Chapter Two

Background on Android Security.

2.1 Importance of Mobile Security Domain Knowledge

Incorporate prior domain knowledge facilitates the process of learning. Nevertheless, the majority of machine learning techniques ignore the domain knowledge In Artificial Intelligence applications, specifically with Machine-Learning, a comprehensive understanding of the basics of the problem's domain knowledge is crucial. For example, if a data analyst/scientist is intending to analyze a financial dataset for prediction purposes, the domain knowledge of this particular situation that they should refer to is economic or finance fields. [4]

Awareness of the domain knowledge is incredibly valuable. Nevertheless, gathering this knowledge can be expensive process, requiring an investment of time, money and involving with researchers or practitioners of the other fields. Without the domain knowledge understanding, an ML analyst who's trying to solve an Android detection problem will find it difficult to identify which features they should focus their analysis on, which dimensions they need to investigate more, or what architecture would suit which situation.

In many Machine Learning literature that tackles Android malware problems, the analysis steps are all expressed with regard to Android malware terminologies and understanding, whether it was data preprocessing/normalization stages, model building, or results interpreting. Thus, the purpose of this chapter is to present the basic understandings of the Android background required to better fathom in the third chapter of why certain Machine Learning approaches were being selected over others.

This understanding will also help the machine-learning expert to determine which Android features help in identifying malware attacks or infections more that any others, and which contribute the least toward the security problem and don't have much discrimination power or usefulness, so that the investigating becomes more focused. Feature-engineering for example will be much challenging later when ML is applied if the domain knowledge of Android malware was not well known.

It is not expected to master the mobile security domain in a detailing manner, but a basic understanding of how features were created, gathered, pre-processed is an important cornerstone. There is still a debate among the data science communities of whether or not a domain-knowledge complete understanding is actually as important as mastering the technical skills of machine learning and the analysis process, or whether they complement each other. A combination of both (domain knowledge + data science skills) is a useful combination and may possibly help the analyst/researcher into approaching the problem differently, or observing an additional dimension or angle to the problem.

Not understanding mobile security at all may result in a garbage-in-garbage-out model that are even more difficult to interpret and justify. Among the many advantages that awareness of domain-knowledge in Android malware would bring: The ability to ask good questions, being derived by intuition in the analytics, and thus, building efficient and effective models, and choosing the right architecture that reflects your prior believes gained from domain knowledge.

Moreover and generally speaking, beside the curse of high dimensionality, comes the problem with bias. Bias in the training data produces incorrect results.

Missing the knowledge about Android malware as the domain-knowledge in our case may possibly result in feeding the algorithm with biased data that lead to false-positive or false-negative identifications.

It is essential to know the internal structure of an Android application, its components and lifecycle, how it processes instructions, how to decompile the application for the examination of vulnerabilities into the application code. The purpose of the following sections is to demonstrate the fundamentals and the minimum knowledge required so as to understand further dynamic and static analysis and how they relate to machine learning methods when applied for Android detection.

The second important prerequisite that follows the Android domain knowledge understanding is identifying what difference does intelligence add; by comparing traditional state-of-the-art Android malware detection with the ML-based detection techniques. A comparison of this kind gives insights into the advantages and limitations of both approaches, which will be discussed more in the third chapter.

2.2 Defining Malware Analysis

Malware analysis or malware reverse engineering is the process of obtaining the source code of a malware in order to understand how it works and the sequence of actions that are executed for the malicious activity to take place.

Malware analysis of android applications can be of a static or dynamic nature. The main difference between the two is that static works on the features of the app that can be viewed, extracted, and examined offline without the need to run the app, while dynamic tries to analyze the app's source code as it is running in a controlled environment. Dynamic analysis is a set of methods that studies the behavior of the malware in execution through gesture simulations. In this technique, the processes in execution, the user interface, the network connections and sockets opening are analyzed.[5]

Both approaches have their advantages and disadvantages, and there is no one that is considered better than the other. Since that every analysis methodology yields different information and final results, it depends on which feature of the Android app the analyst is emphasizing their work on, and the circumstances of the analysis experiment itself.

For instance, in some circumstances where the source code is obfuscated by some techniques, a static analysis will not be of a great help, and thus the dynamic analysis that completely ignores the source code is the one to be chosen here. In contrast, when the significant overhead of the dynamic analysis is not the right option for the resources and capacity available at hand, a static analysis in the method that is followed.

Example of when both methods (dynamic and static) were combined is when researchers attempts to teach a model to detect the Android malware basing on the Intent mechanism. Intents are explained in Android anatomy section. [6][7]

Dynamic	**Static**
extracted features based on code execution[8]	features extracted from the underlying source code, but without executing it.[8]
More informative, particularly with highly obfuscated code. [8]	more efficient[8]
Deals with features involving dynamic code loading and system calls that are collected while the application is running.[8] Monitoring the execution of android malware activity at runtime. The information collected for the dynamic analysis is from the operating system during runtime such as : 1. System calls. 2. Network access. 3. Files and memory modifications. [10]	Rely on features extracted from the manifest file or the Java bytecode.[8] Examples: 1. Permissions requested by an application. 2. API usage: which specific functions were being used. [9] 3. Function calls.
ADVANTAGES: The main advantage of this technique is it detects dynamic code loading and records the application behavior during runtime. [8]	ADVANTAGES: Fast detection speed. Can quickly detect malicious applications and prevent malware application before it was installed. [10]
DISADVANTAGES: 1. There are chances that the applications can fail to execute the malicious code while recording the features. [8] 2. This technique is hard to implement as compared to static analysis, due to the computational overhead of executing the application.[8] 3. Dynamic analysis can not be entirely secure! [12] 4. The efficiency of this kind of approaches depends on code coverage during automatic	DISADVANTAGES: 1. Prone to obfuscating or encryption techniques.[9][10] 2. Several of the proposed static mechanisms are easy to circumvent, for example using kernel-based exploits or API-level rewriting.[7] 3. Instructions may be reordered, branches may be inverted or the allocation of registers may change. [9]

execution. Moreover, some works (e.g., TaintDroid) need to modify Android OS to implement on smartphone, which is technically not feasible.[14] 5. There already exist some techniques to avoid the processes performed by dynamic analysis, where the malware has the capacity to detect sandbox-like environments and to stop its malicious behavior.	4. With only static analysis, some behaviors are difficult to conceal and often indicative for malicious activity [11] 5. Can only detect known feature codes and need to constantly update the existing signature database. [13]

Table 2.1 Dynamic Vs Static analysis of Android applications

2.3 Steps of Malware Analysis of Android Applications

2.3.1 Obtaining the Malware Sample

Mobile reverse engineering is started by obtaining the APK version of the Android mobile app that is meant to be analyzed. For educational purposes, some online repositories offer real malware samples for downloading so that analysts can experiment with. Examples of online repositories that offer real Android malware samples: PGuard, Contagio Mobile Malware Mini Dump[15], Virus Share[16], Virus Total Malware Intelligence Services[17], ADMIRE[18] (Android Marketplaces & Developers Intelligence and Reputation Engine), Android Malware Github Repo. [19]

The obtained Android malware samples will be useful for two tasks: First in analysis, if the analyst is intending to construct a dataset from the scratch instead of working on an already made pre-processed and normalized dataset. And second, in the late stages after building the ML model, this model need to be tested and evaluated against real malware samples to see if the learning model was able to successfully detect them or not.

In other situations where analyst don’t want to build the dataset themselves, but prefer to work on a dataset that has been prepared, cleaned and normalized by other researchers, there are also some online repositories that offer dataset with the malware features and their values.

Examples of Android malware behavioral datasets: DREBIN[20], which includes 179 different malware families in its dataset.

2.3.2 Choosing the Analysis Tools

After the malware android app samples are obtained, analyzing their behavior can be conducted with tools that are designed for this specific purpose. These kinds of tools are dedicated to mobile forensics, analysis and security, and they facilitate the full access to the Android application at hand. As mentioned earlier, tools can do static or dynamic analysis, and each tool has a distinguished role to play.

Santoku for instance, a Linux distribution, is an example of where many useful analysis tools are all gathered at one place. It was specifically designed for Android analysis and is mostly benefited in the context of malware. The Santoku Linux distribution offers a collection of Android tools for reverse engineering, penetration testing, forensics acquisition and mobile application assessment. The specialization of this distribution is an advantage. [21]

The process of malware analysis requires more than a single tool. Security analysis tools included in Santuko evolve with every release, and using environment as Santoku saves times that would be needed to search for different tools from different resources. The word Santoku itself loosely translates as ‘three virtues’ or ‘three uses’. The three uses that support users are: 1-mobile forensics 2-mobile malware and 3- mobile security.

Nevertheless, using Santuko is not the one and only available alternative. The tools that Santuko made available together can be used separately as stand-alone tools. More links are available in the index.

It depends on which task is required from the analyst to achieve, and which answers they are looking for throughout the whole malware analysis or reverse engineering. For instance, the mobile forensics tools are more concerned with acquiring and analyzing data, while mobile security testing tools support security assessment of mobile applications, and the mobile malware analysis tools are more useful with examination of malwares.

Market crawling tools are also available, that download the apps along with details and metadata from official Google play store and other third-party Android markets. Examples of

Market crawlers: Google play crawler, Aptoide downloader [22], ADMIRE (Android Marketplaces & Developers Intelligence and Reputation Engine).

2.4 Anatomy of an Android Application

2.4.1 First, the Files

An android application is simply a zipped file with an extension that is changed to apk. It contains different files with various contents and purposes. The result of decompiling the APK file contains the following three main components:

1- The application resources, which are the files that the application might need to work. Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more.

2- The AndroidManifest is an XML file. It provides other essential information about the mobile application to the Android system, which the system needs before running the application's code, for example: permissions being used, activities, services, broadcast receivers, content providers, libraries the application must be linked against, minimum level of Android API that application requires, and more.
There exist some command-based tools that help in extracting the permissions from the Manifest file, and other tools that enable reading this file in a more elegant and human-friendly way.[23] Example of such tools: AXMLPrinter[24], Drozer[25].

3- Classes.dex files, where all the binary codes of the application reside, and it is where the java virtual machine will execute.
Java source code is compiled by the Java compiler into .class files. Then the dex tool, which is a part of the Android SDK, processes all the .class files into a single file format of DEX. DEX contains Dalvik bytecode
These type of files are understandable by the Dalvik virtual machine. They are generated from the java code that the developer wrote.
.dex are similar to jave classes file but they run under DVM. There are also some reverse engineering techniques to make a jar file or java class file from a .dex file.

Instead of disassembling, a decompiling can be done. The act of decompiling is dex>Jar>Java. But when dex is transformed to Java, some of the metadata information are lost.

So it is more advisable to use decompilers which has been written with Android in mind, and some can go from dex to java directly without going through the jar process.

2.4.2 Second, Android Features and Sources of these Features

Extracting the features' data and constructing the dataset is not a straightforward and easy task itself. Some specific tools and frameworks are specialized for this purpose and they help in getting the required information in a more human-readable formats from the Android data sources.

Two most important Android application sources for data extraction are AndroidManifest.xml file and the Android app source code. Each source provides different features and dimensions to the Android malware application that should be analyzed.

Some features can be extracted as well by running the application in a controlled environment, which would not have been available by only analyzing the source code and the manifestxml file. Since dynamic analysis is out of the scope of this research, we are going to tap briefly but not with details into the dynamic analysis features.

Methods for extracting data rely on the app source code, others on the Manifest.XML file, and there were researching attempts of adopting both source code and Manifest.XML file as the source for constructing the dataset they start their work with.

The following section explains in details the two main sources of data: the manifest file and the source code, and which features can be extracted from each source.

1- AndroidManifest.xml.
   The AndroidManifest file provides data supporting the installation and later execution of the app. The features that an analyst can get from the AndroidManifest file are many, including:

   a. The requested hardware components as camera, GPS module, touch-screen. For the use of certain combinations of hardware often reflects a harmful behavior, e.g.: GPS and Network together to send user location to the attacker.
   b. Requested permissions,

The Android permission system is one of the security permissions that Android platform provides to harden the installation of malware. Permissions are granted at the time of installing of the app, and they allow the application to perform tasks on the mobile phone as sending SMS, accessing resources, and more. Two main points in regard of Android and security is that first, users tend be blindly grant these requested permissions, and second, some malware Android app tends to use specific permissions or combinations of permissions more often to perform their malicious activities.

c. Application components.

There exist four different types of components defined in an Android app: activity, service, content provider, and broadcast receiver. Each app can declare several components in the manifest file as required. Some malware families may share the same name of components. For example, several variants of the DroidKungFu malware use the same name for particular services.

Filtered Intents. Intent is a run-time binding mechanism, which can connect two different components in the process of running program. Through the Intent, the program can express a request or a wish to Android. Android then will select the appropriate component to complete the request according to the content of wish. Activity, Service and Broadcast Receivers are activated through the Intent mechanism. To transfer the data between different Activities, it is necessary to include the corresponding content in the Intent.

Intents perform many tasks, as allowing the inter-process and intra-process communication to be performed, and allowing information about events to be shared between different components and applications. A malware would often listen to these intents, so one of the analyst tasks should involve inspecting which intents does the app in question is listening to.

The Intent mechanism is a very important technology in Android application development, so this research [26] went into studying its in-depth.

2- The application source code.

An Android app as explained in the previous chapter is developed in Java and then compiled into bytecodes. These bytecodes can be efficiently disassembled and provide information about the API calls and data used in the application. The features that can be extracted from the source code, or dex file are many, including:

a. Restricted API calls, by searching for the occurrence of calls in the disassembled code (dex) to gain a deeper understanding of the functionality of the app. A malicious behavior may comprise the use of restricted API calls for which the required permissions have not been requested. This may indicate that the malware is using root exploits in order to surpass the limitations imposed by Android platform.
b. Used permissions: A common practice is to match API calls with requested permissions that were actually used.
c. Suspicious API calls: certain API calls are frequently found in malware samples; and they allow access to sensitive data/resources. Those suspicious API calls have many usages as: collecting sensitive data, communicating over the network, sending and receiving SMS messages, and obfuscation. It worth paying attention to API calls for further investigation.
d. Network addresses: Malware applications regularly establish network connections for number of purposes as retrieving commands and exfilterating data collected from the device. Thus, all IPs, URLs and hostnames that are found in the dex file of the app are collected by data analysts as essential features. Some of these addresses might be involved in botnets and thus present in several malware samples, which can help to improve the learning of detection patterns.

A further analysis is always required, whether static or dynamic, in order to confirm if malicious behaviors are carried out or not. A combination of both can be adopted as well. [27]

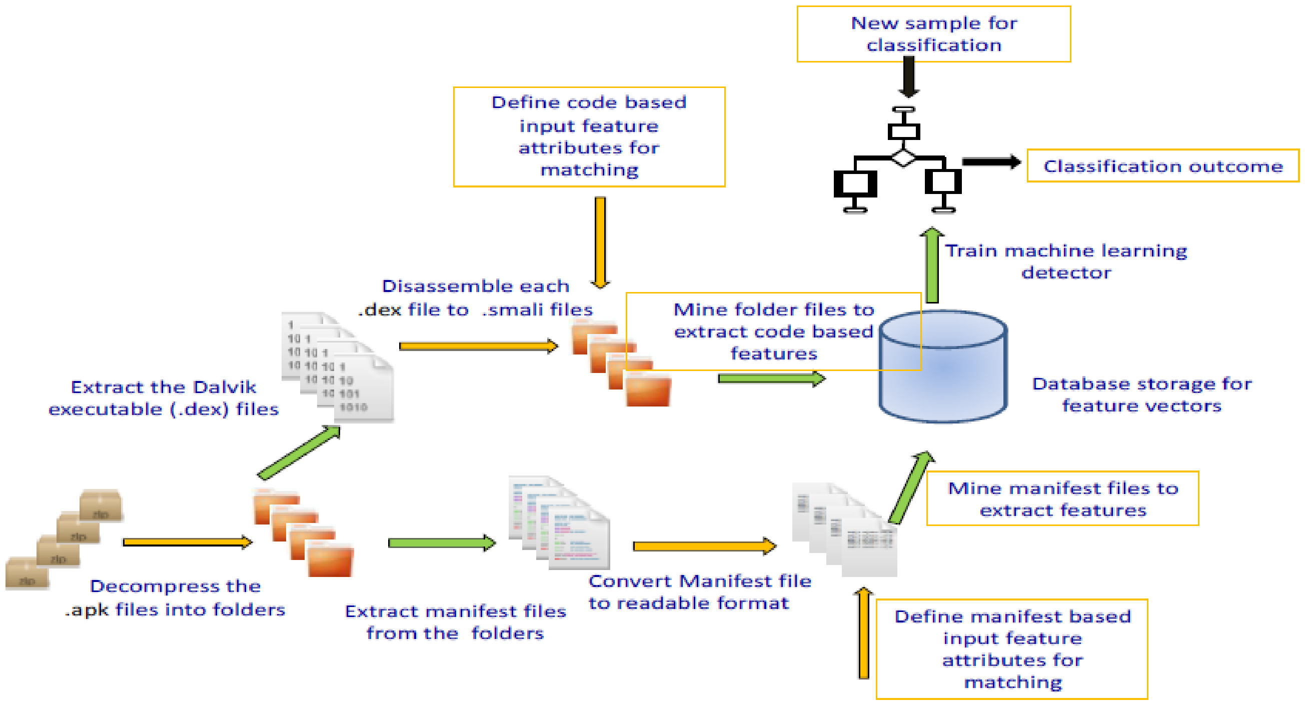
New sample for classification
Classification outcome
Define code based input feature attributes for matching
Train machine learning detector
Disassemble each .dex file to .smali files
Mine folder files to extract code based features
Extract the Dalvik executable (.dex) files
Database storage for feature vectors
Mine manifest files to extract features
Decompress the .apk files into folders
Extract manifest files from the folders
Convert Manifest file to readable format
Define manifest based input feature attributes for matching

Figure 2.1: The process of Android data analysis

2.5 Android Obfuscation Techniques

Since the scope of the research focuses more on the static analysis more than the dynamic, it is important to mention the obfuscation techniques that are conducted by hackers to slow or prevent the static analysis or reverse engineering.

Obfuscation techniques that can obstruct the work of static analysis of mobile applications are many. On some publications, researchers tend to avoid the headache of obfuscation by adopting a dynamic analysis, while others find some workaround that are interestingly machine-learning based to enable them to do static analysis for android malware regardless of it being obfuscated. Obfuscation techniques come in different types and forms, to mention some few:

1- Junk code insertion.
2- Instruction replacement.
3- polymorphism
4- binary packers
5- encryption
6- self-modifying code

7- Fingerprinting + dynamic code loading. [28]
   (More techniques may exist as well.)
8- Binary packers. [from the first dynamic analysis paper.]

Nevertheless, if the feature extraction from dex executable (the source code) failed for example, the permissions would still be obtainable from the Manifest file. Furthermore, if for instance, malware incorporates encryption or any other obfuscation techniques from the ones mentioned above to prevent detection of commands, API call features (including the crypto API) will still be detectable along with permissions to expose the app's malicious intent. [29]

Generally, detection approaches of Android malware are classified as either misuse or anomaly methods. Misuse detection is a signature-based, while an anomaly detection is more of a behavior-based.

Both static and dynamic analysis methods can adopt misuse and/or anomaly. For example, we may find an anomaly detection of Android malware that uses a dynamic analysis. And there exist as well anomaly detection that relies on static analysis. Same thing goes with misuse detection approach. [29][8]