2016

Applying Deep Learning Techniques to the Analysis of Android APKs

Robin Andrew Nix
Louisiana State University and Agricultural and Mechanical College

APPLYING DEEP LEARNING TECHNIQUES TO THE ANALYSIS OF ANDROID APKS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Computer Science

in

The Department of Computer Science

by
Robin Nix
B.S., Franklin W. Olin College of Engineering, 2007
May 2017

Table of Contents

List of Tables

List of Figures

Abstract

Malware targeting mobile devices is a pervasive problem in modern life and as such tools to detect and classify malware are of great value. This paper seeks to demonstrate the effectiveness of Deep Learning Techniques, specifically Convolutional Neural Networks, in detecting and classifying malware targeting the Android operating system. Unlike many current detection techniques, which require the use of relatively rigid features to aid in detection, deep neural networks are capable of automatically learning flexible features which may be more resilient to obfuscation. We present a parsing for extracting sequences of API calls which can be used to describe a hypothetical execution of a given application. We then show how to use this sequence of API calls to successfully classify Android malware using a Convolutional Neural Network.

Chapter 1 Introduction

It is a given at this point that mobile devices are an essential part of most individuals' everyday lives. Furthermore, the vast majority of mobile devices today run Android with sales of Android devices averaging 80% of total market share over the past three years [39]. The continuous expansion of the Android operating system to an increasing number of devices and users worldwide has come with a corresponding uptick in malware targeted at Android devices. Since the first Android malware application was detected in 2009, there has been an alarming increase in the number of malware applications targeting Android devices [38]. Traditional methods for detecting malware rely heavily on hand-engineered features that can be used to detect different families of malware. This approach has two major disadvantages: first, malware must be detected manually and second, features must be handcrafted to identify future variations of that malware family. By necessity, the approach leaves defensive efforts a step behind malicious, offensive efforts to target users since it is reactive in nature. The handcrafted features that are developed are typically very rigid and do not generalize well, which means that often simple modifications can be made to a known malicious package to avoid detection by current algorithms [16]. Further complicating matters, in many cases these modifications can be automated [17,18].

The goal of this thesis research is to demonstrate the effectiveness of detecting malware in Android applications using Deep Neural Networks (DNNs). Deep Neural Networks have the potential to offer several advantages over the current approaches to malware detection. First, DNNs are capable of learning features in an automated manner without the need for direct human intervention in the crafting of features [41]. Second, DNNs tend to exhibit a greater flexibility in dealing with new and novel data after they are trained when compared to systems

that use hand-engineered features [40]. The success of DNNs in image processing and Natural Language Processing (NLP) in particular provide a compelling case for their potential success in malware detection [42].

As a corollary objective, we wish to develop this approach to be as automated and scalable as possible. As of June 2016 there are over 2.2 million distinct applications in the GooglePlay store alone [19]. Given the sheer number of applications and the ease of creating and publishing new applications, it is simply not possible to apply human effort to reviewing even a fraction of the available applications. Additionally, we do not wish to rely on the need for any handcrafted features in the development of the classifier. The more generic the developed detection model can be the more likely it is that the approach will continue to be relevant as new and previously unknown malware variants using different exploits are introduced. This future adaptability will be a key factor influencing design choices whenever possible.

We will first need to develop and demonstrate a suitable method for ingesting Android APK files and converting them into a format that can readily be learned by a deep neural network. The critical aspect here will be determining exactly what features will allow a deep learning technique to succeed while also meeting the design goal of being sufficiently generic to address future malware techniques. Next we will examine the different approaches to using deep neural networks to classify the applications based on the behaviors found within the code. Finally, we will evaluate the model against datasets representative of the currently available benign and malicious applications to determine its predictive ability.

Chapter 2 Structure of Paper

The following is an overview of the structure of the remainder of the document.

Chapter 3: Android Analysis Literature Review

Chapter 4: Parsing APK Files to Create Feature Vectors

Chapter 5: Using Deep Neural Networks to Classify APK Files

Chapter 6: Future Research

Chapter 7: Conclusion

Chapter1: Overview of the Android Operating System

We begin with a brief overview of the Android Operating System (OS) and Android applications, also known as APK files. In particular we will focus on the portions of the Android OS that are most relevant to malware analysis.

Chapter 2: Review of current Android malware analysis techniques

Here we will review the current academic literature describing the state of malware analysis for Android applications. We will cover static and dynamic analysis techniques, particularly those that focus on analysis of Android APKs by monitoring system calls. We will also give a brief overview of some of the recently published works on malware obfuscation techniques.

Chapter 3: An Android parser for creating a feature representation of APK files

This chapter will discuss the design choices made in developing a parser to create a representation of Android applications that is suitable for ingestion by a neural network. The proposed implementation is built on top of the Androguard decompiler and analysis tools. It

seeks to represent each application as a set of strings listing all API calls of hypothetical program execution.

Chapter 4: Review of deep learning techniques

Here we give a brief overview of deep learning techniques, specifically focusing on the convolutional and recurrent neural networks. Given the limited amount of publication of DNNs in the field of malware detection to date, the majority of the applications discussed are for the image and text processing fields where DNNs have seen a large deal of success recently.

Chapter 5: Using Deep Neural Networks to classify Android Malware and Applications

We demonstrate the use of Convolutional Neural Networks to successfully classify Android APKs using the vectorized representation developed in Chapter 3. We compare the results of these approaches to other commonly used machine learning algorithms using the same dataset and find that the deep neural networks provide superior performance and flexibility.

Chapter 6: Future Research

Despite the success of the models presented, given the almost non-existent application of deep neural networks to malware detection in the published literature, there are a number of areas where the models presented could be improved. In this section, we give a high level overview of possible next steps that could further enhance the detection of malware using deep neural networks.

Chapter 3 Android Analysis Literature Review

3.1 Android Operating System

The Android operating system was first released in 2007 as an open source OS developed by Google and designed for mobile devices, specifically phones. Android is based on the Linux kernel, which handles connectivity to the hardware and basic OS functionality. The Android operating system is currently deployed on 1.4 billion devices worldwide and as such represents the most widely deployed operating system in history [43]. This section will give a brief overview of the Android OS with a focus on the components most relevant to malware analysis.

Each application on an Android device runs in its own sandbox and permissions must be explicitly granted by the user at installation. This structure limits the potential damage that can be caused by a malicious application by preventing unnecessary permissions. However, the combination of benign applications over-requesting permissions and malicious applications capable of privilege escalation can circumvent this mechanism by desensitizing users [].

While Android applications are written in Java code, they are compiled into .dex files, which are Dalvik executables containing Dalvik byte code that can run on the Dalvik Virtual Machine (DVM) or Android Runtime Environment (ART) that runs all Android applications. Prior to version 5.0, Android supported execution of Dalvik code using a just-in-time compiler on the DVM. However, starting with Android 5.0 all Android code is executed in ART, which does ahead-of-time compilation of all application code. This change was transparent to both the end user and code developers. In fact the same APK file will operate on both DVM and ART assuming all other system requirements are met [44].

3.2 Android Application Basics

Android applications, often referred to as apps, are the primary programs that can be installed on devices running the Android operating system. All of the necessary files for an Android application are bundled into APK files, which are then distributed. Each APK file contains the following items: res, assets, classes.dex, AndroidManifest.xml, lib, and META-INF. The most relevant components for this research are the classes.dex, and Androidmanifest.xml. Native libraries can also be included inside of APK files. The APK file also contains all other resources needed to run the application such as pictures, sound files, and any other external resources.

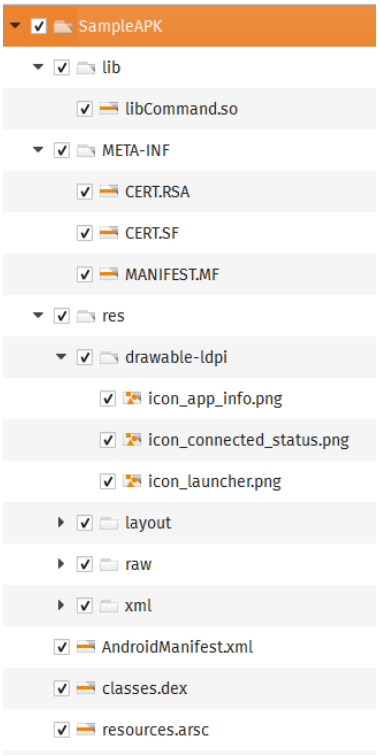

Figure 3-1. A sample APK file layout with the associated elements.

The classes.dex file contains the bytecode for the application and is the main source of executable code for an application. The dex file format is an open specification and can be found in its entirety on the Android Open Source Project website [37].

The permissions allowed to the DVM are set in the Android Manifest file and can be a key indicator of what the file might do. While all dex code is handled inside of an individual DVM instance, native code can be called from the DVM and run inside the normal Linux kernel. This can be done for either compatibility or performance reasons.

As previously mentioned, Android allows for the inclusion of native libraries in addition to the dex source code. Native libraries are stored as .so files in the APK file and present a serious challenge to static analysis because malware can use calls to these native libraries to execute the malicious payloads from static analyzers that only examine dex code. Malware can also contain native libraries to obscure taint tracking because they can pass information into and out of .so libraries to obfuscate that information from taint analysis and static analysis [1].

The final portion of the APK file is the assets and resources included, such as fonts, pictures, sounds, databases, and any other external resources needed for the application to run. These assets and resources have obvious uses for benign applications, but can also serve as a place to hide data for a malicious application. Applications can place code directly in these files or can obfuscate the code and then place them in the files. The code would then be called using DexClassLoader at runtime and could also be reassembled from its obfuscated state [2].

Given the large differences in each of these file types, vastly different design and pre-processing requirements would be necessary to examine each of them for malicious code. This research will not focus on malware that launches exploits through native libraries or places the

malicious code in another asset file. Only malicious code that is included within the classes.dex file will be considered specifically addressed.

In order to obtain source code for analysis, the classes.dex file can be decompiled using several different tools to create .smali versions of the files or java versions (baksmali, dex2jar, etc). Smali is a human readable version of the compiled dex opcode, which is often used for analysis. They contain a limited instruction set, which provides a useful building block towards a machine learning algorithm for analyzing Android applications for malware.

3.3 Background on Android Malware and Analysis

Malware analysis approaches have traditionally been divided into two major fields: static and dynamic analysis. In static analysis the source code is analyzed without executing the application to determine the effects of the application. Typically, in static analysis a decompiler of some form is used in order to convert the source file into human readable code, although some analyzers deal directly with the source file. Static malware analysis often focuses on dataflows within a program or specifically defined features to analyze applications. In dynamic analysis the program in question is executed, typically on a virtual machine or emulator, and its behavior is logged for further analysis. Dynamic analysis allows the investigator to provide different inputs to the program to see the reactions those inputs provoke in the system.

It is commonly accepted that neither static nor dynamic approaches to malware analysis is inherently superior to the other. Most malware analysts believe that the two techniques are complementary and use information gathered in one form of analysis to inform their investigation in the other.

The following is a review of the current literature on Android malware analysis. It is broken up into three major sections: static approaches, dynamic approaches, and obfuscation

techniques. While certainly not an exhaustive review of all published Android malware detection techniques, it gives a good representation of the current techniques employed in the field today.

3.3.1 Malware Analysis Approaches

It is well accepted in the malware analysis community that both static and dynamic analysis techniques have their benefits and neither is inherently superior to the other. In practice malware detection typically involves the combination of both dynamic and static techniques in tandem to completely analyze a program.

This combination of techniques is also the case for Android. Feizollah et al. analyzed 100 papers on Android malware analysis and provided statistics on where the different papers focused their analysis. They break the features down into four overarching categories: Static Features, Dynamic Features, Hybrid Features, and Application Metadata Features. As one might expect, approximately 45% focused on static features, 42% focused on dynamic features, 10% focused on hybrid features and just 3% focused on application metadata [5].

3.3.2 Static Analysis Approaches

Static analysis is the process of examining the code contained within a program either manually or with the help of automated tools to determine its function and purpose. Static analysis is used to improve both in the field of malware analysis to detect malicious applications and also for benign applications to help detect the presence of bugs or problems that may not be readily apparent through traditional program execution [20]. This review will focus entirely on the application of static analysis to malware detection and specifically on the tools that have been developed for APKs on the Android operating system.

3.3.2.1 Flow Analysis

In static analysis one of the most popular techniques for program analysis involves the use of data flow analysis to track how information moves through program code and when and how it might be modified. In particular, data flow analysis makes use of what is typically called a Control Flow Graph (CFG, sometimes called a Call Flow Graph), which is a graphical model of the paths that could be taken during code execution. Given the large amount of information included in any program, CFGs are typically abstracted to cover larger code blocks such as methods, functions, or classes, depending on the application. In a CFG each node in the graph represents a block of code while the edges show the connection between the code segments.

The process of analyzing Call Flow Graphs to determine properties of the underlying program was originally proposed by Frances Allen in his 1970 paper and has become a staple of static analysis since then [21]. The CFG can be used to analyze the flow of information within the program by tracking the flow of variables among the basic blocks. However, constructing and analyzing a complete call flow graph for an application is an undecidable problem and as such approximations are used when attempting analysis [22].

3.3.2.2 String Analysis

In string analysis, the decompiled malware is searched for plaintext strings and the strings are then compared against a database of strings known to be associated with malicious files. The type of strings typically used in this technique are malicious urls, IP addresses, package names, and other similar information. A major limitation of string analysis is that it is relatively trivial for malware to obscure plain text strings by building them dynamically during program execution and only storing encoded or partial versions of the strings in the source code.

3.3.2.3 Signature Analysis

In signature-based analysis, programs or subsections of programs are identified by signatures. The signatures are generated to describe those code sections. There are a variety of methods for generating signatures to describe malware. The simplest approach involves identifying code sections and then using hashes, such as md5, to produce a signature for that code. A slightly more complex approach is to use a known malicious sequence as the signature and compare other code sequences to it using some form of difference metric to compute the similarity. Finally, signatures can be based on heuristics that describe the behavior of the malware in a more general way, which can be distilled into a set of rules for the signature [23]. While signature-based approaches are typically very fast in practice, they can be very susceptible to intentional obfuscation techniques.

3.3.3 Static Analysis Tools for Android

The following is a short overview of some of the recently published static analysis tools available for analyzing Android APK files.

3.3.3.1 Signature-Based Analysis Tools

The following tools attempt to extract features either directly from the application source code or from a secondary representation of the application such as the call flow graph. These feature sets are then used to identify signatures, which can be used to identify and classify malware that exhibit similar features.

3.3.3.1.1 Androsimilar

The approach laid out in Androsimilar by Frauki et al. is unique in comparison to many of the other techniques presented so far in that it deals directly with the raw, unprocessed

bytecode to extract features. The features are generated and ranked based on the entropy values of the underlying byte streams. The entropy is calculated over an ASCII representation of 64 byte segments of the bytecodes in the file based on the occurrence log likelihood of a character appearing relative to the string length. The authors then apply a number of processes to reduce the feature set to only the strong features and then use clustering algorithms to group apps based on similarity [15].

3.3.3.1.2 DenDroid

In creating the DenDroid classifier, Suarez-Tangil, Tagiador, Peris-Lopez, and Blasco first disassemble the code using Androguard and identify a sequence of symbols representing the code within each method of the APK as code chunks. These code chunks are then compared across different applications to determine the similarity between the different applications. The basic supposition is that malware from the same family will have greater commonality between code chunks. Code chunks are then classified using an SVM model over the code chunks within the applications. One of the more interesting assertions of the authors is that the lineage of families that may have succeeded each other, such as the multiple DroidKungFu families and the AnserverBot and Basebridge families, have greater similarity in their code chunks than unrelated families [24]. It should be noted that Dendroid is not designed as a malware detector but instead only focuses on categorizing malware into distinct families.

3.3.3.2 Call Flow Analysis Tools

The following section gives an overview of several tools that primarily make classifications based on analysis of some form of CFG or dependency graph generated from analysis of the application code.

3.3.3.2.1 DroidLegacy

In DroidLegacy, the Deshotels, Notani, and Lakhotia develop a method based solely on using API calls within the Android code in order to classify the malware into known families using 10-fold cross-validation. The authors make the important point that since API calls represent the core functionality of the APK file they are invariant to renaming and some of the other obfuscation techniques commonly employed. This is because an application, particularly malware, must make certain system calls in order to execute its malicious payload, which can be obscured but not completely omitted without disabling the malicious payload.

In order to create signatures for analysis in DroidLegacy, the APK is first partitioned into modules by creating a Class Dependence Graph (CDG) where each class represents a node that is connected by edges to other classes with which it shares a dependence. This graph then clusters classes together based on a similarity threshold. The authors note a key difference between their approach and that of Zhou [63], which is that their approach is more granular because they perform clustering at the class as opposed to the package level. Each clustered module is then compared using the Jaccard similarity to the known malware training set. The document frequency of the API calls in the module is then computed and then used to compare the module to other modules.

The authors make the point that since their module can identify suspicious modules, it can more effectively be used to identify the portions of an APK that should receive attention by a human analysts.

3.3.3.2.2 DroidNative

In this work the Alam, Qu, Riley, Chen and Rastogi use static Call Flow Graph analysis in order in to detect and classify malware. One important contribution of the authors is the

addition of the ability to process native libraries that may be included with the application as well as the Java bytecode. This is a non-trivial task and provides an important insight into other avenues that could be exploited by malware. While the threat of native code malware is just beginning to take root in Android (only 3 of the 20 families the researchers analyzed use native code and the rest use bytecode).

If large sections of the call flow graph match, then the application in question is determined to be malware. They use two different techniques, ACFG and SWOD, which are intended generate signatures from the call flow graphs [64, 65]. However, as the authors note, different obfuscation techniques, which are becoming more common, could seriously hamper the effectiveness of their detector. Techniques such as control flow flattening, irreducible flow graphs, and branch/function obfuscation are all listed as potential hazards to detection [13].

3.3.3.3 Tools with Behavioral Approaches

The following tools use static features and also attempt to take into account the behavior of the application relative to the feature set applied. The importance of moving beyond signature-based classification to behavioral-based classification lies in the improvement in flexibility and potential to counteract some of the obfuscation techniques employed by malware authors. With behavioral-based approaches there exists the possibility to identify new and unknown malware variants [45].

3.3.3.3.1 Risk Ranker

In RiskRanker, Zhou et al. created an application that looks through applications that exhibit suspicious behavior such as exploiting known system vulnerabilities and using code

encryption to flag potentially risky apps. These apps still require manual review after the fact to confirm the presence of malware. This tool was effective in flagging zero day exploits.

The first order analysis employed by the tool is mostly static analysis for non-concealed malware. To flag high risk apps, RiskRanker looks for specific code patterns known to be associated with malware applications. For example, to detect rooting malware code, signatures based on known exploits are compared. For SMS charging type exploits, the tool looks for code paths that could charge users money without involving direct user interaction. A second order analysis was also conducted to look for malicious behavior in apps that might be concealed. Examples of this type of behavior include launching a second child APK inside of the original APK (where the second APK exists as an .apk or .jar file inside the first APK) or launching additional code from an obfuscated source.

The results of both the first and second order analysis are then combined to give the application a 'Risk Score,' which can be used by a human analyst to further inspect suspicious applications. In addition to showing good classification results, the methods proposed in this paper allowed the researchers to identify the previously unknown AnserverBot malware family [2].

What is unique about RiskRanker is that the authors determined a set of classification rules that focused more generally on the behavior of the application than on the specific statically defined features within the malicious files. This approach allowed the application to be more flexible in identifying new malware than some of the other approaches described here.

3.3.3.3.2 AsDroid

In AsDroid, Huang, Zhang, Tan, Wang, and Liang take a different approach to detecting malware in Android applications by focusing on identifying sections of the code where the

functionality of the code does not match the description provided to the user through the user interface. In these types of malware, the malicious applications trick the user into taking certain actions by providing a deceptive or unrelated user interface to launch activities such as sending premium SMSes and installing additional components.

The authors focus on a small set of intent filters such as SendSMS, HTTP Access, and Application Installation to identify sections of code to analyze. The authors then extract the user interface information including the string information presented to the user. This is then compared with a set of statically defined rules to determine whether the application presented is attempting to undertake stealthy or malicious behavior without the user's knowledge. While this approach is interesting in that it attempts to identify malicious applications that try to take advantage of users using legitimate commands such as SendSMS while obscuring their use from the user, it is very limited in the static nature of rules applied and the limited depth of its analysis of the underlying code [46].

3.3.3.3.3 DroidMat

In DroidMat, Wu et al. take a similar approach to detecting Android malware. They focus on combining information obtained from API calls, permissions listed in the Android manifest, and intent filters to create a set of features. The features are then used to make a classification with Singular Value Decomposition along with K-means clustering to group the applications into distinct clusters.

The authors claim to compare the functionality of API calls to the type of component that uses them. This allows them to compare the usage of the API in the calling class to determine if it matches the expected behavior. As an example, the authors cite the use of getDeviceID by a background process as a signal of potential malicious intent [28].

3.3.3.3.4 Execute This

One of the major challenges and potential limitations of static code analysis is that it can be vulnerable to external code loading after the application has been installed. In this scenario, a malicious application may remove the code that undertakes the nefarious activities and publish this application for users to install. However, the application contains hooks to load and execute malicious external code after installation. As described by Poeplau et al. in ExecuteThis there are many legitimate uses for external code loading in applications, making it a challenge to separate out legitimate and malicious uses.

In proposing ExecuteThis, the authors lay out methods to separate potentially legitimate code loading techniques from dangerous ones. This is done by examining the surrounding code as well as the arguments passed in to the code loader to assess the risk [35].

3.3.4 Dynamic Analysis

Broadly, dynamic analysis is the process of using information gathered during execution of the program under test. When examining malware, this process is typically done on an isolated emulation environment such as a virtual machine. During execution the program can be given interaction to stimulate activity either manually or through automated approaches. As with static analysis the choice of which features to log has a decided impact on the quality of the results. Typical features that are tracked are memory accesses, system calls, and external communications.

3.3.4.1 Dynamic Analysis Tools for Android

The following section gives a brief overview of the different dynamic approaches that have been put forward for analyzing Android apps. As described by Feizollah et al.

approximately half of the dynamic analysis techniques surveyed focused on tracking system calls (API calls) while the other half were roughly split between network traffic analysis and other features [5]. While there is certainly merit in techniques other than system calls analysis for malware detection, this review of literature will focus primarily on papers that used API call analysis since that is most relevant to the work being proposed.

3.3.4.1.1 TaintDroid

In TaintDroid, Enck et al. put forward a framework for tracking the taint of information flow in an Android application during dynamic analysis. TaintDroid uses variable level tracking of information flows and thus represents a very fine-grained and detailed approach to information flow tracking. TaintDroid only tracks information flows through the Java bytecode. Native code propagation is not handled directly but is indirectly tagged when information crosses from the Dalvik VM to native code and vice versa. The authors then apply TaintDroid to a number of well-known applications and show how it can successfully track the access and flow of private information to potentially untrusted sinks such as network transmission [30].

3.3.4.1.2 CopperDroid

Tam et al. present an emulation-based behavioral approach for logging malware behavior, which they term CopperDroid. CopperDroid is particularly relevant given the number of dynamic analysis tools that use features extracted from the CopperDroid environment. CopperDroid is built on top of the QEMU emulator and uses system call logging to reconstruct applications. In particular, CopperDroid is capable of connecting multiple system calls together to create a more understandable description of the underlying behavior [11].

3.3.4.1.3 DroidScope

DroidScope is another emulation platform for dynamic analysis of Android APKs built on top of QEMU. DroidScope provides an ability to reconstruct and monitor information from both the Java and OS level. This allows analysis of the application code itself and its effect on the underlying Android operating system. DroidScope provides a number of interesting analysis tools such as native and Dalvik instruction tracer for following sequences of instruction calls, a taint tracker to track the flow of sensitive information sources, and the ability to track memory usage at the OS level. Another valuable feature of DroidScope is that it provides an API to allow users to create custom analysis frameworks [32].

3.3.4.1.4 NDROID

In their paper on tracking information flows through the Java Native Interface (JNI) in Android applications, Qian et al. introduce a dynamic malware analysis engine that is capable of tracking taints through native library codes. The engine is called NDROID and is implemented in the QEMU emulation environment. The researchers propose a dynamic analysis approach that specifically focuses on watching the flow of information between the Java code and native code.

NDROID builds on the taint analysis form TaintDroid, which also checks whether the tagged elements reach selected sinks by monitoring flows as they cross into the native source code. This allows the engine to inspect not only calls made within the Java bytecode but also calls executed in the native code.

One key point that the researchers make about dynamic analysis, particularly with Android applications, is that it is difficult to test apps because of behaviors that are triggered by user interactions, and tools like monkeyrunner have trouble tracking all possible input options.

This is a key limitation with dynamic analysis as some malware behavior is only triggered by specific interactions or after wait timers expire [1].

3.3.4.1.5 DroidScribe

In DroidScribe, Dash et al. build on top of the CopperDroid dynamic analysis framework and use SVM to do the classification into distinct malware families. The system uses the higher level behaviors developed by CopperDroid for classification as opposed to directly working with the raw system calls. One major limitation of the system is that it is not designed to do malware identification, but only to classify files that have already been identified as malware into known families. The authors use the feature identified through CopperDroid analysis and their major contribution is the use of SVM models to fit these features into a classification model [12].

3.3.4.1.6 GetOffMyMarket

In this paper, Zhang et al. outline an approach that uses dynamic analysis informed by statically collected features to help identify potential malware. A set of heuristic rules from a basic analysis of the APK code is used to configure the dynamic analysis, such as required permissions and dynamic loading of code. The dynamic analysis then focuses on logging only code patterns and system calls that are known to be associated with malicious files. The logs from the dynamic analysis are then analyzed in depth by the researchers to determine if the application is malicious [36].

3.3.5 Malware Obfuscation Techniques

As has been alluded to several times in the preceding sections, there are a number of techniques for obfuscating known malware variants to avoid detection. These techniques involve modifying strings and methods or class names, adding additional control flow to change the

shape of the CFG, and hiding or encrypting malicious code in the application. The less flexible the signature used by the malware detector, the more likely it is that malware obfuscations will be effective against it. Purely signature-based algorithms are the most susceptible to even small changes as sometimes even changing a single symbol can cause the detector to fail.

DroidChameleon is an Android malware obfuscation tool that is used to test the ability of malware detectors to identify obfuscated malware. DroidChameleon applies additional code transformations, such as the insertion of junk code, reordering of code, and call indirection. Reordering of code is accomplished in the bytecode by moving instructions and inserting go-to statements in its place to change the layout of instruction in the bytecode. Call indirection and function outline/inlining changes the underlying CFG by splitting or merging functions in the original code into different functions in the obfuscated code while still maintaining the original instruction flow. This type of transformation can seriously inhibit detection techniques based on class or method level analysis as the original methods may no longer exist in the obfuscated code [16].

Maiorca et al. took a number of known Android malware and applied simple obfuscation techniques against them, such as string encryption, class renaming and encrypting, and reflection, to attempt to avoid detection [4]. These two papers cover the major techniques currently being applied in obfuscating Android malware and should give the reader a basic understanding of the challenges. For a more thorough review of Android obfuscation techniques, the reader is referred to Pomilia's overview of the topic [29].

In both cases, the obfuscated malware were then run through a number of virus detection packages leading to relatively poor detection rates. There are many more tools even within the Android ecosystem that can do code obfuscation [18]. Given that Android applications are

relative easy to decompile and thus reverse-engineer, there are a variety of commercial tools that are designed to do code obfuscation for legitimate purposes [53, 54]. Google, the developer of Android, is even encouraging and including simple obfuscation in the Android Studio environment [55]. The presence and ease with which these types of code transformations and obfuscations can be applied clearly shows the need to develop malware detection techniques not based on static features or signatures but instead capable of analyzing the underlying behavior of the malware to improve flexibility in detection.

Dynamic analysis can also be subject to manipulation, although the format is typically different than that taken for static analysis. One tactic taken by malicious applications is to attempt to detect when they are running in a virtual machine and if so, not activate the malicious payload, making it impossible to detect dynamically [47]. This problem is compounded with Android as mobile phones have a bevy of sensors that can be used to test if the environment is real or an emulation [25]. Using simple timers that wait days or weeks is another method used by malware to avoid dynamic analysis as is inspecting device variables or searching for the presence of user information, which is more problematic in the mobile environment than with traditional PCs [26].

Chapter 4 Parsing APK Files to Create Feature Vectors

This section proposes a method for extracting a sequence of API calls from an Android APK file that can later be used for classifying the application. The goal in selecting a feature set for this research was to generate the most generic feature set possible in an automated manner to ensure that the features will continue to be useful with the continuing development of new malware techniques. This section describes the design choices made in selecting this particular feature representation and outlines the specific algorithm employed for parsing APK files to generate the API calls list.

4.1 Choosing a Suitable Representation of the classes.dex Code

The first task necessary to test the effectiveness of DNNs in detecting and classifying malware is to create a representation of the APK file suitable for ingestion by a DNN. As with most machine learning models, DNNs typically require that the data be in a vectorized format. While the raw classes.dex file itself can easily be represented as a series of hexadecimal numbers as is specified in the Android documentation, this is not an ideal representation for training and classification. The placement of different items within the dex file is not well suited to analysis by any of the currently used algorithms. Additionally, there is a large amount of overhead required in the dex file format, which would only serve to reduce the effectiveness of a DNN.

Excluding direct ingestion of the raw classes.dex file, there are two remaining options: decompile the code to Java code or smali bytecodes. Feature extraction can then be performed from that code representation. Some code obfuscation and optimization techniques can result in errors when attempting to recompile dex code into Java leading to additional challenges in analysis [9]. Additionally, the code format in Java is slightly less linear and requires

implementation of a far greater variety of commands and syntax given that it is a higher level language.

Smali is a human readable representation of the dex opcodes.[1] Since smali is quite nearly a literal representation of the Dalvik opcodes, it follows a very tightly structured format, which makes it considerably easier to develop parsing rules to process smali code than Java code. Additionally, the utilities already available in Androguard make executing such a parsing algorithm much simpler than would be the case for a Java representation of the code. Using smali as representation of the code strikes an optimal balance between the higher level Java language and the raw dex format.

4.2 Identifying Malware through Sequences of API Calls

After deciding to work with smali code, we must determine the best way to convert the structure and flow of the smali commands into a vectorized format. There are two principal questions that must be addressed. First, we must determine exactly which of the smali commands should be included in the representation. Second, we must determine how to represent the flow of program execution that is not linear and therefore difficult to encapsulate in a vectorized format.

A basic approach would simply record every possible smali command sequentially as the commands appear in the smali representation of the classes.dex file. This approach presents several significant problems. There are a large number of commands that provide little insight

[1]Technically smali is only a human readable presentation of the Dalvik opcodes contained in the dex file. Given the way in which Androguard and the additional parsing techniques described here work, the model being described here technically is operating on the raw Dalvik opcodes. However, for clarity of presentation, the parsing rules are explained in terms of the relevant smali commands.

for malware detection and including them could overwhelm the DNN with too much noisy and unnecessary information, which at best would require significant additional computation time and at worst could prevent the model from successfully converging. Also, reading the files sequentially as they appear in the smali representation of the code does not reflect the order of execution at run time, giving an incorrect view of the code.

The core API function calls are those which interact with the underlying Android operating system and are defined in the Android specification. Specifically, these API calls can largely be broken up into three major libraries: the Java, Javax, and Android libraries [48]. In total, across the applications downloaded, approximately 500 unique API calls were identified. This set of 500 API calls were treated as the words making up the vocabulary used for the classes.dex files. More specifically, only those API calls initiated by a Dalvik opcode invoke-(virtual, direct, etc.) were considered.

The rationale to limit considered instructions to only system API calls is that standard register-to-register operations are not likely to exploit vulnerabilities that would be considered malicious. Other instructions may provide necessary processing steps to execute a malicious payload (e.g., string processing or function argument preparation), but in order to deliver a malicious payload, the program needs to gain unauthorized access to the underlying system functions (e.g. sending unauthorized SMSs, exploiting system vulnerabilities, etc.). The only way to interact with the underlying Android operating system from an application is through the use of API calls. Thus, while register operations are often important functions of malware programs, the key structure of malware access is accomplished through a series of API calls [2,3].

4.3 Creating a Parser to Extract API Sequences

4.3.1 Handling Branching and Control Flow

With the decision to include only core Android API commands as part of the vectorized representation, we need a way to represent the control flow of the program, specifically method calls and brnaching. As with any programming language, Android code is highly non-linear due to the presence of branching statements. The following image gives a visual representation of just one method call within an APK.

Preliminary testing confirmed the basic intuition that the smali commands for branching and jump statements need to be accounted for during the processing of the classes.dex file. Excluding the control flow information in the APKs led to poor performance on the testing dataset. This result should not come as a surprise since without considering the branching instruction the string representation of the APK file assumes that the order of execution of the program can be disregarded without loss of information. Also, ignoring branching statements often leads to API calls appearing in an unnatural order not found in actual program execution and even allowing for scenarios such as sections of unreachable code to be included in the API representation of the dataset.

We cannot represent all of the control flow information (branching statements, method invocations, etc.) in an effective way for processing by a neural network. This is because the set of all branching operations in a program roughly represents a binary tree, so attempting to cover all branching possibilities would be an exponential problem. Instead of attempting to represent all of the control flow information in one feature set for each APK, we will generate code sequences from the APK by following normal code execution and randomly deciding which branch to take at any conditional branch statement.

Clearly, by randomly following a single branch option at each conditional statement, some portion of the information of the APK will be lost in each pass of the parser. However, it is not necessary to cover every possible combination of branches in a single sequencing of the program in order to gain a sufficient understanding of the overall program functionality. Instead, high coverage of all areas of the code normally accessed during execution can effectively be obtained by allowing the parser to run over the program multiple times in order to gain more complete coverage of the code. Running the parser between approximately ten times for each APK generated sufficiently good results in experimental testing.

4.3.2 Method Invocation

Many of the approaches to static analysis presented in the literature consider each method as a separate entity, analyze the code in the method independently, and then represent relations between methods. The parser presented here takes a different approach. As with actual program execution when a method is invoked, the analyzer identifies the called method and then processes all commands within the called method including recursively processing additional method calls until the return statement is reached. Once the return statement for a method is reached, the analyzer retrieves the information for the calling method and proceeds to process that method from the previous location.

4.3.3 Accounting for Multiple Entry Points in Each APK

Unlike a traditional computer program with only a single entry point, Android APKs have multiple potential entry points, which complicates program analysis [36]. While applications can launch in the traditional manner when a user clicks on the icon, they can also begin execution in response to intents or receivers for which they are registered in the Android manifest. Thus each

registered service or activity within an Android APK represents a potential start point for execution of the program code and they must all be examined to determine the nature of the application. To account for multiple entry points, the parser compiles a list of all entry points listed in the Android manifest file and creates an execution string for each possibly entry point.

4.4 Algorithm for Parsing of Android APKs

The following section describes the specific parsing algorithm applied when converting the classes.dex file into a feature list of API calls. We decompile each APK file from dex format into smali code using the Androguard malware analysis tool, so the decompilation process will not be further described here [56]. Once the file is decompiled, all manipulations described below are performed using the analysis tools and framework provided by Androguard.

After each file is decompiled by Androguard, we retrieve a list of all services and activities registered in the Android manifest. We lookup the initial function call for each service or activity identified and place the function calls into an array to be processed. This array represents all possible starting points for code execution within the APK.

For each initial function called by an activity or service, the parser establishes an instruction pointer that points to the first instruction in that method call. For each smali instruction encountered, the parser processes it according to the rules described below and then increments the program counter based on the length of the instruction as shown in

Table 4-1. The exception to this rule is for branching commands, both conditional and non-conditional, which are described in more detail below. The parser then proceeds to access the instruction stored at the location indicated by the updated program counter. This process continues until either a method call command or return statement is reached.

Table 4-1. Parser actions for each of the major classes of opcodes defined in the dex format.

Command	**Analyzer action**
Invoke- (target is non Android API call)	Look up method called, store current program counter, begin processing called method
Invoke- (target is Android API call)	Log API called, increment program counter
Goto v1	Update program counter with value from v1
Return	Return to the method the called invoke and continue processing from the last program counter
Throw	Return to the method the called invoke and continue processing from the last program counter
If-	Randomly select whether to follow branch, update program counter accordingly
Packed-switch and Sparse-switch	Identify the switch payload; from the switch payload select a random branch target, update program counter based using chosen target
All other instructions	Advance program counter by instruction length

For branching instructions, the instruction pointer is incremented as follows. When the parser reaches a go-to instruction, it increments the program counter in accordance with the variable argument in the instruction. When the parser reaches a conditional branch statement, it first randomly selects whether to take the branch. If the branch is taken, the instruction pointer is updated by the offset variable similar to the go-to instruction. If the branch is not taken, then the instruction pointer is updated by the conditional instruction's length.

The packed-switch and sparse-switch commands are conditional branches in smali similar to the case statement in the Java programming language. While the handling of these switch statements is conceptually similar to what is done for the branch statements, the mechanics differ slightly. When a switch statement is found, first the payload for the switch statement must be located in the dex file and unpacked. The list of target offsets often contains

duplicate values to allow for multiple conditions to point to the same branch. Since the goal of our parser is code coverage rather than recreation of specific runtime behavior, we first eliminate any duplicate values and then randomly select a branch target. The instruction pointer is then updated with this branch target.

When the parser encounters invoke-virtual and invoke-direct commands, it determines from the instruction argument whether the method being invoked is a method defined within the dex file or if it is one of the core Android API method calls. If the method is a core Android API, then the parser appends that API call to the API call list and updates the program counter based on the instruction length.

If the invoke command is not an API call, then it signals calling a method defined within the APK. In this case, the parser retrieves a pointer to the called method using the method name to complete the lookup and increments the program counter. The pointer to the method currently being processed and the current program counter are then pushed onto the method stack. The called method then becomes the current method with a program counter of 0, corresponding to the start point of that method.

Table 4-2. Algorithm for generating string-based representation of API calls within an APK

```
entryPointList  = []
from AndroidManifest.xml push activityList, serviceList onto entryPointList
API_call_list = []
method_stack = []

for entryPoint in entryPointList:
    method = findInitialMethod(entryPoint)
    offset = 0
    do:
      instruction = retrieveInstruction(method, offset)
      if instruction.type == branch:
          choice = random(0,1)
          if choice == 0:
              offset += instruction.getBranchOffset
          else:
              offset += instruction.getLength
      if instruction.type == switch:
          switch_targets = switch.find_payload()
          switch_targets = removeDuplicates(switch_targets)
          offset += randomChoice(switch_targets)
      if instruction.type == 'Invoke':
          if invoke target is Android API:
              API_call_list.append(methodName)
              offset += instruction.getLength
          else:
              method_stack.push(method, offset)
              method = invoke.getInvokedMethod
              offset = 0
      if instruction.type == goto:
          offset += instruction.getBranchOffset
      if instruction.type ==return or throw:
          method, offset = method_stack.pop
      else:
          offset += instruction.length

return API_call_list
```

Whenever a return statement is reached, the analyzer pops the first program counter and method pointer, these correspond to the method that called the current method, off of the method

stack. The parser sets the popped method and program counter as the current method and program counter and continues processing.

When the parser reaches a return statement and the method stack is empty, processing for the particular service or activity is complete, which is functionally equivalent to program termination. The parser then retrieves the next activity or service on the activity stack and repeats the process until the activity stack is empty. The string of API calls is then returned as the output of the algorithm representing potential executions of each service and activity of the APK.

4.5 Discussion

One way to think about the sequential API call list representation generated by this parser is as a hypothetical execution of the target APK. This reason for this is that the list was generated by starting from an entry point to the APK file and following program instructions as they would be encountered in actual program execution. Granted, it is highly unlikely that this execution path would occur in practice given the fact that each branching option was given equal probability while in actual execution many branches will follow one fork 90% of the time or more [cite]. However, this is inconsequential to our analysis since we are focused on obtaining good code coverage and providing repetitive loops would likely be of little or no value to our classifier.

Conceptually this makes the approach somewhat of a hybrid between static analysis and dynamic analysis. It contains static elements in that the API sequence was generated from only examining and not executing the source code. However, it is similar to dynamic analysis in that it could represent an actual execution of the source code. This similarity could be exploited to

allow the classifier proposed in the next section to also work against API call sequences that are generated by either static analysis or dynamic analysis without modification.

There are several other benefits to generating a list of sequential API calls in the manner described here. It should be readily apparent that this approach to feature generation is invariant to any form of function, class, or package renaming typically employed to obfuscate code. This is trivially true since that information is completely omitted in the API call list.

Additionally, since all method calls are unrolled and only the individual API calls are retained, this approach is also invariant to code obfuscations which modify the call tree of a function by subdividing methods or providing additional unnecessary method calls. Again this is trivially true since method calls and relationships are completely abstracted away in our representation. This is a major distinction between the approach proposed here and many of the typical approaches which by a Call Flow Graph to analyze a problem.

There are a few limitations with the parser implementation that bear mentioning. The first is that the parser does not currently do a check to determine the percentage of code coverage obtained after multiple runs. A more sophisticated approach would be to tag sections of the code once covered and then specifically select branches to follow until all code in the source file is covered. This approach would require significantly greater computational complexity and, as we will show shortly, the current approach is more than adequate for the classification task at hand.

The manner in which exceptions are handled could also be improved. Currently, the parser treats exceptions in the same manner as return statements and then continues processing from that point. However, it may be more accurate to determine the target of the exception and begin code execution at that point.

Chapter 5 Deep Learning Literature Review

Neural networks were originally proposed in the late 1940s but gained very little traction due to the computationally expensive nature of training for all but the smallest networks. While there were some limited successes prior to the early 2000s, given the limitations of computational power, it was not practical to train neural networks with enough nodes and depth to produce useful results in most cases. This changed greatly around 2006 based on the general increase in computing power available and the use of GPUs in training. The realization that the training equations for neural networks could be run much more efficiently using the specialized hardware in a GPU to handle the matrix multiplication operations allowed for a speedup of ten times or more in computation time [cite – Graves?]. This made it much more practical to train and employ neural networks for practical applications and led to the renewed interest currently seen.

Neural networks derive their name from the similarity of the individual computational components to neurons in the brain. Neurons or nodes in the network are connected together by weighted edges, which allow for flow of information through the network. At each neuron or node in the network a non-linearity of some form is applied to the summation of the weighted inputs from the incoming connected edges to produce an output. This output value is then propagated to other nodes in the network through the outgoing edges of the node.

The “deep” portion of the term deep learning comes from arranging the neurons in the network in layers of arbitrary size and then stacking these layers on top of each other. This approach effectively increases the computational complexity of the network. While it has been shown that a neural network with two layers is Turing complete, and thus capable of representing

any function, this takes an exponential cost when only expanding the width and not the depth of the network [81, 82].

The major advancements in recent years came in methods to improve training of neural networks for practical applications. The typical method for training neural networks is to use backpropagation in combination with a technique such as stochastic gradient descent to update the weights of the edges and node biases. In backpropagation the output of the neural network is compared with the target or expected value and then the difference between the two values is propagated backwards through all elements of the network. The error at each node is then used with an update algorithm such as stochastic gradient descent to update the weights and biases in the neural network to minimize the training error. This process is repeated with additional training examples until the stopping criteria are met.

There are several modifications to traditional stochastic gradient descent such as Adagrad, which is employed in this research. These modifications aim to improve convergence by taking into account past variations in the gradient descent algorithm [82]. This is done by tacking the weight updates on individual parameters to smooth convergence, usually by applying some form of dampening or momentum tracking. Additionally, in practice the update procedure is normally conducted over a batch of training examples instead of individually to speed convergence. In batch processing a number of training examples are provided at once and the error is calculated across all examples when computing the updates. This can increase processing performance in certain implementations and helps to smooth the path of the learning algorithm by reducing the error introduce by reducing the effects of individual training examples [83].

Convolutional neural networks have recently gained popularity and success in the field of image recognition. Convolutional networks use a series of local filters, typically called receptive fields, applied across the image to generate a feature map and were first demonstrated in Lenet-5 by Lecunn et al. in 1998 [74]. There are two benefits of applying the same convolutional kernel repeatedly across the image. By creating small convolutional filters, with shared weights across the whole network, the computational load is greatly decreased as compared to a fully connected deep network. Additionally, convolutional networks help address the problem of overfitting and increase generalization by looking for the same features across the entire image or document [75].

The last few years have seen several stunning successes with deep neural networks, particularly in the realm of image processing and nature language processing. ImageNet 2012 marked the first time that a deep learning algorithm, named AlexNet, won the iconic image classification challenge. Not only did AlexNet win, it did so by a wide margin compared to the other techniques employed [76]. Since then the competition has been dominated by deep neural networks as opposed to the previous techniques, which focused more on using an ensemble of hand generated features with other machine learning techniques [77]. Some of the other important recent results from the ImageNet competition include GoogleNet in 2014 [78] and the use of Resnets in 2015 [79].

Pooling is often applied in convolutional neural networks for two reasons: to reduce the size of the resultant feature maps from the convolutional layers and to generalize the presence of a feature to a particular area. In the example of a 4x4 max pooling layer, this is accomplished by selecting the maximal filter response in each 4x4 error of the input layer to pass forward to the next layer, removing the specific location information in the process. Pooling also helps reduce

the computational load by effectively shrinking the size of the feature maps passed to the next layer in the network.

AlexNet won the 2012 ImageNet competition using a neural network consisting of eight layers; five convolutional layers followed by three fully connected layers. Pooling was also applied to the first, second, and fifth convolutional layers. At the time, AlexNet's size and depth were state-of-the-art, requiring multiple graphics cards working simultaneously to train the network. However, by modern standards, AlexNet follows a fairly traditional convolutional neural network architecture and can easily be trained on a single GPU [80].

In GoogleNet, the network uses what Simonyan et al. term "inception layers," which are the major distinguishing feature of the network from previous models. Fundamentally, the inception layers use convolutional layers of several different sizes (such as using a 1x1, 3x3, and 5x5 filter) along with a pooling layer to create the feature map for the next layer to process. This allows each layer of the network to learn features at several different scales simultaneously. GoogleNet combines this technique with the additional application of more 1x1 convolutional layers before the filters and after the pooling layer to allow for dimensionality reduction. This allows the construction and training of a much deeper network than would otherwise be possible [78].

The most recent ImageNet winning network used Resnets to achieve a depth of 152 layers. In the simplest terms, Resnets (or Residual Networks) work by applying convolutional layers and then repeatedly feeding the original input to the network back in at later layers. In this regard the approach taken with Resnets is similar to Recurrent Neural Networks (RNNs) and research has shown that Resnets are in fact equivalent to an RNN [84]. The current speculation is that by using the residual approach, the network is able to continuously learn distinct features

in higher levels since it can compare already learned features from lower levels with the input image.

While the majority of CNN applications have focused on image processing, there have also been some successes in other fields such as text classification. Santos and Gatti discuss a network based on employing CNNs to classify sentiment in short sentences. They take the stream of words and develop features based on each word and character level embeddings, which independently apply convolutional layers and are then merged into a second convolutional network, creating a classifier for the whole sentence [85]. In a similar paper on sentence classification, Kim uses a CNN to classify sentences. First each word is converted to a feature vector using pre-trained language data. Then sequential strings of pre-trained word vectors are input to a convolutional neural network to learn features, which are then pooled over time and fed to a dense layer for final classification [86]. Lai et al. combine both recurrent and convolutional networks to achieve a high level of success in classifying document length texts. By adding the recurrent component to the convolutional network, they are able to more effectively deal with inputs of longer lengths [87].

Chapter 6 Using Deep Neural Networks to Classify APK Files

This section develops the process for using neural networks, specifically Convolutional Neural Networks (CNN), to classify APK files. The process for converting the API call sequences generated by the parser into a format amenable to use with a CNN will be described. Then the structure of the classification network will be presented along with a discussion of the hyper parameter selection. The section concludes with a series of experiments testing the ability of the neural networks to classify malware and benign applications and a discussion of the results.

6.1 Neural Network Topology for APK Classification

6.1.1 Formatting Data for Input to a Convolutional Neural Network

Convolutional neural networks require fixed sized matrices as input for classification. As described previously, the parser generates a representation of the APK of the form:

$$R = [a_1, a_2, \ldots . a_n]$$

where a_i is the *i*th API called in the hypothetical execution of the APK. We create a lookup table that maps each API call to a unique number in the range of API calls, replacing the string of API call names with a string of integers representing the program execution. These integers are then be converted into a sequence of one-hot vectors, which are more amenable to use with a CNN (Figure 6-1).

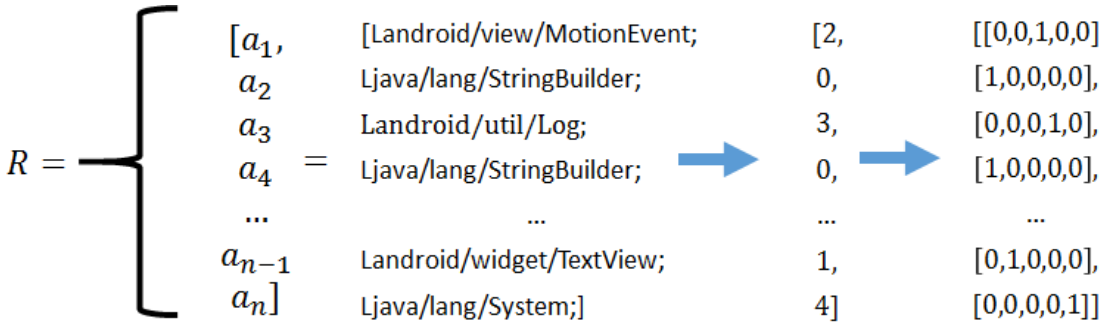

Figure 6-1 A sample conversion of a string of API calls to one-hot vectors by first converting via a lookup to integers and then creating a list of one-hot vectors.

The length of API sequences varied from under 100 to well over 10000 API calls depending on the APKs. However, convolutional neural networks expect a fixed sized input across all samples, so in order to accommodate this discrepancy in sizes of the API lists, we select a set length L_{net}. The list of API calls R can be divided into $R = \{R_1 + R_2 + \ldots R_n\}$ such that each R_1 to R_{n-1} have length L_{net}. This leaves a small remainder of API calls in R_n, of length $Length(R_n) = Length(R)\ mod\ L_{net}$. We repeatedly tile these remaining API calls to fill the final section R_n. Each of these sections R_i can then be fed into a convolutional neural network to provide a classification.

6.1.2 Convolutional Neural Network Topology

In order to classify each individual segment R_i we use a convolutional neural network with the following topology. First there is an input layer followed by a convolutional layer with a filter size of $(vocab\_size, filter_1)$. This allows the network to reduce the dimensionality of the one-hot vector in the first layer and generate features based on the number of API calls in this size of $filter_1$. This layer is followed by a dropout layer to avoid overfitting and co-dependence between different filters within the convolutional layer. Next a pooling layer is applied,

completing the first level of the CNN. This first level is followed by another convolutional layer, dropout layer, and pooling layer. The final pooling layer is fully connected to a softmax output layer with a node for each category within the test set. A diagram of the convolutional network topology is show in Figure 6-2. Dropout layers are only applied during training and serve as a simple pass through in testing.

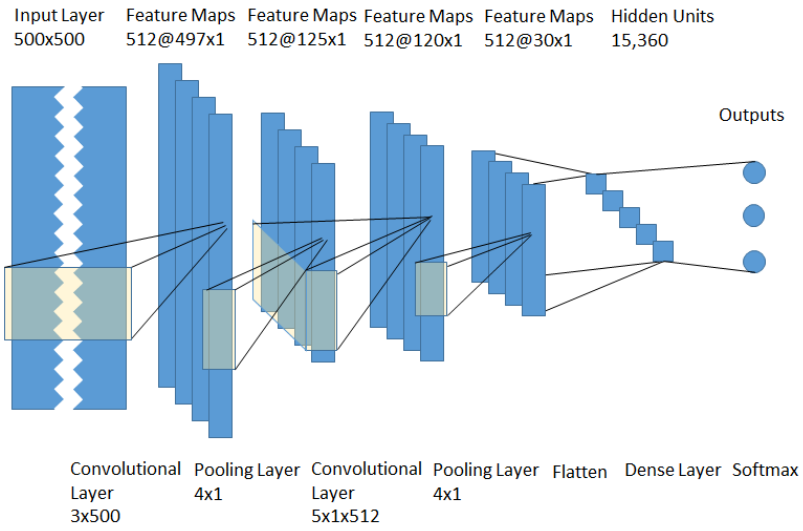

Figure 6-2 Diagram of the Convolutional Neural Network used in testing with two convolutional pooling layers and a dense layer connecting to a softmax output.

The target value for each chunk of program code, R_i, fed to the CNN carries the label for the entire APK file. The training loss function applied over the softmax layer was categorical cross entropy and the ADAGrad update algorithm was used to propagate updates through the network. All functions were implemented using the relevant Lasagne and Theano packages for Python [49,50].

Early stopping was used during training to determine when to stop training for the model. However, the loss used in the early stopping calculation was not categorical cross entropy of each segment R_i as used for backpropagation. Instead, the average prediction accuracy of the

whole APK file, calculated over each of $R_1, R_2, \ldots R_n$, was used as the early stopping metric. Once the model failed to show improvement over a random subset of the validation set, the training was stopped and the best model results were chosen for use in testing.

6.1.3 Final Classification

As noted above, the CNN only generates labels for each segment R_i and does not directly label the whole APK file. It is expected that different segments within each APK may be assigned labels from conflicting classes. To resolve this conflict and provide a final classification for each APK file we take the output s_i for each segment R_i from the softmax output layer and calculate the sum across all output vectors s_i. The maximal position of the resultant vector is selected as the classification for the APK file as shown in the equation below.

$$c_i = argmax(\sum CNN(R_i))$$

6.1.4 Excluding Repetitive API calls

An analysis of the generated API call strings showed that certain API calls tended to be called many times in direct succession. The Java string builder API in particular is often called five or more times in a row. This repetitive calling does not provide meaningful new information to the classifier and the noise can actually significantly impair classification. In order to improve the performance of the DNNs in classifying the API lists, the repetitive call strings were reduced to a single API call in the API list provided to the DNN for training. Testing showed an almost 10% increase in accuracy between classification done when repetitive API calls were included in the API lists and when they were excluded.

6.2 Experiments

6.2.1 Datasets

Two datasets were used for testing the CNN: a benign dataset and a malware dataset. The composition or each dataset is described below.

6.2.1.1 Malware Dataset

Malware samples were obtained from the Contagio Mobile repository [51]. In particular, for this evaluation, we focus on six families of malware from the Contagio dataset consisting of a total of 231 individual APK files. All of these samples are recent with the oldest malware having been detected in May of 2016 and should accurately represent currently employed malware techniques. Table 6-1 shows a breakdown of the different malware families. Additional information on each of the malware samples used can be found from a number of different sources listed in the references [66, 67, 68, 69, 70, 71].

Table 6-1. Malware families and number of samples within each used in testing

Malware Family	**Number of Samples**
Godless	8
Hacking Team Crisis Android	70
SberbBanker	7
Hummingbad[2]	42
OverlayLocker	55
Marcher	5
Xiny	6
AndroidXbot	9

[2]Note that the Hummingbad dataset posted on Contagio mobile contained over 590 distinct APK files. In order to avoid skewing the tests with the large sample size, a random subset of the APKs were used in testing. Additional testing confirmed that increasing the number of Hummingbad examples did not adversely affect detection.

6.2.1.2 Clean Dataset

Android applications for the category classification task were gathered from a variety of third party app stores. The categories used for the classification task were chosen to match the common categories found in app stores for sorting applications for display to users. A small number of files were not able to be decompiled by the Androguard decompiler and were removed from the dataset.

In total 800 unique APK files were used for the classification task spanning nine categories as shown in Table 6-2. To ensure that no duplicate APK files were included in the dataset, an MD5 hash was run over all APKs and any files that had matching signatures were removed from the test dataset. Additionally, to the extent possible different versions of the same applications were excluded as they represent only minor differences in code structure, making classification trivial for one application if the other is included in the training set.

Table 6-2. The categories and number of applications per category for the benign applications used in testing.

Category	**Number of Applications**
System Tools	97
Books & Reference	108
Puzzle Games	101
Health & Fitness	58
Photos	87
Themes & Wallpaper	86
Communications	103
Arcade Games	76
Shopping	68

6.2.2 Comparison Models

In order to give a baseline comparison for the proposed CNN model, several standard classification algorithms were also employed. The model is compared to an N-Grams SVM classifier and a Naïve Bayes classifier, which are both standard classifiers used for classifying text documents. Text documents represent the best analog for the sequence of API calls used in this testing since each API call can be considered as a word in the language and the ordered sequence represents a document. All models used the same set of API string representations as a baseline dataset.

For the N-Gram SVM, each sequence of API calls was converted into a feature vector listing the number of occurrences for each 3-gram, as testing showed this to be the optimal feature size. Term frequency–inverse document frequency (TF-IDF) was used to normalize the feature vectors before classifying with the SVM. SVM was applied using the Scikit-Learn toolkit in Python with an RBF kernel. The Naïve Bayes classifier was also implemented using the Scikit-Learn toolkit [52].

6.3 Malware Classification Experiments

Two tasks were included in the malware classification tests. The first task was malware detection where the CNN needs to differentiate between malicious and benign APKs. To train the CNN for this task the third party app store dataset was used to represent the benign APKs, while the Contagio Mobile dataset was used to represent the malicious APKs. The results in Table 6-3 show the accuracy scores for the various models using 5-fold cross validation.

Since the number of applications included in the clean dataset significantly outnumbers the malware dataset, only a subset of the benign training data was used to train the model. This subset of the clean samples was chosen randomly within each training cross fold. This approach

avoided the typical problems associated with unbalanced datasets when dealing with machine learning models where the model achieves good accuracy at the cost of precision.

To evaluate this approach, two tests were conducted. The classifier was presented with equal sized datasets of malware and clean samples in the first test and unequally sized datasets for which the clean samples greatly outnumbered the malware in the second test. In addition to being able to consistently detect malware, the classifier must also have an acceptably low false positive rate in this type of scenario. The second test is more indicative of reality, in that most real world datasets (i.e. application marketplaces) consist of majority benign applications with only a few malware samples included. Thus the second test gives a better indication of the possible effects of false positives in a real world application.

Table 6-3 shows the accuracy precision and recall rates for the malware detection rate in both the balanced and unbalanced validation sets. The chart also calculates the precision, and recall for each case.

Table 6-3. Results of each model for the malware detection test.

Model	**Accuracy**	**Precision**	**Recall**
CNN	99.4%	100%	98.3%
Ngram SVM	66%	53.3%	34.8%
Naïve Bayes	82.0%	47%	54%

Table 6-4 Confusion matrix for the malware classification test

		Predicted	
		Benign	Malware
Actual	Benign	339	0
	Malware	4	226

In the second malware task the CNN is trained not to identify malware but to classify each malware variant into its respective family. The value of malware family classification is that knowing to which family a piece of malware belongs can help researchers more quickly analyze its code or assess what damage a malicious app may have caused by referring to previous analyses of its familial counterparts.

The same datasets including the benign applications were used again with 5-fold cross validation and the results are shown below in Table 6-5.

Table 6-5. Confusion matrix for the CNN classifier in the malware family classification test

		Predicted Malware Family							
		Crisis Team	SberbBanker	OverlayLocker	Godless	Hummingbad	AndoridXbot	Marcher	Xiny
Actual Malware Family	Crisis Team	69	0	0	1	0	0	0	0
	SberbBanker	0	7	0	0	0	0	0	0
	OverlayLocker	0	0	55	0	0	0	0	0
	Godless	0	0	0	6	2	0	0	0
	Hummingbad	0	0	0	0	42	0	0	0
	AndoridXbot	0	0	0	0	0	9	0	0
	Marcher	0	0	0	0	0	0	5	0
	Xiny	0	0	0	1	0	0	0	5

It is not uncommon for a classifier to have higher accuracy in the malware detection task, comparing malware and benign applications, as opposed to the family classification task where the malware is grouped into specific families. As explained in more detail by Tangil et al. [24], this can be a result of the fact that certain malware families (as classified by malware analysts) may actually be based on the same exploit or basic code fragments.

One of the limitations in testing against the malware datasets is that the variation within available malware families tend to be small. This makes it easy for a classier to identify

additional variants of a particular malware without providing good generality and flexibility. Given the small variation between different APK variants within a malware family, the test and validation sets may contain nearly identical examples. Thus it may be possible to construct a classifier to achieve a high level of performance without actually obtaining good generalization by simply memorizing the features of the training set.[3]

6.4 Classification of APKs into Categories Based on Functionality

As shown in the preceding section, the model does very well in detecting and classifying malware from known families. However, this task provides somewhat less insight into the flexibility offered by the model since many malware families and variants are so internally similar as to make classification a relative easy process. Surprisingly, for machine learning models, it is potentially more challenging to group the benign applications into thematic categories such as System Tools, Games, etc., than it is to identify malware variants. By testing the ability of the model to distinguish APKs within this context, we can better demonstrate its flexibility and understanding of the underlying program functionality.

The malware detection tests only require the identification of small, unique sections of code that can be used as signatures to detect other applications attempting to accomplish the same task, in order to properly classify the malware families. However, given the huge variety of applications within each category in this thematic classification test, the model must demonstrate a higher level of understanding of the application functionality to succeed.

In this task only the benign dataset is used and all applications are classified into the categories based on their functionality and listing in the application markets. Again, 5-fold cross

[3] As a note, this is essentially how signature-based detection of malware works.

validation is used to judge the results and the average score across all validation sets is reported below in **Error! Reference source not found.**.

Table 6-6 Results for different models in thematic classification of benign dataset

Model	**Accuracy**
CNN	61.3%
Ngram SVM	22.2%
Naïve Bayes	18.8%

Table 6-7 Accuracy measurements for each model and confusion matrix for the CNN classifier on the benign dataset.

		Actual Category										
		System Tools	Communications	Books	Health and Fitness	Photos	Wallpaper	Shopping	Food and Drink	Puzzle Games	Arcade Games	Board Games
Predicted Category	System Tools	**64**	20	9	5	3	0	6	6	0	0	0
	Communications	6	**73**	5	0	6	4	8	5	1	2	0
	Books	14	20	**45**	6	7	1	8	2	2	0	0
	Health and Fitness	9	3	4	**52**	10	0	4	3	5	0	0
	Photos	5	8	3	0	**57**	0	3	8	1	0	0
	Wallpaper	2	10	2	0	1	**32**	2	2	1	1	0
	Shopping	3	10	2	0	3	1	**63**	13	0	0	1
	Food and Drink	0	4	1	2	1	1	20	**55**	1	0	0
	Puzzle Games	5	10	5	6	4	1	3	4	**62**	8	3
	Arcade Games	1	3	0	0	6	3	2	5	18	**54**	1
	Board Games	3	6	2	1	8	0	10	6	15	7	**16**

		Actual Category										
		System Tools	Communications	Books	Health and Fitness	Photos	Wallpaper	Shopping	Food and Drink	Puzzle Games	Arcade Games	Board Games
Predicted Category	System Tools	**0.57**	0.18	0.08	0	0.03	0.04	0.05	0.05	0	0	0
	Communications	0.05	**0.66**	0.05	0.04	0.05	0	0.07	0.05	0.01	0.02	0
	Books	0.13	0.19	**0.43**	0.01	0.07	0.06	0.08	0.02	0.02	0	0
	Health and Fitness	0.04	0.19	0.04	**0.6**	0.02	0	0.04	0.04	0.02	0.02	0
	Photos	0.06	0.09	0.04	0	**0.67**	0	0.04	0.09	0.01	0	0
	Wallpaper	0.1	0.03	0.04	0	0.11	**0.58**	0.04	0.03	0.06	0	0
	Shopping	0.03	0.1	0.02	0.01	0.03	0	**0.66**	0.14	0	0	0.01
	Food and Drink	0	0.05	0.01	0.01	0.01	0.02	0.24	**0.65**	0.01	0	0
	Puzzle Games	0.05	0.09	0.05	0.01	0.04	0.05	0.03	0.04	**0.56**	0.07	0.03
	Arcade Games	0.01	0.03	0	0.03	0.06	0	0.02	0.05	0.19	**0.58**	0.01
	Board Games	0.04	0.08	0.03	0	0.11	0.01	0.14	0.08	0.2	0.09	**0.22**

6.4.1 Segment Prediction Accuracy vs APK Prediction Accuracy

The plot in Figure 6-3 shows the accuracy of prediction for the APK as compared to the prediction accuracy of each segment of the APK. It is not surprising that the segment accuracy is

significantly lower than the prediction accuracy, which is the result of the averaging done across all segments in the final classification; however, the reason for this effect may be less clear. While each category of applications has distinct characteristics that allow for successful classification, there are certain aspects that are common across many applications. A good example might be a section of code or library from a third party used in multiple applications. Many applications use external add libraries in their code and therefore it would not be uncommon to see the code for the add library appear across all categories. As such, those chunks of code could not be correctly categorized since they will appear across several or all categories.

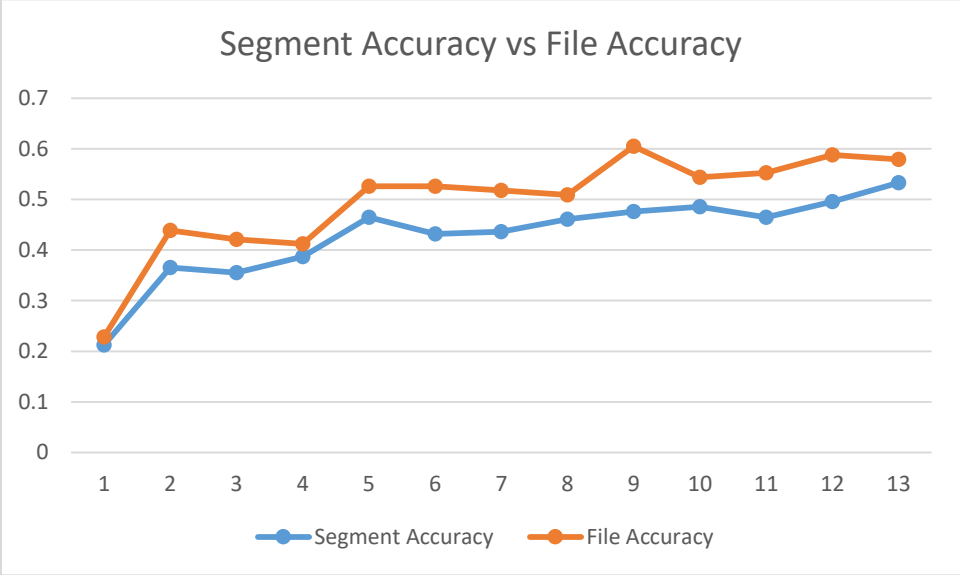

Figure 6-3 Plots of correctly predicted segments of APK files compared with correct predictions for the entire file.

6.4.2 Training Size vs Accuracy

In these experiments the size of the training set was varied and the accuracy of the model plotted against training size. The size and contents of the validation set were held constant throughout these tests. It is important to state the benefit of the small number of training samples

needed to effectively train the model. This is particularly true in the case of malware detection as when new malware families are discovered, there may be only one or a handful of examples of the malware variant with which to train a classifier.

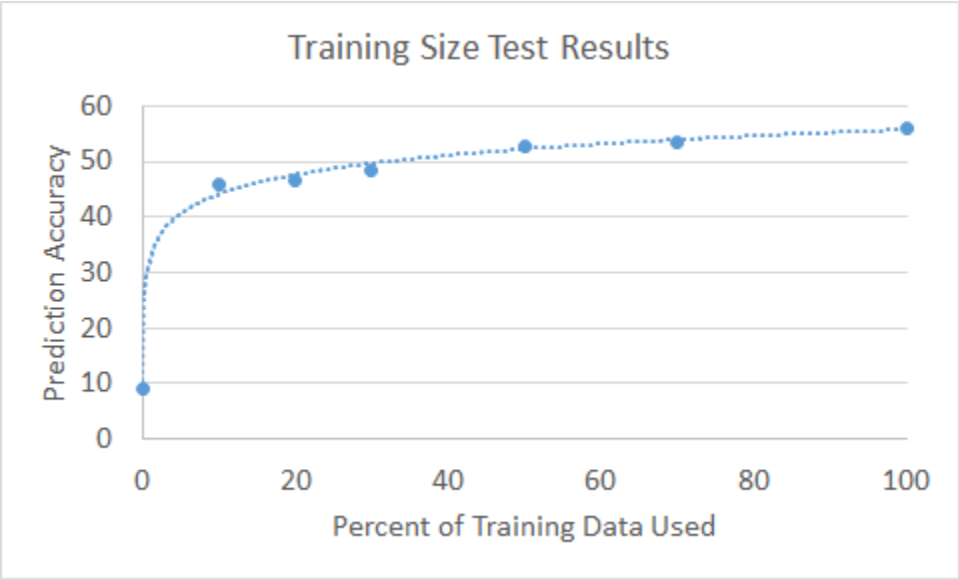

Figure 6-4 Plot of the accuracy of the CNN classifier compared to the percentage of the training data used during the training.

As described earlier, each APK is broken into to 10 API string representations, which are distinct based on the random selections made regarding branching statements within the APK code. The second plot shows how the accuracy is affected when only one API string representation is used per APK. This shows the benefit that is accrued by taking this approach to creating the string representation. In concept this approach is similar to expanding a limit dataset by adding noise, which has been shown to be effective in training neural networks [62]. This strategy has the same effect of introducing small variations that keep the model from over fitting specific API call sequences with the added advantage that this is accomplished not by noisy (and possibly) incorrect data but instead by using potentially real variations of the dataset.

6.4.3 API Calls vs Accuracy Plot

Figure 6-5 shows the accuracy of the model labels for an APK file plotted against the number of API calls in our analyzer's representation of the APK. The trendline suggests that there is no correlation between the number of API calls in an application and the prediction accuracy. This is an encouraging trend since logically we would not expect that the length of a program's code should change the prediction accuracy of the network.

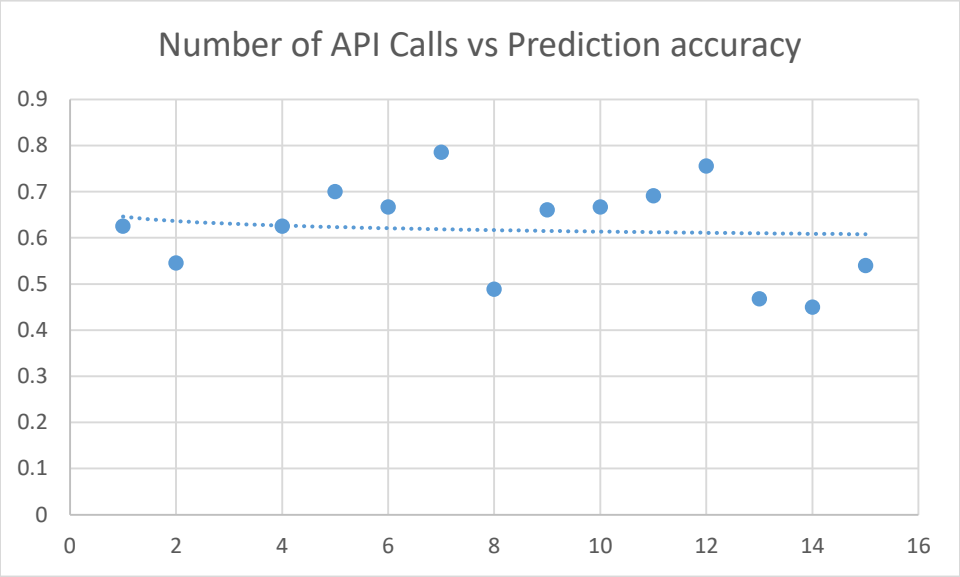

Figure 6-5 Comparison of the number of API calls in the string representation of an APK and the classification accuracy of the CNN classifier

6.4.4 Input Chunk Size and Accuracy

In this test we examine the correlation between the length of the API call segments that are fed into the CNN for classification and the accuracy in predicting the value for that segment along with the accuracy in predicting the APK category (Figure 6-6).

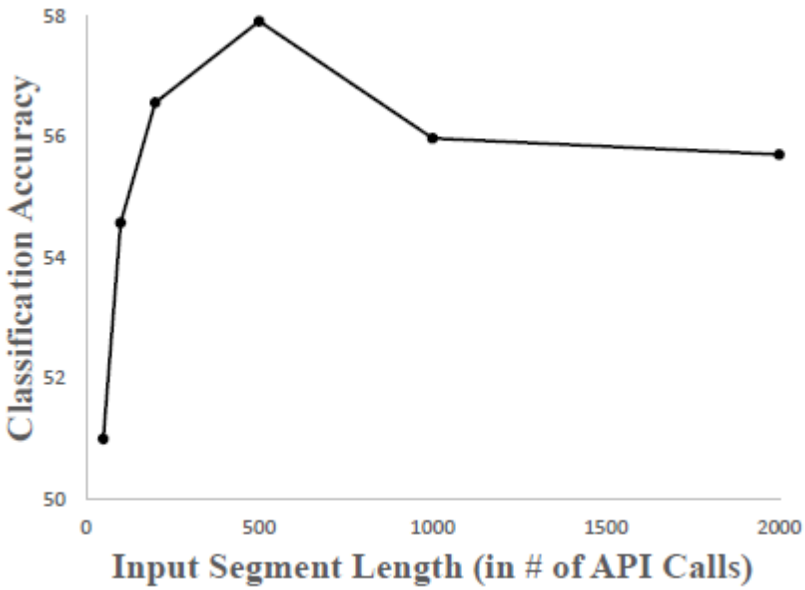

Figure 6-6 Accuracy of the CNN classifier based on the size of the input layer to the classifier

While we would expect to see a correlation between increased length and overall accuracy (up to a point), we also want to find the smallest input length that produces good results. The importance of accuracy over small chunks is that in the malware detection scenario, it allows an analyst to focus on particular sections of code in an APK that has been flagged as potentially malicious. By being able to focus on a smaller section of the code for manual analysis, human analysts can be more efficient and reduce the overall human cost of analyzing malware.

The ideal network length breaks API strings up into sequences of approximately 200-500 API calls each. On an intuitive level this makes sense for several reasons. First, the median API sequence string length for all APKs in the test set is approximately 4000 API calls. Second, and perhaps more importantly, it makes sense that outside of a certain length of time, API calls would be relatively unrelated. Chunks of approximately 200 APIs likely represent the maximum size of a logic block in most programs in the dataset. By segmenting larger files into chunks and

classifying each chunk separately, we allow the classifier to make a determination of each logical block within the program and then combine those individual determinations to make a final classification.

While there is a drop off in accuracy in using sequences under 200 API calls, the model works almost as well at sequences of only 100 API calls. Even at sequences of 50 API calls the accuracy rate is less than 3% below the peak. This means that small API windows could be used to help narrow down the sections of code in an APK that might be malicious to a small handful of function calls out of the entire APK.

6.5 Discussion

As is demonstrated in the tests, a CNN is clearly capable of learning features necessary to classify Android APKs both in the malware detection test and the categorization of benign applications tests. Furthermore, it outperforms the other standard models used for comparison in our testing. One of the major benefits of using a CNN as described in this paper is the lack of any manually generated features and the flexibility of the features to new and novel examples.

To understand why a CNN is well-suited to this task, it is worth considering the fundamental nature of CNNs and an analogy to the field of image processing where CNNs have been widely studied. First, in analyzing program code, feature locality is very important. Two API calls that appear close to each other are much more tightly related than two API calls that exist hundreds of lines apart. CNNs naturally take this fact into account by using small local filters to develop features in each layer of the network, which are consecutively passed upward to create more abstract features in the network. This is very similar to how human experts approach malware analysis, by looking at small sections of code to determine their function and progressively considering the interactions of these code segments with one another.

Also, in the image processing realm, CNNs have shown great strength in being resistant to spatial transformations of images such as deformation, noise, and rotation [62]. While code is not deformed in the same way as images, the insertion of extra API calls between segments of code containing key features is very similar to elongation of an object in an image. The ability to tolerate noise and deformations makes it harder for typical code obfuscation techniques to defeat analysis without changing the underlying effect of the code and potentially eliminating its malicious payload.

Interestingly, CNNs have also demonstrated decent accuracy in providing labels to partially occluded objects [73]. In the malware classification task, this might correlate to instances where only partial features are available for analysis either because the malware was obfuscated or portions of the code were launched from external resources. Specifically, we can see this in the classifier's ability to categorize the Godless examples. Two of the Godless samples are distinct from other members of the family because they obtain the majority of the malicious payload via a web download after the application is installed, only having a minimum amount of code within the APK available for analysis.

For comparison, the very insightful and effective work done by Zhou et al. in RiskRanker produced excellent results but relied heavily on manually generated features to aid in detection [2]. As malware evolves, these features need to constantly be updated to keep up with the new techniques of malware writers. The classifier proposed here will automatically update and generate new features after training with new examples reducing the burden on human analysts.

Many static analysis techniques described in the literature review rely on two commonly used techniques: package or class name analysis and call flow graphs. As detailed in the literature review, both of these techniques are vulnerable to obfuscation techniques that can be

relatively easily employed by malware authors. The problem with and limitation of these techniques is that they rely on transient properties of the APK file for analysis, which can be modified without changing the effect of the underlying code by several different obfuscation techniques.

Package renaming is a relatively straightforward undertaking; there are a plethora of online tools and IDEs to assist in this process [88]. Since the proposed model never considers package or class names directly in its analysis, it is clearly invariant to these obfuscation techniques, representing a clear improvement in durability when compared to models that require that form of information.

While call flow graphs are slightly more difficult to alter without affecting the end function of the code, there exist a number of known techniques for obfuscating call flow graphs [59, 60, 61]. Several of the papers discussed in the literature review focus specifically on trying to improve the resiliency and flexibility of CFGs, but they remain susceptible to manipulation. By unrolling the sequence of API calls that would be encountered during a real execution of the program, the model presented here has the potential to avoid many of these techniques. Additionally, analysis of CNNs in the realm of image processing have shown that with proper training techniques, the networks are resilient to rotations, deformations, and noise. While the field of image processing is not a direct analogue to that of code analysis, there is reason to believe that CNNs for code analysis could maintain some of this invariance, making it more difficult for obfuscation techniques to be applied [72].

6.6 Limitations of the Current Model

There are several limitations to the model presented above that could limit its effectiveness. These limitations include the failure to consider native libraries or additional

resources in the APK file, the potential for reflection to reduce the parser's effectiveness in processing all relevant code section, and the failure to confirm all code sections have been followed.

The methods described in this approach only consider the code contained within the classes.dex file. Recently there have been a number of malware strains that launch their attacks from native code libraries or hide code in additional resource files. While it is possible that the model could still detect these types of attacks through the calling procedures needed to launch the code from the classes.dex file, this is somewhat less likely as these operations also have legitimate functions [35].

The model also does not directly address the concept of reflection where instead of calling a package directly, the application uses Java-based reflection techniques to generate the function call. This type of approach could cause the parser to skip important sections of application code, effectively hiding malicious code from analysis. This limitation could potentially be deal with in two ways. First, the parser could be enhanced to recognize reflection and determine the correction function to follow during parsing. Second, we could keep track of all code sections in the APK and after initial parsing, identify any methods that were not covered and individually cover those methods (independent of the start point list). Both of these approaches should adequately address the use of reflection.

The model currently does not confirm that all sections of code contained within the classes.dex file have been covered during parsing. This could lead to critical sections of the code potentially being omitted based on the random selections made when the parser encounters conditional branches. This could be addressed by tagging code sections and conditional

branches that have been parsed and ensuring that subsequent passes of the parser go to the unvisited branches.

Chapter 7 Future Research

There are a number of areas for further research that could lead to a model with greater understanding and predictive power than what is presented here. Some of these topics include enhancing the test and training dataset, combining the proposed model with other types of analysis, and converting the model to work with dynamic analysis tools. This section will be devoted to a brief discussion of some of these key areas.

7.1 Inclusion of Code from Native Libraries

Only the classes.dex file is considered in this analysis. None of the native libraries or additional resources included in the APK file are considered. As was shown in DroidNative there have recently been a small number of known malicious APKs that use native library code to launch their attacks. It is likely that this approach will be increasingly adopted by malware writers both to avoid detection and to take advantage of system vulnerabilities that may only be present using native code [13].

While the structure and nature of API calls in native libraries is different from that present in the dex file, the same concept for classification using a CNN could apply. The major difference would be in the decompilation process for the native library source code. While decompiling native library source code is not a trivial task, it is one that has been addressed by multiple other researchers [57, 58].

7.2 Applying a More Sophisticated Neural Network Architecture

Although the network used in this paper was sufficient to obtain excellent results in the malware detection and classification scenario, it is relatively simplistic compared to many of the recently released Convolutional (and Resnet) architecture being employed in the field of image

recognition. While there is not a direct link between image recognition tasks and malware detection, the high level of success obtained by these networks does suggest that using a more sophisticated neural network may provide additional insight and predictive power than the CNN described here. This is an area for future consideration.

7.3 Including Techniques to Examine Additional APK Resource Files

Malware or parts of malware can be hidden with ancillary resources within the APK and be reconstructed and launched by the code in the classes.dex file. The analysis of these types of files is mostly orthogonal to the analysis of the API calls in the classes.dex file and as such could be combined with the model described here with relative ease.

7.4 Complementary Techniques

The model does not take into account information relating to package or class names, or string information included in the dex file. While these types of analyses are subject to relative simple obfuscation methods such as dynamic string building within the dex code and package/class renaming, they are nonetheless still effective in analyzing malware.

7.5 Configuring to Work with Dynamic Analysis Techniques

The approach described here of using a CNN to classify APKs based on API call sequences can be directly translated from static analysis as described in this paper to dynamic analysis. Multiple dynamic analysis tools, such as CopperDroid, already log the API calls made by an application under test. It would therefore be a relatively straightforward process to convert these sequences of API calls into a format similar to the one presented in this paper for classification by a CNN. As discussed in the literature review, tools such as NDROID already log not only API calls from ART, but also calls made from native libraries and communication

across the VM and underlying OS. This could allow the method described here to detect and classify malware that is launched from native code.

There are several potential advantages of moving from a static to a dynamic approach. Code that is hidden in other APK resources or downloaded from external sources must eventually be executed at run time. While the malicious exploit API calls made by these types of malware may not be observed in static analysis, they must be executed at run time to be successful and could then be detected and logged by a dynamic analysis tool.

Chapter 8 Conclusion

In this paper we have shown the effectiveness of using Deep Neural Networks and specifically Convolution Neural Networks to identify and classify malware. This approach represents one of the first attempts to apply deep learning techniques to Android Malware classification, as there are no other comparable approaches in the published literature. While this paper accomplishes the goal of proving the effectiveness of deep learning techniques to malware analysis, there is considerable room for improvement going forward. There are many potential benefits to continuing research in deep learning, drawing from the fundamental ability of deep learning techniques to automatically derive rich feature sets from the input data.

We demonstrated a parser capable of generating a representation of an APK file as a list of API calls. These API calls were then converted into a vector, segmented into smaller chunks, and used as the input to a CNN for classification. The CNN used two convolutional and pooling layers followed by a dense layer connected to a softmax output to classify each chunk of the API string representation. The classification applied to each of the API list segments were then averaged to obtain a final classification for the APK.

Experiments with six recent malware variants showed that the combination of the vector representation created by the APK parser in conjunction with the CNN classifier successfully detected malware files from benign applications nearly 100% of the time with 100% precision. Furthermore, through tests of classifying benign applications into categories based on their functionality, the CNN classifier showed that it achieves far better generalization of the functional relationship between then API calls than standard text classification methods such as n-grams classified by SVM or a Naïve Bayes classifier.

Finally, we discussed methods that could be used to expand upon the initial success demonstrated here in using deep learning techniques to analyze Android APKs. One of the key benefits of using a list of API calls in sequential order of execution as the input to the CNN classifier is that it allows for the classification techniques discussed in this paper to be easily adapted to work with dynamic techniques or to incorporate analysis of native code libraries in APK files. This means that two of the biggest weaknesses of the proposed approach, namely malware launched through native code and malware that is dynamically loaded after installation, could potential be addressed with minor modifications.

References

[1] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T.s. Chan. 2014. On Tracking Information Flows through JNI in Android Applications. 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks(2014). DOI:http://dx.doi.org/10.1109/dsn.2014.30

[2] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and accurate zero-day Android malware detection. Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12 (2012). DOI:http://dx.doi.org/10.1145/2307636.2307663

[3] Wenhao Fan, Yuan'An Liu, and Bihua Tang. 2015. An API Calls Monitoring-based Method for Effectively Detecting Malicious Repackaged Applications.IJSIA International Journal of Security and Its Applications 9, 8 (2015), 221–230. DOI:http://dx.doi.org/10.14257/ijsia.2015.9.8.19

[4] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. Computers & Security 51 (2015), 16–31. DOI:http://dx.doi.org/10.1016/j.cose.2015.02.007

[5] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Ainuddin Wahid Abdul Wahab. 2015. A review on feature selection in mobile malware detection.Digital Investigation 13 (2015), 22–37. DOI:http://dx.doi.org/10.1016/j.diin.2015.02.001

[6] Shina Sheen, R. Anitha, and V. Natarajan. 2015. Android based malware detection using a multifeature collaborative decision fusion approach.Neurocomputing 151 (2015), 905–912. DOI:http://dx.doi.org/10.1016/j.neucom.2014.10.004

[8] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing droids: Program Slicing for Smali Codes.Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13 (2013). DOI:http://dx.doi.org/10.1145/2480362.2480706

[9] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of android application security. In Proceedings of the 20th USENIX conference on Security (SEC'11). USENIX Association, Berkeley, CA, USA, 21-21.

[10] Long Short Term Memory Networks for Anomaly Detection in Time Series. Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, Puneet Agarwal. Proceedings, p. 89. Presses universitaires de Louvain, 2015.

[11] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.Proceedings 2015

Network and Distributed System Security Symposium(2015).
DOI:http://dx.doi.org/10.14722/ndss.2015.23145

[12] Santanu Kumar Dash et al. 2016. DroidScribe: Classifying Android Malware Based on Runtime Behavior. 2016 IEEE Security and Privacy Workshops (SPW)(2016). DOI:http://dx.doi.org/10.1109/spw.2016.25

[13] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, DroidNative: Semantic-Based Detection of Android Native Code Malware. http://arxiv.org/abs/1602.04693, 2016

[14] Luke Deshotels, Vivek Notani, and Arun Lakhotia. 2014. DroidLegacy.Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW'14 (2014). DOI:http://dx.doi.org/10.1145/2556464.2556467

[15] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, M.s. Gaur, and Vijay Ganmoor. 2015. AndroSimilar: Robust signature for detecting variants of Android malware. Journal of Information Security and Applications 22 (2015), 66–80. DOI:http://dx.doi.org/10.1016/j.jisa.2014.10.011

[16] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. DroidChameleon.Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13 (2013). DOI:http://dx.doi.org/10.1145/2484313.2484355

[17] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2014. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. IEEE Transactions on Information Forensics and Security 9, 1 (2014), 99–108. DOI:http://dx.doi.org/10.1109/tifs.2013.2290431

[18] Min Zheng, Patrick P.C. Lee, and John C.S. Lui. 2013. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. Detection of Intrusions and Malware, and Vulnerability Assessment Lecture Notes in Computer Science (2013), 82–101. DOI:http://dx.doi.org/10.1007/978-3-642-37300-8_5

R. Fedler, J. Schutte, M. Kulicke, On the Effectiveness of Malware ¨ Protection on Android, Tech. rep., Technical report, Fraunhofer AISEC, Berlin (2013

[19] Number of apps available in leading app stores 2016 - Statista. Retrieved October 24, 2016 from https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[20] TQSO - An overview on the Static Code Analysis approach in Software Development. Ivo Gomes, Pedro Morgado, Tiago Gomes, Rodrigo Moreira. Faculdade de Engenharia da Universidade do Porto, Portugal. 2009.

[21] Frances E. Allen. 1970. Control Flow Analysis. In Proceedings of a Symposium on Compiler Optimization (1970), pp. 1-19, doi:10.1145/800028.808479 Key: citeulike:1596451

[22] Thomas Reps. 2000. Undecidability of context-sensitive data-independence analysis. ACM Transactions on Programming Languages and Systems ACM Trans. Program. Lang. Syst. TOPLAS 22, 1 (January 2000), 162–186. DOI:http://dx.doi.org/10.1145/345099.345137

[23] Alexander Hanel. An Intro to Creating Anti-Virus Signatures: - blogspot.com. Retrieved October 24, 2016 from http://hooked-on-mnemonics.blogspot.com/2011/01/intro-to-creating-anti-virus-signatures.html

[24] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco. 2014. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Systems with Applications 41, 4 (2014), 1104–1117. DOI:http://dx.doi.org/10.1016/j.eswa.2013.07.106

[25] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine.Proceedings of the Seventh European Workshop on System Security - EuroSec '14 (2014). DOI:http://dx.doi.org/10.1145/2592791.2592796

[26] Common Weaknesses of Android Malware Analysis Frameworks

Lars Richter

[27] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016 (2016). DOI:http://dx.doi.org/10.1145/2931037.2931044

[28] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. 2012 Seventh Asia Joint Conference on Information Security (2012). DOI:http://dx.doi.org/10.1109/asiajcis.2012.18

[29] A study on obfuscation techniques for Android malware
Matteo Pomilia

[30] William Enck et al. 2014. TaintDroid. ACM Trans. Comput. Syst. TOCS ACM Transactions on Computer Systems 32, 2 (January 2014), 1–29. DOI:http://dx.doi.org/10.1145/2619091

[31] BELLARD, F. 2005. QEMU, a fast and portable dynamic translator. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference. 41-41.

[32] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. Presented as part of the 21st USENIX Security Symposium (USENIX Security 12. 569-584.

[33] Xiao Zhang, Yousra Aafer, Kailiang Ying, and Wenliang Du. 2016. Hey, You, Get Off of My Image: Detecting Data Residue in Android Images. Computer Security – ESORICS 2016 Lecture Notes in Computer Science (2016), 401–421. DOI:http://dx.doi.org/10.1007/978-3-319-45744-4_20

[34] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid.Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11 (2011). DOI:http://dx.doi.org/10.1145/2046614.2046619

[35] S Poeplau, Y Fratantonio, A Bianchi, C Kruegel, G Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)

[36] Shuying Liang, Andrew W. Keep, Thomas Gilray, Peter Aldous, and David Van Horn. Sound and Precise Malware Analysis for Android via ... Retrieved October 17, 2016 from http://arxiv.org/pdf/1311.4201v1

[37] Anon. Dalvik Executable format. Retrieved October 17, 2016 from https://source.android.com/devices/tech/dalvik/dex-format.html

[38] Tim Wyatt. Lookout Security Blog. Retrieved October 17, 2016 from https://blog.lookout.com/blog/2010/08/10/security-alert-first-android-sms-trojan-found-in-the-wild/

[39] Anon. Mobile OS market share 2016 | Statista. Retrieved October 17, 2016 from https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[40] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks ... Retrieved October 17, 2016 from https://www.researchgate.net/publication/268079628_how_transferable_are_features_in_deep_neural_networks

[41] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence 35, 8 (August 2013), 1798–1828.

[42] ürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. Neural Networks 61 (2015), 85–117. DOI:http://dx.doi.org/10.1016/j.neunet.2014.09.003

[43] John Callaham. 2015. Google says there are now 1.4 billion active Android devices worldwide. (February 2015). Retrieved October 19, 2016 from

http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide

[44] Anon. ART and Dalvik. Retrieved October 19, 2016 from https://source.android.com/devices/tech/dalvik/

[45] Grégoire Jacob, Hervé Debar, and Eric Filiol. 2008. Behavioral detection of malware: from a survey towards an established taxonomy. Journal in Computer Virology 4, 3 (2008), 251–266. DOI:http://dx.doi.org/10.1007/s11416-008-0086-0

[46] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014). DOI:http://dx.doi.org/10.1145/2568225.2568301

[47] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14* (2014). DOI:http://dx.doi.org/10.1145/2590296.2590325

[48] Android Development Team. Introduction to Android. Retrieved October 19, 2016 from https://developer.android.com/guide/index.html

[49] Theano: A Python framework for fast computation of mathematical expressions, Theano Development Team

[50] Eric Battenberg, Sander Dieleman, Daniel Nour, Eben Olson, Aäron van den Oord, Colin Raffel, Jan Schlüter, Søren Kaae Sønderby, et. al. Lasagne: First Release. 2015. http://dx.doi.org/10.5281/zenodo.27878. DOI: 10.5281/zenodo.27878

[51] Mila Parkour. contagio mobile. Retrieved October 19, 2016 from http://contagiomi-nidump.blogspot.com/

[52] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[53] Godfrey Nolan. Android Obfuscation Tools Comparison. Retrieved October 19, 2016 from http://riis.com/blog/android-obfuscation/

[54] Smardec Inc. Allatori Java Obfuscator - Android Obfuscation. Retrieved October 19, 2016 from http://www.allatori.com/features/android-obfuscation.html

[55] Android Development Team. Shrink Your Code and Resources. Retrieved October 19, 2016 from https://developer.android.com/studio/build/shrink-code.html

[56] Androguard: Reverse engineering, Malware analysis of Android applications ... and more. Retrieved October 19, 2016 from https://github.com/androguard

[57] ODA - The Online Disassembler. Retrieved October 21, 2016 from http://onlinedisassembler.com/odaweb

[58] Anon. IDA: About - Hex-Rays. Retrieved October 21, 2016 from https://www.hex-rays.com/products/ida/

[59] J.-M. Borello and L. Me. Code obfuscation techniques for metamorphic viruses. J. in Computer Virology, pages 211–220, 2008.

[60] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In NDSS, 2008.

[61] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In BWCCA, pages 297–300, 2010

[62] P.y. Simard, D. Steinkraus, and J.c. Platt. Best practices for convolutional neural net-works applied to visual document analysis. *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* DOI:http://dx.doi.org/10.1109/icdar.2003.1227801

[63] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of "Piggybacked" mobile applications. *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13* (2013). DOI:http://dx.doi.org/10.1145/2435349.2435377

[64] S. Alam, I. Traore, and I. Sogukpinar. Annotated Control Flow Graph for Metamorphic Malware Detection. The Computer Journal – Section D: Security in Computer Systems and Networks, 2014

[65] S. Alam, I. Sogukpinar, I. Traore, and R. Nigel Horspool. Sliding Window and Control Flow Weight for Metamorphic Malware Detection. Journal of Computer Virology and Hacking Techniques, 2014.

[66] Veo Zhang. 2016. 'GODLESS' Mobile Malware Uses Multiple Exploits to Root ... (June 2016). Retrieved October 24, 2016 from http://blog.trendmicro.com/trendlabs-security-intelligence/godless-mobile-malware-uses-multiple-exploits-root-devices/

[67] Checkpoint Security. 2016. From HummingBad to Worse report - blog.checkpoint.com. (July 2016). Retrieved October 24, 2016 from http://blog.checkpoint.com/wp-content/uploads/2016/07/hummingbad-research-report_final-62916.pdf

[68] Shivang Desai. 2016. Android Banker malware goes social | Zscaler Blog. (May 2016). Retrieved October 24, 2016 from https://www.zscaler.com/blogs/research/android-banker-malware-goes-social

[69] Morgan Marquis-Boire, John Scott-Railton, Claudio Guarnieri, and Katie Kleemola. 2014. Hacking Team's Tradecraft and Android Implant. (June 2014). Retrieved October 24, 2016 from https://citizenlab.org/2014/06/backdoor-hacking-teams-tradecraft-android-implant/

[70] Cong Zheng, Claud Xiao, and Zhi Xu. 2016. New Android Trojan "Xbot" Phishes Credit Cards and Bank ... (February 2016). Retrieved October 24, 2016 from http://researchcenter.paloaltonetworks.com/2016/02/new-android-trojan-xbot-phishes-credit-cards-and-bank-accounts-encrypts-devices-for-ransom/

[71] Jordan Pan. 2106. Fake Bank App Ramps Up Defensive Measures. (May 2106). Retrieved October 24, 2016 from http://blog.trendmicro.com/trendlabs-security-intelligence/fake-bank-app-phishes-credentials-locks-users-out/?utm_source=feedburner&utm_medium=feed&utm_campaign=feed: anti-malwareblog (trendlabs security intelligence blog)

[72] Ian Goodfellow, Quoc Le, Andrew Saxe, Honglak Lee, and Andrew Ng. Measuring invariances in deep networks - dl.acm.org. Retrieved October 24, 2016 from http://dl.acm.org/citation.cfm?id=2984166

[73] Kyunghee Kim. Measuring Occlusion Invariances in Deep Networks. Retrieved October 24, 2016 from http://cs229.stanford.edu/proj2010/kim-measuringocclusioninvariance-sindeepnetworks.pdf

[74] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.

[75] Yann LeCun and Yoshua Bengio. 1998. Convolutional networks for images, speech, and time series. In The handbook of brain theory and neural networks, Michael A. Arbib (Ed.). MIT Press, Cambridge, MA, USA 255-258.

[76] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. (2012) 1097–1105

[77] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. 2016. Deep Networks with Stochastic Depth. Computer Vision – ECCV 2016 Lecture Notes in Computer Science (2016), 646–661. DOI:http://dx.doi.org/10.1007/978-3-319-46493-0_39

[78] Simonyan, K., Zisserman, A.: Very deep convolutional networks for largescale image recognition. arXiv preprint arXiv:1409.1556 (2014), Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper

with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2015) 1–9

[79] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385 (2015)

[80] Tianjun Xiao. Walkthrough: AlexNet · dmlc/minerva Wiki · GitHub. Retrieved October 24, 2016 from https://github.com/dmlc/minerva/wiki/walkthrough:-alexnet

[81] Yoshua Bengio and Olivier Delalleau. 2011. On the Expressive Power of Deep Architectures. Lecture Notes in Computer Science Algorithmic Learning Theory (2011), 18–36. DOI:http://dx.doi.org/10.1007/978-3-642-24412-4_3

[82] Nadav Cohen, Or Sharir, and Amnon Shashua. On the Expressive Power of Deep Learning: A Tensor Analysis. Retrieved October 24, 2016 from http://jmlr.csail.mit.edu/proceedings/papers/v49/cohen16.pdf

[82] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. J. Mach. Learn. Res. 12 (July 2011), 2121-2159.

[83] Michael Nielsen. Neural Networks and Deep Learning, Determination Press 2015.

[84] Qianli Liao and Tomaso Poggio. Bridging the Gaps Between Residual Learning, Recurrent ... Retrieved October 24, 2016 from http://arxiv.org/abs/1604.03640

[85] Cicero Santos and Maira Gatti. Deep Convolutional Neural Networks for Sentiment Analysis ... Retrieved October 24, 2016 from http://www.aclweb.org/anthology/c14-1008

[86] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014). DOI:http://dx.doi.org/10.3115/v1/d14-1181

[87] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI'15). AAAI Press 2267-2273.

[88] Android-rename - npmjs.com. Retrieved October 25, 2016 from https://www.npmjs.com/package/android-rename

Vita

Robin Nix graduated from the Franklin W. Olin College of Engineering in Needham, Massachusetts in 2007 with a B.S. in Electrical and Computer Engineering. He received the Franklin W. Olin full tuition and room scholarship. He will receive his M.S. in Computer Science from the Louisiana State University in Baton Rouge, Louisiana in 2017. His primary research interests are in deep learning and cybersecurity.