                       }
                             };
                   }).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)
```