

Automatic Android Malware Analysis

Author:

Waqar Rashid
Matr. # 927923

Supervised By:

Prof. Dr. Nils Gruschka (FH Kiel)
Lukas Bernhard (QuoScient GmbH)

Project report for
Master Information Engineering



Department of Computer Science and Electrical Engineering,
Kiel University of Applied Sciences,
Germany
July 27, 2018

Contents

1	Introduction	1
1.1	Android application fundamentals	1
1.1.1	Application components	1
1.1.2	Intents messages	2
1.2	APK file	2
1.2.1	APK file contents	2
1.3	Dex file	3
1.3.1	Dex file format	3
1.3.2	Multiple Dex files in single APK	6
1.4	Malware Analysis	6
1.4.1	Static Analysis	6
1.4.2	Dynamic Analysis	6
2	Static Analysis	7
2.1	Androguard	7
2.1.1	Androguard usage example	7
2.2	Information Extracted from an APK using Androguard	10
2.2.1	Example usage	14
2.3	Chapter Conclusion	16
3	Dynamic Analysis: CuckooDroid	17
3.1	CuckooDroid architecture	17
3.2	CuckooDroid Installation	18
3.3	CuckooDroid required patching for Android 4.1	18
3.3.1	Android 4.1 Persistent Rooting	19
3.3.2	Bug fixes in Analyzer code	19
3.4	CuckooDroid with Android 5.1	19
3.4.1	Android 5.1 AVD persistent rooting procedure	19
3.4.2	Python 2.7 on Android 5.1	21
3.5	CuckooDroid with Android 7.0	22
3.6	Improvements in CuckooDroid and pull request	23
3.7	Chapter Conclusion	23
4	Dynamic Analysis: Evasion and Cloaking	25
4.1	Introduction	25
4.1.1	Emulator Detection or Evasion	25
4.1.2	Anti-Emulator Detection or Cloaking	26
4.2	emulation Detection or evasion Methods	26
4.3	Emulator testing app	29
4.4	Chapter Conclusion	31
5	Instrumentation: Frida with Android	32
5.1	Introduction	32
5.2	Frida with Android	32
5.2.1	Installation	32
5.2.2	Examples	32
5.3	Chapter conclusion	39
6	Conclusion and future work	40

Appendix A Android Activity Manager tool(am)	45
Appendix B Compiling C/C++ code for android	46
Appendix C Malwares that uses Anti-Emulator detection	48

List of Figures

1.1	Files inside an APK	2
1.2	Dex file format [1]	4
2.1	Groups of information extracted using APK	11
2.2	APK info	11
2.3	Class info	12
2.4	Method info	12
2.5	Nodes info	13
2.6	Edges info	13
2.7	Reused methods in sonicspy variants	14
2.8	Reused of methods in sonicspy variants	15
3.1	CuckooDroid architecture with AVD	17
4.1	Root detection techniques [2] [3]	26
4.2	Methods in stub dex [2]	27
4.3	Pie chart of evasion techniques [4]	27
4.4	Executing our test app without hooking "getPhone"	30
4.5	Executing our test app with hooking "getPhone"	31
5.1	Source code of the method getPhone()	33
5.2	Stack/call trace of the non-hooked method getPhone()	33
5.3	Hooked output on app screen	35
5.4	terminal output of app with hooked getPhone()	35
5.5	Terminal output	36
5.6	interactive frida output	37

List of Tables

1.1	Dex File Format	5
2.1	Some classes of androguard and their description	8

Statement of Originality

I hereby declare that

- the work reported here is composed by and originated entirely from me,
- information derived and verbatim from the published and unpublished work of others has been properly acknowledged and cited in the bibliography,
- this report has not been submitted for a higher degree at any other University or Institution.

Waqar Rashid
Matr. # 927923
July 27, 2018

Dedications

Dedicated to my family whose continuous and unconditional support allowed me to take risks and follow my heart.

It is also dedicated to my teachers who guided in the right direction and taught me with dedication.

I also dedicate this work to Muhammad Shafiq, my brother-in-law who always believed in me and convinced me to continue university after my Bachelors degree.

Acknowledgments

I would like to thank Lukas Bernhard, who was my supervisor at QuoScient GmbH¹. Thanks to his continuous support and guidance that enabled me to finish this work. This work would have been impossible without him. I also learned and improved several programming and related concepts from him. He had been a great mentor.

I would also like to thank Prof. Dr. Nils Gruschka, who agreed to be my supervisor at the University of Applied sciences Kiel and provided me his support and thoughtful insights, throughout the course of this internship.

I would also like to thank Fabien Dombard (CEO of QuoScient GmbH) who gave me the freedom and support to work in an independent manner.

I would thank Patrick Ventuzelo² whose suggestions and understanding of the topics in Mobile security helped me a lot in making some critical decisions.

Lastly, thanks to all QuoScient team who supported me and this work would have been impossible without them.

¹<https://www.quoscient.io/>

²<https://github.com/pventuzelo>

Abstract

Today an ever growing number of population is using smartphones for their day to day tasks like social media, financial or other personal tasks. This fact hadn't gone unnoticed from the cyber-criminals and they are coming up with newer ways to bypass current security defenses to infect devices with banking malwares, crypto miners, adwares etc. Android having more than 80% of market share, is the prime target and due to the huge number of apps that needs to be analyzed every day the current android malware analysis tools are not completely up to the tasks of automatic scalable malware analysis.

To address the issues of automatic malware analysis, in static analysis we extracted data from an APK file and showed that how that extracted data can be used to identify malicious samples. For dynamic analysis we used CuckooDroid. We fixed several issues in CuckooDroid and made some enhancements. We also made a pull request to CuckooDroid official repository.

We also identified several areas in which future work can be done for example, frida integration with CuckooDroid, Enhancing anti-emulator detection capabilities of CuckooDroid, separating malicious and android API methods from static analysis data, improving performance of android emulator and latest Android support in CuckooDroid.

Chapter 1

Introduction

Smartphones had become an integral part of our lives. We use it for communication, financial transactions, entertainment and many other activities. Past several years had seen an increase in the number of malwares targeted toward smartphones and it is still increasing. The major target of such malwares are the smartphones running Android. Due to the high number of Android apps created or modified everyday, its impossible to analyze them manually for malicious behavior and thus the need for good automatic malware analysis tools and techniques is growing. Google is already using some automatic tools (aka Bouncer) to analyze every app uploaded to Google Play Store but hackers always comes up with new ways to fool the system and thus enable themselves to upload malwares to Google Playstore. Therefore it is very important to have the tools to analyze android samples for malicious behavior even though they are analyzed before. In this project we worked on developing the tools that will allow analysts to analyze samples dynamically or statically in automatic fashion.

In this chapter we will discuss some fundamental concepts related to android applications and android malware analysis.

1.1 Android application fundamentals

Android application are mostly written in Java. The Android SDK tool compile this code along with any data and resources files into an APK file, an Android Package. One APK file contain all contents of an Android app and is the file that Android devices use to install the application [5]. We will discuss the structure of an APK file in section 1.2. In this section we will discuss some basic parts of an Android application.

1.1.1 Application components

The essential building blocks of an Android application are called components. Each type serves a distinct purpose and has a distinct life-cycle that defines how the component is created and destroyed. The communication between these components (except Content providers) is done using messages called "Intents" (section 1.1.2). It is also important to note that all of these components needs to be listed in AndroidManifest.xml file, for more detailed description of this file please look at section 1.2.1. There are four different types of components and three out of these serve as entry points to an android application. These four components are described below:

- **Activities** An activity is the entry point for interacting with the user. It represents a single screen with a user interface. Each activity is independent from others [5].
- **Services** A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface [5].
- **Broadcast receivers** A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. Although broadcast receivers don't display a user interface, they may create a status bar notification to

alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a gateway to other components and is intended to do a very minimal amount of work [5].

- **Content providers** A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify that data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query the content provider, such as `ContactsContract.Data`, to read and write information about a particular person [5].

1.1.2 Intents messages

Three out of the four component types activities, services, and broadcast receivers can be activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime. You can think of them as the messages that request an action from other components, whether the component belongs to your app or not [5]. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

- Starting an activity
- Starting a service
- Delivering a broadcast

Readers more interested in this topic are recommend to have a look at [6].

1.2 APK file

Android Application Package(APK) is the file format used for an android application. It contains all the resources required for an application to run on android operating system. Its basically a zip file or a jar file with extension of ".apk" [7].

1.2.1 APK file contents

Normally an apk file contains following files or folders:

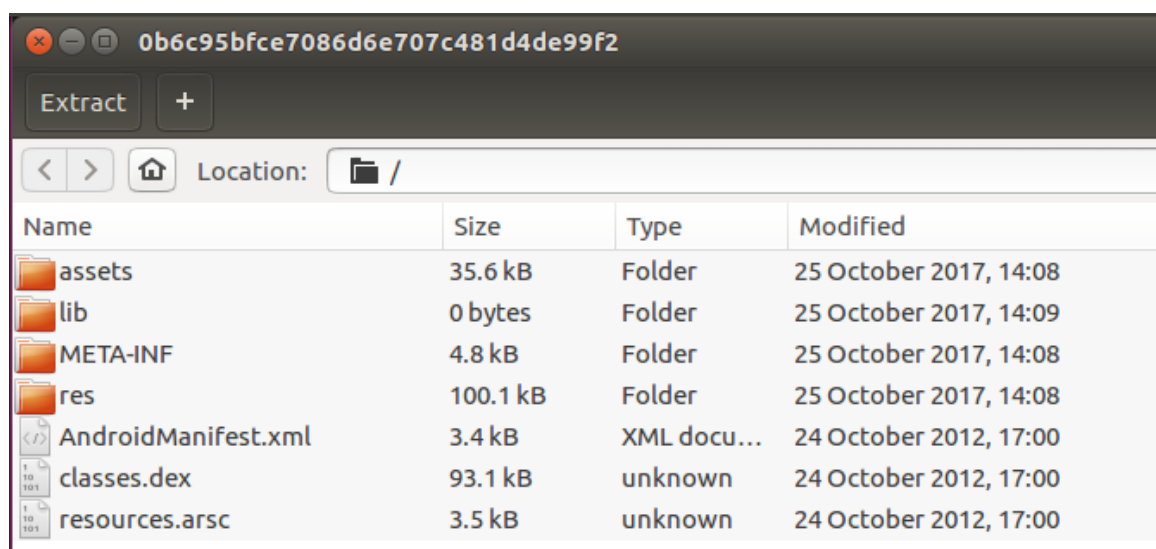


Figure 1.1: Files inside an APK

- **assets/**: It provides a way to include arbitrary files like text, xml, fonts, music and video in your application and allow you to access your data raw/untouched. `AssetManager` is used to read this data [8]. Due to raw access sometimes this directory contains executable payloads

and dynamically loaded code. One interesting usage is storing Dex files in it to avoid its reverse engineering. [2]

- **lib/**: This directory is for natively compiled code. This directory contains a subdirectory for each platform type, like armeabi, armeabi-v7a, arm64-v8a, x86, x86_64, and mips [7]. This code is run directly on CPU and have access to android API using Java Native Interface(JNI). Natively compiled code is more suitable for CPU intensive jobs because of less overhead and good performance of programming language like c/c++. Most of the android static analysis tools work on Java level- that is, they process either the decompiled Java source code or Dalvik Byte Code [9]. This raise several interesting scenarios in which malware authors can avoid detection by hiding malicious parts in native code, allowing them to redistribute benign applications with malicious injections or completely modifying behavior of an application. Readers interested in this topic are encouraged to have a look at [9]. Android NDK can be used to compile native code for android. Have a look at appendix B to see how C/C++ code can be compiled for android using android NDK.
- **META-INF/**: This directory contains the following three files:
 1. **MANIFEST.MF**: Its a text file and contains a list and base64 encoded SHA-1 hashes of all files included in the APK.
 2. **CERT.SF**: This file again contain a list of all files but this time with the base64 encoded SHA-1 hashes of the corresponding lines in the MANIFEST.MF file. It also contain based64 encoded SHA-1 hash of MANIFEST.MF file.
 3. **CERT.RSA**: It contains developers public signature, used for validation of upgrades. Its basically singed content of CERT.SF file along with public key to validate the contents.
- **res/**: This directory contain resources which are not compiled into "resources.arsc" (see below) [7]. These resources can be accessed from inside the application code using resource ID. All resource IDs are defined in "R" class of the project. Application developers can specify alternate resources to support specific device configurations e.g, alternative drawable resources for different screen sizes, alternative strings for different languages etc.
- **AndroidManifest.xml**: Every application must have an AndroidManifest.xml file. This file provide essential information about the application like entry points, package name, components, permissions, minimum level of Android API, libraries, intents etc. For static analysis purposes a lot of information can be extracted from this file.
- **classes.dex**: This is the most important file inside an apk. It contains classes compiled into the DEX file forma [10]. In section 1.3 we will describe its structure in more details.
- **resources.arsc**: This file contain compiled resources. This file contains the XML content from all configurations of the res/values/ folder. The packaging tool extracts this XML content, compiles it to binary form, and archives the content. This content includes language strings and styles, as well as paths to content that is not included directly in the resources.arsc file, such as layout files and images [7]. These resources can also be accessed using the "R" class.

1.3 Dex file

Dex file is the heart of an android application. First Java source code of an application is compiled to Java byte code (".class" extension). Then this Java byte code is compiled to Dalvik Byte Code or Dalvik Executable(DEX) using Dex-compiler or dexter tool. This code is then executed by Dalvik Virtual Machine (DVM, deprecated) or Android Runtime (ART). In case of ART, this code is compiled at install time to the native code and its called Ahead of time Compilation in contrast to Just in time compilation of DVM.

1.3.1 Dex file format

In this section we will discuss the dex files format. For more in depth and up to date specifications, readers are encouraged to have a look at android official documentation on dex format [10]. A more graphical representation of dex file is shown in Figure 1.2. In figure 1.2 links between some elements are shown with arrows. Janus Vulnerability [11] discovered by GuardSquare can be interesting read

here. A few months after the discovery of Janus Vulnerability TrendMicro also reported that they found an app which was exploiting this vulnerability [12]. Although a bit off the topic but around the same time security researcher at CheckPoint discovered ParseDroid [13] which effected tools like apktool and others used by Android security researchers. Table 1.1 describes each and every section contained in a Dex file.

DALVIK EXECUTABLE

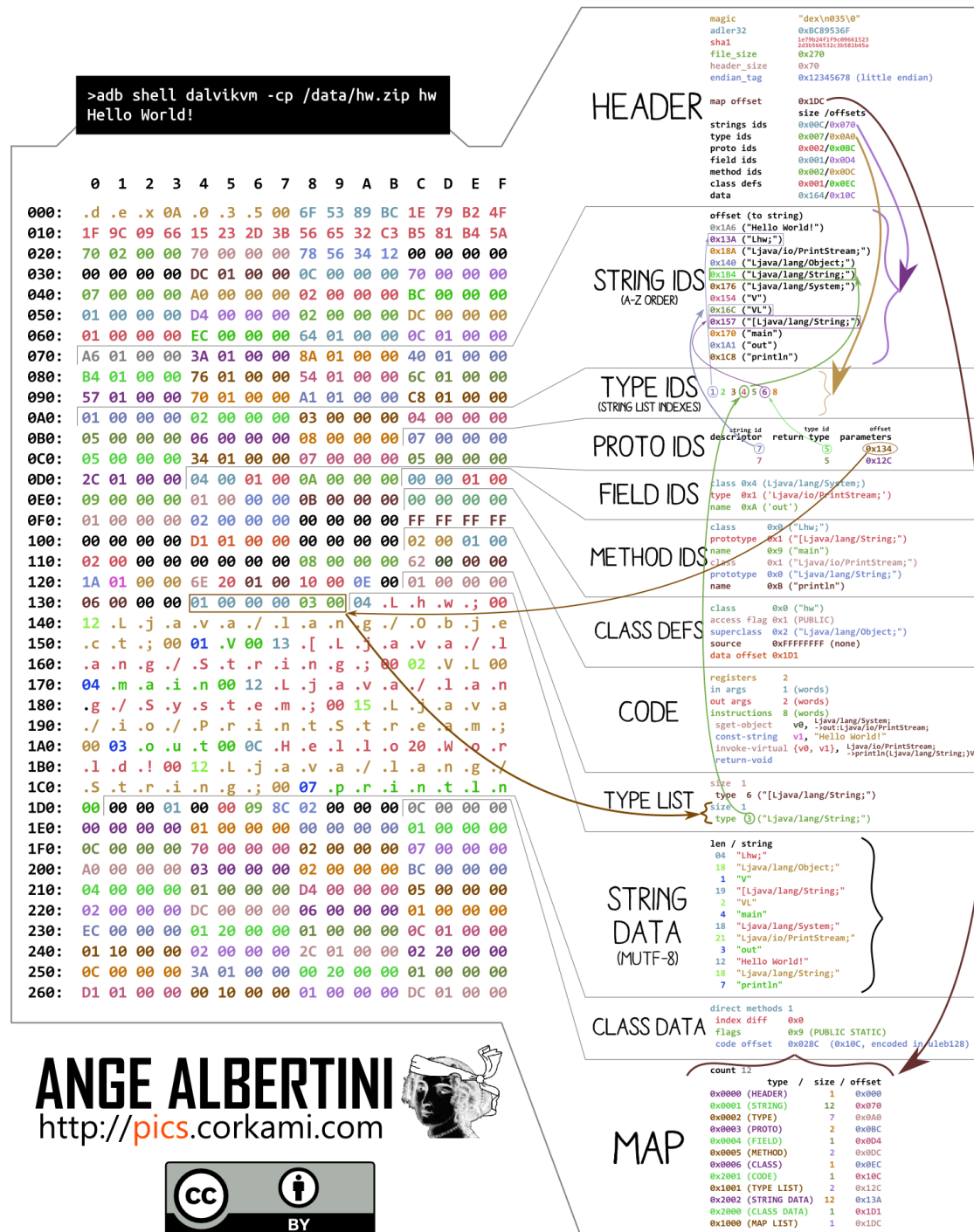


Figure 1.2: Dex file format [1]

Table 1.1: Dex File Format

Name	Format	Description
header	header_item	The header contain information about how the dex file is organized.
string_ids	list of string_id_items	Its a list of string identifiers. Each item points to a location in data section (see below) where the original string is stored.
type_ids	list of type_id_items	This list contain type identifiers for all types (classes, arrays or primitive types) referred to by this file, whether defined in the file or not. The actual identifier string is stored in data section. Items in this list points to items in string_ids list and which in turn points to type identifier string stored in data section.
proto_ids	list of proto_id_items	<p>Its a method prototype identifier list. Each item of this list contain three elements:</p> <ul style="list-style-type: none"> • shorty_idx Points to string_id.item of shorty descriptor for this prototype • return_type_id Specify return type by pointing to corresponding type_id .item • parameter_off Offset from start of file to the list of parameter types for this prototype. It must point to location in data section. The data there should be in "type_list" format. This value would be zero in case no parameters.
field_ids	list of field_id_items	These are identifiers for all fields referred to by this file, whether defined in the file or not.
method_ids	list of method_id_items	These are identifiers for all methods referred to by this file, whether defined in the file or not.
class_defs	list of class_def_items	The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. Furthermore, it is invalid for a definition for the same-named class to appear more than once in the list.
call_site_ids	list of call_site_id_items	These are identifiers for all call sites referred to by this file, whether defined in the file or not.
method_handles	list of method_handle_items	A list of all method handles referred to by this file, whether defined in the file or not. This list is not sorted and may contain duplicates which will logically correspond to different method handle instances.
data	unsigned bytes	Containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment.
link_data	unsigned bytes	The format of the data in this section is left unspecified by this document. This section is empty in unlinked files, and runtime implementations may use it the way they see fit.

1.3.2 Multiple Dex files in single APK

APK files contain executable bytecode files in the form of Dalvik Executable (DEX) files, which contain the compiled code used to run your app. The Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to 65,536 including Android framework methods, library methods, and methods in your own code. This limit is referred to as the '64K reference limit' [14].

Android apps contains relatively large number of methods and thus some applications use multiple dex files inside a single APK. Android 5.0 or older versions (API level 21) use the Dalvik runtime, which limits apps to a single dex file per APK. Multidex support library can be used to workaround this limitation. Android 5.0 (API level 21) and higher uses a runtime called ART which natively supports loading multiple DEX files from APK files. Because of this support its not uncommon these days to come across APKs that contain multiple dex files e.g, Facebook, instagram etc. [14]

1.4 Malware Analysis

Malware analysis is done to understand the behavior of a malware sample and to come up with steps to avoid or remove the malware from infected systems, identify the threat actors, determine indications of compromise (IOCs) or to find other interesting information. There are main two categories of malware analysis:

1.4.1 Static analysis

In this category, the file is analyzed by looking at its source code. This source code can be obtained by decompiling the sample and then looking for interesting patterns in this code like some well know strings, signatures, commonly used malicious code sections etc. This type of analysis doesn't give us any information about which code is executed and which code is not (dead code). It is also not good in analyzing samples that download the payload and the sample is only a dropper. There are different ways to compensate for the drawbacks discussed above like data flow analysis, taint analysis etc but that are beyond the scope of this report. For Android one of most popular static analysis tool is Androguard [15]. Other tools include APKtool, IDA, smali/backsmali.

1.4.2 Dynamic Analysis

In this category of analysis, the sample is executed in a controlled environment to see what it is doing when executed. Which kind of network traffic it generates, which files does it interacts with, which system functions does it use. There are some challenges that this method of analysis faces like Emulator detection and delayed execution. For android the most popular dynamic analysis tool is CuckooDroid [16].

Both of the above categories have their own limitations but are widely used in hybrid mode. A malware analysis system use both of these on the sample to extract information. CuckooDroid uses Androguard along with instrumentation framework "Xposed Framework" to perform analysis. In coming chapters we will talk in more details about these two categories and how we used it in our analysis tools [16].

Chapter conclusion

This is a basic introductory chapter. In this chapter we tried to answer "what is an APK file?", "What is a Dex File?" and "what is Malware Analysis?". We also explained the components an Android Application and intent messages, which would be used later in this report.

Chapter 2

Static Analysis

Static analysis is the analysis of a sample without actually executing it. Various techniques and tools are used like decompiling, string searches, data flow analysis etc. The purpose is to understand the behavior of a malware and sometimes extract some useful information that can be used in cuckoodroid for dynamic analysis. In this project there are two kinds of static analysis that are being performed.

First type of analysis is being performed during or before dynamic analysis to improve code coverage or just to be added to the final report. It include extracting activites, intent messages, services which are later used as parameters to the android Activity Manager Utility to increase code coverage (section 3.6). Other information extracted in this category are extracting permissions, package name, signatures, looking for specific function calls(encryption, decryption etc). Have a look at appendix A for more on Activity Manager and how to use it.

Second type of static analysis is performed to extract as much useful information as possible. The goal of this is to have a lot of information about a sample that will be stored in a database and can be compared to detect similarities between malwares or to identify new malware as most of them share code. It can also be used to find newly added features to a malware family as they adapt with time. We will talk about this type of static analysis in more details in this chapter. Androguard is used for these analysis because it can be easily automated and have very good community support. For a literature review on android static analysis please have a look at [17].

2.1 Androguard

Androguard is an open source tool written in python for analyzing android applications. Its been used in several tools including Virustotal and Cuckoodroid among others. It can process APK files, dex files or odex files. It can disassemble Dex/Odex files to smali code and can decompile Dex/Odex to Java code. Being python based and open source it allows for automating most the analysis process. The analysis is done on need-to basis which means we only use our CPU cycles for the tasks that we need. It has the also the option to be used in interactive mode by using androlyze.py script.

2.1.1 Androguard usage example

Installing androguard is very simple. You can download it from its official github repository [15] and then can install it by executing the following in androguard directory:

```
$ python setup.py install
```

On Ubuntu it can be installed with simply executing:

```
$ sudo apt-get install androguard
```

It can be also be installed using python pip by executing:

```
$ pip install androguard
```


We recommend downloading it from github repository under releases link, because there you can download the specific version that you need. Currently, pip and Ubuntu repositories doesn't have the same version of androguard.

As we mentioned above, androguard can be used in two way, its a good idea to start with interactive CLI mode of androguard just to get some idea of what it is capable of. You can use the below command in your terminal to start the interactive shell. It will import all the required androguard packages and you can start analyzing APKs right away. Depending on how you installed androguard, you may need to execute androlyze.py or just androlyze.

```
$ androlyze -s
```

This will open a python interactive shell and you can now specify the APK file as show below:

```
In [1]: apk, dvm, dx = AnalyzeAPK("path/to/app.apk")
```

The analyzeAPK method returned three objects. These classes are also described in table 2.1. Keep reading after the table 2.1 to know more about these returned objects.

Classes for Parsing	Classes for Analysis
<ul style="list-style-type: none"> • APK Used for accessing all elements inside an APK, including information from Manifest.xml like permissions, activities etc. • DalvikVMFormat It parses the dex file and gives access classes, methods, strings etc. defined inside the dex file. • ClassDefItem Class for interacting with class information inside the dex file. • EncodedMethod Class for interacting with method information inside the dex file. • Instuction Class for interacting with instructions, it contains mnem, opcodes etc. Its a base class and a androguard derive a class for each instruction format from this class. 	<ul style="list-style-type: none"> • Analysis Its the main analysis class and contain instances of all other analysis classes discussed below. create_xref() method needs to be called after an instance of this class is created to populate all defined fields in this class. • ClassAnalysis This class contain analysis data of a class like cross references and external methods etc. • MethodAnalysis Contain analysis information of a method like the basic blocks it is composed of etc. • DvmBasicBlock Represents a simple basic block of a method. It contains information about that basic block like its parents, children etc.

Table 2.1: Some classes of androguard and their description

apk The first returned class is an object of APK class. This one can be used to get information about the APK most of which are extracted from manifest.xml file. Below is the example on how to get some information:

```
1 user@workstation:~$ androlyze.py -s
2 Androguard version 3.1.0
3 In [1]: pwd
4 Out [1]: '/home/user'
5
```

```

6 In [2]: apk, dvm, dx = AnalyzeAPK("/home/user/workspace/apps/evasion.apk")
7
8 In [3]: apk.get_activities()
9 Out[3]: [b'com.ouz.evasion.EvasionMainActivity']
10
11 In [4]: apk.get_permissions()
12 Out[4]: []
13
14 In [5]: apk.get_libraries()
15 Out[5]: []
16
17 In [6]: apk.get_package()
18 Out[6]: 'com.ouz.evasion'
19
20 In [7]: apk.get_max_sdk_version()
21
22 In [8]: apk.get_min_sdk_version()
23 Out[8]: '14'
24
25 In [9]: apk.get_files()
26 Out[9]:
27 ['res/layout/activity_evasion_main.xml',
28 'res/menu/evasion_main.xml',
29 'AndroidManifest.xml',
30 'resources.arsc',
31 'res/drawable-hdpi/ic_launcher.png',
32 'res/drawable-mdpi/ic_launcher.png',
33 'res/drawable-xhdpi/ic_launcher.png',
34 'res/drawable-xxhdpi/ic_launcher.png',
35 'classes.dex',
36 'META-INF/MANIFEST.MF',
37 'META-INF/CERT.SF',
38 'META-INF/CERT.RSA']

```

There are a lot of methods that can be called on the APK object, to see them just type "apk." in the CLI and then press tab. The auto-complete feature will show you all the possible methods you can call. For more details you can have a look at the source code or the documentation for this class.

dvm Second returned object, the dvm class represent DalvikVMFormat class, which basically parse the dex file and gives us information about the classes, methods, Basic Blocks and instructions.

```

39 In [10]: type(dvm)
40 Out[10]: androguard.core.bytecodes.dvm.DalvikVMFormat

```

In below script we will demonstrate how to get the name of some of the methods using this dvm object:

```

41 In [11]: methods = dvm.get_methods()[10:15]
42
43 In [12]: for method in methods:
44 ...:     print(method.get_name())
45 ...:
46 getSettingsActivityName
47 <clinit>
48 <init>
49 capabilityToString
50 feedbackTypeToString

```

The auto-complete feature, documentation or source code can be used in similar way as mentioned above to discover more features more methods or fields of these objects.

dx The third returned object is an instance of Analysis class which contain cross reference information of classes and methods, string analysis. This object also allows us to access more analysis classes like ClassAnalysis and MethodAnalysis. Decompilation is performed here and on slower systems it can take some time.

```

51 In [13]: for key in dx.get_strings_analysis():
52 ...:     print(key)
53 ...:
54 Unable to create files subdir
55
56 mCommitted=
57 mDeliveredData=
58 Resolving type
59 No fake drag in progress. Call beginFakeDrag first.
60 android.app.Notification$Action
61 mAvailIndices:
62 ; boundsInScreen:
63 RemoteInput
64 Can't change tag of fragment
65 mAvailBackStackIndices:
66 Couldn't fetch mRecreateDisplayList field; dimming will be slow.
67 Operations:
68 is not a sliding drawer
69 NULL
70 mRetaining=
71 command cannot be null or empty
72 mAccessibilityDelegate
73 mProcessingChange=
74 op #
75 Callback may not be null
76 onCreateView: id=0x
77 .
78 .
79 .
80 .
81 =>
82 :
83 Unable to instantiate fragment
84 onLoadComplete:
85 android:fragment:
86 icon
87 .
88 , tag
89 Can not perform this action after onSaveInstanceState
90 mediaSession is not a valid MediaSession object
91 mMenuVisible=
92 tag must not be null or empty

```

Below script demonstrates how to further examine the results provided by Analysis class object dx.

```

93 In [14]: string_analysis = list(dx.get_strings_analysis().values())[10]
94
95 In [15]: string_analysis.get_orig_value()
96 Out[15]: "Can't change tag of fragment "
97
98 In [16]: string_analysis.get_xref_from()
99 Out[16]:
100 {(<androguard.core.analysis.analysis.ClassAnalysis at 0x7fd62588d320>,
101 <androguard.core.bytecodes.dvm.EncodedMethod at 0x7fd626197b00>)}
102
103 In [17]: string_analysis.get_value()
104 Out[17]: "Can't change tag of fragment "

```

We encourage reader to play with the androguard interactive CLI. After you have some idea of what can you do, then you proceed to write some python scripts and automate some of the tasks. In section 2.2 we will talk about the information that we extracted from APK files using androguard.

2.2 Information Extracted from an APK using Androguard

Before getting into details, we would like to mention that the information we extract will be stored in a database and similarity search would be performed on this data to figure out its relations with other malware or goodware samples. The more information we have about an APK, more relations we can find.

Now coming towards, the extracted information, we divide it into different groups as shown in figure 2.1.

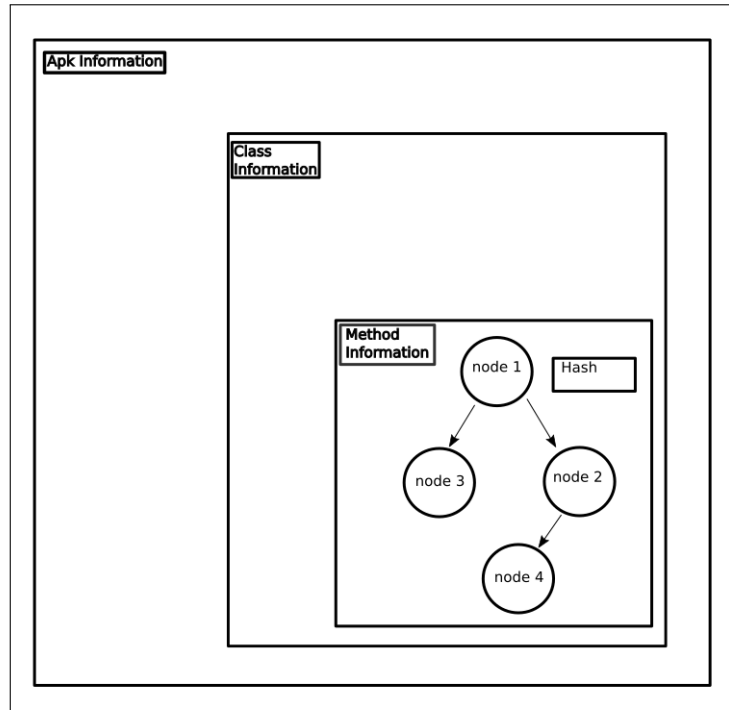


Figure 2.1: Groups of information extracted using APK

- APK information** This contain general information about the APK like Permissions, Package name, Libraries, Certificates, components (activities, services, receivers) and information about all classes contained in this APK. We also wrote some code to extract strings from layout files and other binary xml files contained in the APK because most of the bitcoin wallet addresses contained in malware samples were hard coded in the those files. Figure 2.2 shows the APK information stored in JSON format.

JSON	Raw Data	Headers
Save Copy		
target_sdk_version:	null	
providers:	[]	
app_icon:	"res/drawable-mdpi/icon.png"	
min_sdk_version:	"7"	
permissions:	[...]	
is_valid_APK:	true	
androidversion_name:	"1"	
total_classes:	"210"	
app_name:	"定制机终极克星"	
libraries:	[]	
certificates:	[...]	
activities:	[...]	
internal_methods:	"370"	
androidversion_code:	"1"	
number_of_external_classes:	"122"	
main_activity:	"com.triangle.rootinsecon...zhjikexing.MainActivity"	
max_sdk_version:	null	
services:	[...]	
classes_info:	{...}	
package:	"com.triangle.rootinsecondszhjikexing"	
api_version:	19	
files:	{...}	
requested_third_party_permissions:	[]	
receivers:	[]	
internal_classes:	"88"	
filename:	"/home/wra/workspace/apks...e7086d6e707c481d4de99f2"	
activities_filters:	{...}	

Figure 2.2: APK info

- **Class information** It represents a single element of the class_dictionary contained in APK information. It contains information like fields in the class, its access flags, name, superclass name, number of internal methods, inherited methods etc. It also contain information about methods which are part of this class in the form of a list. This information can be seen in figure 2.3.

▼ classes_info:	
▼ Lcom/zipmaster/w;:	
▶ methods_info:	[...]
▶ fields:	[...]
access_flags_string:	"final"
number_of_internal_methods:	"1"
interfaces:	[]
inherits_methods:	[]
length:	32
access_flags:	16
name:	"Lcom/zipmaster/w;"
super_class_name:	"Ljava/lang/Object;"
▶ Lcom/zipmaster/b;:	{...}
▶ Lcom/zipmaster/e;:	{...}
▶ Lcom/zipmaster/ab;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/n;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/R;:	{...}
▶ Lcom/zipmaster/av;:	{...}
▶ La/a/d;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/f;:	{...}
▶ Lcom/zipmaster/f;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/b;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/q;:	{...}
▶ Lcom/zipmaster/ZipMasterOfferActivity;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/R\$id;:	{...}
▶ Lcom/triangle/rootinsecondszhjikexing/o;:	{...}

Figure 2.3: Class info

- **Method information** As we said above, class_information contain a list of methods_info. Each element of that list contain information about a method such as method name, class name, address, method descriptor, length of method, its cross references, shortly descriptor, java decompiled source code for that method, control flow graph of that method and the calculated hash(canonical certificate) for this method. Method information can be seen in 2.4.

▼ La/a/d;:	
▼ methods_info:	
▼ 0:	
▶ cfg:	{...}
Encoded_length:	4
▶ registers:	[...]
size:	7
java-source-code:	"\n private d()\n {... return;\n }\n"
hash:	"e446997e7553418abd0c917b_d436c72a3f56c4ac20c83a7"
class_name_type:	"a.a.d"
▶ xref_from:	[...]
address:	24384
method_name:	"<init>"
class_name:	"La/a/d;"
descriptor:	"()V"
return:	"void"
is_internal:	"True"
▶ xref_to:	[...]
access_flags_string:	"private constructor"
▶ 1:	{...}
▶ 2:	{...}
▶ 3:	{...}
▶ 4:	{...}

Figure 2.4: Method info

- **Control Flow Graph** Control flow graph contain edges and nodes. Nodes are basic blocks and are basically a list of instructions. Figures 2.5 and 2.6 shows this part of information

extracted from an apk and stored in JSON format.

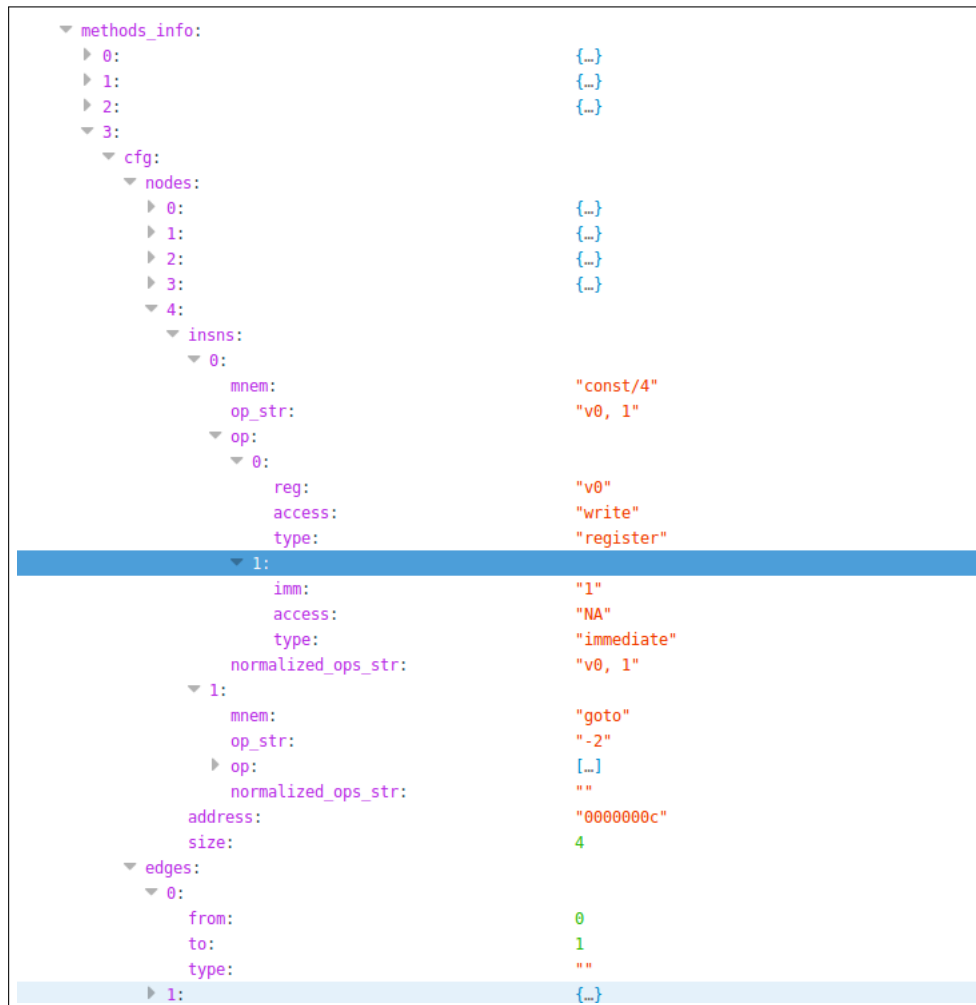


Figure 2.5: Nodes info

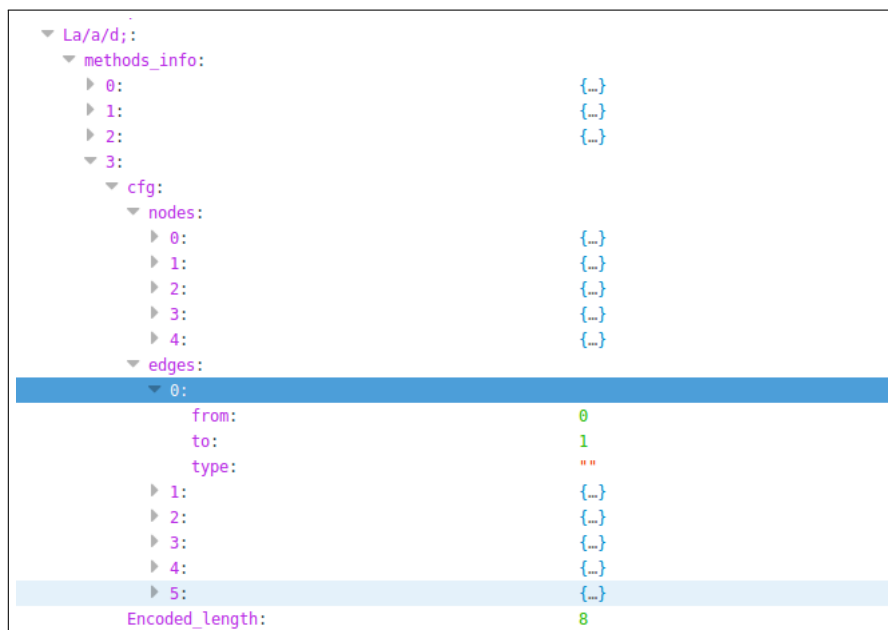


Figure 2.6: Edges info

- **Canonical Certificate** In order to compare two methods, we basically have to compare the corresponding control flow graphs. The control flow graph can change for the same method so we need to find the Canonical form of this Control flow graph. For more details about the topic readers are encouraged to search the Internet for "Graph canonization" and also have a look at the webpage of "bliss" [18]. We used pybliss python library [18] [19] to get the canonical certificate of this graph. Before computing this certificate, the operands of small instructions are normalized according to a specific criteria so that compilation specific changes like offsets etc are ignored. The canonical certificate can be seen in figure 2.4 with key of "hash". An example of normalized instruction can be seen in figure 2.5 with key of "normalized_ops_str".

2.2.1 Example usage

Just to make the usage of this extracted data more clear. In this example we process some of the sonicspy samples and will try to figure out how much code they share. We analyzed 16 samples and the result is shown in figure 2.7.

```

In [23]: jp.freq_methods_nb
Out[23]:
{1: 2,
 2: 1882,
 3: 533,
 4: 225,
 5: 3350,
 7: 42,
 9: 14,
11: 831,
13: 34,
14: 124,
16: 910}

In [24]: hash_ = jp.freq_methods[16][3]
In [25]: code = jp.get_method_code(hash_)
In [26]: print(code)

    public boolean dispatchKeyEvent(android.view.KeyEvent p2)
    {
        if ((!super.dispatchKeyEvent(p2)) && (!this.executeKeyEvent(p2))) {
            int v0_2 = 0;
        } else {
            v0_2 = 1;
        }
        return v0_2;
    }

In [27]: █

```

Figure 2.7: Reused methods in sonicspy variants

Output line 23 in figure prints the result, in this dictionary keys represent frequency or number of samples in which a method is used. Value represent numbers i.e, number of all such methods that has been used in "key" samples, or number of all methods with corresponding frequency. From this analysis we can see that a large portion of code is shared between these samples but later we found that most of this code is not malicious. Most of it is standard android API methods and non-malicious general purpose methods like wrappers etc. It would be very interesting to identify and separate API code from this as it is just noise. It can be topic for further research to identify common non-malicious pieces of code to make analysis easier.

In the figure 2.7, "jp" is an object of a class JasonParser which we wrote just to verify information extracted from APKs. In line 24 we get the hash of a specific method and in line 25 and 26, we print its Java source code. Here we can see that its a very general method and its been used in all of the 16 samples analyzed.

Figure 2.8 shows the code reuse graphically. Here we still have the same problem that we can't separate standard API code and malicious parts of code. Figure 2.8 was result of processing following 15 samples:

- 9a6b7418729d0b6dc22e8e4925006c6567b303a9
- 5a341606fc391d669f1c01c5dd7e93fc49dca0d7
- ed84300fb1ba7feef63351e49ddd2930f865bea9
- 515453379a26d19c49e60edb8affd801b6001234
- ea9079c800cf8d77cccdc393de1d7366b00ba7ec
- 96548a4054aa1c798a8318302ab416cdbbbedd5bb
- 82bc64931d5e86d02792685141da0cfc5fcf147a
- 275b55b2addb08912d973a6e181fd70e6bbe5d90
- a8b3013c4998196672967028f5612f73c0be23ef
- 7bd0f90e75941939671e430f960674c566f7167c
- 9f69841f38501e4e527d4563d97838c21006083c
- 3f33367040dc423ff97aab7196aa6748ff11cc45
- b22f7611916ff2c57514bc40e59924268fb64452
- 9d2aa0bcecaf564eb06238086c53ce56e8ca1bf4
- 29967d4afd2e11beb85e2f7eec5ce4c1778026a2

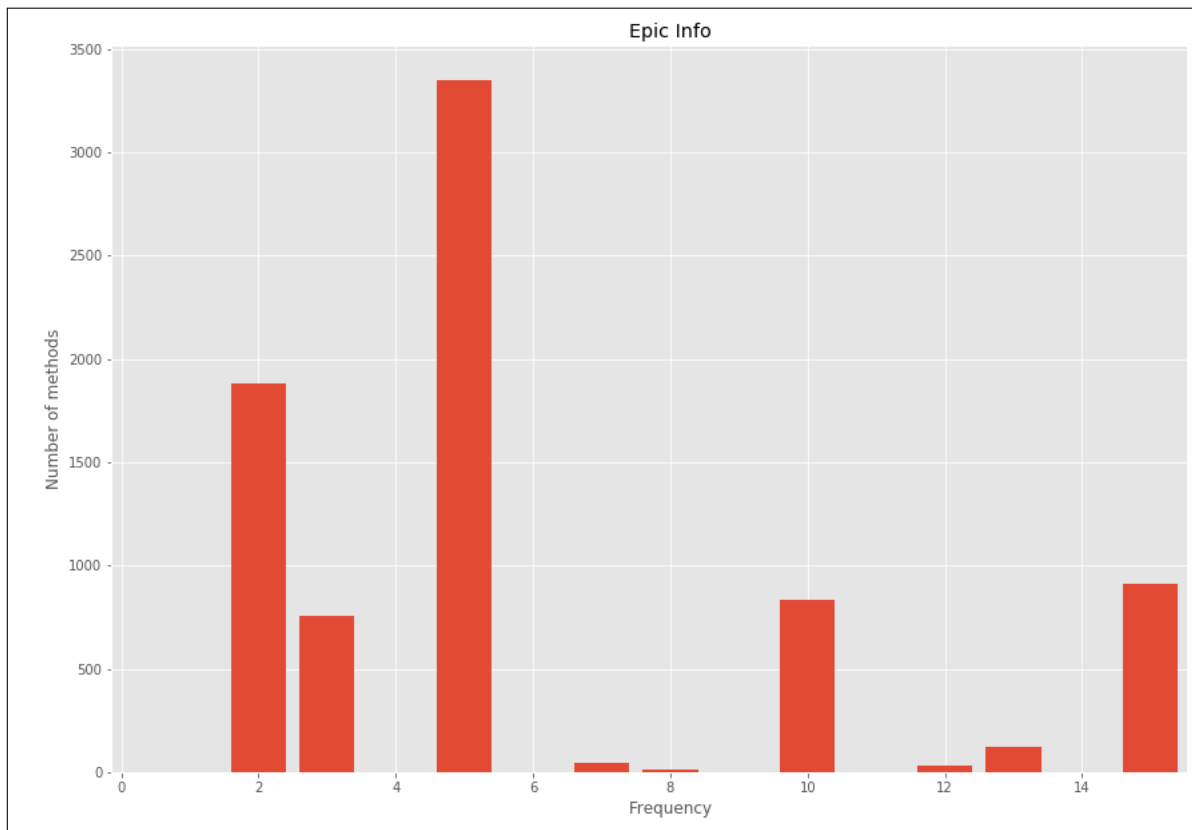


Figure 2.8: Reused of methods in sonicspy variants

2.3 Chapter Conclusion

In this chapter, we introduced androgaard and showed how to use it with a few examples. After that, we discussed the information extraction that can be used to find similarities between different samples or for other purposes. During the course of this work we also identified some bugs in androguard and communicated them to the maintainers of the official androguard github repository [15]. Example of these are issues number 324, 325, 529, 329. At the end, I would appreciate the maintainers of androgaard repository for their help especially reox [20].

Chapter 3

Dynamic Analysis: CuckooDroid

CuckooDroid is an extension of Cuckoo Sandbox the Open Source software for automatic dynamic malware analysis. It enables cuckoo to execute and analyze android applications [21]. For more information about the cuckoo sandbox, readers are encouraged to visit the cuckoo sandbox website [22]. Currently, CuckooDroid only supports Android 4.1 and is based on cuckoo 1.2.

CuckooDroid can be downloaded from the CuckooDroid github repository [16] by following the guidelines provided there. For more step-by-step installation guide, readers are encouraged to have a look at CuckooDroid documentation [21]. Because of changes in android emulator (goldfish) and android SDK the CuckooDroid documentation are not precisely accurate and some deviations are required from it in order to make the CuckooDroid work ¹. By following the CuckooDroid documentation with a few changes discussed later in this chapter, it should be fairly easy for reader to configure his CuckooDroid setup.

3.1 CuckooDroid architecture

In this section we will describe the architecture of CuckooDroid. There are main two parts a "Host" and a "Guest". We will be configuring CuckooDroid with Android Emulator (Goldfish) and figure 3.1 shows the architecture of CuckooDroid with Android Emulator.

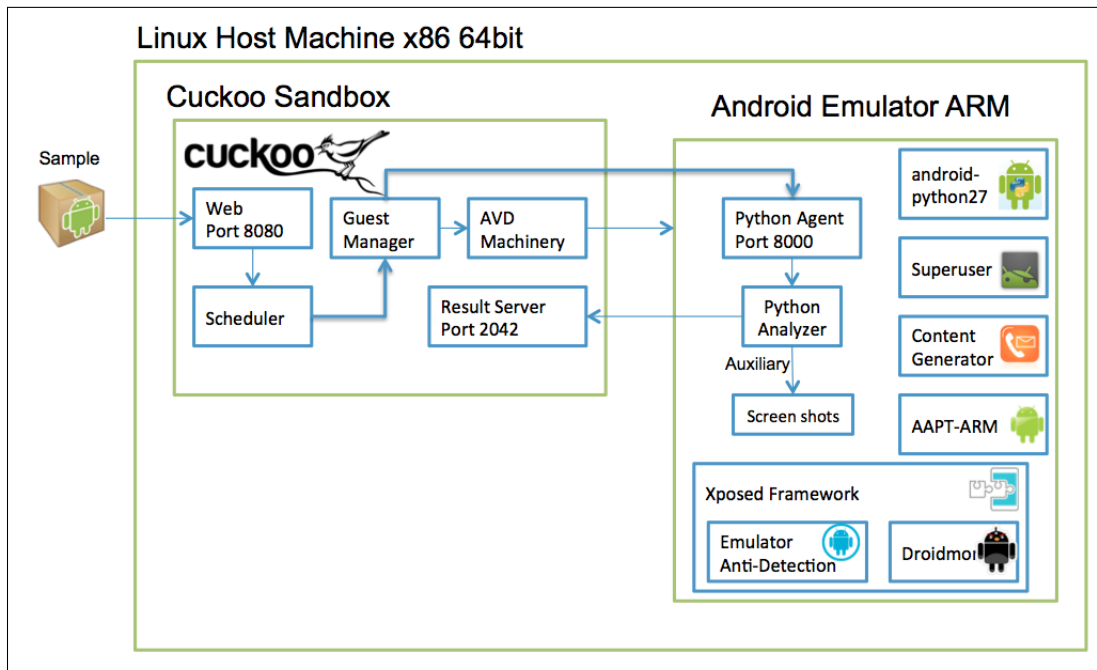


Figure 3.1: CuckooDroid architecture with AVD

¹For some readers who just want a working a CuckooDroid as quickly as possible, have look at the QuoScient CuckooDroid repository [23]. As of today July 27, 2018, the pull request from this repository is still waiting to be approved by CuckooDroid maintainers.

Cuckoo Sandbox is responsible for managing the android emulator and generating report at the end of analysis. Android Emulator executes the application, gather some information from it and reports it back to Cuckoo Sandbox. Below is the description of some of the main parts shown in figure 3.1

- **Python Agent** Executed on AVD and is responsible for receiving APK file, Analysis code, configuration and executing analysis. It also provides constant status updates to Host.
- **Python Analyzer** Android analyzer component that is sent to the guest machine at the beginning of the analysis. This is the main part that executes application, send dropped files back to host, send screenshots back to host, interact with the application if required. It is also responsible for ending the analysis and sending back some log files back to host. It has modular structure and new modules can be added very easily.
- **Xposed Framework** A framework for modules that can change the behavior of the system and apps without affecting any APKs. The version in use only work up to Android 4.1.2 (API 16). In CuckooDroid two modules are used with this Framework:
 1. **Droidmon:** Dalvik API Call monitoring module, it hooks up API calls and prints it into logcat, anaylsis code takes it form logcat and store it in a log file which is sent back to host at the end of anaylsis.
 2. **Emulator Anti-Detection:** Implements some Emulator Anti-detection techniques. Its not specified anywhere as to exactly which techniques does it use. For more details on this topic see Chapter 4.

In the rest of this chapter we will walk through configuring CuckooDroid and some of the changes which are necessary to make it work with latest Android Emulator and latest Android versions.

3.2 CuckooDroid Installation

Installation of CuckooDroid is pretty straight forward and can be easily done by following the Easy integration script available on the CuckooDroid github repository [16]. For the configuration and requirements follow the instructions in the CuckooDroid documentation [21]. In the next two paragraphs we will give a bit more intuitive description of how CuckooDroid works which will help in understanding some of steps in installation and configuration step. Please note that there are some changes which are required to make CuckooDroid work and those are discussed in section 3.3.

The basic components of CuckooDroid can be seen in figure 3.1. There are two basic parts of CuckooDroid "Host" and "Guest". We run the cuckoo sandbox on host which handles the emulator. In order to do analysis our guest needs to be a rooted AVD with Xposed Framework [24] and with two of its modules Droidmon and Emulator Anti-Detection. We also need Python 2.7 to run the python agent and analyzer code on our guest.

Once we run the Cuckoo sandbox and submit a sample to it, it makes a new copy of our specified AVD and launches it. Once the machine boots, the cuckoo sandbox runs the python agent using Android Debugger (adb) and sends the APK file, configuration file, and analysis code to it and orders it to start executing the analysis code. Agent then executes analyzer, which installs the sample, executes it. It also starts the Auxiliary modules and waits for the signal to wrap up the analysis. For the period of execution of the app, the DroidMon module of xposed Framework monitors the specified function calls using hooks and prints it on the logcat. Screenshot modules takes screenshots of the android screen and sends it to host. Once the analysis is completed, the analyzer sends all the required files back to host and terminates. Then host kills the emulator and starts making a report out of these files.

3.3 CuckooDroid required patching for Android 4.1

Initially, we were expecting that CuckooDroid will work out of the box but that wasn't the case. In this section we will talk about some of the changes which were required to make CuckooDroid work with Android 4.1. There are two main steps, one is the persistent root problem and the second one is some bug fixes in python analyzer code.

3.3.1 Android 4.1 Persistent Rooting

Now coming towards the first fix which was that the root wasn't staying persistent across reboots. It was the result of changes in Android emulator. More details and solutions about this problem can be found in issue 4 of CuckooDroid github repository [25]. The solution can be found in the same issue, according to which we needed to copy the system.img file from the android SDK directory to the AVD directory. The following command can be used to do that, make sure you replace the paths according to the location of your android SDK and AVD.

```
$ cp ~/Android/Sdk/system-images/android-16/default/armeabi-v7a/system.img ~/.android  
↪ /avd/aosx.avd/system-qemu.img
```

After the above step, we need to start the emulator with command given below:

```
$ emulator -avd aosx -qemu -nand -system,size=0x1f400000,file=~/.android/avd/aosx.avd  
↪ /system-qemu.img&
```

Then follow the standard steps of rooting described in cuckoodroid documentations [21]. I also made some video tutorials to help the readers with the process. You can find the related video tutorial here [26].

3.3.2 Bug fixes in Analyzer code

The second fix is in the python code that is executed on Android. The CuckooDroid analyzer and agent code heavily relies on Subprocess Management library [27] (specifically subprocess.Popen() method). During debugging we noticed that Most of the calls to subprocess.Popen() method never returns resulting in analysis critical time out. Upon further research we found an issue relating to a similar problem in python 3.7 [28], which seems to be the case here. The subprocess.Popen function uses /bin/sh in Unix environments. Android is detected as a Unix environment, but has moved that executable to "/system/bin/sh". This can be worked around by adding a parameters "shell=True" and executable="/system/bin/sh" to all the subprocess.Popen calls. One related problem was that in some parts of the analyzer, like in adb.py os.popen was used to execute shell commands. This function was also failing to return so we have to replace it with subprocess.Popen. The related video tutorial can be found here [29].

The next limitation that we faced was that CuckooDroid only supports Android upto 4.1, which is quite older and in order to analyze applications that are targeting higher version of androids we need to find a way to run CuckooDroid on higher version of Android. In the next sections we will describe the steps and results of our attempts to upgrade to Android 5.1 and Android 7.0.

3.4 CuckooDroid with Android 5.1

Just like with Android 4.1, here we again faced the persistent root problem. Another problem was that we couldn't use the python binaries for android that comes with CuckooDroid because from this version Android make it a requirement for binaries to support PIE mode. In the following subsections we will describe in more detail about how we solved these problems. Here we used the x86 image just to save time during testing. This rooting procedure works on arm image too but that speed of arm image is very slow and was wasting a lot of time during the development and debugging.

3.4.1 Android 5.1 AVD persistent rooting procedure

The rooting procedure for Android 5.1 is a bit different than that of Android 4.1 in the sense that we needed updated version of tools like SuperSU, XposedFramework etc. Below are the required steps to root Android 5.1 AVD and keep the root persistent.

1. First copy the system.img to your avd directory and rename it as system-qemu.img: For example for Nexus One API 22 x86:

```
cp ~/Android/Sdk/system-images/android-22/default/x86/system.img ~/.android/avd/  
↪ Nexus_One_API_22_x86.avd/system-qemu.img
```

2. Then start the AVD Machine with

```
emulator @Nexus_One_API_22_x86 -verbose -writable-system
```

3. Run the script below which was made with the help of stackexchange answer by a user with the name xavier.fakerat [30].

```
1  #!/usr/bin/env bash
2  # Inspired by: https://android.stackexchange.com/questions/171442/root-android-
   ↳ virtual-device-with-android-7-1-1/176447
3  #this script is meant for easy creation on an analysis machine for android
   ↳ emulator avd
4
5  #Path to the local installation of the adb - android debug bridge utility.
6  #ADB=/home/wra/Andriod/Sdk/platform-tools/adb
7  ADB=$(which adb)
8  if [ ! -f $ADB ]
9  then
10 echo -e "\n-Error: adb path is not valid."
11 exit
12 fi
13 echo -e "\n-adb has been found."
14 # First we root the emulator using Superuser
15 echo -e "\n-Pushing /system/xbin/su binary"
16 $ADB root
17 $ADB remount
18 # one of these is correct, we do both just to be on safe side
19 $ADB push nougat/SuperSU-v2.82-201705271822/x86/su.pie /system/xbin/su
20 $ADB shell chmod 06755 /system/xbin/su
21 $ADB push nougat/SuperSU-v2.82-201705271822/x86/su.pie /system/bin/su
22 $ADB shell chmod 06755 /system/bin/su
23 echo -e "\n-Installing application Superuser"
24 $ADB install nougat/eu.chainfire.supersu_2.82.apk
25 # Install Xposed Application
26 echo -e "\n-Installing Xposed Application"
27 $ADB install nougat/XposedInstaller_3.1.4.apk
28 # Install Droidmon Application
29 echo -e "\n-Installing Droidmon Application"
30 $ADB install hooking/Droidmon.apk
31 # Install Termux application
32 $ADB install nougat/com.termux.apk
33 # Install Anti Emulator Detection Application
34 echo -e "\n-Installing Anti Emulator Detection Application"
35 $ADB install hooking/EmulatorAntiDetect.apk
36 $ADB push anti-vm/fake-build.prop /data/local/tmp/
37 $ADB push anti-vm/fake-cpuinfo /data/local/tmp/
38 $ADB push anti-vm/fake-drivers /data/local/tmp/
39 # Install Content Generator
40 echo -e "\n-Installing Content Generator"
41 $ADB install apps/ImportContacts.apk
42 # Install Cuckoo Agent and Python for ARM
43 echo -e "\n-Pushing Agent executing scripts"
44 $ADB push ../../agent/android/python_agent/. /data/local/
45 $ADB shell chmod 06755 /data/local/aapt
46 $ADB shell chmod 06755 /data/local/get_terminal.sh
47 $ADB shell chmod 06755 /data/local/run_agent.sh
48 echo -e "\n***** Run the following commands and you are good to go*****\n"
49 echo "adb shell"
50 echo "cd /system/bin/"
51 echo "su root"
52 echo "su --install"
53 echo "su --daemon&"
54 echo "setenforce 0"
55 echo -e "\n"
56 echo -e "*NOTE"
57 echo " To make sure everything is working fine after the rooting is done, do the
   ↳ following:"
58 echo " Open android emualtor and run termux. Then type python2 to make sure to
   ↳ that python is installed"
59 echo " Then open adb shell Run the get_terminal.sh script and then type python2
   ↳ to verify that you "
60 echo " that you can run python with root previledges. After that run run_agent.
   ↳ sh and verify that agent is running."
```

4. Run the SuperSu app and if the device is rooted, it will ask for update, update the app, don't reboot yet.
5. After update, open Xposedframework app and click install there, it will ask for administrator permission, click grant.
6. Enable Droidmon and anti-emulator modules.
7. Soft reboot the machine, (It will be somewhere in xposed app)
8. After reboot verify root by opening SuperSU app and it doesn't give you an error and show xposedframework app, the device is rooted.
9. Close the machine, (The emulator 27.0.1 will save the snapshot of the machine and will load it at next start)
10. Start the machine with -writable-system option (without it the loaded machine is not rooted)and it will automatically load the previously saved snapshot, that means it will be up and running in no time then verify root access by opening SuperSu app or by running another app that required root access, then close the machine.
11. Start the machine again with adding one more option to above command, that is -no-snapshot, this time the machine will boot properly and verify the root by running SuperSU app or another app that needs root.

3.4.2 Python 2.7 on Android 5.1

In order to do the analysis we need to have python 2.7 in our Android 5.1 guest. CuckooDroid comes with python 2.7 binaries which are compiled for Android 4.1, which doesn't work with Android 5.1 because with the release of Android 5.1 Android requires all dynamically linked executables to support PIE (position-independent executables). So we need python compiled with PIE support. We can either compile it ourselves or we can use pre-compiled python with PIE support. Compiling python from source for Android was harder than we thought and we left it in favor of the pre-compiled solution because of simplicity.

Pre-Compiled python for Android 5.1+

Now, as we said above we used the pre-compiled python 2.7 that can be installed inside the Termux application [31]. Termux is an Android terminal emulator and Linux environment application that provides several linux command line tools on android making it really powerful. The app doesn't require root permissions which means it can't access or modify files outside of application. We needed a workaround so that we can run termux shell as root and have access to all system. Below are steps required to achieve that:

1. Download Termux apk from google playstore or apkmirror and install it to a rooted emulator
2. Open the installed app and type "pkg search python"
3. Install python2 using pkg install python2 (take help from step2 to determine exact command)
4. Now type python2 and you will have a working python. But this only works inside the app and can't be run as sudo/root.
5. To be able to run python(that comes with termux) as root from adb, we will need to download the script [32] (A fork of this which support optional command is here [33]). Tsu is an su wrapper for the terminal emulator, Termux [31].
6. Push this script to your emulator and run it with -e option.
7. It will drop you into root shell from termux.
8. Now we need to add the code inside the script so that instead of launching a shell it run our agent.py script as root.
9. To do that go to line 100 which is:

```
exec "$s" --preserve-environment -c "LD_LIBRARY_PATH=$LD_LIBRARY_PATH
↳ $ROOT_SHELL"
```

Step 10: Edit the line as follows:

```
exec "$s" --preserve-environment -c "LD_LIBRARY_PATH=$LD_LIBRARY_PATH python2 /
↳ data/local/agent.py"
```

10. Now run this script with "-e" and you will see that our agent.py script is been fired. We will save this script as agent.sh and replace our old agent.sh by it.
11. Now we need to add "-e" option to the part of code which fires up agent.sh, it can be located in the start_agent() method of "cuckoo-droid/modules/machinery/avd.py"
12. Due to some reasons we ran into some permissions problems like not being able to read hooks.json file. In logcat droidmon was complaining about not having permission to read hooks.json. This can be fixed by setting the global read flag on hooks.json after it is copied in cuckoodroid/analyzer/android/analyzer.py Below is the snippet of code:

```
1 def prepare(self):
2     # Initialize logging.
3     init_logging()
4
5     # Parse the analysis configuration file generated by the agent.
6     self.config = Config(cfg="analysis.conf")
7
8     # We update the target according to its category. If it's a file, then
9     # we store the path.
10    if self.config.category == "file":
11        self.target = os.path.join("/data/local/tmp", str(self.config.file_name))
12        shutil.copyfile("config/hooks.json", "/data/local/tmp/hooks.json")
13        os.chmod("/data/local/tmp/hooks.json", 0754) # Give set global read flag on
14        ↳ hooks.json
15    # If it's a URL, well.. we store the URL.
16    else:
17        self.target = self.config.target
```

13. Another problem was in the screenshot module, it wasn't able to read screenshots and this can be solved by using the same method as above i.e, setting global read at the place where the file is created and also changing the location of file, which in this case was because of sdcard being read-only and this function wasn't able to save the image there. Below is the modified code of take_screenshot() method in adb file:

```
1 def take_screenshot(filename):
2     proc1= subprocess.Popen("/system/bin/screencap -p /data/local/screenshots/"+
3     ↳ filename, # "/data/local/screenshots/" Fix for /sdcard read only
4     stdout=subprocess.PIPE,
5     shell=True,
6     stderr=subprocess.PIPE,
7     executable='/system/bin/sh')
8     stdout, stderr = proc1.communicate()
9
10    if len(stdout)>0:
11        log.info("take_screenshot stdout: %s", stdout)
12    if len(stderr)>0:
13        log.info("take_screenshot stderr: %s", stderr)
14
15    os.chmod("/data/local/screenshots/"+filename, 0744)
16    return "/data/local/screenshots/"+filename
```

3.5 CuckooDroid with Android 7.0

We didn't have much success with Android 7.0. We couldn't pass the persistent rooting problem and the machine wouldn't boot after the rooting procedure as described for Android 5.1. Furthermore it was extremely slow and was almost unusable so we decided to move on to more pressing matters :).

3.6 Improvements in CuckooDroid and pull request

Once we had the cuckoodroid working, we decided to add some features to it. We wrote one additional module for CuckooDroid analyzer. The goal was to improve code coverage by interacting with app in intelligent manner and then to introduce some random events using Monkey [34] tool. Monkey tool is described as "The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner" [34]. Our module is called IntentGun and the meaning of this name will become apparent in the next paragraph.

As we said in chapter 1 that an android application have different entry points called activities, receivers and services and that these can be started using Activity Manager (am) tool with intents. So in this module we make a list of all activities, receivers and services contained inside an app and then use the activity manager utility [35] to start these activities, receivers and services. The commands are give below:

```
# For Activity
$ am start -n Package/Activity -a action -c category data

# For receivers
$ am broadcast -n Package/Receiver -a action -c category

# For services
$ am startservice -n Package/Activity -a action -c category data
```

This module is still work in progress as the data section is not easy because only Multipurpose Internet Mail Extensions (MIME) type is given in the Manifest.xml file and we need to provide data as input to this command and on top of that sometimes these MIMEs are custom defined by the app developers. Once we are done with executing all entry points to an application, then we run the Monkey to generate random events. We used the following command to run the Monkey:

```
$ monkey --ignore-crashes --ignore-security-exceptions --monitor-native-crashes -p
↪ package --throttle 500 500
```

Its important to note here that this tool is used to stress test applications and we couldn't manage to make it work in stable condition. Usually it crashes the emulator altogether [36].

We also upgraded Cuckoo-Droid to support latest androgaurd and made a pull request to the Cuckoo-Droid official repository [16]. We decided not to include the intent gun module because of its unstable nature. The pull request can be found here [37]. Due to the Cuckoo-Droid maintainers being inactive, we also decided to keep a mirror of the update version at QuoScient github with updated documentation so that people who want to use it can see the updated documentation [23]. With this pull request we hope to make cuckoodroid stable enough that it works right out of the box. We were also continuously in touch with the Cuckoo Sandbox team.

3.7 Chapter Conclusion

In this chapter we explained how CuckooDroid works, which kind of changes we needed to make and what improvements we made. After doing the work described, in this chapter we also reached to following conclusions:

Emulator Anti Detection Although during our dynamic analysis we didn't analyzed enough malicious samples to make an estimate of the usage of emulation detection techniques in these samples, but a little search on the Internet revealed several techniques that were very easy to implement and integrate into an application and thus it can be very easy for a determined threat actor to add them to their code and can make its way to popular android application stores like google play store easily or can stay undetected in general. We will further explore this topic in the chapter 4.

Support for higher version android We also reached a conclusion that there is dire need for android malware analysis platforms that run the latest Android because only 1.5% users are using Android 4.1 [38] and this number is declining, also more and more apps are dropping support for android 4.1. For future development of cuckoodroid we think Frida, a dynamic instrumentation framework is a best candidate for instrumentation, because it is well maintained, support latest android operating systems, can be modified very quickly and is cross-platform. We will introduce Frida in Chapter 5. For a more quick solution, we recommend to compile python with PIE support or get it the way we did it i.e from Termux.

Slow emulator performance With the android emulators running higher versions of Android, there is big performance penalty. Running an arm image of android on x86 based PC system is also one of the reason of performance degradation. One solution to the problem can be using Houdini translation that allows android apps written for arm processors to be executed on x86 android image. We didn't investigated this solutions further and can be done as a separate project.

Native Code Currently CuckooDroid only detects if native code is being used in an application. It would be nice to have the capabilities of decompiling/disassembling of native code and include that into analysis results. In Chapter 5, We will talk about a tool called Frida, which supports hooking native code on Android and be xposed framework can be replaced by it.

Chapter 4

Dynamic Analysis: Evasion and Cloaking

4.1 Introduction

Fundamentally, the concept of emulator detection is rooted in the fact that complete system emulation is an arduous task. By discovering or observing differences between virtual and physical execution an attacker can create software virtualization checks that can be used to alter overall program behavior. Such differences may be an artifact of hardware state not correctly implemented in the virtual CPU, hardware or software components that have yet to be virtualized, or observable execution duration [39]. Emulator detection(evasion) and anti-emulator detection(cloaking¹) is an interesting area of research because its a continuous arms race on both ends. There are some valid reasons for developers to deploy emulator detection techniques into their apps and these same techniques can easily be applied by malicious actors to avoid detection or analysis.

Because of the popularity and wide use of Android, we felt the urge to study the common tricks and techniques that are used today in this area. Before diving into detailed discussion on these techniques, we will first introduce emulation detection and anti-emulation detection. Then at the end of this chapter we will show some results from our own application that can be used to test emulator against some common detection techniques.

We would like to note here that due to time constraints we couldn't explore Cloaking as much as we would have liked to. This topic seems to be not well researched and we can recommend it for further study. Since we are already talking about cloaking, We would like to mention a very interesting approach that is discussed in [40] which is based on splitting an APK to multiple smaller apks and then dynamically analyzing these smaller apks, furthermore cuckoodroid does have an xposed framework module installed that is called "Android Blue Pill" and during its reversing we found out that it is hooking some of the methods that are commonly used for emulation detection. We tried figuring out the exactly which and how many techniques it uses to cloak the emulator but there was no information available online and we didn't want to spend more time reversing it. Most of this chapter will focus on common emulator detection (evasion) methods.

4.1.1 Emulator Detection or Evasion

Its the techniques that are used by malicious or no malicious applications to detect whether its being executed in emulated environment. The goal is to change the behavior of the application if its being executed inside an emulator. Malicious actors can use this technique to avoid analysis or detection. Non Malicious actors can use it to protect their intellectual property or to stop cracking of their apps for accessing premium features in apps or games.

¹The term cloaking is normally used to evade root detection but we are using it here to evade emulator detection. Root detection can also be used to detect analysis.

4.1.2 Anti-Emulator Detection or Cloaking

On the other hand there are people (mostly security researchers) who wants to know what an application is doing and mostly they use emulators to do the dynamic analysis of applications. Now for an app that uses emulator detection techniques, its not possible to see its actual behavior. In order to solve this problem, anti-emulator detection techniques are used that tries to cloak the emulator from emulation detection checks and thus the application will be executed as it should on a real device.

4.2 emulation Detection or evasion Methods

In this section we will describe some of the well known techniques used for emulator detection and will present a list of the emulator detection techniques that was the result of our thorough research on the internet. A general introduction to some of well known techniques can be found in [41] and readers are encouraged to have a look at it. Root detection techniques can also be used to detect analysis and some of these techniques are described in [3]. These techniques can be easily evaded by using method hooking as demonstrated by [2]. Figure 4.1 from [2] lists the root detection techniques presented in [3]. There are also several open source packages on github which provides very easy to integrate root checking capabilities to android application, one such example is "root beer" [42].

Detection methods	Description
D1: Check installed packages	check whether rooting-related apps are installed.
D2: Check installed files	check whether binary files (such as busybox and su) that usually appear in rooted devices are installed.
D3: Check the BUILD tag	check whether Android image on the device is a stock image or a custom image built by third-party developers.
D4: Check system properties	check the value of system properties, <code>ro.debuggable</code> and <code>ro.secure</code> , that allow root-privileged shell.
D5: Check directory permission	check whether write permission is given to some directories that should have read-only permission.
D6: Check processes, services and tasks	check whether apps with root privilege are running.
D7: Check Rooting Traits using shell commands	check rooting traits in a separate process by using shell commands such as <code>su</code> and <code>ps</code>

Figure 4.1: Root detection techniques [2] [3]

The paper [2] also present its own approach to avoid analysis or reverse engineering. They use a stub dex file which contains all of emulation detection code, root detection code and code for loading original Dex file. They repackage the APK so that it loads the stub dex first, which performs emulator checking and if it detects no emulation, then it loads the original dex file from resources otherwise it doesn't load the original dex file. This behavior looks very similar to a recent wave of an Android malware category called "Droppers" but these droppers are only used to download payload and its very difficult to identify them as they don't need any dangerous permissions and had developed a new business model called "Downloader-as-a-Service" [43].

Checklist	stub DEX's method
custom image flashing	<code>detectTestKeys()</code>
change of system properties	<code>checkForSystemProperties()</code>
installed su binary	<code>checkForBinary()</code>
new commands from busybox	<code>checkForBusybox()</code>
filesystem attributes	<code>checkForFilesystem()</code>
rooting-related apps	<code>detectRootManagementApps()</code>
running root process	<code>detectRootProcess()</code>

Figure 4.2: Methods in stub dex [2]

They use an effective detection scheme called "call stack analysis" which can be used to detect method hooking and the application is then terminated. For more details on this topic, have a look at section 5.2.2

The paper [4] provides on some introductory information about Mobile Malware and some statistics which gives a general idea of mobile malware evasion techniques in use that can be seen in figure 4.3. We couldn't find any up-to date study discussing Mobile malware evasion techniques currently in use and that can be topic for future research. The most complete works on the topic that we found were [39] and recently published [44].

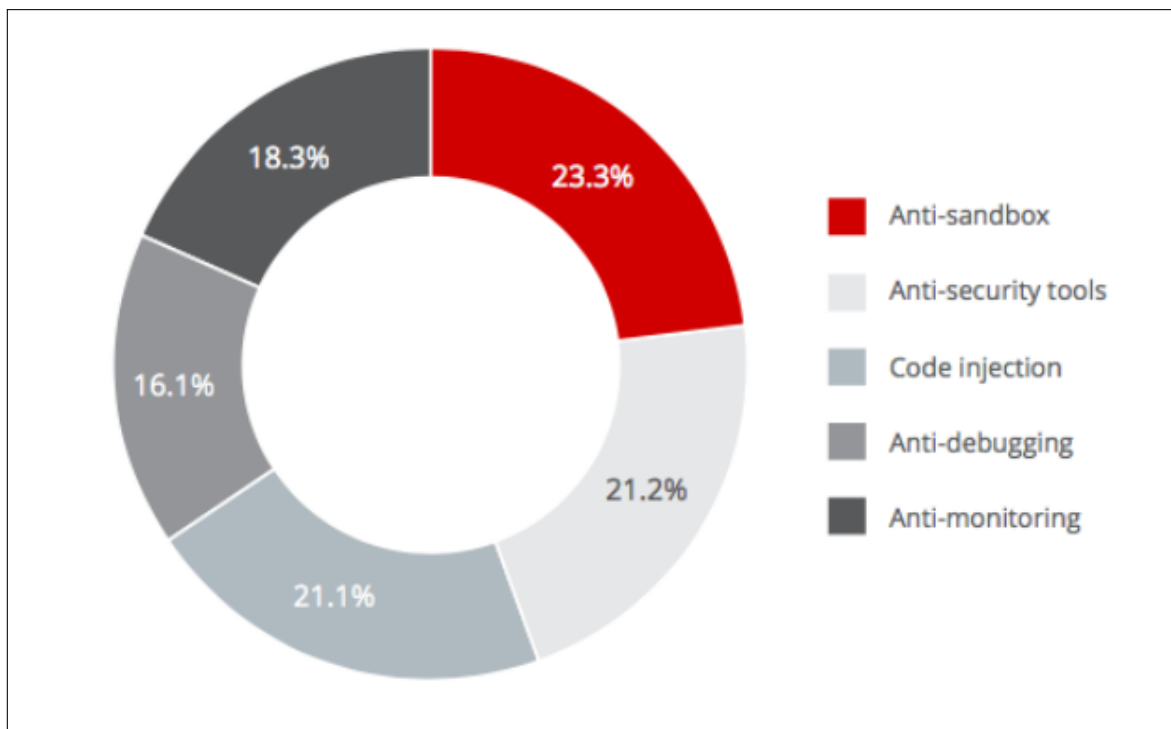


Figure 4.3: Pie chart of evasion techniques [4]

We create the below list with the help of [39] [45] and with our own techniques. We also implemented some of easier to implement tests and will talk about it in the next section. [44] implemented different evasion techniques into 22 samples and for those samples please refer to their paper. Below is the list:

1. Reading telephone number [45] [39]
2. Mobile country code [39]

3. Build.Host property [45] [39]
4. Build.Fingerprints property [45] [39]
5. Reading device IMEI number
6. Network address e.g. 10.0.0.2/24 and routing table (netH) [39]
 - (a) virtual router (address space: 10.0.2/24)
 - (b) Emulated network (IP address: 10.0.2.15)
7. Checking device uptime
8. Checking ping www.example.com results
9. CPU serial number mostly in 0's [39]
10. CPU frequency files absent [39]
11. Number of sensors (SensorManager API) [39]
12. Bluetooth not present [39]
13. Battery level [39]
14. Checking for Manufacturer related software which comes pre-installed [39]
15. Checking for other usage related data e.g image, sms, calls, WLAN networks, browser history, installed applications e.g facebook etc.
16. Checking for Google internet services [39]
17. Checking for marketplace application [39]
18. Checking for carrier added applications [39]
19. Delaying execution [45]
20. Detecting lack or abundance of user input
21. Checking if Qemu can handle interrupt while executing blocks [45]
22. isUserMonkey(), setRunAsMonkey(false) in Android API
23. Checking other network configurations
24. Using native code to detect Qemu based detection in CPU implementations like some bugs
25. CPU Performance [39]
26. GPU performance (Low FPS, Large variation(May not work with new emulators because of improved integration)) [39]
27. Input from sensors is not spread enough [39]
28. Camera detection [39]
29. Checking for Superuser.apk
30. Checking for some specific applications with package names [39]
31. Checking for apps which run only on rooted devices, e.g, busybox, executing su and id command, [39]
32. Checking for files e.g, /system/bin/, /system/sbin/, /sbin/. Or searching them in \$PATH (manually or by which "file")
33. File permissions: Some rooting tools make certain root folders readable (e.g.,/data) or writable (e.g., /etc, /system/sbin, /system, /proc./vendor/bin) to any process on the device. The File.canRead and File.canWrite Java APIs, or access C API can be used to check whether such condition exists.

34. Shell Command Execution: Applications can use Runtime.exec Java API, ProcessBuilder Java class, or execve C API to execute a specific command in a separate process, and then examine the output of the shell command to detect rooting traits. Commonly employed shell commands are:
 - (a) su: If the su binary exists, this shell command will exit without error; otherwise, an IOException will be thrown to the calling application.
 - (b) which su: This check involves executing the Linux which command with parameter su and verifying if the result is 0 (indicating the su binary was found). This is essentially the same as parsing the PATH variable, but requires less work for the caller.
 - (c) id: This check is based on executing the command id, and verifying the UID to determine if it corresponds to root (uid=0 is root)
35. System Mounted: Normally the /system partition on an Android device is mounted ro (read only). Some rooting methods require this partition to be remounted rw (read-/write). There are mainly two variants of this check. The first simply runs the mount command and looks for a rw flag. The second attempts to create a file under /system/ or /data/. If the file is successfully created, it implies the mount is rw.
36. Ability to mount: This method attempts to mount the /system partition with the command mount -o remount,rw /system, and then checks if the return code was 0.
37. Check Processes/Services/Tasks: This check consists of finding whether a specific application that requires root privileges is running on the device. The ActivityManager.getRunningAppProcesses method returns a list of currently running application processes. Similarly, getRunningServices or getRecentTasks APIs are also used by applications to retrieve a list of current running services or tasks.
38. System properties: If the value of system property ro.secure equals 0, the adb shell will be running as root instead of shell user. System.getProperty Java API can be used by applications to examine this property value. In addition, examining ADB source code reveals that the adbd daemon is also running as root user if both ro.debuggable and service.adb.root properties are set to 1. [45] [39]
39. hooking detection: Detecting hooks using call stack analysis [2]
40. Check debugger and installer: This one is not an anti-emulator but its purpose is also to obstruct the dynamic analysis. Like this skinner adware reported by checkpoint [46], it uses Debug.isDebuggerConnected() and Debug.waitForDebugger() to check if a debugger exists. More interesting, it also gets the installer using getInstallerPackageName and sees if its installed by Google Play (com.android.vending). So if you install the program to a device with adb, like most analysts do, the application wont work. [45]

4.3 Emulator testing app

After collecting information about how emulator detection works, we wanted to make an app which can help us test emulators against evasion techniques so we decided to implement some of these techniques in a very simple app. Below is the list of techniques that we implemented.

- read Build information
 - Build.FINGERPRINT
 - Build.HOST
 - Build.HARDWARE
 - Build.PRODUCT
 - Build.MODEL
- Getting stack trace to check for abnormal calls
- Writing to a file
- Getting network information

- Is wifi connected
- Is mobile network connected
- Get information related to connected network
- Get Phone data
 - Phone number
 - Country
 - getDeviceID
- Checking for Root using RootBeer [42]
- Pinging www.example.com
- Trying to get the IP address of the phone for all interfaces

Our app will print the result of these tests on the screen. The screenshots of our app can be seen in figure 4.4 and figure 4.5. Figure 4.4 shows result when the method "getPhone" is not hooked. Also notice the absence of Bluetooth device, device uptime, ping result, device ID and rootbeer showing that device is rooted.

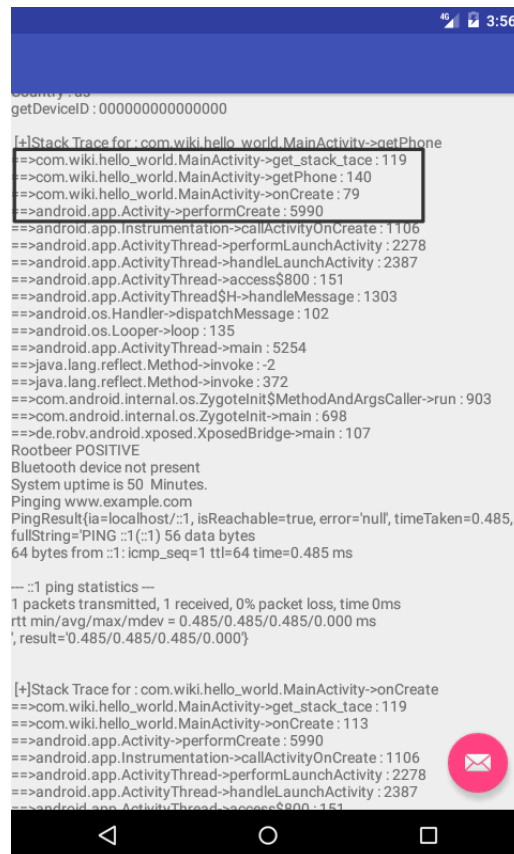


Figure 4.4: Executing our test app without hooking "getPhone"

Figure 4.5 shows the result when the method "getPhone" is hooked. Notice the difference in call trace. In the next chapter we will explain the output of this app in more details along with introduction to another instrumentation tool called "Frida".



Figure 4.5: Executing our test app with hooking "getPhone"

4.4 Chapter Conclusion

In this chapter we gave an overview of the android emulation detection and anti-emulator detection techniques. We listed some of the techniques that can be used to check for emulators. We showed how we made a basic android application to detect emulator and showed us the call trace of function when hooked and when not hooked. We also came to the conclusion that the anti-emulation detection part is worthy of future research.

Chapter 5

Instrumentation: Frida with Android

5.1 Introduction

Frida is a dynamic instrumentation toolkit that supports Windows, macOS, GNU/Linux, iOS, Android, and QNX. Its widely available for these platforms and lets you inject snippets of JavaScript or your own library into native apps on supported platforms. Frida also provides you with some simple tools built on top of the Frida API. These can be used as-is, tweaked to your needs, or serve as examples for how to use the API. It injects Google's V8 engine into the target processes, where your JS gets executed with full access to memory, hooking functions and even calling native functions inside the process. [47]

The goal of this chapter is to give readers an introduction to this tool as the work we did on Frida during this internship is still incomplete and can not be reported here. In the next section, We will show you how to install Frida and will do a few examples that will explain the use cases of this tool.

5.2 Frida with Android

5.2.1 Installation

We will be using Ubuntu 16.04 for installation but its pretty simple on other platforms as well.

1. Download frida using

```
pip install frida
```

2. Check version by using

```
frida --version
```

3. Go to frida github and download the same version of server for android and correct architecture. (Currently x86 server on x86 image is crashing the app)

4. Push frida-server to android device, chmod 775 it and execute it with

```
adb shell "/path/to/frida-server -D"
```

5. On host test frida with

```
frida-ps -U
```

5.2.2 Examples

In this section we will show you some examples on how frida can be used.

example 1: Hooking a method In this example we will run one of our own developed application. This application is for testing the anti-emulator capabilities of our AVD that is used for dynamic analysis. It obtains different information from the system which can be used to detect emulation environment or root. It also displays stack trace of some of its methods and that information can be used to detect hooking. In this example we will use Frida to hook the "getPhone()" method inside MainActivity. Figure 5.1 shows the source code of this method and Figure 5.2 shows the stack trace or call trace of this method. From the call trace it can be seen that there are no unexpected calls.

```
private String getPhone() {
    TelephonyManager phoneMgr = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

    String phoneData = "getSimOperatorName : \t " + phoneMgr.getSimOperatorName() + "\n" +
        "\tgetNetworkOperatorName : \t " + phoneMgr.getNetworkOperator() + "\n";

    if (ActivityCompat.checkSelfPermission(activity, wantPermission) != PackageManager.PERMISSION_GRANTED) return "";

    phoneData = phoneData.concat("Phone Number : " + phoneMgr.getLine1Number() + "\n");
    phoneData = phoneData.concat("Country : " + phoneMgr.getSimCountryIso() + "\n");
    phoneData = phoneData.concat("getDeviceID : " + phoneMgr.getDeviceId() + "\n");
    phoneData = phoneData.concat(get_stack_trace());
    return phoneData;
}
```

Figure 5.1: Source code of the method getPhone()

```
[+]Stack Trace for :com.wiki.hello_world.MainActivity->getPhone
==>com.wiki.hello_world.MainActivity->get_stack_trace: 119
==>com.wiki.hello_world.MainActivity->getPhone: 140
==>com.wiki.hello_world.MainActivity->onCreate: 79
==>android.app.Activity->performCreate: 5990
==>android.app.Instrumentation->callActivityOnCreate: 1106
==>android.app.ActivityThread->performLaunchActivity: 2278
==>android.app.ActivityThread->handleLaunchActivity: 2387
==>android.app.ActivityThread->access$800: 151
==>android.app.ActivityThread$H->handleMessage: 1303
==>android.os.Handler->dispatchMessage: 102
==>android.os.Looper->loop: 135
==>android.app.ActivityThread->main: 5254
==>java.lang.reflect.Method->invoke: -2
==>java.lang.reflect.Method->invoke: 372
==>com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run: 903
==>com.android.internal.os.ZygoteInit->main: 698
==>de.robv.android.xposed.XposedBridge->main: 107
Rootbeer POSITIVE
Bluetooth device not present
System uptime is 50 Minutes.
```

Figure 5.2: Stack/call trace of the non-hooked method getPhone()

Now we will hook this method using frida, below is the procedure:

1. Execute frida-server on guest, if it not running already.
2. Execute the below python script

```
1 import frida, sys, time
2 def on_message(message, data):
3     if message['type'] == 'send':
4         print("[*] {0}".format(message['payload']))
5     else:
6         print(message)
7     jscode = """
8     Java.perform(function () {
9         // Function to hook is defined here
10        var MainActivity= Java.use('com.wiki.hello_world.MainActivity');
11        // Whenever button is clicked
12        MainActivity.getPhone.implementation = function () {
13            // Show a message to know that the function got called
14            send('getPhone');
15            // Call the original onClick handler
```

```

16     var output = this.getPhone();
17     send(output);
18     // Modify return value of getPhone method
19     return output+"NNN0000TTTT";
20 };
21 });
22 """
23 device = frida.get_usb_device()
24 pid = device.spawn(['com.wiki.hello_world'])
25 process = device.attach(pid)
26 script = process.create_script(jscode)
27 script.on('message', on_message)
28 print('[*] Running CTF')
29 script.load()
30 device.resume(pid)
31 sys.stdin.read()

```

The above code comprises of two languages, one is the python code that is being executed on the host. Second is the JavaScript code that is injected in the target process. The python script is doing the following:

1. It connects with adb in line "device = frida.get_usb_device()",
2. Opens the application process (its paused and need to be resumed)
3. Attach it to process, adds the JavaScript code store in jscode variable
4. Registers the callback function to receive messages from guest (send using send("message"))
5. Load the script and resumes the start executing the application opened in 2.
6. Add stdin to stop application from exiting

In the JavaScript code we are doing following:

1. We open MainActivity using Java.use
2. We hook the getPhone method, send its original return value and modify the return value.

Figure 5.3 shows the messages sent by frida-server to host including our messages and confirms that the method getPhone is hooked by sending back the string "getPhone" and return value of this method. We made this output(build info+ stack trace) rather short to be able to show it in on one screen. and figure 5.4 , and modified returned value in app window by appending an easy to spot string of "NNNNNOOOOOOTTTTT" to the return string of the getPhone and this can be seen in figure 5.3. Also notice the extra calls in stack trace and compare it with the previous one in which no hooking was involved.



Figure 5.3: Hooked output on app screen

```
wra@wra01ws: ~/workspace/cuckoo_frida
wra@wra01ws:~/workspace/cuckoo_frida$ python hello_world.py
[*] Running CTF
[*] getPhone
[*] getSimOperatorName :      Android
    getNetworkOperatorName :      310260
Phone Number : 15555215554
Country : us
getDeviceID : 0000000000000000

[+]Stack Trace for : com.wiki.hello_world.MainActivity->getPhone
==>com.wiki.hello_world.MainActivity->get_stack_tace : 119
==>com.wiki.hello_world.MainActivity->getPhone : 140
==>com.wiki.hello_world.MainActivity->getPhone : -2
==>com.wiki.hello_world.MainActivity->onCreate : 79
==>android.app.Activity->performCreate : 5990
==>android.app.Instrumentation->callActivityOnCreate : 1106
==>android.app.ActivityThread->performLaunchActivity : 2278
==>android.app.ActivityThread->handleLaunchActivity : 2387
==>android.app.ActivityThread->access$800 : 151
==>android.app.ActivityThread$H->handleMessage : 1303
==>android.os.Handler->dispatchMessage : 102
==>android.os.Looper->loop : 135
==>android.app.ActivityThread->main : 5254
==>java.lang.reflect.Method->invoke : -2
==>java.lang.reflect.Method->invoke : 372
==>com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run : 903
==>com.android.internal.os.ZygoteInit->main : 698
==>de.robv.android.xposed.XposedBridge->main : 107
```

Figure 5.4: terminal output of app with hooked getPhone()

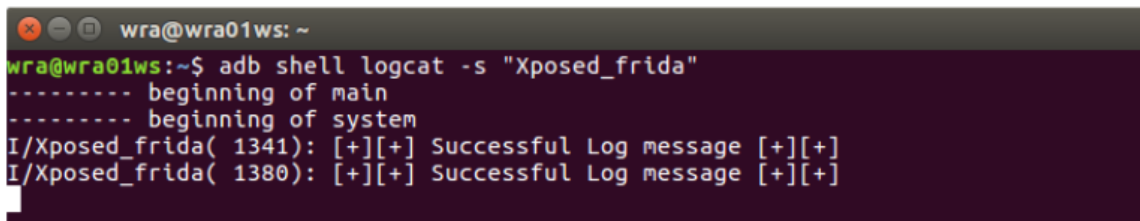
Example 2: Getting Javascript wrapper for Java classes (Logcat example): In this example we will demonstrate on how to use the Java API of android inside our injected JavaScript code. Frida allows us to get wrappers for Java methods of android and use them inside of our injected code as we wish. Python code for this example is below. In this code it can be seen that we use Java.use method provided by JavaScript API of frida to get a JavaScript wrapper for class 'android.util.Log'. We then check for log.i method and if it is present, we call it.

```

1 import frida, sys, time
2 def on_message(message, data):
3     if message['type'] == 'send':
4         print("[*] {0}".format(message['payload']))
5     else:
6         print(message)
7     jscode = """
8     Java.perform(function () {
9         // Function to hook is defined here
10        var MainActivity = Java.use('com.wiki.hello_world.MainActivity');
11        var Log = Java.use('android.util.Log');
12        if (Log.i){
13            send("Sending message");
14            Log.i("Xposed_frida", "[+][+] Successful Log message [+][+]");
15        }
16    });
17    """
18 device = frida.get_usb_device()
19 pid = device.spawn(['com.wiki.hello_world'])
20 process = device.attach(pid)
21 script = process.create_script(jscode)
22 script.on('message', on_message)
23 print('[*] Running CTF')
24 script.load()
25 device.resume(pid)
26 sys.stdin.read()

```

The result of running this script can be seen in figure 5.5. This example demonstrate the power of frida, we can use all the Java functionality inside our script.



```

wra@wra01ws: ~
wra@wra01ws:~$ adb shell logcat -s "Xposed_frida"
----- beginning of main
----- beginning of system
I/Xposed_frida( 1341): [+][+] Successful Log message [+][+]
I/Xposed_frida( 1380): [+][+] Successful Log message [+][+]

```

Figure 5.5: Terminal output

Example 3: Interactive Frida (Frida-CLI): One can get an interactive frida shell by entering the following command:

```
frida -U --no-pause -f com.package.name
```

This command will launch the app and gives an interactive shell where you can type Javascript to do your stuff. If you want to write your script before the app launches, omit the `--no-pause` parameter. Because of the auto-complete feature as show in figure 5.6 this is a great place to discover methods and features supported by frida.

```

wra@wra01ws: ~
Failed to spawn: unable to connect to remote frida-server: closed
wra@wra01ws:~$ frida -U --no-pause -f com.wiki.hello_world

  /_/_/  |  Frida 10.6.32 - A world-class dynamic instrumentation framework
 |(_|_|_|
 >_|_|_|
 /_/_/_|_|
 . . . .
 . . . .
 . . . .
 . . . .
 . . . .
 . . . .
 More info at http://www.frida.re/docs/home/
Spawned `com.wiki.hello_world`. Resuming main thread!
[Android Emulator 5554::com.wiki.hello_world]-> help
Help: #TODO :)
[Android Emulator 5554::com.wiki.hello_world]-> var a = new Fil
ReferenceError: identifier 'Fil' undefined
[Android Emulator 5554::com.wiki.hello_world]-> var a = new File(Math
  InvocationListener
  InvocationReturnValue
  JSON
  Java
  Kernel
  Math
  Memory

```

Figure 5.6: interactive frida output

Example 4: Compile a Frida script comprised of one or more Node.js modules (File reading test case): Its not much useful to code in simple JavaScript in our use case scenario. Our use case scenario is same as server side programming using JavaScript and its important to be able to use NodeJS. Frida provides a tool for that called "Frida-Compile" [48]. This tool compile a Frida script comprised of one or more Node.js modules. In layman terms, its takes a NodeJS projects and create a single file payload/agent that we can load using the "load_script" method that comes with frida.

Before using first we need to install frida-compile and it comes a node package and we can use node package manager (npm) to install it. We will install it with -g parameter to make it available globally.

```
$ sudo npm install frida-compile -g
```

To make sure that frida-compile is installed correctly and to see help we will type the following command:

```

1 $ frida-compile --help
2
3 Usage: frida-compile [options] <module>
4
5 Options:
6
7 -V, --version                output the version number
8 -o, --output <file>         set output <file>
9 -w, --watch                  watch for changes and recompile
10 -b, --bytecode               output bytecode
11 -x, --no-babelify            skip Babel transforms
12 -S, --no-sourcemap           omit sourcemap
13 -c, --compress               compress using UglifyJS2
14 -a, --use-absolute-paths     use absolute source paths
15 -h, --help                   output usage information

```

Now the command to compile a NodeJS project/code is below. Here app.js the NodeJS main file and payload.js is the output file that we will load in python binding using load_script.

```
1 $ frida-compile app.js -o payload.js
```

Below are scripts that I used to open a text file located on the device and send its content to host. Make sure that you install the required packages frida-fs [49] using npm. Here is the source code of app.js:

```
1
2   'use strict';
3   const fs = require("frida-fs");
4   Java.perform(function () {
5       var readStream = fs.createReadStream("/data/local/tmp/hooks.json");
6       var text = "";
7       readStream
8           .on('readable', function () {
9               var chunk;
10              while (null !== (chunk = readStream.read())) {
11                  text = text.concat(chunk);
12              }
13          })
14          .on('end', function () {
15              send(text);
16          });
17  });
```

In this python code we first compile the NodeJS code using frida-compile and then use the output script as a normal JavaScript payload shown in every other example.

```
1  import frida, sys, time, os, subprocess
2
3  def on_message(message, data):
4      try:
5          if message['type'] == 'send':
6              print("[*] {0}".format(message['payload']))
7          else:
8              print(message)
9      except:
10         print(message)
11
12  frida_ready = False
13  device = frida.get_usb_device()
14  while not frida_ready:
15      try:
16          time.sleep(5)
17          pid = device.spawn(['com.wiki.hello_world'])
18          frida_ready = True
19      except Exception as e:
20          print("Frida not ready yet " + str(e))
21
22  time.sleep(1)
23
24  cwd = os.path.dirname(__file__)
25  print("Working Directory is : " + cwd)
26
27  print ("Compilation result is: \n")
28  print(subprocess.Popen("frida-compile index.js -o compiled.js", shell=True).
29      ↪ communicate())
29
30
31  try:
32      process = device.attach(pid)
33      script = process.create_script(
34          open("/home/wra/workspace/cuckoodroid/frida-scripts/file_io/compiled.js").read())
35      script.on('message', on_message)
36      print('[*] Running Frida')
37      script.load()
38      device.resume(pid) # At end Important, otherwise app will start and you will miss
39      ↪ some early code execution as hooks won't be inplace at that time.
40  except Exception as e:
41      print("Frida python binding code error : " + str(e))
42  sys.stdin.read(1)
```

Below the output, you can see that the contents of hooks.json file was sent by guest and we displayed it on host.

```
1 Working Directory is :
2 Compilation result is:
3
4 (None, None)
5 [*] Running Frida
6 [*] {
7   "hookConfigs": [
8     {
9       "class_name": "android.telephony.TelephonyManager",
10      "method": "getDeviceId",
11      "thisObject": false,
12      "type": "fingerprint"
13    },
14    {
15      "class_name": "android.telephony.TelephonyManager",
16      "method": "getSubscriberId",
17      "thisObject": false,
18      "type": "fingerprint"
19    }
20  ],
21  "trace": false
22 }
```

5.3 Chapter conclusion

In this chapter we gave an introduction to Frida. We tried to show how powerful it is by some simple examples. We also concluded that integrating Frida to CuckooDroid can have several benefits and it will help keep the CuckooDroid in supporting higher versions of Android.

Chapter 6

Conclusion and future work

In this project we created tools for automatic static and dynamic analysis of Android application. We used androguard for static analysis, discussed the extracted data and how this data can be used to compare android apps. For dynamic analysis we used an extension of cuckoo called CuckooDroid. We made several bug fixes in it and made improvements by adding an auxiliary module IntentGun and upgrading to Android 5.1. We also researched the currently known emulation detection techniques.

During the course of this project we identified several areas in which future work is recommended and we will describe them here:

Separating API or commonly used methods from possible malicious methods During our static analysis we extract information about every method in an APK and then calculate a signature/hash for each of these methods which can be used for finding similar methods as shown in figure 2.7 and 2.8. We can see in these figures that these samples share code but its not possible to tell whether this is malicious or not without looking at it manually. We need some tools that identify the standard API methods and some other frequently used parts of code in that shared code. It will simplify our search for malicious code segments inside an app and help us calculate similarity measures based on code parts that really matters.

MIME type to data As we discussed earlier that we wrote an auxiliary module "IntenGun" for cuckoodroid in order to increase code coverage. This module generates intent messages for each and every component of an Android app to activate that component. Sometimes these intent messages need to have data appended to them and currently we don't have any mechanism to add this data. This can be done checking the MIME-type in the intent-filter for that component and then appending the appropriate data according to that MIME-type.

Cuckoo-Droid support for higher versions of Android Currently cuckoo-droid only supports up to Android 4.1, which is very old and needs to be upgraded. There are some challenges in achieving that and should be done in future as most of new android apps doesn't run on it.

Slow android emulator We found out that running arm image of Android 5.1 or higher on Android emulator is very slow. This problem needs to be solved. One possible solution can be to use x86 image and use Houdini arm translation on top of it.

Anti-Emulator-detection We concluded that there is a lot of emulator detection techniques available and it is only a matter of time that their usage picks up the pace. There are already some anti-emulator detection techniques available but this topic is not well researched and can be explored further.

Real life examples of Android emulator detection techniques used We noted in the literature there is not much related to real world examples and there is a empty space between literature and whats going in the real world. It would be interesting to explore what kind of techniques real life android malware uses to avoid detection or to detect emulators. Might also be worthy to explore some non-malicious applications using these techniques to discourage reverse engineering.

Support of Native Code, Frida Android supports native compiled libraries but till now most of the android malware analysis tools focus on the Dex part. Developers (malicious or non malicious) had been using this as a place to hide their code from the eyes of analyzers. Therefore support for the analysis of native compiled libraries can boost up the detection capabilities of a malware analysis system and can present analyzer with more information and we recommend for future work. Frida is nice candidate for this task as it supports hooking into native functions on android.

Cuckoo-Droid on Physical devices As we talked about emulation detection techniques, one of way of fighting it is using real devices for analysis. Real devices will also be way more faster than using an emulator and can be made to run latest android. This year (2018), there was a similar project offered in Google Summer of Code and will be most probably offered in future as well [50]. Interested students can also apply there and do this project under the supervision of honeynet project.

Upgrading Cuckoo-Droid to latest cuckoo sandbox Currently, Cuckoo-droid is on cuckoo 1.2. Recent version of cuckoo is 2.0 which way ahead of Cuckoo-Droid and is resulting in Cuckoo sandbox community distancing themselves from Cuckoo-Droid. Cuckoo-Droid should be upgraded to cuckoo 2.0 and can be a nice topic for future work.

Large number of methods in APKs As compared to other executable files, android apks contains very huge number of methods and visualization or processing tools used for those other executable files format are not sufficient for APKs. Separate tools are required for APK files to visualize methods and other related meta data of an APK.

Bibliography

- [1] “Dalvik executable picture by ange albertini.” *corkami github repository* [Online] . Available: <https://github.com/corkami/pics/blob/master/binary/DEX.png>. [Accessed: Dec 20, 2017].
- [2] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, “An android application protection scheme against dynamic reverse engineering attacks,” *JoWUA*, vol. 7, no. 3, pp. 40–52, 2016.
- [3] S.-T. Sun, A. Cuadros, and K. Beznosov, “Android rooting: methods, detection, and evasion,” in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 3–14, ACM, 2015.
- [4] B. Amro, “Malware detection techniques for mobile devices,” *arXiv preprint arXiv:1801.02837*, 2018.
- [5] Android, “Android application fundamentals.” *Android Developers* [Online] . Available: <https://developer.android.com/guide/components/fundamentals.html>. [Accessed: Dec. 20, 2017].
- [6] Android, “Android intents class documentation.” [Online] . Available: <https://developer.android.com/reference/android/content/Intent.html>. [Accessed: Dec 20, 2017].
- [7] Android, “Reduce the apk size.” *Android Developers* [Online] . Available: <https://developer.android.com/topic/performance/reduce-apk-size.html>. [Accessed: Dec 20, 2017].
- [8] Xamarin, “Using android assets.” *Xamarin developers website* [Online] . Available: https://developer.xamarin.com/guides/android/application_fundamentals/resources_in_android/part_6_-_using_android_assets/. [Accessed: Dec 19, 2017].
- [9] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, “Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy,” in *NDSS*, 2016.
- [10] “Dalvik executable format.” *Android Open Source Project* [Online] . Available: <https://source.android.com/devices/tech/dalvik/dex-format>. [Accessed: Dec 20, 2017].
- [11] GuardSquare, “New android vulnerability allows attackers to modify apps without affecting their signatures.” *Guardsquare blog post* [Online] . Available: <https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-attackers-modify-apps-without-affecting-their-signatures>. [Accessed: Dec. 20, 2017].
- [12] V. Z. (TrendMicro), “Janus android app signature bypass allows attackers to modify legitimate apps.” *Trendmicro blog post about janus vulnerability* [Online] . Available: <https://blog.trendmicro.com/trendlabs-security-intelligence/janus-android-app-signature-bypass-allows-attackers-modify-legitimate-apps/>. [Accessed: 27 Dec 2017].
- [13] E. Vaknin, G. Elbaz, A. Boxiner, and O. Vanunu, “Parsedroid: Targeting the android development & research community.” *Checkpoint website* [Online] . Available: <https://research.checkpoint.com/parsedroid-targeting-android-development-research-community/>. [Accessed: 20 Dec 2017].
- [14] Android, “Enable multidex for apps with over 64k methods.” *Android Developers* [Online] . Available: <https://developer.android.com/studio/build/multidex.html>. [Accessed: Dec 20, 2017].

- [15] A. Desnos, “Androguard,” *URL: <https://github.com/androguard/androguard>*, 2017.
- [16] idanr1986, “Cuckoo-droid github.” *Cucoo-Droid github repository* [Online] . Available: <https://github.com/idanr1986/cuckoo-droid>. [Accessed: Dec 28, 2017].
- [17] L. Li, T. F. Bissyand, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, “Static analysis of android apps: A systematic literature review,”
- [18] “bliss: A tool for computing automorphism groups and canonical labelings of graphs.” *Webpage for bliss* [Online] . Available: <http://www.tcs.hut.fi/Software/bliss/>. [Accessed: Feb 21, 2018].
- [19] T. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs,” in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 135–149, SIAM, 2007.
- [20] “reox: Androguard github repository maintainer.” *reox github user account* [Online] . Available: <https://github.com/reox>. [Accessed: Feb 21, 2018].
- [21] “Cuckoo-droid documentation.” *cuckoo-droid documentation* [Online] . Available: <http://cuckoo-droid.readthedocs.io/en/latest/>. [Accessed: Dec 28, 2017].
- [22] Cuckoo-Sandbox, “Cuckoo website.” *Cuckoo official website* [Online] . Available: <https://cuckoosandbox.org/>. [Accessed: Dec 28, 2017].
- [23] Q. GmbH, “Cuckoodroid quoscient repository.” *CuckooDroid QuoScient repository* [Online] . Available: <https://github.com/quoscient/cuckoo-droid>. [Accessed: July 25, 2018].
- [24] “Xposed module repository.” *Xposed Module* [Online] . Available: <http://repo.xposed.info/module/de.robv.android.xposed.installer>. [Accessed: Jan 2, 2018].
- [25] melonaerial, “Android avd root access fail.” *github repository* [Online] . Available: <https://github.com/idanr1986/cuckoo-droid/issues/4>. [Accessed: Jan 2, 2018].
- [26] W. Rashid, “Cuckoo-droid tutorials part-03: Rooting avd (android virtual device).” *Cuckoo-droid tutorial 03* [Online] . Available: <https://www.youtube.com/watch?v=dhu5iIQEymU>. [Accessed: April 03, 2018].
- [27] “Subprocess management python 2.7.” *Python Documentation* [Online] . Available: <https://docs.python.org/2/library/subprocess.html>. [Accessed: Jan 2, 2018].
- [28] “Subprocess.popen issue 16255.” *Python Issue tracker* [Online] . Available: <https://bugs.python.org/issue16255>. [Accessed: Jan 2, 2018].
- [29] W. Rashid, “Cuckoo-droid documentation.” *Cuckoo-droid tutorial 04* [Online] . Available: <https://www.youtube.com/watch?v=HRBIZaA5N-w>. [Accessed: May 27, 2018].
- [30] “Android emulator 7.1 rooting procedure.” *Stack Exchange Question on Rooting AVD* [Online] . Available: <https://android.stackexchange.com/questions/171442/root-android-virtual-device-with-android-7-1-1/176447>. [Accessed: Jan 2, 2018].
- [31] “Termux website.” *Termux Website* [Online] . Available: <https://termux.com/>. [Accessed: Jan 2, 2018].
- [32] cswl, “tsu.” *tsu github repository* [Online] . Available: <https://github.com/cswl/tsu>. [Accessed: 27 Dec 2017].
- [33] cswl and kiney, “tsu.” *tsu github repository* [Online] . Available: <https://github.com/kiney/tsu>. [Accessed: 27 Dec 2017].
- [34] Android, “Ui/application exerciser monkey.” *Android Developers* [Online] . Available: <https://developer.android.com/studio/test/monkey>. [Accessed: 26 Dec 2017].
- [35] Android, “Android activity manager.” *Android Developers* [Online] . Available: <https://developer.android.com/studio/command-line/adb#am>. [Accessed: 26 Dec 2017].
- [36] waqar rashid, “Using monkey for interactions with app to increase code coverage.” *Cuckoo-droid github issue* [Online] . Available: <https://github.com/idanr1986/cuckoo-droid/issues/52>. [Accessed: 26 Dec 2017].

- [37] W. Rashid, “Pull request to cuckoodroid.” *Pull request to CuckooDroid* [Online] . Available: <https://github.com/idanr1986/cuckoo-droid/pull/74>. [Accessed: July 25, 2018].
- [38] Android, “Distribution dashboard.” *Android Developers* [Online] . Available: <https://developer.android.com/about/dashboards/>. [Accessed: 14 Jun 2017].
- [39] T. Vidas and N. Christin, “Evading android runtime analysis via sandbox detection,” in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 447–458, ACM, 2014.
- [40] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, “Harvesting runtime values in android applications that feature anti-analysis techniques.” in *NDSS*, 2016.
- [41] D. Geist, M. Nigmatullin, and R. Bierens, “Jailbreak/root detection evasion study on ios and android,” 2016.
- [42] scottyab, “root beer by scottyab.” *rootbeer github repository* [Online] . Available: <https://github.com/scottyab/rootbeer>. [Accessed: 10 Feb 2018].
- [43] C. Cimpanu, “Droppers.” *Droppers Is How Android Malware Keeps Sneaking Into the Play Store* [Online] . Available: <https://www.bleepingcomputer.com/news/security/droppers-is-how-android-malware-keeps-sneaking-into-the-play-store/> [Accessed: July 25, 2018].
- [44] L. Bello and M. Pistoia. *Ares: Triggering Payload of Evasive Malware for Android*. 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILE-Soft 2018) 2018.
- [45] B. BRENNER, “Android malware anti-emulation techniques.” *Sophos website* [Online] . Available: <https://news.sophos.com/en-us/2017/04/13/android-malware-anti-emulation-techniques/>. [Accessed: 05/01/2018].
- [46] checkpoint, “The skinner adware rears its ugly head on google play.” *Checkpoint website* [Online] . Available: <https://blog.checkpoint.com/2017/03/08/skinner-adware-rears-ugly-head-google-play/>. [Accessed: 30 May 2018].
- [47] Frida, “Frida.” *Frida website* [Online] . Available: <https://www.frida.re/>. [Accessed: Feb. 28, 18].
- [48] O. A. V. Ravens, “frida-compile.” *frida-compile github repository* [Online] . Available: <https://github.com/frida/frida-compile>. [Accessed: 30 Dec 2017].
- [49] O. A. V. Ravens, “frida-fs.” *github repository* [Online] . Available: <https://github.com/nowsecure/frida-fs>. [Accessed: 30 Dec 2017].
- [50] H. Gonzalez, “Project #19 - android instrumented sandbox in a real device.” *honey.net project GSoC 2018 projects* [Online] . Available: <https://www.honey.net.org/gsoc2018/ideas>. [Accessed: 20, March 2018].
- [51] L. Stefanko, “Multi-stage malware sneaks into google play.” *welivesecurity website* [Online] . Available: <https://www.welivesecurity.com/2017/11/15/multi-stage-malware-sneaks-google-play/>. [Accessed: 20, Dec 2017].

Appendix A

Android Activity Manager tool(am)

Within an adb shell, you can issue commands with the activity manager (am) tool [35] to perform various system actions, such as start an activity, force-stop a process, broadcast an intent, modify the device screen properties, and more. While in a shell, the syntax is:

```
am command
```

You can also issue an activity manager command directly from adb without entering a remote shell. For example:

```
adb shell am start -a android.intent.action.VIEW
```

For detailed documentation on activity manager commands please visit [35].

In this project we used this tool in the IntentGun module. We pass the intent message to this tool as parameter and it starts the corresponding Activity or Service or Broadcast that intent message. Some common used commands are:

```
# For Activity
$ am start -n Package/Activity -a action -c category data

# For receivers
$ am broadcast -n Package/Receiver -a action -c category

# For services
$ am startservice -n Package/Activity -a action -c category data
```

Appendix B

Compiling C/C++ code for android

Below is an example on how to compile a C program with PIE support using Android Native Development Kit (NDK).

Compiling C code for Android 4.1

1. Download and extract NDK (My NDK directory is /android-ndk-r16)
2. Create a new standalone toolchain using the script /android-ndk-r16/build/tools/make_standalone_toolchain.py

```
python ~/android-ndk-r16/build/tools/make_standalone_toolchain.py --arch arm --  
↪ api 16 --install-dir ~/android-ndk-r16/toolchains/arm-16
```

3. Write your c program and compile it

```
1 cd ~/android-ndk-r16/toolchains/arm-16/bin  
2 touch main.c  
3 cat > main.c  
4 #include<stdio.h>  
5 int main()  
6 {  
7     printf("Hello world!\n");  
8     return 0;  
9 }  
10 ^C  
11 # Compiling  
12 arm-linux-androideabi-gcc main.c -o main.o
```

4. Start emulator, copy file to it, make it executable and run it

```
1 emulator @aosx_1 -writable-system # Recently the emulator is not starting  
↪ without writable, check if yours do  
2 adb push main.o /data/local  
3 adb shell chmod 06755 /data/local/main.o  
4 adb shell /data/local/main.o  
5  
6 Hello world!
```

Compiling C code for Android 5.1+

1. Download and extract NDK (My NDK directory is /android-ndk-r16)
2. Create a new standalone toolchain using the script /android-ndk-r16/build/tools/make_standalone_toolchain.py

```
python ~/android-ndk-r16/build/tools/make_standalone_toolchain.py --arch x86 --  
↪ api 22 --install-dir ~/android-ndk-r16/toolchains/x86-22
```

3. Write your c program, compile it and link it separately with required flags for PIE.

```

1 cd ~/android-ndk-r16/toolchains/x86-22/bin
2 touch main.c
3 cat > main.c
4
5 #include<stdio.h>
6 int main()
7 {
8     printf("Hello world!\n");
9     return 0;
10 }
11 ^C
12 # compiling and linking
13 ./i686-linux-android-gcc -c main.c -fPIE
14 ./i686-linux-android-gcc -lm main.o -pie

```

4. Start emulator, copy file to it, make it executable and run it

```

15 emulator @Nexus_One_API_22_x86 -writable-system
16 adb push a.out /data/local
17 adb shell chmod 06755 /data/local/a.o
18 adb shell /data/local/a.o
19 WARNING: linker: /data/local/a.out: unused DT entry: type 0x6ffffffe arg 0x32c
20 WARNING: linker: /data/local/a.out: unused DT entry: type 0x6fffffff arg 0x1
21 Hello world!

```


Appendix C

Malwares that uses Anti-Emulator detection

The purpose of this list is to help other researchers who are looking for samples that uses anit-emulation detection.

- 0ef01f50b829355b73418adcb30b20a3a1f6cb95123afce2c5469cdb25c7f7d0 Skinner
- 9AB5A05BC3C8F1931A3A49278E18D2116F529704 (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- 2E47C816A517548A0FBF809324D63868708D00D0 (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- DE64139E6E91AC0DDE755D2EF49D60251984652F (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- 6AB844C8FD654AAEC29DAC095214F4430012EE0E (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- C8DD6815F30367695938A7613C11E029055279A2 (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- 47442BFDFBC0FB350B8B30271C310FE44FFB119A (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- 604E6DCDF1FA1F7B5A85892AC3761BED81405BF6 (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]
- 532079B31E3ACEF2D71C75B31D77480304B2F7B9 (Android/TrojanDropper.Agent.BKY) Uses delayed execution [51]