Spring 2018

Analyzing Android Adware

Supraja Suresh
San Jose State University

Analyzing Android Adware

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Supraja Suresh

May 2018

The Designated Project Committee Approves the Project Titled

Analyzing Android Adware

by

Supraja Suresh

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2018

Dr. Mark Stamp	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science
Fabio Di Troia	Department of Mathematics

ABSTRACT

Analyzing Android Adware

by Supraja Suresh

Most Android smartphone apps are free; in order to generate revenue, the app developers embed ad libraries so that advertisements are displayed when the app is being used. Billions of dollars are lost annually due to ad fraud. In this research, we propose a machine learning based scheme to detect Android adware based on static and dynamic features. We collect static features from the manifest file, while dynamic features are obtained from network traffic. Using these features, we initially classify Android applications into broad categories (e.g., adware and benign) and then further classify each application into a more specific family. We employ a variety of machine learning techniques including neural networks, random forests, adaboost and support vector machines.

ACKNOWLEDGMENTS

I would like to thank Dr.Mark Stamp for his guidance and support throughout the project and my degree program. I am grateful to Fabio Di Troia for his suggestions to improvise the project and to Dr. Katerina Potika for being helpful and for her valuable time.

I would also like to thank my family and friends for being my strength and support throughout the course of my Master's degree.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Advertisements are used to promote or sell a product, an idea or a service. The advent of the internet, and later the smartphone, has pushed marketing strategies towards digitizing advertisements as it's the new norm and the digital channel is rapidly growing with no signs of slowing down [6]. These ads are displayed while browsing a website online or while using a mobile application. A majority of the smartphones are built on the Android platform which led to the growth of a huge ecosystem of apps. This rapid growth gave rise to the increase in the apps with malicious intent. Most of the apps are available for free use. Based on statistics from the 4th quarter of 2017, 93.99% of the apps available in Google playstore are available for free [7].

The major revenue generators for several free mobile apps and online web services are through advertisements, but they are plagued by fraudulent activities [8]. Ads fetched from ad providers are launched by interleaved ad libraries in the web page or in the mobile apps and are displayed to the users. The ad providers pay the developers of the app or the web page certain amount for each ad impression, i.e., each ad that is fetched and displayed to the user. In addition, the developer gets paid for each click the user makes on the displayed ads. Unfortunately, the problem of ad fraud has been disrupting the advertising industry for a long time. When the developers include code that fetches an ad but does not display it to the user, or when the app clicks on an ad without user activity, this is fraudulent activity [9]. Software that commits ad fraud is called Adware.

Cybercriminals profit from the ad industry and it affects the users and businesses in several ways. Adware can be structured to steal sensitive information from a user's phone and pass it on to third parties. Adware not only displays annoying pop-up messages, but attackers can root users phone and gain access to the devices [10]. Adware can be also used for other purposes, such as performing DoS attacks.

There are slight differences in how ad fraud is committed on the web and on mobile devices. In the web context, ad fraud is often perpetrated by botnets, which are collections of compromised user machines called bots. Fraudsters issue fabricated impressions and clicks using bots so that the traffic they generate is varied (i.e., by IP address), making such fraud difficult to detect [8]. The user system has to be infected with malware so that it acts as a bot and receives a command from a central server to display ads and clicks on them automatically. In the mobile world, at most one application runs in the foreground, while several can run in the background. If the application running in the background fetches ads or clicks on the ads, it is committing ad fraud. Fraudulent activity also includes the case in which the application fetches the ad but does not display it to the user, or clicks on the ad automatically. Such fraud can be caused by code in the application which could turn the phone into a bot by downloading modules that infect the device and display ads based on commands from the botnet command and control server.

The main purpose of this research is to analyze the use of machine learning as a possible solution to detect adware. This can be done by classifying the apps based on the features obtained from the static and dynamic analysis. Static features may involve features extracted from Java byte code or from the Android manifest file, AndroidManifest.xml. As static analysis cannot detect dynamic code injection or loading we also extract dynamic information. Dynamic analysis can be done by

running the code and monitoring API calls, network traffic, and other similar aspects. In this paper, we consider dynamic information obtained by monitoring network traffic while executing the application. These features can be used for isolating fraudulent traffic from legitimate traffic and identifying different families of adware.

The remainder of the report is organized as follows. Chapter 2 gives a basic overview of advertising in the web and mobile domains and detection mechanisms. Chapter 3 discusses the proposed methodology. Chapter 4 explains the experiments performed and the results obtained while Chapter 5 outlines future enhancements.

CHAPTER 2

Background

This chapter provides the needed background information related to advertising, android, adware, ideas and existing solution to detect adware and the basic machine learning techniques used in the research.

2.1 Overview of Advertising in the Web Domain

Advertising in the web involves ads displayed as a part of the website to the user. The website owner called the publisher embeds ads in the website using libraries provided by the third party ad providers in the <iframe> or <script> tags whose src attribute points to the ad provider's ad servers [8]. These ad providers are responsible for finding and selecting advertisements and they also pay the publishers for the ads displayed. When a user loads the website, an ad request is made along with the publisher id and user related information so that appropriate ads can be selected as, different ads target different groups of users. The response to this contains the pixel URL, click URL and content URL. The content is provided by the ad provider itself and the marketers who want their ads to be displayed via a provider use a tracking pixel to track the ads so that the marketers are not charged for fraudulent activities by the provider [8]. The click URL contains the corresponding page that will be loaded when an ad is clicked.

The web front is plagued by several types of ad fraud due to botnets, ghost sites, ad stacking or purchased traffic. Bots are programs on normal user computers that can be controlled by a central Command and Control server (C&C). The C&C server together with the bots form a botnet which can be used to click on ads on a web

page or generate excessive web traffic and overwhelm a server. Ghost sites are legit websites whose content are old or copied from other websites and they have several ad slots. These sites are usually not visited by the users but by the bots generating ad impressions with no return on investment. At a given time only one ad can be viewed and stack ads on top of the other generate multiple ad impressions but only one ad is displayed to the user. In addition to ghost sites, bots can also visit legitimate and popular websites and view ads thus creating fraudulent impressions [11]. This traffic can be purchased by the publishers thus increasing their revenue.

2.2 Overview of Advertising in the Mobile Domain

Mobile advertising usually comprises of ads displayed in android apps most of which are freely available in numerous app stores. The app developer generates revenue when ads are displayed to the user and when the user clicks the ads. To achieve this, the app developer obtains a publisher id by registering to an android ad provider and embeds the ad library provided in the application code. This library takes care of fetching the ads through an HTTP request and displays it to the user after receiving the pixel, click and content URL from the ad server. Most ad libraries implement the requesting ad and display it by simply loading an ad element in a web view [8].

Advertising in smartphones also suffers from ad fraud in the form of fraudulent impressions, auto-clicking, fat finger fraud, multiple sdk libraries and pixel stuffing. In an app, it is possible to have several sdks from different ad providers and ad networks but ideally, only one of the ads will be displayed in a spot; the rest being counted as fraudulent impressions. Pixel stuffing occurs when an ad, say, of dimensions 1024×480 pixels are crammed to a 1×1 pixel on the app screen which is not visible to the user.

Pixel stuffing also occurs in the web domain where it is also termed iframe stuffing. Fat finger fraud occurs when an ad is placed near the navigation buttons in a phone with the intention to make the user accidentally click it. It is also a common practice in mobile applications to give a reward like an extra life in a game or extra points for watching an ad. The user could open the ad and even click on it without being interested in the content, this user behavior is not addressed as a part of the research.

Figure 1: Ad Fraud Common Examples [1]

2.3 Overview of Android

Android is one of the most popular operating systems to run applications on the mobile phone. It is an open source project, with Google as one of its major

contributors. It allows apps developed on this platform to run on different hardware devices provided it has Android installed. Figure 2 shows the major components in the architecture of an Android operating system.

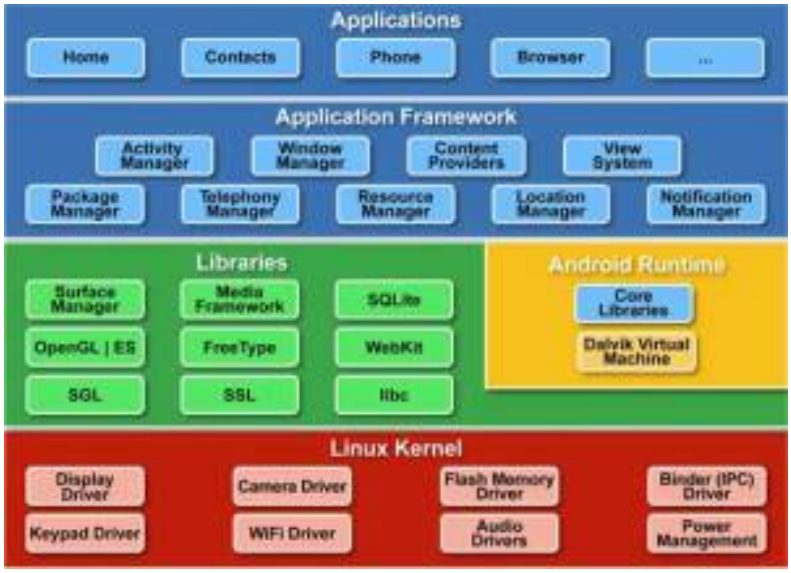


Figure 2: Android Architecture [2]

It is built on top of Linux and provides abstractions needed to interact with the hardware components like keyboard, camera, and screen. It contains several libraries on top of it for storage of information, web browsing, working with audio and other functions. Several java-based android libraries are available for development purposes, a developer can use *app* library to access the application, *webkit* library to incorporate browsing functionalities into the application, *text* to control the displayed text and *view* for building the interface the user interacts with to name a few.

The *Android Runtime* contains the core libraries and also provisions a Dalvik Virtual Machine (DVM) which is similar to java virtual machine that has been improvised for android. The Dalvik bytecode generated from java bytecode of an application is run on this DVM. It is worthy to note that each application in android runs in its own DVM. The *Application Framework* layer exposes several classes that can be

used by the applications to perform needed functions. Some of the available classes include activity manager, content provider, and notification manager. *Applications* are built on top of the application framework and it provides the interface for the user. It comes packaged as a .apk file with the needed libraries, AndroidManifest.xml, and classes.dex file. While the manifest file contains the list of permissions, content providers, and resources used, the classes.dex file contains the packaged application code without duplicated sections.

2.4 Android Adware

Adware is a malicious software that presents the user with unwanted ads. This often annoys the user more than causing harm. The main target of adware is to generate revenue for the publisher, some types of adware also collect information regarding the user and his activities without user's consent. This type of adware is also called spyware. The behavior of an adware depends on the family the application belongs to, the recent adware families used in this research are described below.

2.4.1 GhostClicker

Applications belonging to the GhostClicker family are known for automatically clicking the ads once downloaded. According to Trendlabs, the auto clicking routines are embedded in the Facebook Ad software development kit (SDK) and in Google Mobile Services (GMS) as a legit logs package thus avoiding suspicion. It also tries to evade detection by checking the system property and triggering the auto clicker routine only when it is not nexus as android emulators used for detecting malware usually are termed as "Nexus XXX". Some of the apps also request device admin permission which when provided makes it difficult to uninstall. The auto clicker is

not implemented in javascript code but inserts itself into Google's advertising platform and gets the ad location and simulates clicking [12].

2.4.2 Judy

In 2017, researchers from Checkpoint crawled Google play and identified around 41 applications that automatically clicked on ads and termed this family as Judy. All these applications were from a Korean firm and it invaded Google Plays Bouncer, a program that prevents malicious apps from entering play store. Once the app was installed it relies on C&C server for operation. The C&C server gives a user agent, javascript code and url's. The URL's are opened using the user agent and then the javascript code takes care of clicking on the ads [13].

2.4.3 Copycat

Copycat is an adware campaign identified by researchers at Checkpoint that reached its peak during April and May 2016. It affected around 14 million devices worldwide, rooted around 8 million devices generating around $1.5 million in revenue to the group [14]. The infected apps rooted the user's device and gave control of the user's device to the attackers. It steals revenue in two ways, first by substituting the referrerID with a fraudulent id when apps are downloaded and the second by displaying unwanted ads based on conditions. It was the initial family of adware that injected code to the Zygote process.

2.4.4 HummingBad

HummingBad installs a rootkit to the device and it generates revenue for the attackers by installing fraudulent apps and generates fraudulent ad revenue. The

malicious components are encrypted making it harder to detect and it uses two attack vectors so that if one fails the other can meet the objective. The apps look for certain events like a timeout or whether the screen is on and then it roots the device and tries to connect to the C&C center. The other attack strategy kicks in when it does not get the root privileges. At first, it tries social engineering methods through a component called 'qs' that gets decrypted and connects to the C&C server from where it can download apps, initiate referrer requests for generating Google Play advertisement revenue or launch applications [15].

2.4.5 Hamob

Hamob is an adware that usually does not do much harm other than display ads when installed in the device [16]. It captures information about the user and then uses it to send a large number of ads and annoys the user.

2.4.6 MarsDaemon/Marswin

Once installed the infected app displays ads and will not stop even when the application is force stopped. It spawns several processes and creates a file and locks it [17]. The processes check the file and see if they are locked and start a process when a related process is dead. This ensures that the ad libraries can inject apps by not killing the apps even when forced.

2.4.7 Origin

Origin is a family of applications that poses to be benign but in the background sends user information to a C&C server and delivers an ad to the user. Thus it acts as a trojan and adware. Email, appid, gcmid ,imei identifier are some of the information

that is sent to the server and based on the command received it can show banner ads, redirect to facebook or chrome and installing shortcuts on the screen. At least 100 applications are estimated to be affected by this adware according to analysts at Dr.Web. As the behavior depends on several factors it makes it hard to detect.

2.5 Related Work

In the web advertising realm, Daswani provided a detailed study of the clickbot architecture and various techniques for detection to help other researchers interested in this area. Detection techniques for fabricated impression, clicks, and duplicate clicks (where the publisher clicks the same ad many times) have been proposed in [18, 19] but these techniques fail when trying to combat botnets.

Considerably less research work has been done with a focus on ad fraud through mobile devices. Security companies like TrendMicro, Checkpoint, and others identify fraudulent apps in the Google Play store and report them to be removed so that it does not affect further users. By then, the damage is already done and we need mechanisms to detect before it affects the users. On this front, Liu [20] made a significant contribution by trying to identify fraud through ad stacking and hidden ad rendering. The experiment used a technique to analyze UI of the apps and then detect but it fails when ad traffic is generated in the background.

Initial work in this field was done by Miller and his team where they performed a detailed analysis of clickbot families Fiesta and 7cy [21]. This paved the way to understand the working of a clickbot. In the recent years, Crussell [8] proposed an analysis tool called MADFraud which detected fraudulent impressions and click after running the apps automatically. This approach used machine learning for identifying ad requests and used HTTP request trees and heuristics to detect adfraud. On the

other hand, Arp and his team proposed a lightweight method to detect android malware using only static features fed to a machine learning model [22]. They gathered extensive static features and used SVM for classification but it does not take into account the features from dynamic analysis. This is a drawback as attacks due to code transformation appear only during dynamic analysis and cannot be detected by this mechanism.

Grace and team [23] analyzed the permissions used by in-app ad-libraries to determine the potential risks caused by these libraries. The authors from [24] used a two order risk analysis scheme to identify and classify malware on basis of risks. The first order depends on permissions of an application while the second order uses certain heuristics to identify risky applications. Researchers in [25] also used permissions to identify malicious applications.

Detection using dynamic analysis has also been performed earlier on the apps. The authors of TaintDroid [26] and DroidScope [27] performed detailed analysis of the applications by running in a controlled environment but are complex to be deployed on the mobile phones. Authors of [28] detected malware using network traffic features and achieved accuracy above 90% using 8 distinguishing features. More recently, a network-based android malware detection and characterization mechanism was proposed by Lashkari [29] and his team which aims to segregate malware, adware, and benign apps. They proposed that 9 traffic features are needed for classifying the apps using machine learning. Though this provided a good method it had few limitations in data collection method. An efficient algorithm to minimize the number of network traffic based features to detect malware was introduced in [30]. Several deep learning methods were proposed to segregate malware based on network traffic like [31], [32] and [33] outlined a method to use deep learning using static features. We employ

several static and dynamic methods to derieve features and propose a solution to detect adware and classify them according to their families using machine learning techniques.

2.6 Machine Learning Models

Machine learning relates to the field where a computer can learn by itself given some data without programming for it explicitly [34]. It involves training a model or an algorithm with data that has the final outcome, this is called *Supervised Learning*. When the model is trained with data without providing the final outcome it is called *Unsupervised Learning*. For this research, we know that if a given apk is an adware or not so we use supervised learning models like random forest, adaboost, support vector machine (SVM) and a deep learning based model called Multilayer Perceptron (MLP).

2.6.1 Random Forest

Random Forest algorithm is an ensemble supervised classification algorithm that combines several weak learners to form a strong learner. The weak learners used here are Decision Trees. It is a non-linear model and in addition to using boosting it also uses a bagging approach, here not only are the samples chosen at random for each decision tree but also the features are chosen at random. This randomness prevents the problem of overfitting that the decision trees face.

In simpler terms, for a dataset with 5 samples X_1,X_2,X_3,X_4,X_5 with labels L_1,L_2,L_1,L_2,L_1 and features f_1,f_2,f_3 each. Then Random Forest may create 3 weak

classifiers with inputs as

$$\text{Weak classifier 1: } (W_1) = [X_1, X_3, X_5] \text{ with features } f_1, f_2$$

$$\text{Weak classifier 2: } (W_2) = [X_2, X_4, X_5] \text{ with features } f_2, f_3$$

$$\text{Weak classifier 3: } (W_3) = [X_1, X_2, X_3] \text{ with features } f_1, f_3$$

Here we can see that each weak learner takes a subset of the input with overlap and only a subset of the features is used. After training the final outcome of a test sample is the majority outcome of those of the weak learners. Say for a test sample X_6 the 3 weak classifiers output 1,-1,1 then, the final outcome is 1. An added advantage of this ensemble model is that it can also show the importance of each feature that can be used for gaining various insights.

Figure 3 shows the steps in prediciting the label of a given input

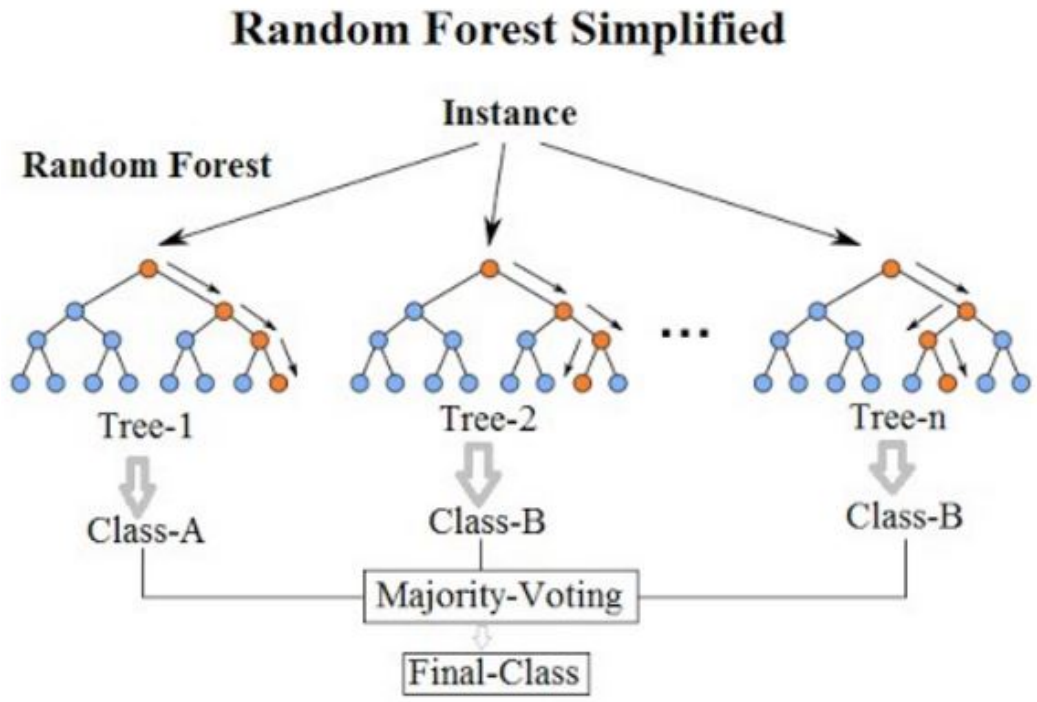


Figure 3: Predicting a label using Random Forest [3].

2.6.2 Adaboost

The Adaboost algorithm was the pioneer in boosting based Machine Learning models. It improves the prediction accuracy by considering a number of inaccurate results and combining them. Similar to random forest the weak classifiers are trained on a subset, with overlap, of the training data and each has a weight associated to it that determines the probability of its occurrence in the subset. The weight of an entry is increased when it is misclassified so that the next classifier can perform well on these misclassified samples. While combining the results of the weak classifiers, ones with higher accuracy are given more weights which inturn influences the final prediction. The weighted training samples are used to train the weak classifiers which are added one by one till a pre-defined number of classifiers or when no more improvement can be achieved during training. In the end, it has a set of weak classifiers each with a stage value which decides the final output. For Example, consider the output labels for the 5 weak classifiers to be 1,-1,1,-1,1 then the output can be expected to be 1 as that's the maximum value, but in adaboost the final output is linear weighted combination of the predicted values. If each stage weight is 0.3,0.9,0.2,0.7,0.5 and using a linear combination we get -0.6 which gives the final output as -1. Unlike random forests, adaboost can be used with any classifier like decision trees, SVC and others.

2.6.3 Support Vector Machines

Support Vector Machine is a supervised algorithm that works by identifying the optimal hyperplane. It accepts an n-dimensional input and then it generates an n-1 dimensional hyperplane to classify the samples and predict the label. For a 2-dimensional input with 2 labels the hyperplane is a line that separates one class of

labels from the other. SVM is based on the following concepts :

Maximizing the Margin: For a binary classification with 2-dimensional input in Figure 4, the hyperplane is the solid yellow line and the margin is formed by the dashed line that represents the minimum distance between the hyperplane and a sample from the training set. The main aim in SVM is to maximize the margin and the solid black lines to the hyperplane are called the support vectors.

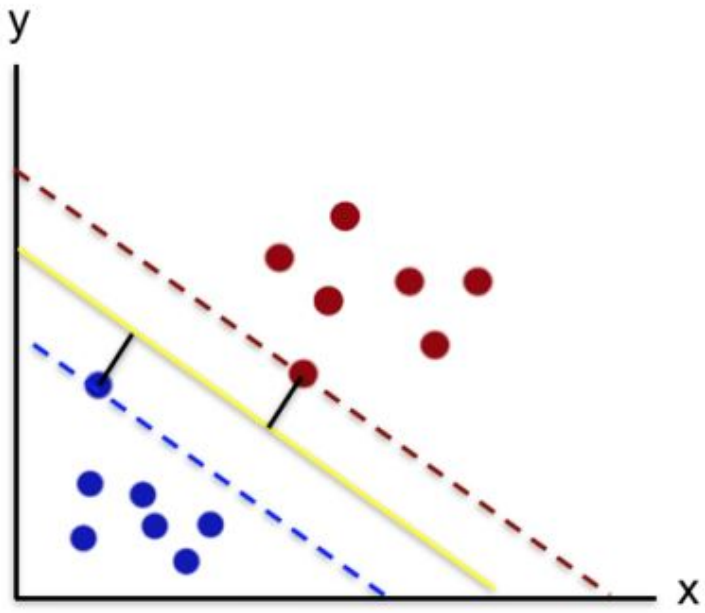


Figure 4: SVM Line Hyperplane and Maximum Margin of Separation.

Working in a higher dimensional space: Hyperplane can be identified and used for separation only when the data is linearly separable. When the data is not linearly separable, to classify the training set we first need to map the input space to a higher dimensional feature space where the data is linearly separable; and then identify the separating hyperplane. In Figure 5 we can see that for a non-linearly separable input data when a transformation is applied it becomes linearly separable.

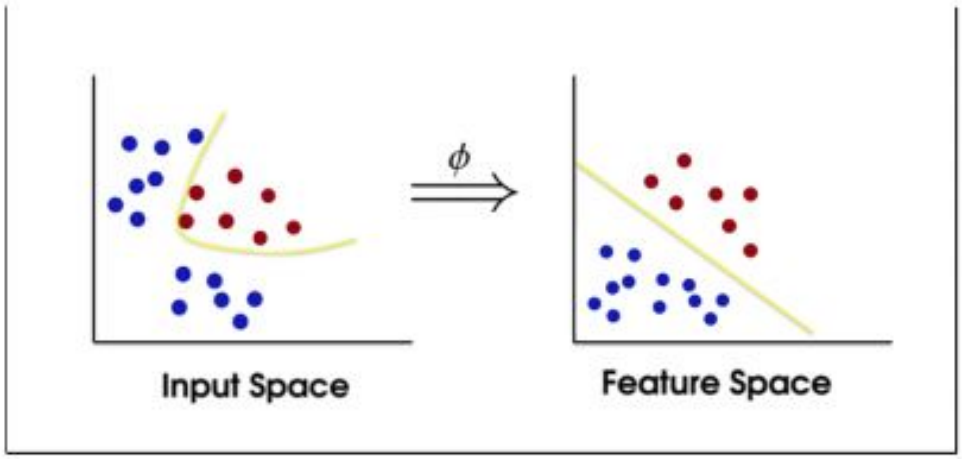


Figure 5: Input Space to Higher Dimensional Feature Space.

Kernel trick: The mapping function that makes the non-linearly separable data to linearly separable is called the kernel trick.

2.6.4 Deep Learning

In this experiment we used Multilayer Perceptron (MLP), a deep learning model. It consists of a network of perceptron/neurons. The output of a perceptron is a linear combination of inputs based on weights passed through an activation function.

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b_i\right) \tag{1}$$

In Equation 1, x forms the input vector, w the weight vector, b the bias vector and the ϕ forms the activation function. Figure 6 shows a single flow of data through a perceptron.

Several activation functions like RELU, logistic sigmoid, hyperbolic tangent are available. These functions add the needed non-linearity for making the network more

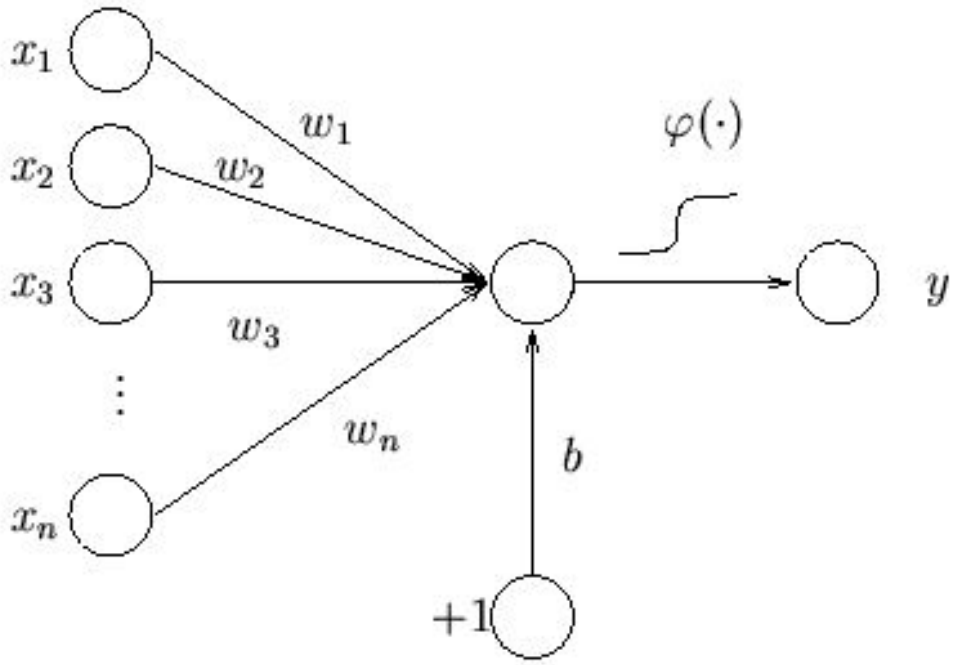


Figure 6: Single Perceptron Flow [4]

powerful. For our experiments, we used hyperbolic tangent which squishes the input to be within -1 to 1. It is given by

$$\tanh(x) = \frac{2}{1 + \exp^{-2x}} - 1$$

.

A single layer perceptron is less effective and hence multiple layers are stacked to form a multilayer perceptron. It usually has one *Input Layer*, several *Hidden Layers* and an *Output Layer* [35]. The data passes from one layer to the other and is subjected to several transformations after which it is given as the output.

MLP, when used for a supervised algorithm, tries to understand the relationship between the input and labels of a training set through transformations and then tries to predict the label for a test set. It relies on *Backpropagation Algorithm* for updating

the weights. It has two steps

1. *Forward Pass:* The inputs are used to calculate the output values using Equation 1

2. *Backward Pass:* The weights are adjusted by backpropagating the partial derivatives of the cost function and the error derivatives. This can be achieved by simply applying the chain rule.

Figure 7 shows an MLP with 2 hidden layers, one input, and one output layer. It can also be considered as fully connected layers as the outputs from all the nodes in a layer are propagated to the next.

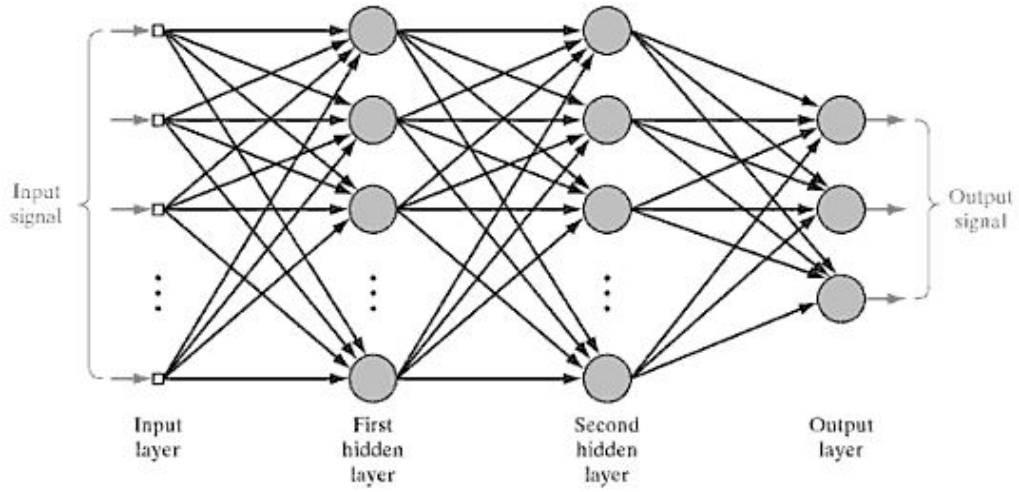


Figure 7: MLP with 2 Hidden Layers [5]

The error function used here in the experiment is Cross-Entropy Loss Function along with Softmax layer. This allows the output to be treated as a probability measure of a sample belonging to a class. Cross-Entropy is a distance measure between probability distributions. The activations from the output layer are normalized using

softmax given by $s(o_j)$ where

$$s(o_j) = \frac{\exp(o_j(x))}{\sum_k \exp(o_k(x))}$$

Training involves updating the weight vectors through backpropagation and once it is completed it can be used to predict the label for the test set. Thus from the output of the softmax layer we obtain the class the test sample belongs to.

CHAPTER 3

Methodology

This section initially describes the dataset collected and used. Then it describes the procedure to collect and extract the features from static and dynamic analysis.

3.1 Dataset

The dataset used for this experiment was mostly collected manually. We initially identified android adware families in the recent times from blogposts which listed the infected apps package name or SHA keys or both. Then we downloaded the apks and verified if it is infected or not on VirusTotal, an online tool that runs different antivirus engines, websites and URL scanners and gives an aggregate result. Apps for the adware families Hamob and Copycat the apks were obtained from the Drebin dataset [22] and then verified to be malicious and then used. The benign apps were obtained by first choosing apps from the playstore with and without ads, downloading them and then making sure the apps are not infected in VirusTotal. A total of 266 apps were used in this experiment; the application family names and the count of apps in each family are shown in Table 1.

3.2 Feature Collection and Extraction

The features collected and extracted play an important role in determining and improving the model accuracy. The methodology used to extract the static and dynamic features are described in this section.

Table 1: Android Application Family and their Count

Family Name	Count
Judy	45
GhostClicker	29
Origin	20
HummingBad	20
Hamob	16
CopyCat	12
Marswin	29
Benign without ads	48
Benign with ads	47

3.2.1 Static Features

Android applications distributed as .apk files contain AndroidManifest.xml, classes.dex and other resources. Static analysis is a part of reverse engineering the apk files without executing it. Static analysis of an apk involves analyzing the code segments of the apk without running it on an emulator. It is difficult to detect obfuscated code and dynamic code loading using this method. The major advantage of static analysis is that the cost of computation is low and requires low resources. Static analysis was conducted using a tool called *ApkAnalyzer* [36], a java tool to extract detailed information about the APK files. This in turn uses apktool to generate a JSON describing the permissions, activities, services used and other information. The tool gets information in several categories like:

1. *Basic Metadata* — The file name, size, source of download, size of compiled dex and arc sizes

2. *Manifest Metadata* — The number of activities, listed and used permissions, services, content providers and broadcast receivers, min, max and targeted sdk

version and supported screen size.

3. *Certificate Metadata* — Signing algorithm used, issuer name, start and end date, MD5 public key and hash of the certificate

4. *Resources Metadata* — The number of xml, gif, jpg, idpi, hdpi, layouts and menus used to name a few.

5. *File Hashes* — Hashes of the dex files, resources, drawables, layout and all other files in Manifest.MF

Permissions are one of the most important features in detecting malicious apk. In addition to the permission we also used the activities as it represents the entry point for user interaction. An app can have several activities and an activity of one app can be used by another app if allowed. Next, the number of services was extracted as it represents the long-running operations that run in the background for an app. The broadcast receivers allow the apps to react to broadcast notifications or announcements. The content provider manages the app data like the user's contact information on an android phone. It can also be used to store and retrieve data private to an app.

One approach would have been to list all the values of the components mentioned above and construct a feature vector, it would have been of very high dimensions. Instead of building a high dimensional feature vector and reducing it, we wanted to try and see if just using the count of these components like permissions, activities and so on helps in identifying adware. The static features used for the experiments are as shown in Table 2 .

For each app in the dataset, the ApkAnalyzer is used on the file to obtain the

Table 2: Extracted Static Features

Number	Static Feature Name
1	numberOfActivities
2	numberOfServices
3	numberOfContentProviders
4	numberOfBroadcastReceivers
5	numberOfPermissions

JSON file with app details. A custom parser to extract the above-mentioned features were used on the JSON file and the static features were extracted and normalized. The steps involved in extracting static features are shown in the below Figure 8 and the sample partial json for adware, benign with and without ads are shown in Figure 9:

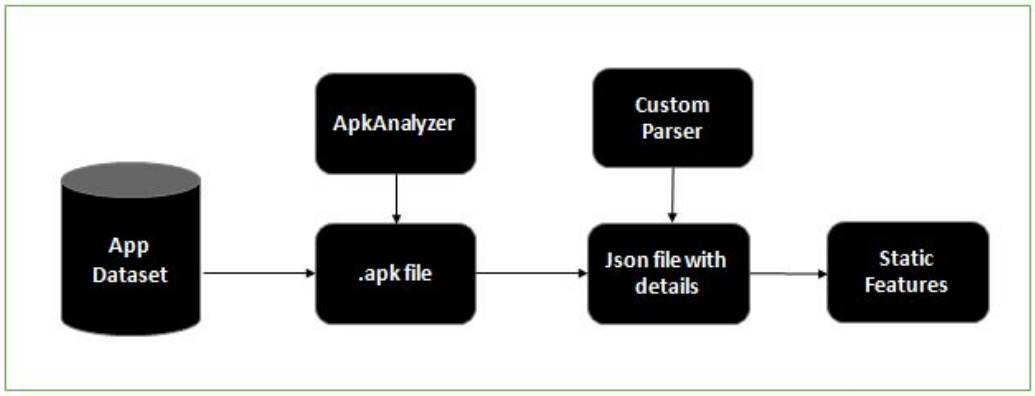


Figure 8: Flow Chart of Static Feature Extraction

3.2.2 Dynamic Features

Dynamic analysis involves running the apk in a real/emulated environment and collecting information. It can detect dynamic code loading and gather application information. The dynamic analysis could also involve monitoring the system call or network traffic and the like. As the ads are from the provider the proposed methodology monitors network traffic and tries to extract features based on the flow and count

```
{
  "fileName": "com.dictamp.englishpoems.apk",
  "fileSize": 50656568,
  "dexSize": 8532980,
  "arscSize": 888740,
  "androidManifest": {
    "packageName": "com.dictamp.englishpoems",
    "versionCode": "7",
    "installLocation": "auto",
    "numberOfActivities": 7,
    "numberOfServices": 3,
    "numberOfContentProviders": 1,
    "numberOfBroadcastReceivers": 2,
    "namesOfActivities": [
      "com.dictamp.mainmodel.SettingActivity",
      "com.dictamp.mainmodel.HomeActivity",
      "com.dictamp.mainmodel.SplashScreenActivity",
      "com.dropbox.core.android.AuthActivity",
      "com.google.android.gms.ads.AdActivity",
      "com.dictamp.mainmodel.MainActivity",
      "com.google.android.gms.common.api.GoogleApiActivity"
    ],
    "namesOfServices": [
      "com.google.android.gms.analytics.AnalyticsService",
      "com.google.android.gms.analytics.CampaignTrackingService",
      "com.google.android.gms.analytics.AnalyticsJobService"
    ],
```


Benign app with ads


```
{
  "fileName": "Animal Judy Dog care_v1.250_apkpure.com.apk",
  "fileSize": 33100917,
  "dexSize": 7215668,
  "arscSize": 57464,
  "androidManifest": {
    "packageName": "air.com.eni.AnimalJudy002",
    "versionCode": "1250000",
    "installLocation": "auto",
    "numberOfActivities": 14,
    "numberOfServices": 3,
    "numberOfContentProviders": 0,
    "numberOfBroadcastReceivers": 6,
    "namesOfActivities": [
      ".AppEntry",
      "com.google.android.gms.ads.AdActivity",
      "com.inmobi.rendering.InMobiAdActivity",
      "com.tnkfactory.ad.AdWallActivity",
      "com.tnkfactory.ad.AdMediaActivity",
      "com.igaworks.adpopcorn.activity.ApOfferWallActivity_NT",
      "com.igaworks.adpopcorn.activity.ApBridgeActivity_NT",
      "com.igaworks.adpopcorn.activity.ApBridgeActivity",
      "com.igaworks.adpopcorn.activity.ApCSActivity_NT",
      "com.igaworks.adpopcorn.activity.ApVideoAdActivity",
      "com.nextapps.naswall.NASWallBrowser",
      "com.nextapps.naswall.NASWall",
      "com.pozirk.payment.BillingActivity",
      "com.unity3d.ads.android.view.UnityAdsFullscreenActivity"
    ],
    "namesOfServices": [
```


Adware

Figure 9: Partial Json Result of Apps from ApkAnalyser

information. The apk was run on an Android Emulator called Bluestacks and then the network traffic was captured using Wireshark in a Windows Virtual Box.

Steps:

1. Start Bluestacks and attach it as one of the emulators using ADB command – adb connect 127.0.0.1:5555.
2. Store the apks in an input folder and pass the python script to Monkeyrunner to automate installation.
3. Monkeyrunner picks each apk and installs it in bluestacks and starts the application.
4. The python script also captures the network traffic using wireshark by running the application for 30 minutes when the user is interacting with the app manually.
5. After the desired time, wireshark is stopped and the apk is uninstalled and moves on to the next apk.
6. This process continues for each of the apks in the input folder.

The features from network analysis are extracted on several basis:

1. *Packet-Based*: Packet-based features are used to capture the packet statistics like number of packets sent, number of packets received, the total number of packets exchanged and the ratio of packets received to packets sent. We also focused on DNS and HTTP protocol based packets as ads requests are served over HTTP and ad domains are resolved using DNS in certain cases. *DNS*

related - features like the number of DNS requests, DNS responses, the max, min and average value of DNS answer length is also accounted for. *HTTP related* - features like the number of HTTP requests, max, min and average header length. Percentage of TCP, UDP, DNS requests out of the total packets is also extracted.

2. *Time-Based*: Time based features have been proved effective in classifying the network traffic [37] and using it as a basis we extracted features like the time the flow was active (active time), the time the flow was idle (idle time), the average number of packets per second and the average length of the packets in both the directions per second.

3. *Flow-Based*: A flow is a sequence of packets with the same source and destination, IP and port along with the same protocol. The flow-based features include number of flows and ratio of number of IP from which packets were received to the number of IP to which packets were sent to.

4. *Byte-Based*: This captures the information transferred between the sender and receiver and so the min, max and average size of the packets and the payload of HTTP requests was extracted as byte-related features.

The pcap file thus generated also has few packets other than that of those from the application which was filtered out. The features as shown in Table 4 were extracted with a python script using scapy, a powerful package manipulation program in python. Scapy can perform tasks like scanning, filtering, probing, attacks or network discovery [38]. The steps involved in extracting static features are shown in the below Figure 10.

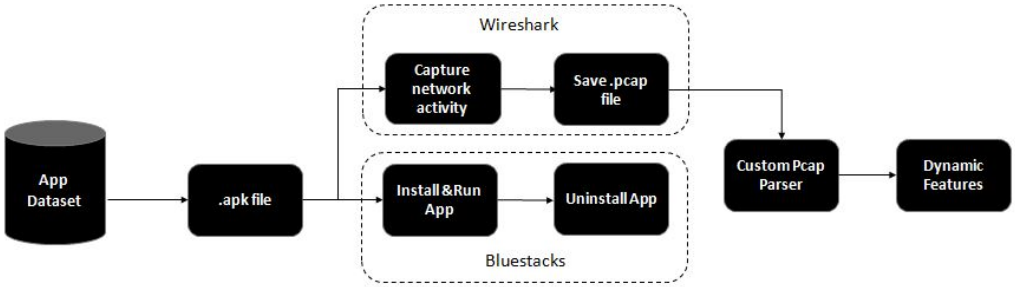


Figure 10: Flow Chart of Dynamic Feature Extraction.

3.3 Recursive Feature Elimination

One can get a varied number of features for a task but one of the major questions plaguing the machine learning field is ' how many features is enough ?' to effectively and accurately address the task at hand. Not all features generated will be important, we tried to identify the optimal number of features needed to identify adware by using RFE with ten-fold cross-validation. We started with using the most important features, then with $2, 3, \ldots, 37$ features and identified the number of features that gave better results. The major advantage of using RFE is that it takes an algorithm that gives feature importance and uses the specified number of features to train and test. If for example, we specified the number of features to be used as 3 then RFE runs and gets the most important feature, then runs the model with the remaining features to identify the most important feature from the feature subset and includes it; then does the same till the required features are identified. The major advantage of using RFE is that it does not remove several features at a time as it would give sub-optimal features. This approach of removing features recursively by building a model on the remaining subset of features is proved to be effective.

Table 3: Extracted Dynamic Features

S.No	Dynamic Feature Name	Description
1	dnsReqCount	No of DNS requests
2	dnsResCount	No of DNS responses
3	dnsAnswerLength	Min, Max and Avg DNS answer length
4	PayloadHttp	Min, Max and Avg payload size of the HTTP requests
5	noOfPkt	Total no of packets exchanged
6	httpCount	No of HTTP requests/responses sent or received
7	HeaderLength	Min, Max and Avg header length in the HTTP requests made
8	PktSize	Min, Max and Avg size of the packets
9	noOfPktsSent	No of packets sent
10	noOfPktsReceived	No of Packets received
11	Percentage of TCP, UDP, DNS	Percentage of the TCP,UDP and DNS requests made out of the total values
12	ActiveTime	Min, Max and Avg flow active time
13	IdleTime	Max, Min and Avg Idle time between Flows
14	flowCount	No of flows observed
15	ratioOfRBySIP	Ratio of number of IP from which packets are recevied to the IP's sent to by the application
16	avgpktcountPerSec	Average packets count per second
17	avgpktLenPerSecond	Average packets length per second
18	ratioOfRBySPkts	Ratio of packets received to packets sent in total

CHAPTER 4

Experiments

This chapter discusses the experiments performed on the applications and the corresponding results. First, the metrics used for evaluation are discussed and then the results from the static and dynamic analysis are presented.

4.1 Experiments Conducted

We performed multiple experiments using static, dynamic and combined features for the following scenarios.

4.1.1 Scenario A

We started off with a binary classification by considering all the adware families as one group called *adware* and the benign apps with and without ads as *benign*, to see how well can the models differentiate between adware and benign apps.

4.1.2 Scenario B

As in machine learning extracting features can be hard and there is no guarantee that the features are useful for classification; Out of the total static(5), dynamic(32), and combined(37) features we tested the ideal number of features needed to effectively perform the binary classification using Recursive feature elimination(RFE).

4.1.3 Scenario C

In the next set of experiments, we tried to label the apps after classifying it to adware or benign using a two-level classification mechanism where the first step as

described in Scenario A and then ran a trained model to label the family the apps belong to.

4.1.4 Scenario D

Lastly, we performed experiments to gauge the difficulty in identifying each family of adware from the benign apps without ads and benign apps with ads. This formed several sets of classifiers the results of which are described in the next section.

4.2 Evaluation Metrics

Accuracy is a measure that is used to determine how well the model detected adware and the correctness of the predicted family of an apk. If

- *True Positives (TP)*: number of positive examples, labeled as such.
- *False Positives (FP)*: number of negative examples, labeled as positive.
- *True Negatives (TN)*: number of negative examples, labeled as such.
- *False Negatives (FN)*: number of positive examples, labeled as negative.

Then accuracy is defined by

$$\text{accuracy} = (\text{TP} + \text{TN})/(\text{TP} + \text{TN} + \text{FP} + \text{FN})$$

In addition to accuracy, we also consider area under the ROC curve. Roc curve is a graph that plots True positive Rate vs False Positive Rate (TPR vs FPR). A model with AUC-ROC values closer to 1 is considered to be a good classifier and if it is near 0.5 then it can be considered to perform worse than flipping a coin.

$$\text{TPR} = \text{TP}/(\text{TP} + \text{FN})$$

$$\text{FPR} = \text{FP}/(\text{FP} + \text{TN})$$

4.3 Results

We experimented with algorithms like Random Forests(RF), Support Vector Classifier(SVC), Adaboost and MLP for the above-mentioned use cases. Where in each of the model was trained with 80% of the data and the remaining 20% of the data was used for testing. The results of which are discussed below.

4.3.1 Results for Scenario A

Here we classified apps as adware or benign for all 3 sets of features as shown in Figure 11. To have an idea of how the model would perform on new data we performed ten-fold cross-validation where we split the dataset into 10 parts and used 9 for training and one for testing. This is repeated 10 times and each time a different set is chosen for testing.

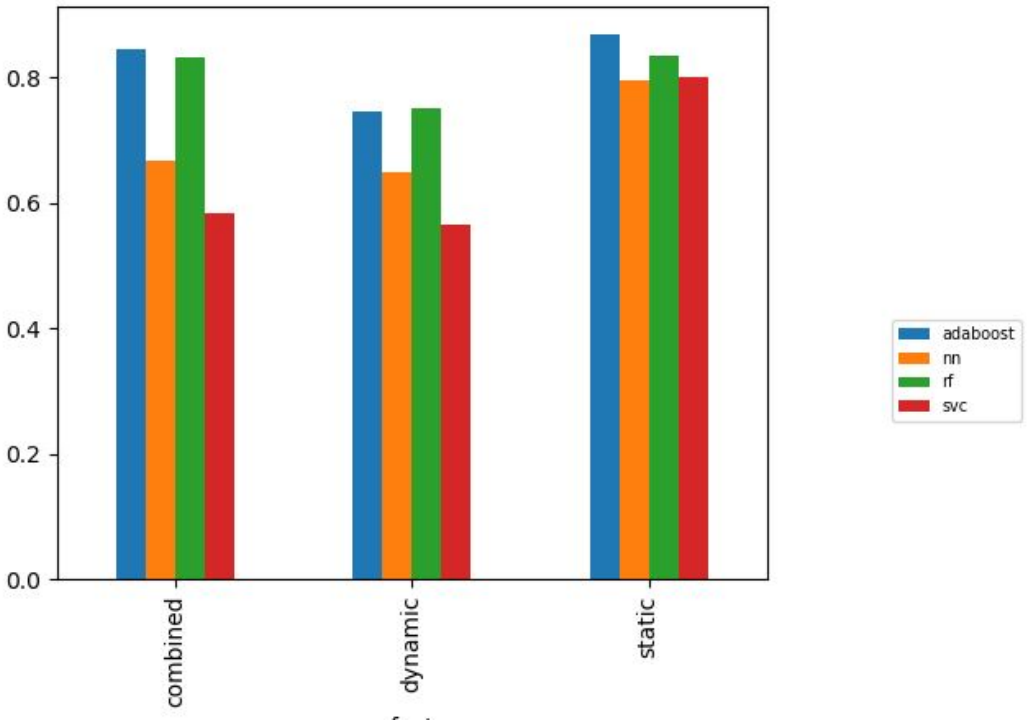
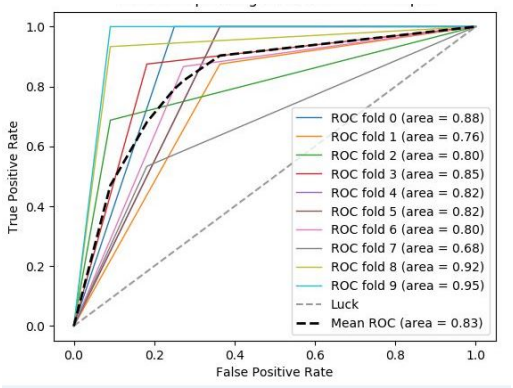


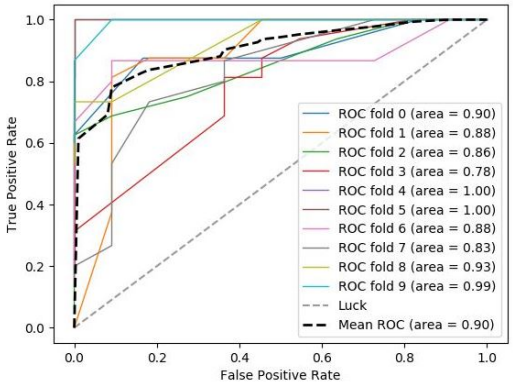
Figure 11: Accuracy of Models for Static, Dynamic and Combined Features for Adware vs Benign Classification

Our experiments showed that using dynamic features alone for detecting adware is not sufficient as the accuracy was only about 76% when experimented with combined features the accuracy improved to 84%. Though the overall accuracy for binary classification is higher for static features, about 85%, we still prefer using combined features as we suspect the model overfits a little as the dataset is small. From Figure 11, we see that linear SVC is not ideal for this case and tree-based techniques like adaboost or random forest works well. Adaboost results are slightly higher than random forest or MLP irrespective of the feature set used. This shows that dynamic features are not sufficient enough to detect adware and using tree-based ensemble methods with combined features is more suitable for this task.

Plotting Area under ROC for the models with combined features shows that the value is higher for random forest (0.9) than adaboost (0.83) though, the accuracy for adaboost was slightly higher. Figure 12 shows the ROC curve for ten fold cross validation for adaboost and randomforest when combined features were used for detecting adware. The area under the curve for SVC in Appendix A shows that it did no better than flipping a coin thus validating that accuracy alone is not enough to evaluate a model.



Adaboost



Random Forest

Figure 12: Area under ROC for Combined Features for Adware vs Benign

4.3.2 Results for Scenario B

Once the results showed that combined featureset gave better results in detecting adware, the next experiment was to identify the minimum number of features needed for detection. We used Recursive Feature Elimination on the combined 37 feature set which was trained on different models like random forest, adaboost and SVC without cross-validation. The results as shown in the Figure 13 indicates that the accuracy does not improve after 12 features when using adaboost, around 88%. SVC based RFE does not suit for identifying the important features and with random forest, the accuracy though is the highest at 87.4% for 17 features but it does not give consistent results like adaboost. Also, the accuracy is high when trained with 12 features and remains fairly stable with lower accuracy after that for adaboost.

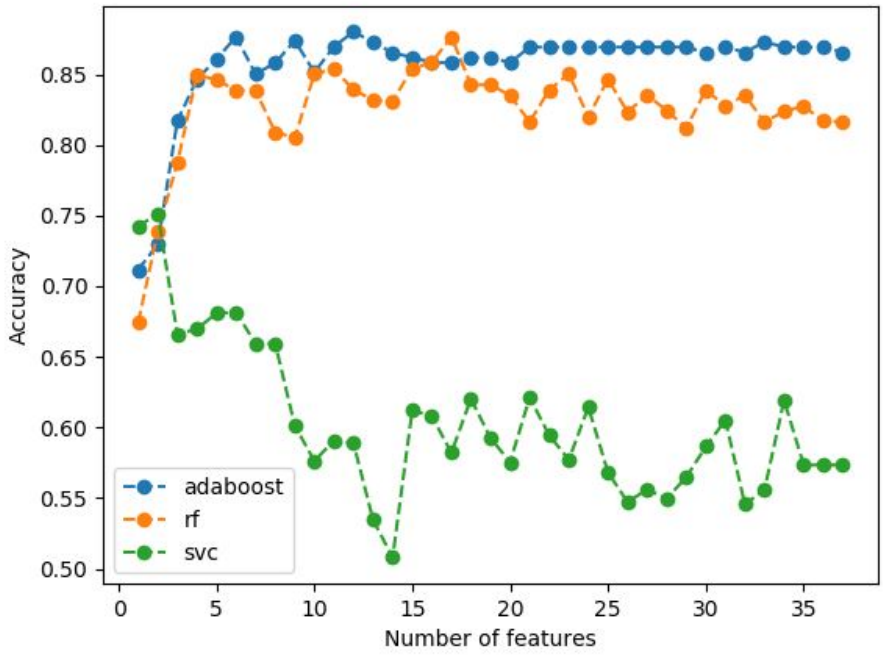


Figure 13: RFE on Combined Features

4.3.3 Results for Scenario C

In addition to a binary classification, we evaluated the performance of a two level classifier with combined features where, the first model was trained to detect adware as in Scenario A and the second classifier was trained to identify the family an app belongs to. Multiclass classifiers like One vs Rest and One vs One were used where, the former trains one classifier per class and the latter trains classifiers for each pair of classes. The results in Figure 14 show that the family of an app can be detected with an accuracy of 99% when one vs rest strategy is used for adaboost and random forest.

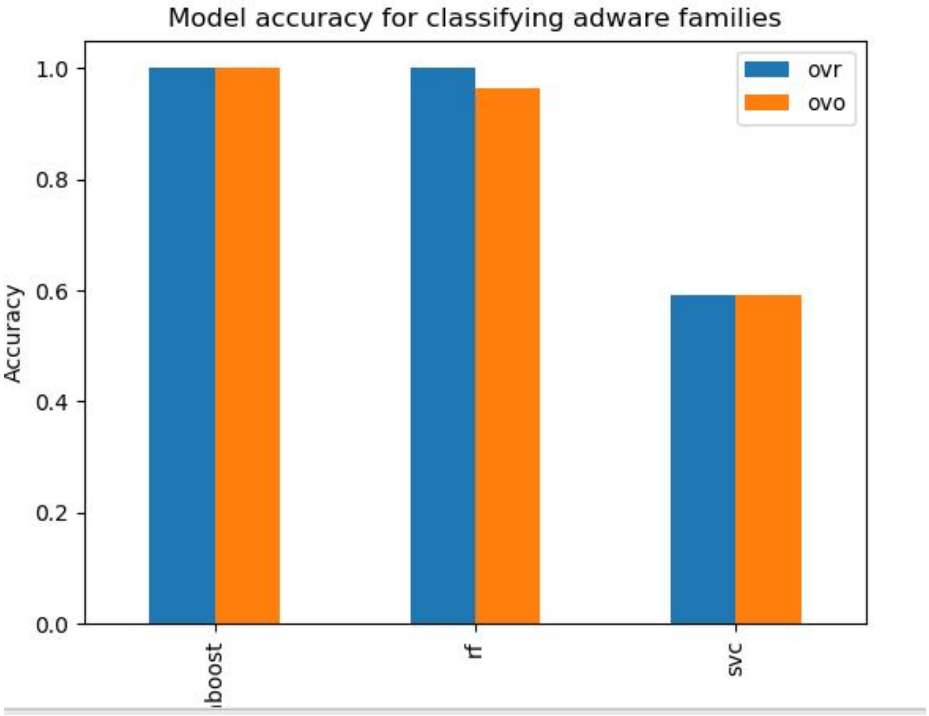


Figure 14: Accuracy of identifying family of apps using combined features

4.3.4 Results for Scenario D

This section describes the results obtained by training a model to detect each of the adware families from benign apps with and without ads. Specifically, we discuss the results from classifying each family with benign apps with ads as both kinds fetch ads but one is benign and the other is adware. The graph in Figure 15 shows the accuracy of classifying each family against the benign apps with ads. We can see that MLP identifies each family with the highest accuracy, that is above 84% when combined feature set is used. Next, tree-based ensemble techniques like random forest and adaboost produce comparative results but more towards the lower side. On the other hand, linear model like SVC did not perform well. Combined features accurately classifies benign apps with ads from individual family while dynamic features can help in detecting apps from judy, marswin, hamob and origin family. Detecting humming bad with dynamic features alone is insufficient. Static features though gives good results for all families we do not recommend it as the dataset used is small and thus model might is doing better. A similar graph for detecting apps from a particular family from benign apps is given in Appendix A.

4.4 Discussion

From the experiments, we infer that detecting adware can be tricky as several benign apps also display ads using ad libraries. Our results show that it is better to use combined features than using static or network-based dynamic features individually to avoid bias. Adware can be identified with an accuracy of 84% using tree-based ensemble techniques when using combined features and with an accuracy of 88% when suing only 12 features instead of all 37 for adaboost. The important features are shown in Table 4 for adaboost and in Table 5 for random forests. Ensemble

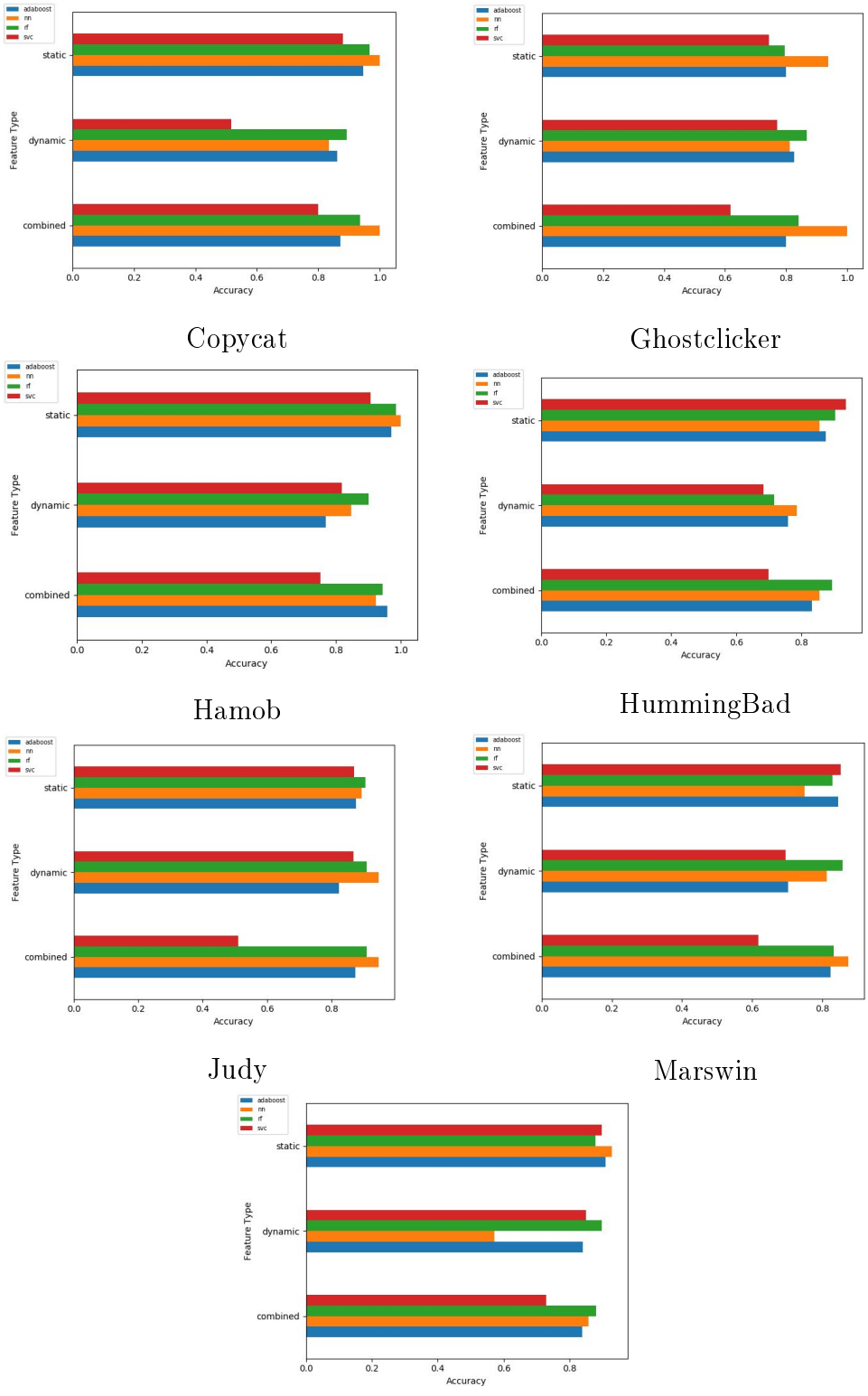
based techniques when used as a second level classifier plays a vital role in identifying the family label each app belongs to with higher accuracy than the linear models or neural networks. MLP can effectively separate each adware family from apps with ads with higher accuracy compared to other models, but when all the apps are combined together it's accuracy decreases. This may be due to the fact that the neural network model does not have enough data to learn from.

Table 4: Important Features by Adaboost

Number	Feature Name	Importance
1	numberOfPermissions	0.238420
2	numberOfActivities	0.143715
3	httpCount	0.092341
4	numberOfContentProviders	0.074321
5	dnsMaxAnswerLength	0.073395
6	flowCount	0.061364
7	maxIdleTime	0.049825
8	ratioOfRBySPkts	0.039481
9	numberOfServices	0.037938
10	noOfPkts	0.035264
11	maxPayloadHttp	0.032320
12	ratioOfRBySIP	0.015808

Table 5: Important Features by Random Forest

Number	Feature Name	Importance
1	numberOfPermissions	0.101284
2	numberOfActivities	0.088013
3	flowCount	0.076565
4	noOfPktsSent	0.064180
5	avgPayloadHttp	0.052946
6	numberOfContentProviders	0.050890
7	maxIdleTime	0.047395
8	avgIdleTime	0.044745
9	dnsAvgAnswerLength	0.042894
10	maxPayloadHttp	0.042866
11	httpCount	0.039604
12	dnsResCount	0.035854
13	noOfPkts	0.034824
14	numberOfBroadcastReceivers	0.034635
15	numberOfServices	0.027704
16	noOfPktsReceived	0.026180
17	dnsMaxAnswerLength	0.020036



Copycat Ghostclicker

Hamob HummingBad

Judy Marswin

Origin

Figure 15: Accuracy of classifying each family vs benign apps with ad

CHAPTER 5

Conclusion and Future Work

To keep up with the growth in the mobile and the advertising industry it is important to prevent fraudulent attacks to ensure that the right content reaches the user to gain a return on investment from the advertisements. In the mobile front the first step is to detect adware from benign apps and once the malicious apps are detected it is important to remove them from being available to the general public to prevent billion-dollar losses. In this experiment, we analyzed the importance of static and dynamic features for classifying the adware and static features turned out to be the winner when used with ensemble models. Thus machine learning approaches provide a good mechanism to detect and classify adware and it also fares well when it encounters adware in real time.

From the experiments conducted, we conclude that dynamic features alone are not sufficient to detect and classify adware, a combination of static and dynamic features performs better. As the next step, we can evaluate the above experiments on a larger dataset. Features related to system calls and UI components could also be used for detection. Periodicity can also be considered to detect adware as some adware frequently tries to connect to the C&C center for further processing. Another frontier to test would be to determine the difficulty in segregating malware from adware. One could extract information by running the apps on a real smartphone instead of using an emulator as malware developers sometimes disable malicious routines on emulators.

LIST OF REFERENCES

[1] "Sampi: Chinese programmatic ads — 6 most common types of ad fraud in China," https://sampi.co/6-most-common-types-ad-fraud-in-china/.

[2] "Android for all: Android architecture," https://letsknowaboutandroid.wordpress.com/about/, June 2013.

[3] "Random Forest based Classification (YouTube video)," https://www.youtube.com/watch?v=ajTc5y3OqSQ, June 2016.

[4] A. Honkela, "Multilayer perceptrons," http://www.helsinki.fi/~ahonkela/dippa/node41.html, 2001.

[5] "ADG Efficiency: Forecasting UK imbalance price using a multilayer perceptron neural network," http://adgefficiency.com/forecasting-uk-imbalance-price-using-a-multilayer-perceptron/, December 2016.

[6] M. Tomita, "Marketo: 7 reasons digital advertising wins," https://blog.marketo.com/2015/11/join-the-big-league-7-reason-to-go-digital-with-your-advertising.html, September 2016.

[7] "Statista: Distribution of free and paid Android apps 2017," https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/, January 2018.

[8] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in Android applications," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2014, pp. 123–134.

[9] "Whatis.com: What is ad fraud?" http://whatis.techtarget.com/definition/ad-fraud.

[10] "Whatis.com: What is adware?" https://searchsecurity.techtarget.com/definition/adware.

[11] "Sizmek: Impressions that inspire," https://www.sizmek.com/media/filer_public/eb/13/eb13ee88-972e-441a-a879-8e641609b4c2/casestudy_060514_fraud.pdf.

[12] "TrendLabs Security Intelligence Blog: GhostClicker adware is a phantomlike Android click fraud," http://blog.trendmicro.com/trendlabs-security-intelligence/ghostclicker-adware-is-a-phantomlike-android-click-fraud/, August 2017.

[13] "CheckPoint Blog: The Judy malware — possibly the largest malware campaign found on Google Play," https://blog.checkpoint.com/2017/05/25/judy-malware-possibly-largest-malware-campaign-found-google-play/, May 2017.

[14] "CheckPoint Blog: How the CopyCat malware infected Android devices around the world," https://blog.checkpoint.com/2017/07/06/how-the-copycat-malware-infected-android-devices-around-the-world/, July 2017.

[15] "CheckPoint Blog: HummingBad — a persistent mobile chain attack," https://blog.checkpoint.com/2016/02/04/hummingbad-a-persistent-mobile-chain-attack/, March 2017.

[16] J. Geater, "How to remove Android:Hamob-D," https://www.solvusoft.com/en/malware/potentially-unwanted-application/android-hamob-d/.

[17] "Naked Security: The Google Play adware apps that just won't die," https://nakedsecurity.sophos.com/2017/06/16/the-google-play-adware-apps-that-just-wont-die/, June 2017.

[18] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *The 28th International Conference on Distributed Computing Systems*, ser. ICDCS'08. IEEE, 2008, pp. 77–84.

[19] A. Metwally, D. Agrawal, and A. El Abbadi, "Detectives: detecting coalition hit inflation attacks in advertising networks streams," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 241–250.

[20] B. Liu, S. Nath, R. Govindan, and J. Liu, "DECAF: Detecting and characterizing ad fraud in mobile apps," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USENIX Association, 2014, pp. 57–70.

[21] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson, "What's clicking what? Techniques and innovations of today's clickbots," in *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'11. Springer-Verlag, 2011, pp. 164–183.

[22] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 2014.

[23] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[24] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys'12, 2012.

[25] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious Android applications," *Future Generation Computer Systems*, vol. 36, pp. 122–132, 2014.

[26] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, p. 5, 2014.

[27] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis." in *USENIX Security Symposium*, 2012, pp. 569–584.

[28] D. Sharma, "Android malware detection using decision trees and network traffic," *International Journal of Computer Science and Information Technologies*, vol. 7, no. 4, pp. 1970–1974, 2016.

[29] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, "Towards a network-based framework for android malware detection and characterization," in *Proceeding of the 15th International Conference on Privacy, Security and Trust*, ser. PST'17, 2017.

[30] A. Arora and S. K. Peddoju, "Minimizing network traffic features for android mobile malware detection," in *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ser. ICDCN '17. ACM, 2017, pp. 32:1–32:10.

[31] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *2017 International Conference on Information Networking (ICOIN)*, 2017, pp. 712–717.

[32] R. K. Rahul, T. Anjali, V. K. Menon, and K. P. Soman, "Deep learning for network flow analysis and malware classification," in *Communications in Computer and Information Science Security in Computing and Communications: Proceedings of the 5th International Symposium on Security in Computing and Communications*, ser. SSCC 2017, 2017, pp. 226–235.

[33] T. L. Wang, "Blackhat: AI based antivirus: Detecting android malware variants with a deep learning system," https://www.blackhat.com/docs/eu-16/materials/eu-16-Wang-AI-Based-Antivirus-Can-Alphaav-Win-The-Battle-In-Which-Man-Has-Failed.pdf.

[34] M. Stamp, *Introduction to Machine Learning with Applications in Information Security*. Boca Raton: Chapman and Hall/CRC, 2017.

[35] "Statista: Multilayer perceptrons," http://www.helsinki.fi/~ahonkela/dippa/node41.html, January 2018.

[36] M. Styk, "Github: Martinstyk/apkanalyzer," https://github.com/MartinStyk/ApkAnalyzer.

[37] A. H. Lashkari, G. D. Gil, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of Tor traffic using time based features," *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, 2017.

[38] "Scapy," http://www.secdev.org/projects/scapy/.

APPENDIX

Results for Adware vs Benign Classification

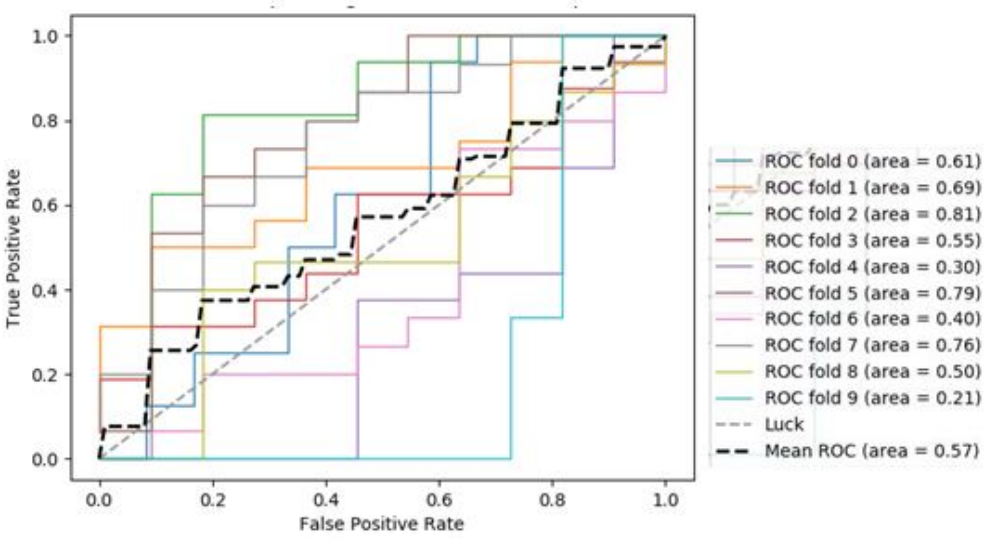


Figure A.16: Area under ROC with combined features for SVC

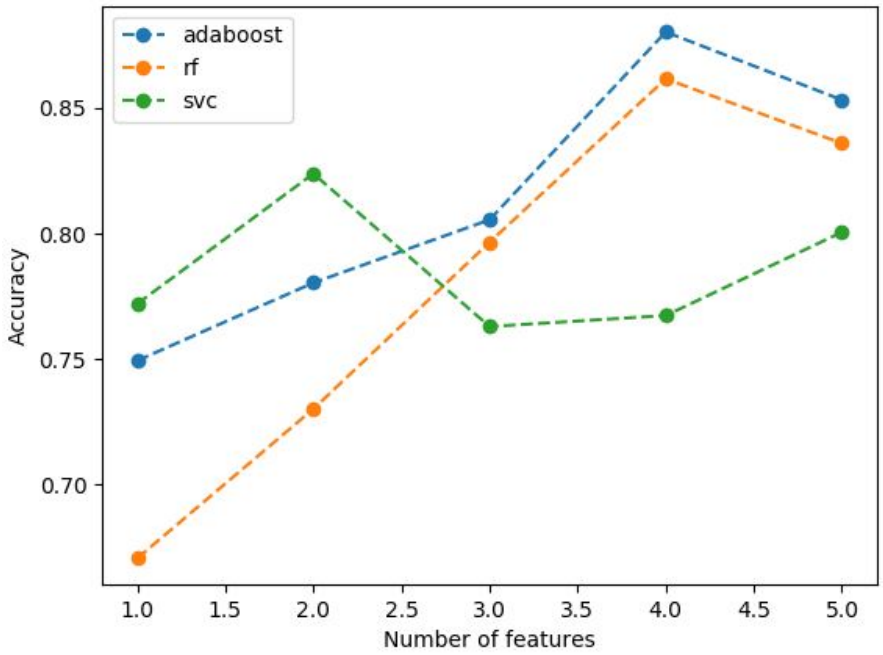


Figure A.17: RFE of Static Features

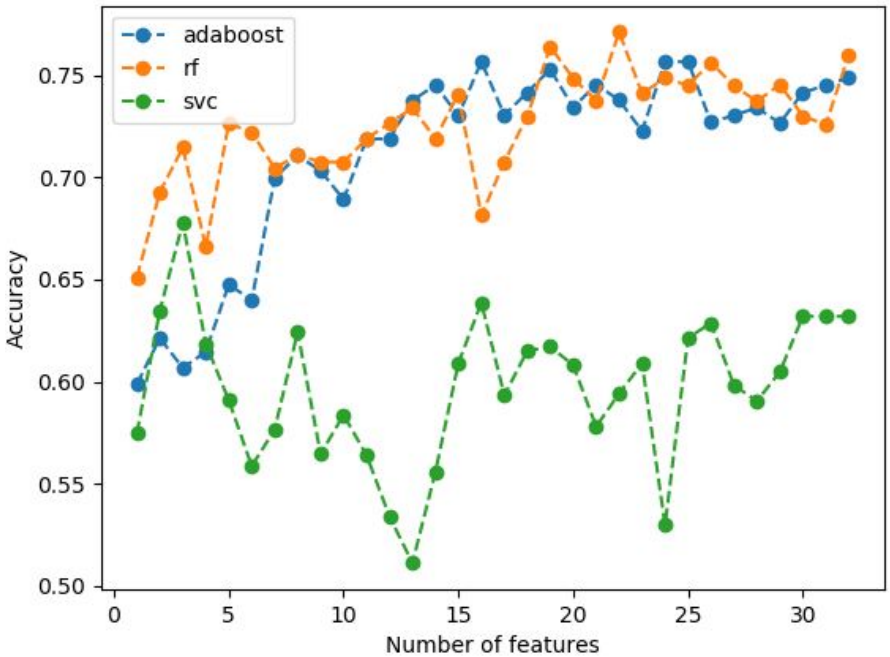
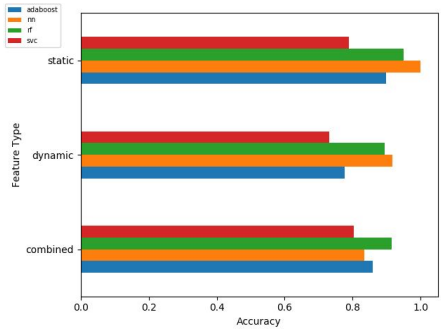
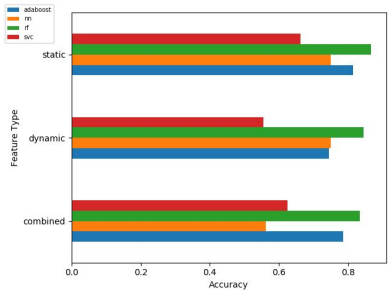


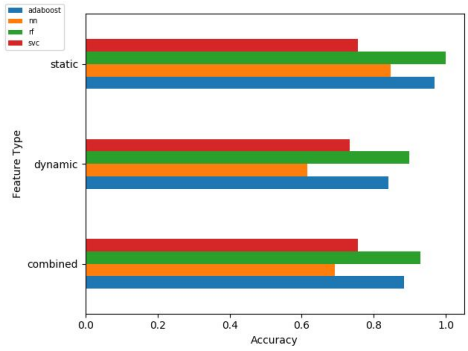
Figure A.18: RFE of Dynamic Features



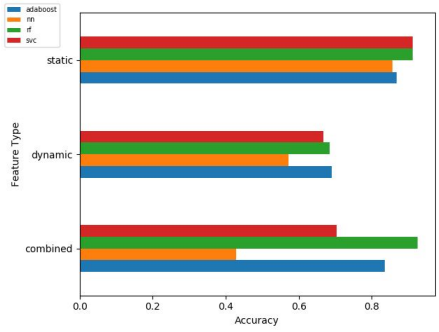
Copycat



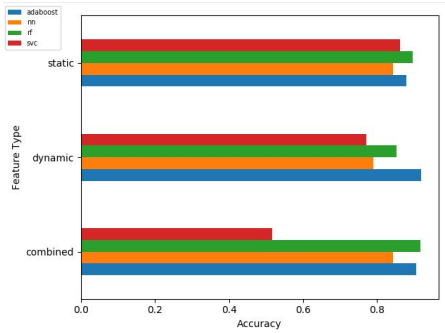
Ghostclicker



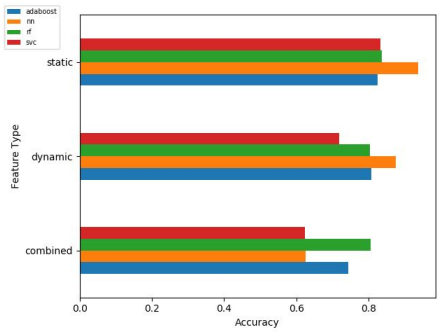
Hamob



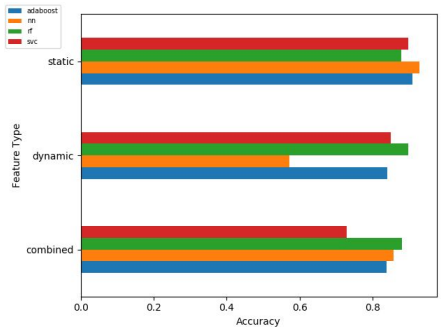
HummingBad



Judy



Marswin



Origin

Figure A.19: Accuracy of Classifying each Family against Benign Apps without Ads