ISSN: 2641-2039
DOI: 10.33552/GJES.2022.09.000720
Global Journal of Engineering Sciences
Iris Publishers

Research Article

Malicious Android Applications' Classification using Machine Learning

Kanwalinderjit Kaur[1]*, Jesal Patel[2], Alex Kiss[2] and Michael Walen[2]

[1]*California State University, Bakersfield, USA*

[2]*Florida Polytechnic University, Lakeland, FL, USA*

***Corresponding author:** Kanwalinderjit Kaur, Department of Computer and Electrical Engineering and Computer Science, California State University, Bakersfield, USA.

Received Date: May 17, 2022

Published Date: May 27, 2022

Abstract

Smartphones have been an integral part of everyday life for society since their development. Naturally, the popularity of these devices has brought an equal amount of malware designed specifically for these smartphone devices. The struggle to keep these devices secure and in turn the sensitive information stored on these devices from getting into the wrong hands has become an ever-evolving endeavor. With the sheer amount of malware produced coupled with the intelligent, polymorphic nature of the malicious software, it has become increasingly difficult to protect against them. In this paper, we propose a dynamic approach to classifying malicious Android applications that does not rely solely on the signatures of said applications. Instead, we analyze the Android Manifest of the dataset to classify whether an application should be considered malicious or benign.

Keywords: APK; repackaging; Android Manifest; Smali; Dex; Dalvic executable; Google play store

Introduction

With the proliferation of smartphones within the past few years they have become a part of most cellphone users' everyday lives [1]. Due to how integral they have become in today's society coupled with ease of access these devices have brought to contact one another, it is only natural that these devices have very sensitive information hosted on them. Knowing this, potential attackers see these devices as very enticing targets.

What makes Android devices even more enticing to potential attackers than the other platforms are the sheer amount of them being used. According to a worldwide market share survey, 85% of smartphone devices are Android based leaving iOS devices as the second largest with 14.7% and other devices as effectively non-existent [2]. Moreover, the fact that Android based devices use Java as the coding language leaves the Android devices vulnerable to reverse engineering. This is because Java applications in general have this characteristic. Attackers have used this to their advantage and as explained by Yajin et al. [3] that 86% of malicious software developed for Android has been through repackaging attacks.

A repackaging attack is performed by downloading a known to be benign and most likely popular application, reverse engineering the code, injecting a small snippet of malicious code within the application, and finally resigning the APK with the original signature to make it resemble the original application. Thus, resulting in applications that look familiar and benign on the Google Play Store, but has malicious code hiding within them.

Related Work

The paper \Kernel-based Behavior Analysis for Android Malware Detection" by Isohara et al. focuses on using a lower level approach to malware detection. Instead of using the manifest file, it

classifies malware based on system calls at the OS level of processing. It collects logs of all valuable application activity on the kernel, as well as process trees for suspicious applications, and sends them to a server so they can be analyzed. The advantage of doing this is that all applications must run through the kernel in some form or another, so even if a malicious app tries to circumvent its own analysis, the logger will have evidence of the processes being used to do so. Out of 230 popular third-party applications that were used as a sample, the logs found that 37 of the apps leaked information, 14 of them ran exploitation code, and 13 ran with the "su" command, giving the application access to other accounts. However, the server that the logs are being sent to uses a signature-based detection system of analyzing the system calls, so the accuracy of this detection method may not be as high as we want it to be.

Motivation

The majority of currently used pre-existing work in this area focuses on signature-based malware detection. While this method is very effective it requires a large database of known malware samples that must be kept up to date. Additionally, the app in question is being signed with identical signature those are same as the original benign app then it will go under the radar of the signature-based detection system. Another factor that could prove difficult to detect with this methodology is when polymorphic malware is introduced into the scenario. All these issues can be solved using a dynamic malware detection approach. Using a dynamic approach combined with machine learning can solve the issues that static signature-based detection systems have. These dynamic solutions reverse engineer the APK, examine the Dalvic Executable code contained within to determine whether the application in question is malicious or benign. The main issue with the dynamic approach is the limited resources found on mobile phones such as power consumption, CPU, memory, and storage. Keeping these limitations in mind we will take a look at alternative solutions.

Data Acquisition and Preprocessing

Working with malware detection, we need to have a baseline of benign samples with examples of known malicious samples. Our malware samples came from a combination of Koodous [4] and a collection on Github [5].

Something to keep in mind is that due to the popularity of these datasets it is also likely that static signature-based virus detection systems also use all or portions of these datasets. The APKs on these websites were double checked to be malicious using the VirusTotal tool [6] as a confirmation that these samples are indeed malicious. So, if the malicious sample is indeed malicious, it is very likely that VirusTotal will also confirm that.

When an Android application is compiled it is referred to as an APK. A compiled APK comprises of 3 modules; the Android Manifest made up of .xml permission requests coupled with API calls, the Java code that determines the logic of the application and lastly the resource file that consists of any music, images, colors, or anything else to produce an application. When we gathered a dataset of 100 malicious APK samples, we also gathered 100 benign APK samples taken from Google Play Store [7] and once again confirmed through VirusTotal that they were benign. Once we had our samples, we began the reverse engineering process. To do this we utilized a well-known tool known as APKTool [8] to extract the Android Manifest from an APK file. Our approach is to analyze the Android Manifest file included in every Android application that holds the permissions as well as the intent actions found within the Android Manifest. Intent Actions work as a message object that triggers an action in the component of some other app. To analyze the Android Manifest, we wrote a python script utilizing the xml.dom.minidom [9] library that parses through a Manifest file and organizes the requested permissions accompanied with the intent actions found within the application.

Malware Detection

The process of our dynamically detecting malware algorithm from an Android application's Manifest begins with sifting through the parsed permissions, and later on intent actions, to determine correlations between certain amounts of permissions called, the frequency of those called and assign a malignancy score to the most frequent permission requests and intent actions. Then, the number of permission requests and intent actions is recorded. Finally, the malignancy score for each app is calculated, and the application is classified as benign or malicious with the Logistic Regression, kNN, and Naive Bayes algorithms.

A dataset is developed consisting of 100 malicious applications and 100 benign applications and gathered the number of occurrences of the permissions from each. This was done using our python script that utilizes the xml.dom.mindom library in combination with second script that utilizes APKTool. This second script allows us to extract the manifests of a folder of APK files and gather the permission requests to score the application (Figure 1).

```
ch.nth.android.contentabo_101_ech_univ.permission.C2D_MESSAGE
com.google.android.c2dm.permission.RECEIVE
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_NETWORK_STATE
```

Figure 1: Example of permission requests gathered.

Through the process of gathering this dataset it was noticed that we could cut down on the complexity of the algorithm by determining unique and infrequently used permission requests and cutting those out of the permission requests that will be considered. In doing so we were able to cut the complexity down so that we can focus resources where they will be the most useful. Table 1 shows the top occurrences of permission requests from the benign data set and Table 2 shows the top occurrences of permission requests from the malware data set.

Table 1: Occurrences of permission requests in benign data set.

Permission	Occurrences
android.permission.INTERNET	85
android.permission.ACCESS NETWORK STATE	80
android.permission.WAKE LOCK	62
android.permission.WRITE EXTERNAL STORAGE	62
android.permission.ACCESS WIFI STATE	56
android.permission.VIBRATE	52
com.google.android.c2dm.permission.RECEIVE	49
android.permission.READ EXTERNAL STORAGE	47
android.permission.RECEIVE BOOT COMPLETED	43
android.permission.GET ACCOUNTS	35
android.permission.ACCESS FINE LOCATION	32
android.permission.CAMERA	30
com.android.vending.BILLING	29
android.permission.READ PHONE STATE	29
android.permission.READ CONTACTS	26
android.permission.ACCESS COARSE LOCATION	26
com.google.android.providers.gsf.permission. READ GSERVICES	21
android.permission.MANAGE ACCOUNTS	20
com.google.android.finsky.permission. BIND GET INSTALL REFERRER SERVICE	19
android.permission.USE CREDENTIALS	19
android.permission.GET TASKS	19
Total	**841**

Table 2: Occurrences of permission requests in malicious data set.

Permission	Occurrences
android.permission.WRITE EXTERNAL STORAGE	423
android.permission.INTERNET	361
android.permission.READ PHONE STATE	345
android.permission.ACCESS NETWORK STATE	344
android.permission.ACCESS WIFI STATE	308
android.permission.SEND SMS	270
android.permission.READ SMS	267
android.permission.RECEIVE SMS	260
android.permission.CHANGE WIFI STATE	234
android.permission.CHANGE NETWORK STATE	223
android.permission.MOUNT UNMOUNT FILESYS	184
android.permission.WRITE SMS	181
android.permission.ACCESS COARSE LOCATION	173
android.permission.GET TASKS	164

android.permission.VIBRATE	142
android.permission.READ EXTERNAL STORAGE	137
android.permission.RECEIVE BOOT COMPLETED	133
android.permission.WRITE SETTINGS	112
android.permission.ACCESS FINE LOCATION	109
android.permission.SYSTEM ALERT WINDOW	104
com.android.launcher.permission.INSTALL SHORTCUT	100
Total	4574

From this data, we observe that the malware dataset had about 443.9% more permission requests than the benign dataset. This is likely due to some manifests having duplicate permission requests. Requesting a permission that has already been requested before, has no effect on changing the approval or denial status of that permission. Therefore, the max number of requests for a particular permission should be as the number of APKs (Table 3).

Table 3: Scores assigned to permissions.

Permission	Score
android.permission.WRITE EXTERNAL STORAGE	3.61
android.permission.INTERNET	2.76
android.permission.READ PHONE STATE	3.16
android.permission.ACCESS NETWORK STATE	2.64
android.permission.ACCESS WIFI STATE	2.52
android.permission.SEND SMS	2.62
android.permission.READ SMS	2.56
android.permission.RECEIVE SMS	2.5
android.permission.CHANGE WIFI STATE	2.16
android.permission.CHANGE NETWORK STATE	2.09
android.permission.MOUNT UNMOUNT FILESYS	1.82
android.permission.WRITE SMS	1.77
android.permission.ACCESS COARSE LOCATION	1.47
android.permission.GET TASKS	1.45
android.permission.VIBRATE	0.9
android.permission.READ EXTERNAL STORAGE	0.9
android.permission.RECEIVE BOOT COMPLETED	0.9
android.permission.WRITE SETTINGS	0.96
android.permission.ACCESS FINE LOCATION	0.77
android.permission.SYSTEM ALERT WINDOW	0.87
com.android.launcher.permission.INSTALL SHORTCUT	0.86

In this research the number is 100 for both datasets. The next step in our malware detection process is to assign a malignancy score to the app. To do this, we first assign a malignancy score to each of the permission requests in Table 4. The score for each permission request is calculated using the following Eq. 1. n represents the number of APKs.

$$\text{Malignancy Score} = \frac{(\text{Malicious Occurrences} - \text{Benign Occurrences})}{n} \quad (1)$$

Shown below in Table 3 are the scores that were calculated.

Table 4: Occurrences of intent actions in benign data set.

Intent Action	Occrrences
android.intent.action.VIEW	372
android.intent.action.MAIN	138

android.intent.action.BOOT COMPLETED	91
com.google.android.c2dm.intent.RECEIVE	89
com.google.android.gms.gem.ACTION TASK READY	85
com.android.vending.INSTALL REFERRER	83
android.intent.action.MY PACKAGE REPLACED	53
android.intent.action.SEND	45
com.google.firebase.INSTANCE ID EVENT	44
android.intent.action.SEARCH	44
android.intent.action.LOCALE CHANGED	40
android.appwidget.action.APPDWIDGET UPDATE	35
android.net.conn.CONNECTIVITITY CHANGE	32
com.google.firebase.MESSAGING EVENT	31
android.content.Syncadapter	28
com.audible.mobile.download.action. CANCEL DOWNLOAD	22
com.audible.mobile.download.action. START DOWNLOAD	22
android.intent.action.EDIT	22
android.intent.action.TIME SET	20
com.google.android.c2dm.intent.REGISTRATION	18
com.google.android.gms.iid.InstanceID	18
Total	1332

Next, the number of permission requests found in an app's manifest are recorded. The malignancy score for an app is then calculated by adding up the scores of each scored permission found in the app's manifest .

From the results of the algorithms, it was decided that another attribute besides the number of permission requests and the permission request score for each app is required. The results from the algorithms for only permission requests are presented in the Results and Analysis section. We limited our search to items that are found in the Android Manifest le to make the implementation consistent. It was found that intent actions would be the best addition to the malware detection process. Intent actions are an attribute of an item found within the Android Manifest called an intent filter. Intents are a data structure that allows for an Android app to request actions from other apps. Two types of intents exist: explicit and implicit. Where, explicit intents specify the app that will perform the action and implicit intents do not specify the app that will perform the action, but rather the action that is being requested. Implicit intents are specified using intent filters [10]. The actions that the intent filters specify are under the <action> tag. To try and improve the accuracy of our malware detection process, we gathered the intent actions of all manifests in our dataset using the same process as the permission requests. Figure 2 shows an example of how the intent actions are gathered.

```
android.intent.action.USER_PRESENT
android.intent.action.ACTION_SHUTDOWN
android.net.conn.CONNECTIVITY_CHANGE
android.intent.action.SIM_STATE_CHANGED
android.intent.action.SERVICE_STATE
android.bluetooth.adapter.action.STATE_CHANGED
```

Figure 2: Example of intent actions gathered.

Shown in Table 4 and 5 are the occurrences of the most frequent intent actions of both data sets. With this data, it was noticed that both data sets have intent actions that exceed 100 occurrences. Unlike permission requests however, duplicate intent actions are justified because there could be multiple interactions in an app that perform the same action. Despite this, it was decided to use the same scoring method as the permission requests because most of the occurrences came from a few intent actions. Most notably, about 55.2% of the occurrences of the malware data set came from one intent action: MAIN. So, it was concluded that the use of this intent action is common in malware, and that the previous scoring method would be able to reflect that. The calculated scores are shown below in Table 6.

Table 5: Occurrences of intent actions in malicious data set.

Intent Action	Occrrences
android.intent.action.MAIN	1591
android.provider.Telephony.SMS RECEIVED	263
android.net.conn.CONNECTIVITY CHANGE	165
android.intent.action.USER PRESENT	154
android.intent.action.BOOT COMPLETED	139
android.provider.Telephony.SMS DELIVER	65
android.intent.action.ACTION POWER CONNECTED	39
android.net.wifi.STATE CHANGE	39
android.intent.action.MEDIA MOUNTED	39
android.net.wifi.WIFI STATE CHANGED	38
android.intent.action.ACTION POWER DISCONNECT	37
android.intent.action.SERVICE STATE	35
android.intent.action.ACTION SHUTDOWN	34
android.intent.action.ANY DATA STATE	34
android.intent.action.SIM STATE CHANGED	33
android.provider.Telephony.WAP PUSH RECEIVED	32
android.intent.action.MEDIA EJECT	31
android.bluetooth.adapter.action.STATE CHANGED	30
android.intent.action.BATTERY CHANGED	29
android.intent.action.PACKAGE ADDED	28
android.net.wifi.supplicant.CONNECTION CHANGE	26
Total	2881

Table 6: Scores assigned to intent actions.

Intent Action	Score
android.intent.action.MAIN	14.53
android.provider.Telephony.SMS RECEIVED	2.61
android.net.conn.CONNECTIVITY CHANGE	1.33
android.intent.action.USER PRESENT	1.49
android.intent.action.BOOT COMPLETED	0.48
android.provider.Telephony.SMS DELIVER	0.64
android.intent.action.ACTION POWER CONNECTED	0.3
android.net.wifi.STATE CHANGE	0.37
android.intent.action.MEDIA MOUNTED	0.31
android.net.wifi.WIFI STATE CHANGED	0.32
android.intent.action.ACTION POWER DISCONNECT	0.31
android.intent.action.SERVICE STATE	0.35
android.intent.action.ACTION SHUTDOWN	0.25

android.intent.action.ANY DATA STATE	0.34
android.intent.action.SIM STATE CHANGED	0.29
android.provider.Telephony.WAP PUSH RECEIVED	0.3
android.intent.action.MEDIA EJECT	0.28
android.bluetooth.adapter.action.STATE CHANGED	0.28
android.intent.action.BATTERY CHANGED	0.29
android.intent.action.PACKAGE ADDED	0.12
android.net.wifi.supplicant.CONNECTION CHANGE	0.26
Total	2881

Various Parameters

The parameters that are used to compare this experiment with previous Android Malware research are given as follows:

- TPR representing True Positive Rate
- FPR representing False Positive Rate
- Accuracy
- Area under the ROC curve
- Precision
- Recall
- F1-score

TPR computes if the given vector supports in detecting malware.

$$TPR = \frac{TP}{(TP+FN)} \qquad (2)$$

True Positives representing TP is how many malwares are classified correctly. False Negatives representing FN is how many malwares are classified incorrectly.

FPR computes if a benign app is falsely detected as malware.

$$FPR = \frac{FP}{(FP+TN)} \qquad (3)$$

False Positives representing FP is how many benign apps are classified incorrectly. True Negative representing TN is how many benign apps are classified correctly.

Accuracy is how many benign apps and malware apps are classified correctly.

$$ACC = \frac{TP+TN}{(TP+FN+FP+TN)} \qquad (4)$$

If the TPR is high and the FPR is low that means algorithm has a good performance. An algorithm with good performance will have an accuracy closer to 1.

This will be displayed in the ROC curve. If this curve is closer to the axis of TPR and TPR 1, then the algorithm has good performance. The area under the curve will show how well the classifier can distinguish each class. Precision represents the percentage of apps classified correctly as malware, as they actually being malware.

$$Precision = \frac{TP}{(TP+FP)} \qquad (5)$$

Recall is the percentage of apps that are malware are also classified correctly as malware.

$$Recall = \frac{TP}{(TP+FN)} \qquad (6)$$

F1-score is average of the Recall and Precision.

$$F1 = \frac{2 * Precision * Recall}{(Precision + Recall)} \qquad (7)$$

The F1-score is closer to the smaller value. Generally, if these values are closer to 1, then the algorithm is performing well.

Results and Analysis

The classification algorithms chosen to work with are Logistic Regression, Naive Bayes, and kNN [11]. This section will provide explanations of these classification algorithms, along with the results for each of them. The results of each algorithm include the ROC curve for both classes, the confusion matrix, and then a table containing other results such as area under the curve and classification accuracy. The ROC curve represents the TPR vs the FPR related to each test case. The confusion matrix represents how many apps are classified for each class, and how many apps that were actually in each class. The experiments started with the default properties for each algorithm, and any changes that were made to increase accuracy is mentioned in their respective explanation. The testing method used is 10-fold cross validation. This means that the data will be split into 10 groups, the model will learn from 9 random folds, test itself on the remaining fold, then repeat the process for each other fold. The first algorithm tested with the data set was Logistic Regression [12] (Figure 3).

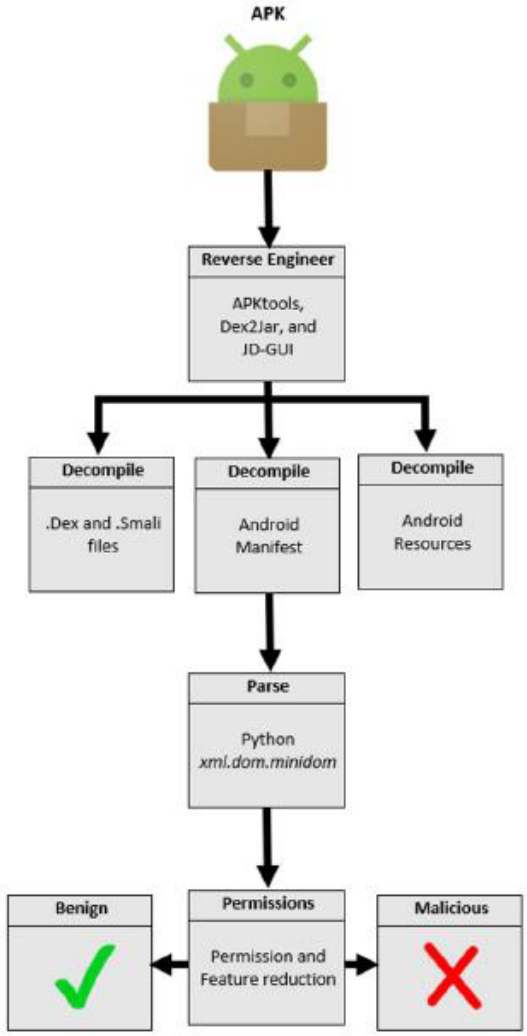

Figure 3: Overview of our malware classification process.

This algorithm uses a logit transform to calculate the probability that an instance is a member of a class. The predicted classification of the instance is the classification that has the highest probability. The probability that an instance X is a member of class j with k possible classes is calculated with Equation 8. If an instance has m attributes, then β is the $m * (k-1)$ parameter matrix.

$$P_j(X_i) = \frac{\exp(X_i * \beta_j)}{((\sum_{j=1}^{k=1} X_i * \beta_j) + 1)} \qquad (8)$$

Eq. 8 does not give the probability of the last class.

The probability of the last class is given below in Eq. 9:

$$1 - \left(\sum_{j=1}^{k=1} X_i * X_j \right) = \frac{1}{((\sum_{j=1}^{k=1} \exp X_i * \beta_j) + 1)} \qquad (9)$$

With all the probabilities calculated, the algorithm then uses a log-likelihood function to optimize the probabilities.

$$L = -\sum_{i=1}^{n} \left(\sum_{j=1}^{k-1} (Y_{ij} * \ln P_j(X_i) \right) + (1 - (\sum_{j}^{k-1} Y_i j)) * \left(\sum_{j=1}^{k-1} P_j((X_i)) \right) + \text{ridge} * \beta^2 \qquad (10)$$

Ridge is a parameter entered by the user to determine how much the algorithm will optimize the loglikelihood. The classification of each instance is the class with the highest probability. With Logistic Regression, we obtained the best results when using 20-fold cross validation instead of 10-fold cross validation. Only difference

between 20-fold and 10-fold is how many folds the data is split into. Logistic Regression had a classification accuracy of 91% with the ridge parameter set at 8. Logistic Regression did not have the highest classification accuracy out of the three algorithms, it did have 0.95 area under the ROC curve. The ROC curves for both classes are shown below in Figures 4 and 5, along with the confusion matrix in Figure 6. Table 7 is the full table of evaluation results for Logistic Regression.

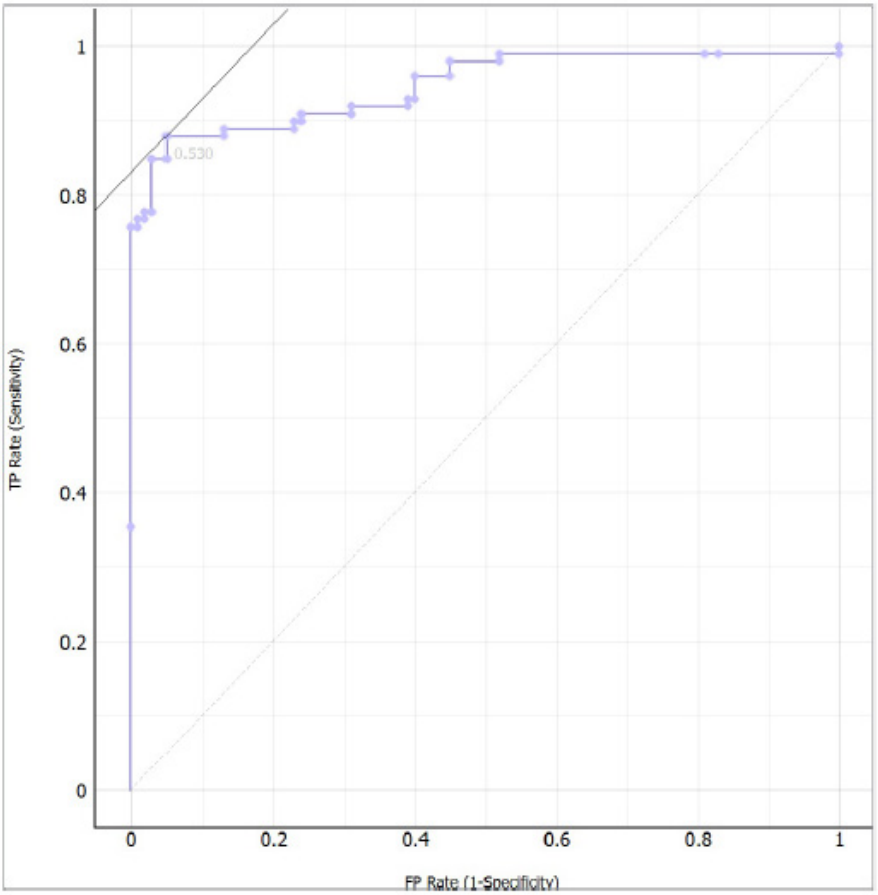

Figure 4: ROC curve for Logistic Regression for malicious classifications.

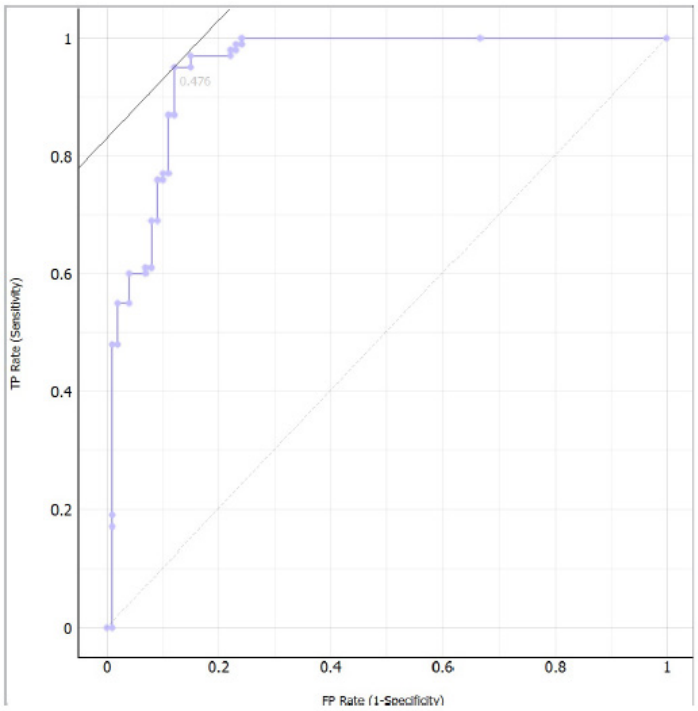

Figure 5: ROC curve for Logistic Regression for benign classifications.

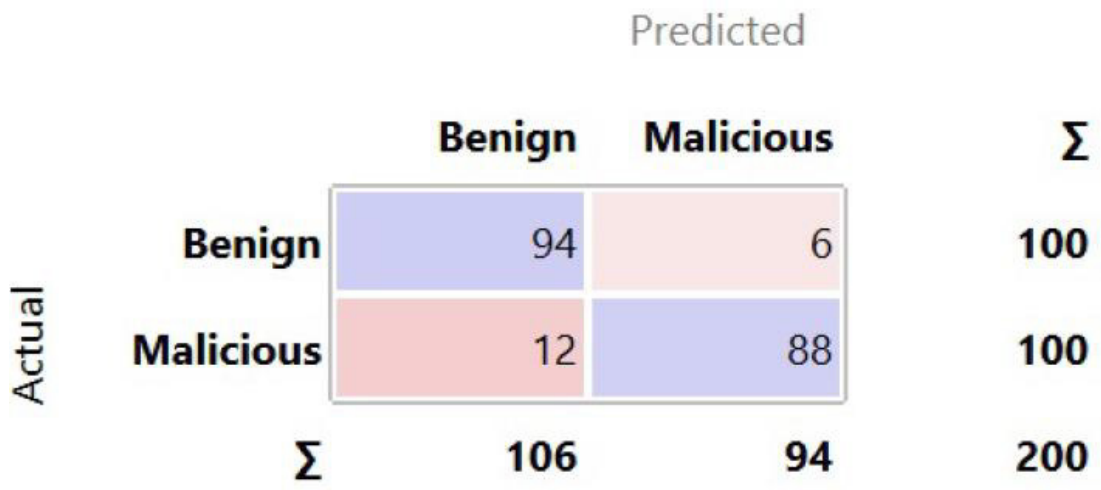

Figure 6: Logistic Regression Confusion matrix.

Table 7: Evaluation Results for Logistic Regression.

	AUC	CA	F1	Precision	Recall
Benign	0.95	0.91	0.913	0.887	0.94
Malicious	0.95	0.91	0.907	0.936	0.88
Average	0.95	0.91	0.91	0.911	0.91

Table 8: Evaluation Results for Naive Bayes.

	AUC	CA	F1	Precision	Recall
Benign	0.852	0.785	0.79	0.771	0.81
Malicious	0.852	0.785	0.779	0.8	0.76
Average	0.852	0.785	0.785	0.786	0.785

The next algorithm tested with the data was Naive Bayes [13]. Naive Bayes assumes that the attributes used for prediction are always independent from the classes, and that there are no underlying factors that will alter the predictions. Like Logistic Regression, Naive Bayes calculates the probabilities that a test instance is part of a class. The Bayes' Theorem is used to determine if a test instance x being a member of a class c. Let X be the vector that contains all predictor attributes, x represents the values of each attribute that particular test instance has, and C be the vector that contains the classification variable (Table 8).

$$p(X = x) = \frac{p(C = c) * p(C = c)}{p(X = x)} \quad (11)$$

Because the prediction attributes are conditionally independent from the class, Equation 12 is obtained.

$$p(C = c) = p(\wedge_i X_i = x_i | C = c) = \prod_i p(X_i = x_i | C = c) \quad (12)$$

Equation 12 only calculates the probabilities of discrete attributes. The probabilities for continuous attributes are given by Equation 13. $p(C = c) = g(x; \mu_c; \sigma_c)$,

$$Where\ (x; \mu, \sigma) = \frac{1}{\sqrt{2 * \pi * \sigma}} * e^{\frac{(\infty - \mu)^2}{2 * \sigma^2}} \quad (13)$$

and are the standard deviations and mean respectively of the continuous attribute. Once the probabilities are calculated, the algorithm simply selects the class that has the highest probability. With Naive Bayes, we achieved the best results using 20-fold cross validation. Naive Bayes was the least accurate algorithm in our testing with a classification accuracy of 78.5%. The ROC curves for both classes are shown below in g. 6 and 7, and Table 9 presents the confusion matrix.

Table 9: Evaluation Results for kNN.

	AUC	CA	F1	Precision	Recall
Benign	0.92	0.92	0.921	0.912	0.93
Malicious	0.92	0.92	0.919	0.929	0.91
Average	0.92	0.92	0.92	0.92	0.92

Naive Bayes has 0.852 area under the ROC curve. The results are shown in Table 9 (Figure 8 & 9).

The last algorithm tested was the kNN algorithm. It is an instance-based k-nearest neighbor's algorithm [14]. An instance is a single APK with the predictor attributes being the number of permission requests and the malignancy score, and the category attribute being the true classification of the application. The training of this algorithm could be considered as storing instances into a set with their classifications known. When the algorithm is done training, it compares an instance with an unknown classification to the stored instances. The similarity between two instances x and y with n predictor attributes is calculated with a distance formula. The default formula is a basic Euclidean distance shown in Equation 14.

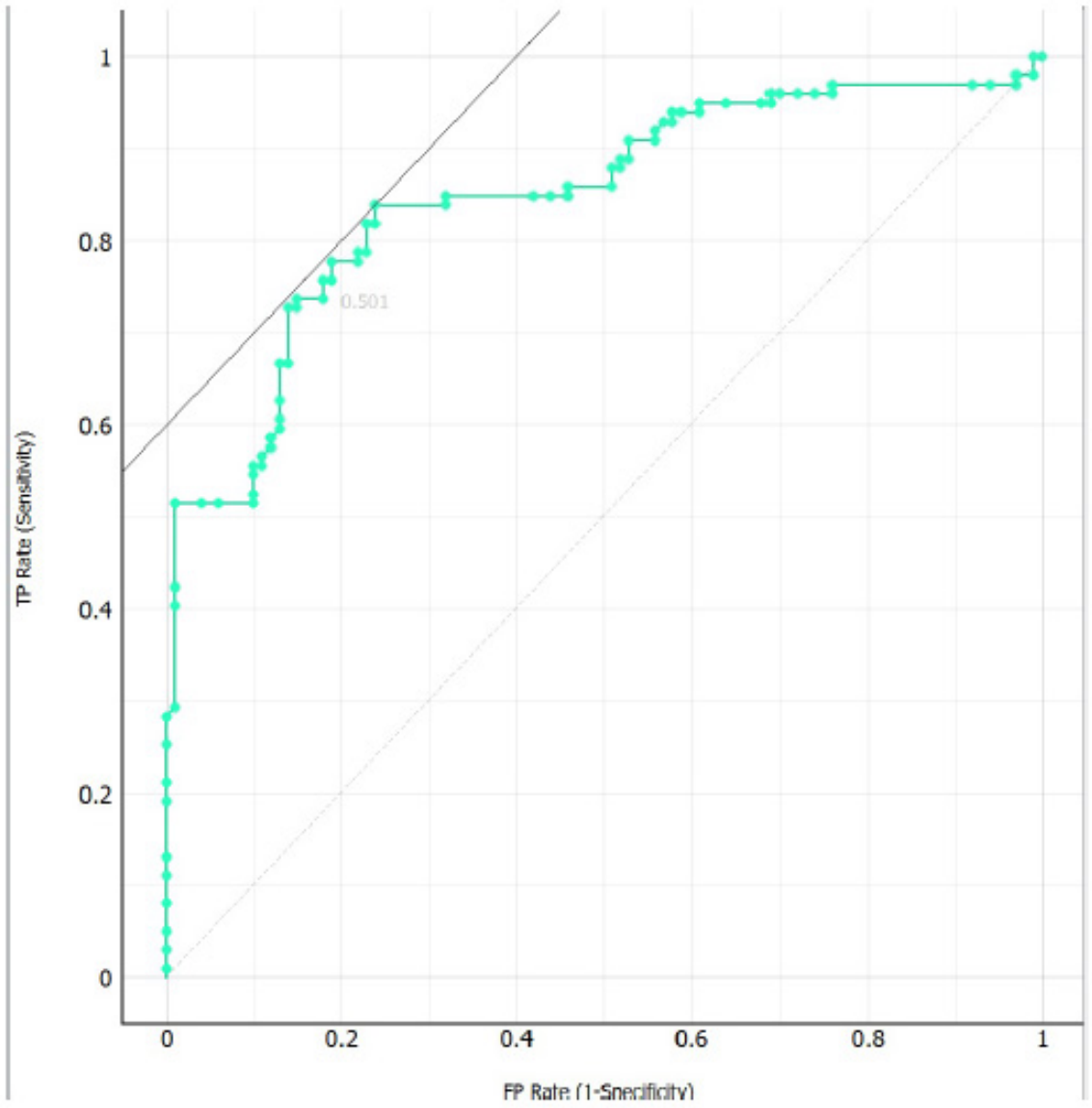

Figure 7: ROC curve for Naive Bayes for malicious classifications.

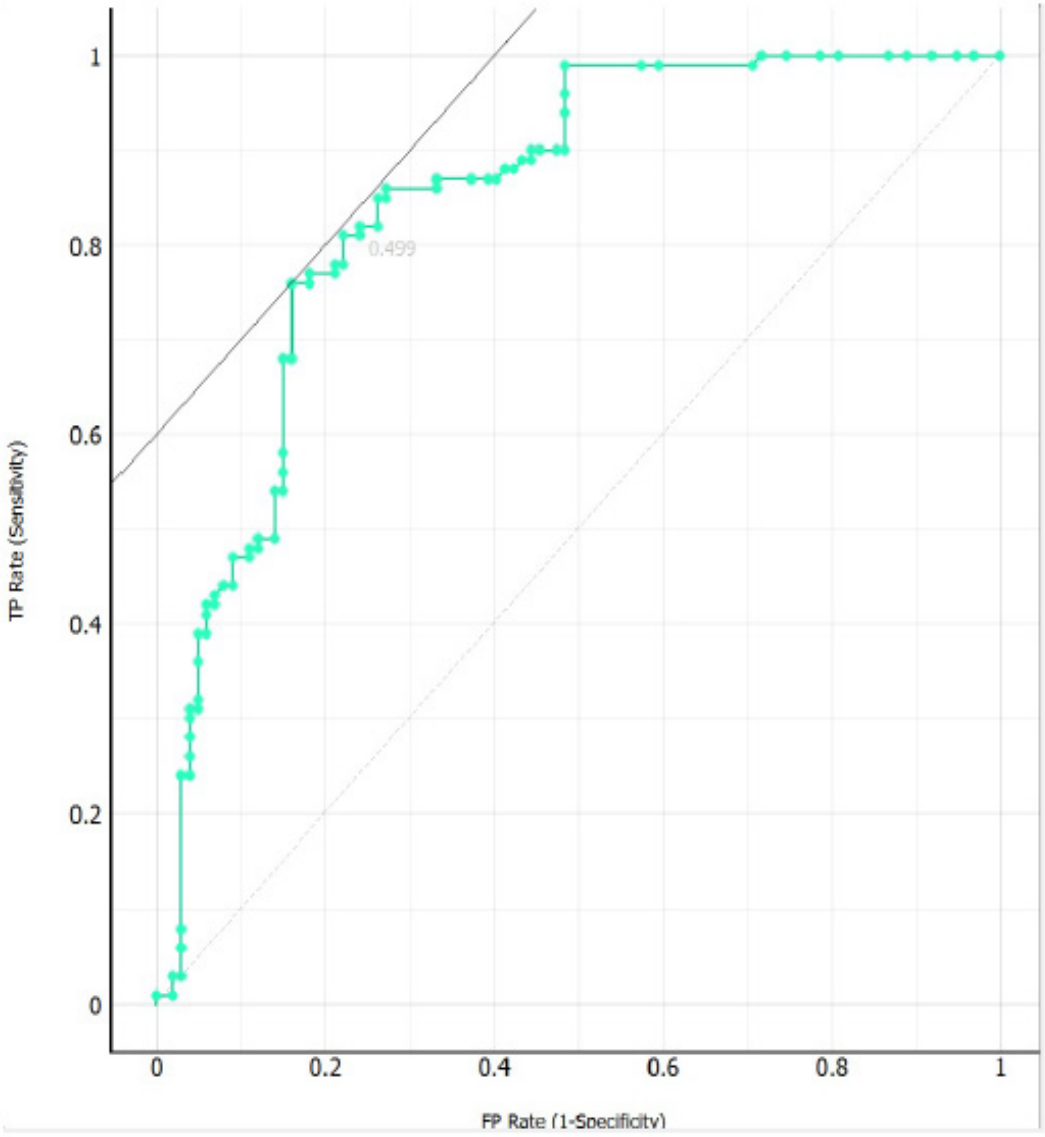

Figure 8: ROC curve for Naive Bayes for benign classifications.

		Predicted		
		Benign	**Malicious**	**Σ**
Actual	**Benign**	81	19	**100**
	Malicious	24	76	**100**
	Σ	**105**	**95**	**200**

Figure 9: Naive Bayes Confusion matrix.

$$\text{Similarity}(x; y) = -\sqrt{\left(\sum_{i=1}^{n}(x_i - y_i)^2\right)} \quad (14)$$

The similarity can also be calculated using a Manhattan, Chebyshev, or Mahala Nobis distance formula. The Manhattan distance formula is shown below in Equation 15 and was used in this research.

$$\text{Similarity}(x; y) = \sum_{i=1}^{n}|x_i - y_i| \quad (15)$$

Once all the similarities are calculated, the next step is to classify the unknown instance. The classification of instance is dependent on the popular classification of k smallest similarities. kNN was the most accurate classification algorithm with a classification accuracy of 92.0% with k set to 1, and the distance parameter set to Manhattan. The ROC curve for both classes is represented in Figures 10 and 11, and the confusion matrix is shown in Figure 12. The full evaluation results for kNN are shown in Table 9 [15].

From these results, clearly kNN is the algorithm that should be used in our malware detection process [16-19].

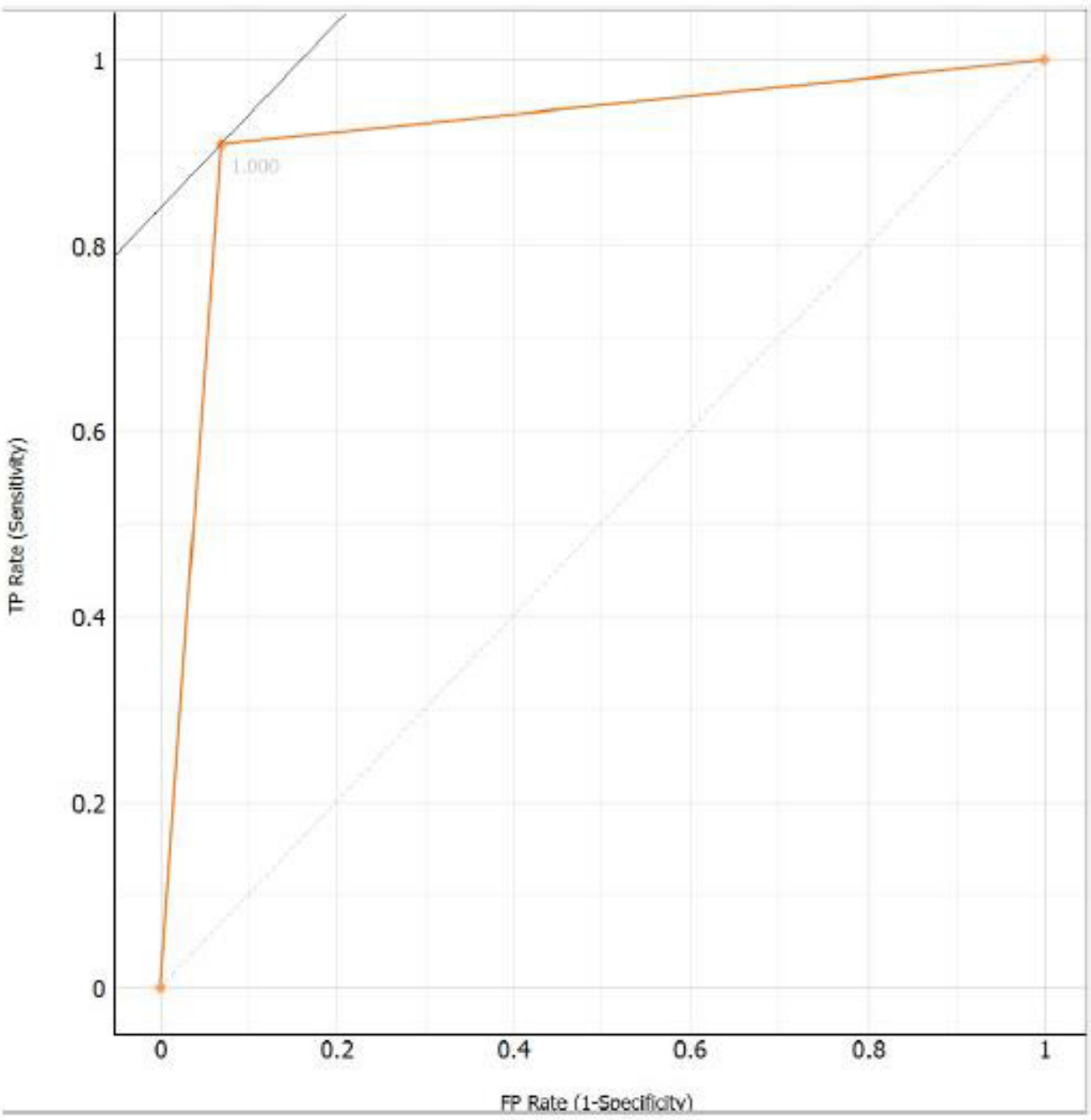

Figure 10: ROC curve for kNN for malicious classifications.

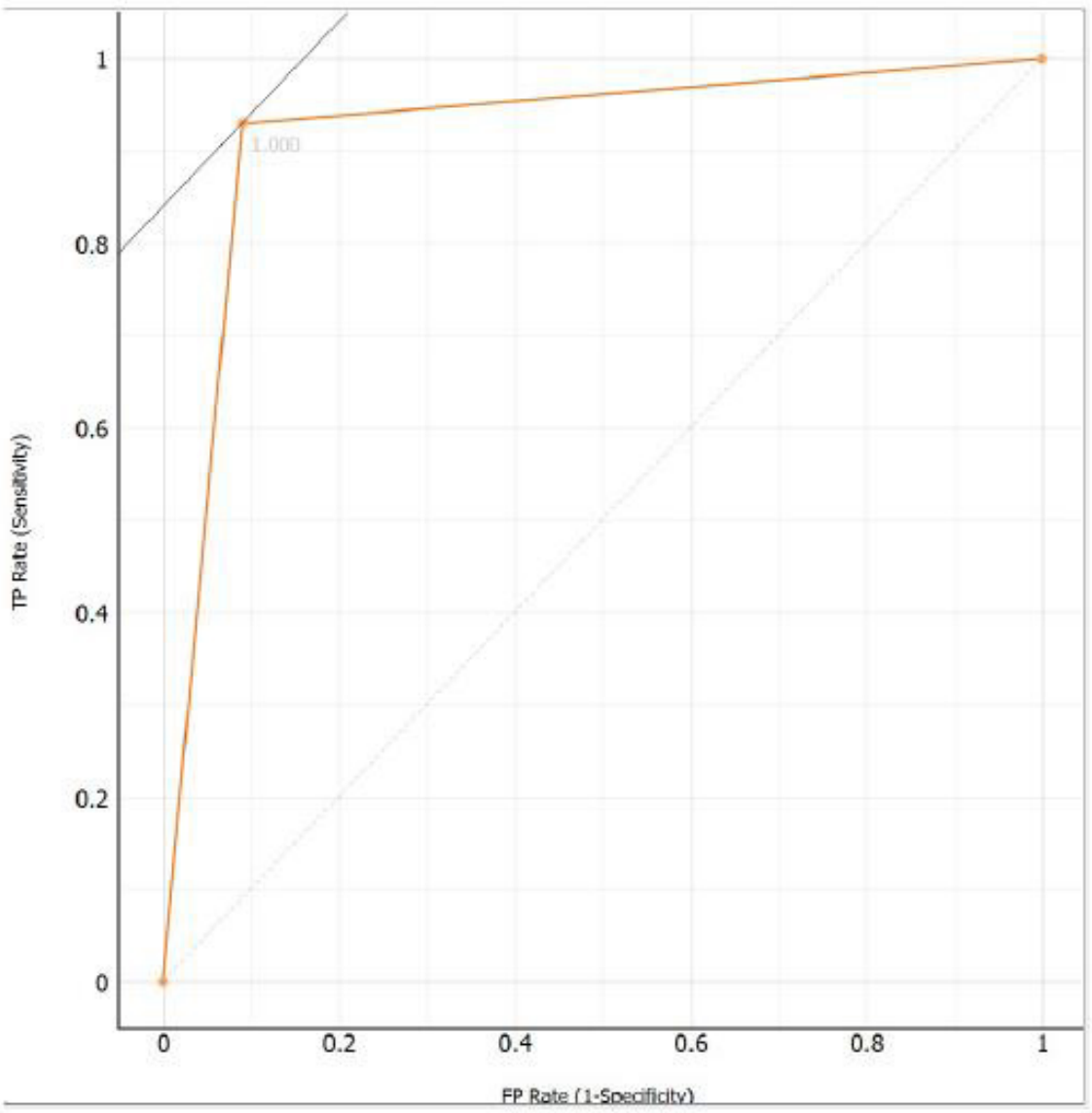

Figure 11: ROC curve for kNN for benign classifications.

		Predicted		
		Benign	**Malicious**	**Σ**
Actual	**Benign**	93	7	**100**
	Malicious	9	91	**100**
	Σ	**102**	**98**	**200**

Figure 12: kNN Confusion matrix.

Conclusion

With smartphones becoming ever popular devices and Android being the dominant operating system within the field, the battle of keeping these devices secured against malware has become equally more important. In this paper, we have discussed the problems with static signature-based malware detection methodologies and proposed a new dynamic algorithm-based framework instead that keeps power consumption concerns in mind.

Thus, our proposed method brings together the best of both worlds: the lightweight properties of signature-based detection and the effectiveness of other more in-depth, resource heavy methodologies. To accomplish this, we reverse engineer the Android APKs, parse thro' the Android Manifest for permission requests and intent actions. Then apply Logistic Regression, kNN, and Naive Bayes algorithms. In doing so, we were able to provide a general framework to dynamically classify Android malware, in which classification algorithms can be ne-tuned to any existing data points to provide more accurate results, providing a lightweight and effective solution to malware detection.

Acknowledgement

We thank Cyber Florida for funding this research work with grant #3910-1004-00-D.

Conflict of Interest

Conflict of Interest: The authors declare that they have no conflict of interest. This research does not involve Human Participants and/or Animals.

References

1. (2014) Billion Consumers to Get Smart (phones) by 2016. eMarketer Inc.
2. "IDC - Smartphone Market Share - OS." IDC: The Premier Global Market Intelligence Company.
3. Zhou, Yajin, Xuxian Jiang (2012) Dissecting Android Malware: Characteristics and Evaluation. IEEE Symposium on Security and Privacy.
4. Koodous. https://koodous.com/
5. Ashish B (2016) Android Malware. github.
6. VirusTotal. https://www.virustotal.com/gui/home/upload
7. Google Play Store. https://play.google.com/store=en
8. Apktool. https://ibot.github.io/Apktool/
9. Minimal DOM implementation.
10. https://developer.android.com/guide/
11. Demsar J, Curk T, Erjavec A, Gorup C, Hocevar T, et al. (2013) Orange: Data Mining Toolbox in Python. Journal of Machine Learning Research 14: 2349-2353.
12. Le Cessie S, van Houwelingen JC (1992) Ridge Estimators in Logistic Regression. Applied Statistics 41(1): 191-201.
13. George H John, Pat Langley (1995) Estimating Continuous Distributions in Bayesian Classifiers. Eleventh Conference on Uncertainty in Artificial Intelligence, San Mateo, 338-345.
14. D Aha, D Kibler (1991) Instance-based learning algorithms. Machine Learning 6: 37-66.
15. Takamasa Isohara, Keisuke Takemori, Ayumu Kubota (2011) Kernel-based Behavior Analysis for Android Malware Detection. Seventh International Conference on Computational Intelligence and Security, China, pp. 1011-1015.
16. Gagneja KK, Nygard K (2012) Key Management Scheme for Routing in Clustered Heterogeneous Sensor Networks," IEEE NTMS 2012, Security Track, Istanbul, Turkey, pp. 1-5, 7-10.
17. Tirado E, Turpin B, Beltz C, Roshon P, Judge R, et al. (2018) A New Distributed Brute-Force Password Cracking Technique," Future Network Systems and Security, FNSS Communications in Computer and Information Science 878: 117-127.
18. KK Gagneja, KE Nygard, N Singh (2012) Tabu-Voronoi Clustering Heuristics with Key Management Scheme for Heterogeneous Sensor Networks," IEEE ICUFN 2012, Phuket, Thailand, pp. 46-51.
19. Kanwalinderjit Kaur Gagneja, Riley Kiefer (2012) IoT Devices with Non-interactive Key Management Protocol. 2020 Sixth International Conference on Mobile And Secure Services, MobiSecServ 2020, Miami, USA.