Automatic Android Malware Analysis

Contents

1 Introduction

Write introductory paragraph here

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, 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 [1]. 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 App components. Each component is an entry point to the application. 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 app components:

- **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 [1].
- **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[1].
- **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 [1].
- **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 the 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 [1].

1.1.2 Intents messages

Three of the four component typesactivities, services, and broadcast receiversare activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime. You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another [1]. 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 [2].

Add ref to section describing the use of am and broadcast to fire activities and intents

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"[3].

1.2.1 APK file contents

Normally an apk file contains following files or folders:

Add captions and made the picture available in list of figures

- **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[4]. 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. [5]

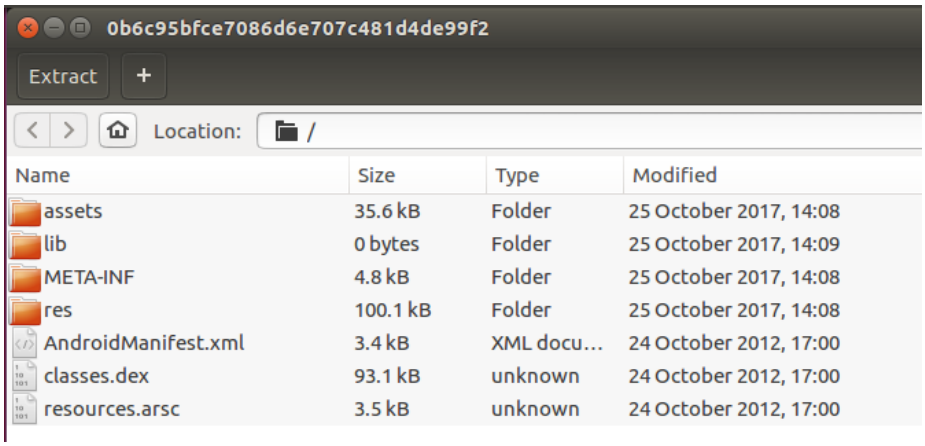

Figure 1: Files inside an APK

- **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 [3]. 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[6]. This rises several interesting scenarios in which malware authors can avoid detection, can redistributing benign applications with malicious injections or completely modifying behavior of an application. Readers interested in this topic are encouraged to have a look at [6]. Android NDK can be used to compile native code for android.

  compile hello world in c for android in apendix

- **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 resource which are not compiled into "re-

sources.arsc" (see below) [3]. 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 insude an apk. It contains classes compiled in the DEX file format which can be understood by the Dalvik/ART virtual machine [3]. In the next section we will describe this file 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 [3]. 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 dexer tool. This code is then executed on Dalvik Virtual Machine (deprecated) or in case of Android Runtime (ART), this code is compiled at install time to the native code.

1.3.1 Dex file format

In this section we will briefly discuss the file format for dex files. For more in depth and up to date specifications readers are encouraged to have a look at android official documentation on dex format [7]. A more graphical representation of dex file is shown in Figure 2. In 2 we had shown that how one element of proto_ids points to different locations in a dex file with the help of lines.

structure of Dex file

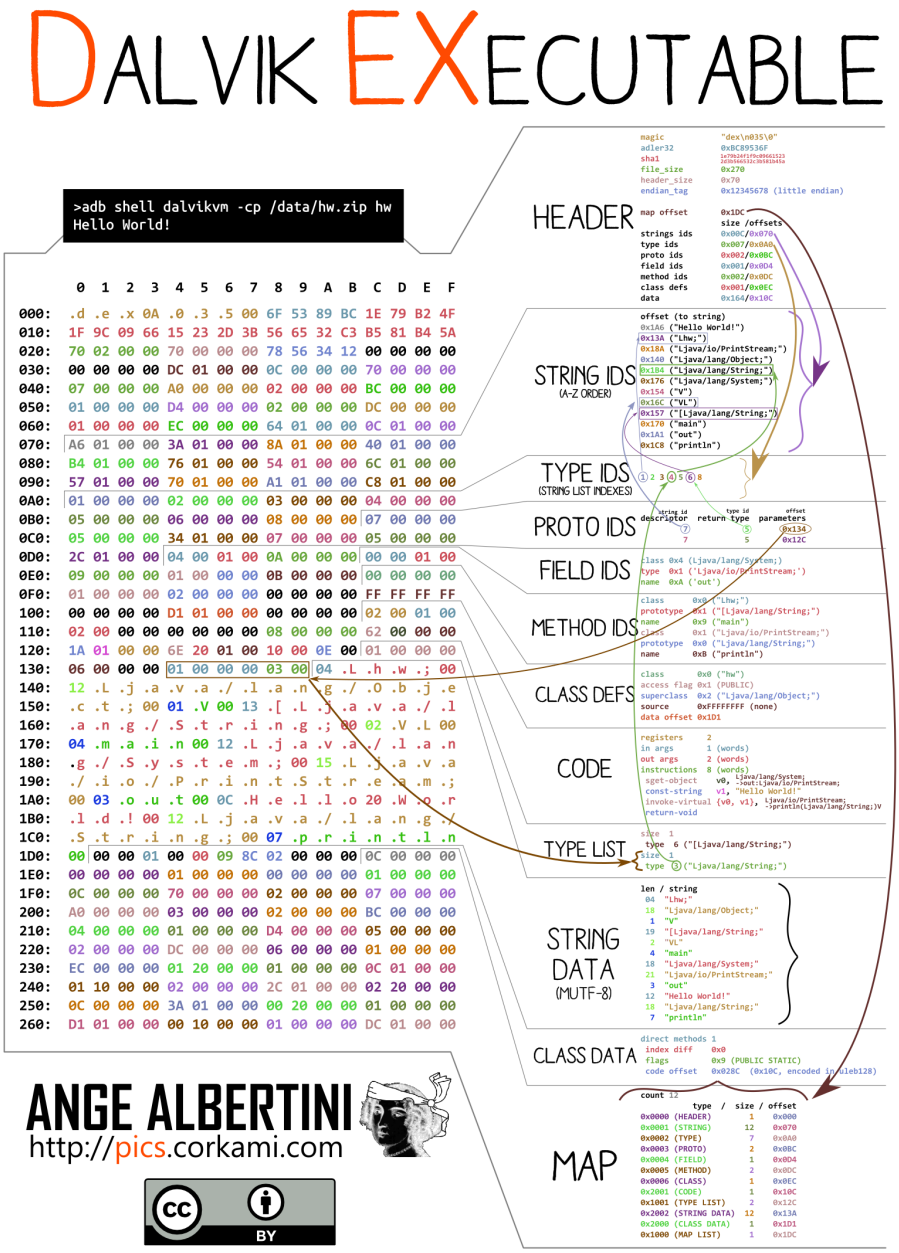

Figure 2: Dex file format [8]

change table to multipage table

Add information about Janus vulnerability to make the reading interesting

Name	Format	Description
header	header_item	The header contain information about how the dex file is organized, sizes of different sections inside the dex file, size of dex file, size of data section, version of dex format etc.
string_ids	list of string_id_items	Its a list of string identifiers. These are identifiers for all the strings used by this file e.g, class names, method names, constant objects. 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	Its a method prototype identifier list. Each item of this list contain three elements: • **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.

1.3.2 Multiple Dex files in single APK

Android app (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'

Cite https://developer.android.com/studio/build/multidex.html

. [9]

Versions of the platform prior to Android 5.0 (API level 21) use the Dalvik runtime for executing app code. By default, Dalvik limits apps to a single classes.dex bytecode 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

Cite https://developer.android.com/studio/build/multidex.html

. Because of this support its not uncommon these days to come across APKs that contain multiple dex files e.g, Facebook, instagram etc. [9]

1.4 Android Application Analysis

TODO: Write introduction section after the significant part of report is done and the structure is more clear

To be done later, In this chapter we include the problem statement, See fh kiel project report structure for missing parts.

2 Static Analysis

There are several static analysis tools available for APKs, each one having its own strengths and weaknesses.

Add info from http://orbilu.uni.lu/bitstream/10993/26879/1/tr_slr_article.pdf

Add some info about common tools

Add info about similarity search to identify malwares or relation between them

2.1 Smali/Backsmali

2.2 IDA pro

2.3 JADX - Dex to Java decompiler

2.4 Apktool

APKTool is one of the major reverse engineering tools for android applications.

Add more info

2.5 Androguard

Introduce androguard

MalloDroid, extension of androguard https://www.dfn-cert.de/dokumente/workshop/2013/FolienSmith.pdf

Androguard used in http://lilicoding.github.io/SA3Repo/papers/2013_guo2013characterizing.pdf

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 and one can make desired improvements.

Androguard doesn't have a lot of documentation available online and most of the time one has to figure it out from source code of androguard. For easier understanding and use, we can generalize the classes androgaurd contain into two groups as shown in table 1.

2.5.1 Androguard usage example

Add androguard demos

Fix the position of table

Classes for Parsing	Classes for Analysis
• **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.	• **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 1: Some classes of androguard and their description

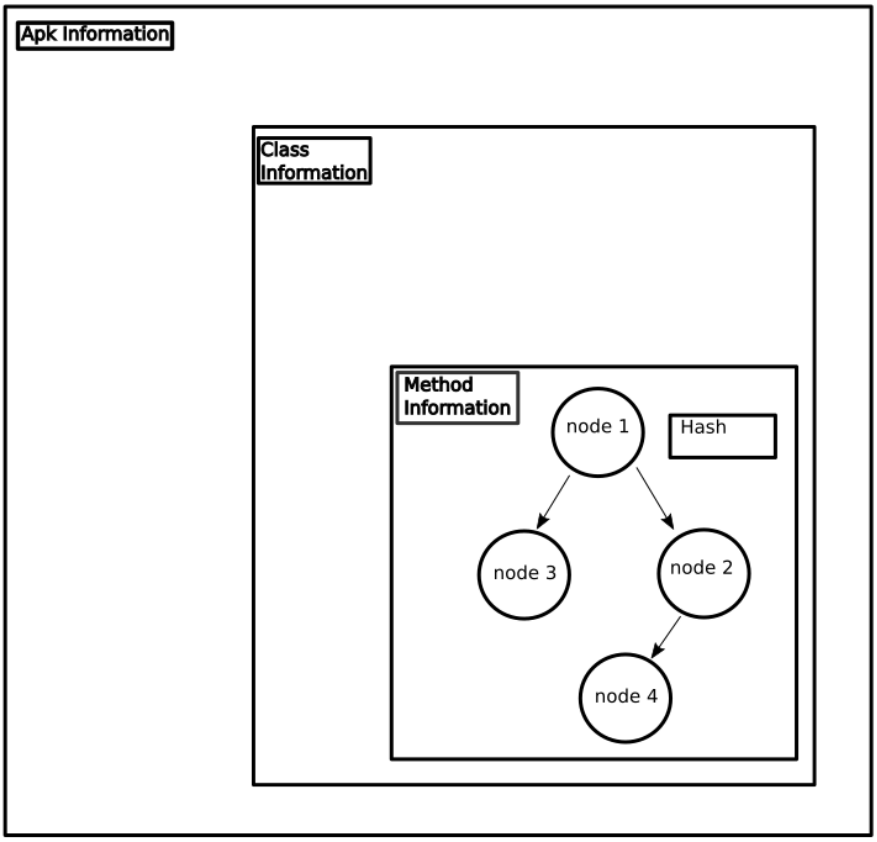

Figure 3: Groups of information extracted using APK

2.5.2 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 other malware or goodware samples. The more information we have about an APK, more relations we can figure out.

Now coming towards, the extracted information, we divide it into different groups as shown in figure 3.

- **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.

- **Class information** It represent 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.
- **Method information** A 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, shorty descriptor, Java decompiled source code for that method, Control flow graph of that method and the calculated hash for this method.
- **Control Flow Graph** Control flow graph contain edges and nodes. Nodes are basic blocks and are basically a list of instructions.
- **Hash** Using the control flow graph a canonical hash for is computed for each method. Before computing this hash, the smali instructions are normalized according to a specific criteria so that compilation specific changes are ignored like offsets etc,

  Ask lukas about adding information about the hash

  How much details about the normalization

2.5.3 Example usage

Just to make the usage of this extracted data more clear. In this example we process some of the sonicspy samples and tried to figure out how much code the share. We analyzed 16 samples

Add hashes, probably redo the whole analysis

and the result is shown in figure 4.

Line 23 in figure prints the result, in this dictionary keys represent frequency or number of times a method is been reused. Value represent numbers i.e, number of methods reused a specific number of times. From this analysis we can see that a large portion of code is common between these samples but a large part of this code is not malicious. Most of it 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 chunk. It can be topic for further research to identify common non-malicious pieces of code to make analysis easier.

In the code snippet, "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.

```
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 4: Reused methods in sonicspy variants

2.6 Dex2Jar and jd-gui

TODO: Do androguard basic usage examples

Discuss the changes we made including normalization, canonical hasing for similarity search

Discuss the info we are extracting from apks for platform

TODO: Do androguard comparison apks to see how many functions has added and how many removed, make a table out of it

TODO: Find reused code section in sonicspy or bankbots or lokibot

Usage of androguard for extracting features for AI/ML, prepare for talk in AIOLI-FFM group

Ask lukas for some results from platform

Improvements in androguard

3 Dynamic Analysis: CuckooDroid based on Cuckoo sandbox

CuckooDroid is an extension of Cuckoo Sandbox the Open Source software for automating analysis of suspicious files. CuckooDroid brigs to cuckoo the capabilities of execution and analysis of android application [10]. For more information about the cuckoo sandbox, readers are encouraged to visit the cuckoo sandbox website [11].

CuckooDroid can be downloaded from the CuckooDroid github repository [12] by following the guidelines provided there. For more step-by-step installation guide, readers are encouraged to have a look at CuckooDroid documentation [10]. 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. 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". Below is an excerpt from the CuckooDroid documentation [10]:

"This documentation refers to Host as the underlying operating systems on which you are running Cuckoo (generally being a GNU/Linux distribution) and to Guest as the Windows virtual machine used to run the isolated analysis."

We will be configuring CuckooDroid with Android Emulator (Goldfish) and figure 5 shows the architecture CuckooDroid with Android Emulator. As it can be seen in the figure 5 that there are two main parts, Cuckoo Sandbox and Android Emulator.

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 5

- **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

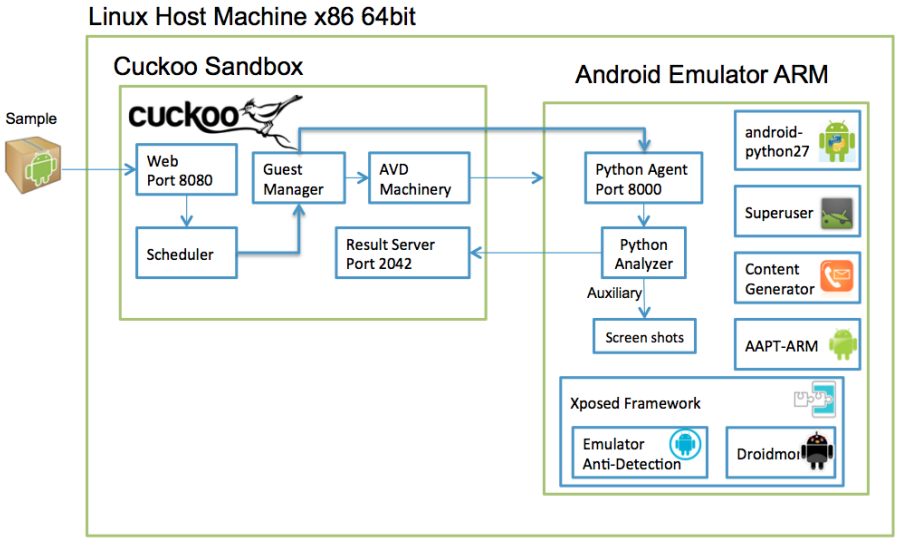

Figure 5: CuckooDroid architecture with AVD

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 know Emulator Anti-detection techniques (For more details on this topic see Chapter 4)

3.2 CuckooDroid required patching

Fixing cuckoodroid

3.3 Android emulator and its rooting

Persistent root problem

3.4 Upgrading to higher versions of Android

Latest android

3.4.1 NDK hello world, python termux

python compilation workaround, termux

3.5 Future work

Slow android emulator

Emulator anti detection

4 Dynamic Analysis: Anti-Emulator Detection

Common methods employed for emulator detection, some literature

Good and bad uses of anti-emulator detection

Testing results of cuckoodroid against common emulator detection methods

Adding some new anti-emulator detection features to cuckoodroid

result of analysis before and after

References

[1] "Android application fundamentals."

[2] "Intents."

[3] "Reduce the apk size."

[4] "Using android assets."

[5] 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.

[6] 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.

[7] "Dalvik executable format."

[8] "Dalvik executable picture by ange albertini."

[9] "Enable multidex for apps with over 64k methods."

[10] "Cuckoo-droid documentation."

[11] "Cuckoo website."

[12] "Cuckoo-droid github."