Android Malware Detection based on AI and Machine Learning Algorithms

reported by
Kubilay Kürtür, 121200146

for
MTH 430: AI-based Security Analysis
Report

1 Introduction

This report discusses 3 different research: De-LADY, Dynamic Permissions based Android Malware Detection using Machine Learning Techniques, and Deep Android Malware Detection. The main purpose of these researches is to develop a methodology or a framework for Android malware detection based on artificial intelligence and machine learning algorithms. Malware detection is a rising issue, particularly on mobile platforms. Due to the widespread use of mobile devices and the app shops that go along with them, there are too many new apps available to carefully check each one for bad actions. Traditionally, malware detection relied on manually creating malware signatures by analyzing the behavior and decompiled code of known malicious programs. The static nature of signature-based malware detection, which allows new malware to be created to circumvent old signatures, makes it difficult for this procedure to scale to high application volumes. As a result, a lot of work has been done lately on automatic malware detection that makes use of machine learning concepts and AI. To stop app reversal, modern malware is improved with dynamic loading, encryption, and emulator detection techniques. Malware analysis, which is divided into two categories: dynamic (behavioral) analysis and static (code), is the process of comprehending the operation and infection of malware. While the dynamic technique watches

the run-time execution of code, the static approach analyzes code sequences without running them.

2 Methodologies

2.1 De-LADY

De-LADY basically uses dynamic features for malware detection on Android, with a deep learning foundation. In this methodology, an Android application's Android Application Package (APK) file must first be run in an emulated environment in order to determine if the application is malicious or benign. The emulator produces logs, which are then parsed and preprocessed to provide an application feature vector representation. The deep learning model is trained using this feature vector. In order to detect malware, the trained model is assessed using the testing dataset. De-LADY's framework consists of four main parts: dynamic analysis, feature extraction and preprocessing, deep learning classifier, and evaluation. To log application behavior, the sample is executed in an emulated environment during dynamic analysis. Copperdroid is used to rebuild RPC (Remote Procedure Call) and IPC (Inter-Process Communication) interactions and objects particular to Android. The logs that were taken specifically detailed the tracking of system calls, binder analysis, network traffic capture, and interaction between composite behaviors. Monkey, a tool for controlling the emulator, is used to create user interactions. To obtain behavioral insight, system calls, and binder information are used during the feature extraction process. JSON format is used for the logs indicated above that were extracted during dynamic analysis. To extract the features, each APK file's JSON log files are parsed. Consequently, a feature vector is created that represents all low-level behavioral data. Yeo-Johnson power transformation is used to normalize a dataset of feature vectors. Rescaling data to have a mean of zero and a standard deviation of one (unit variance) is commonly associated with standardization. Data becomes more Gaussian-like through power transformation. For the deep learning classifier, the deep neural network receives the standardized feature vectors. For each of the hidden layers, a leaky Rectified Linear Unit (ReLU) activation function is utilized. A leaky ReLU activation function, whose input is a neuron input, forwards the positive portion of the argument and permits a tiny positive gradient on the negative argument. The sigmoid function is utilized as an activation function for the output layer. The deep learning classifier is trained to distinguish between two classes: benign and malicious. There were 500 training epochs for the model. For testing,

samples from 30% of the dataset were chosen at random. Confusion matrix was used to assess the classifier model's performance effectiveness. To assess the performance of De-LADY, a collection of 13533 samples for Android applications is assembled. Of those, 10,712 were harmful samples while the remaining 2821 were benign ones. A variety of applications from areas like banking, gaming, utilities, and media player applications are included in it. Evaluation metrics' definitions are listed in Figure 1.

Term		Definition
True Positive Rate	TP	Percentage of samples correctly detected as malware or correctly classified as malware.
True Negative Rate	TN	Percentage of samples correctly detected as benign or correctly not classified as malware.
False Positive Rate	FP	Percentage of sample incorrectly detected as malware or incorrectly classified as malware.
False Negative Rate	FN	Percentage of sample incorrectly detected as benign or incorrectly not classified as malware.
Precision	p	$TP/(TP+FP)$
Recall	r	$TP/(TP+FN)$
F-measure	F_1	$2rp/(r+p)$
ROC Area	AUC	Area under ROC curve
Accuracy	Acc	Percentage of malwares correctly detected or classified
Error Rate	ER	Classification error rate

Figure 1: Definitions of Evaluation Metrics [1]

For the evaluation and results, the emulator was set up to give an example application in an environment that looked like a real device. The application was executed in a clean condition after each modification was made to the IMEI number, IMSI number, contact information, SMS, etc. The deep learning model has one output layer with a single neuron for binary classification, fine-tuned hidden layers with neurons the size of feature vectors, and input layer neurons that match the size of the feature vector. To determine the ideal configuration, the number of hidden layers and hidden neurons are adjusted. The loss function that is employed is binary cross entropy. The model is trained for 800 iterations using 100 mini-batch sizes of randomly chosen samples during the learning process. Figure 2 shows the results.

Hidden Layers	Hidden Layer Neurons	p	r	F1	AUC	Acc	ER
2	[100, 100]	98.72	98.69	98.70	94.16	97.86	2.14
	[200, 200]	98.55	98.96	98.75	93.47	97.93	2.07
	[400, 400]	98.60	98.72	98.66	93.62	97.78	2.22
	[300, 150]	98.04	99.07	98.55	91.68	97.61	2.39
3	[100, 100, 100]	98.66	98.84	98.75	93.96	97.93	2.07
	[200, 200, 200]	98.72	98.63	98.67	94.12	97.81	2.19
	[400, 400, 400]	98.69	98.60	98.64	93.97	97.76	2.24
	[360, 240, 120]	98.39	99.04	98.72	93.02	97.88	2.12
4	[100, 100, 100, 100]	98.51	99.04	98.77	93.76	97.98	2.02
	[200, 200, 200, 200]	98.42	**99.25**	**98.84**	93.38	**98.08**	**1.92**
	[400, 400, 400, 400]	98.71	98.77	98.74	93.68	97.93	2.07
	[360, 270, 180, 90]	98.28	99.13	98.70	92.76	97.86	2.14
5	[100, 100, 100, 100, 100]	98.36	98.89	98.63	93.02	97.73	2.27
	[200, 200, 200, 200, 200]	98.45	98.74	98.59	93.33	97.68	2.32
	[400, 400, 400, 400, 400]	98.60	98.92	98.76	94.08	97.96	2.04
	[400, 320, 240, 160, 80]	98.19	99.01	98.60	92.30	97.68	2.32
6	[100, 100, 100, 100, 100, 100]	98.48	98.95	98.71	93.78	97.88	2.12
	[200, 200, 200, 200, 200, 200]	98.54	98.92	98.73	93.82	97.91	2.09
	[400, 400, 400, 400, 400, 400]	**98.89**	98.51	98.70	**94.89**	97.86	2.14
	[420, 350, 280, 210, 140, 70]	98.39	98.83	98.61	93.12	97.71	2.29

Figure 2: Results of De-LADY [1]

Every architecture had a good f1 score, with over 98% of them doing well. Out of all the model architectures, two had good performance. Superior precision and AUC were achieved via an architecture with six hidden layers, each including 400 neurons. Four hidden layer architectures (each with 200 neurons) perform better than other architectures with 99.25% recall, 98.84% f1 score, 98.08% accuracy, and 1.92% error rate. Figure 3 shows the comparison of De-LADY with some well-known machine learning algorithms.

Method	Accuracy	F-measure	Error Rate
KNN	94.48	79.87	5.52
Naive Bayes	91.60	65.40	8.40
SVM (linear)	95.77	85.16	4.23
Decision Tree	97.09	90.26	2.91
Random Forest	97.01	89.6	2.99
XGBoost	96.98	89.53	3.02
De-LADY	98.08	98.84	1.92

Figure 3: Method Comparisons [1]

2.2 Dynamic Permissions based Android Malware Detection using Machine Learning

The methodology in this research has 3 phases. In the first phase, The Android application packages, or .apk files, are gathered from many resources. The permissions that these apps need during installation and startup are gathered to create the dataset in the second phase, which involves doing dynamic analysis (just like De-LADY) on the gathered Android application packages (.apk). In the final stage, different machine learning algorithms are applied to assess the data collection. Using the Android Botnet dataset, DroidKin dataset, Android Malware Genome Project, and AndroMalShare, 13,000 unique Android application packages (.apk) are gathered in the first phase of the methodology. These comprise 6029 normal Android application packages (.apk) from Appchina, Hiapk, Android, Mumayi, Gfan, Pandaapp, and Slideme, and 6971 malicious applications from several malware families, including worms, trojans, backdoors, botnets, and spyware. These gathered Android application packages (.apk) are executed using the Bluestack emulator. Additionally, a Java method is used to extract permissions and the dataset is created. 123 unique permissions are collected which these applications demand. The dataset is built of 11,000 distinct applications by obtaining these permissions. These 123 permissions are divided into two states: safe and unsafe. Naive Bayes (NB), Decision Tree (J48), Random Forest (RF), Simple Logistic, and k-star are the five machine learning classifier approaches that are used to assess the dataset. The configuration of them is shown in Figure 4.

Algorithms	Configuration
Naive Bayes (NB)	N/A
Decision Tree (J48)	Size of tree=19
Random Forest (RF)	8 random features
Simple Logistic	Number of classes =2
k-star	k=1

Figure 4: Configuration of Algorithms [2]

In the evaluation stage, the dataset is tested in WEKA by utilizing its three accessible options. The first is to provide a training set and assess it first, followed by cross-validation in second place and percentage-wise data splitting in third place. The results are shown in Figures 5,6 and 7.

Techniques	TPR	FPR	Prec.	Recall	F-Measure
Naive Bayes	0.987	0.007	0.988	0.987	0.987
J48	0.996	0.003	0.996	0.996	0.996
Random Forest	0.996	0.002	0.996	0.996	0.996
Simple Logistic	0.997	0.002	0.997	0.997	0.997
k-star	0.952	0.028	0.957	0.952	0.952

Figure 5: Testing of Training and Supply Dataset with All Classifiers [2]

A training set is a file that is imported, preprocessed, and tested initially. After that, a dataset is provided for assessment. 30% of the dataset is used for testing, while 70% is used to train the classifier. The accuracy rank-wise for Simple Logistic, J48, Random Forest, Naive Bayes, and k-star classifiers is 0.997, 0.996, 0.996, 0.987, and 0.952, respectively. TPR (True Positive Rate), FPR (False Positive Rate), precision, recall, and f1 score are shown in Figure 5.

Techniques	TPR	FPR	Prec.	Recall	F-Measure
Naive Bayes	0.984	0.009	0.985	0.984	0.984
J48	0.996	0.003	0.996	0.996	0.996
Random Forest	0.996	0.003	0.996	0.996	0.996
Simple Logistic	0.996	0.003	0.997	0.996	0.996
k-star	0.952	0.028	0.957	0.952	0.952

Figure 6: Testing during Cross-Validation of Dataset with All Classifiers [2]

Using a technique known as 10-fold cross-validation, WEKA splits the dataset into ten segments, or "folds," distributes each segment sequentially, and then averages the scores. Thus, every data point in the dataset is used nine times for training and once for testing. The results are shown in Figure 6.

Techniques	TPR	FPR	Prec.	Recall	F-Measure
Naive Bayes	0.987	0.007	0.988	0.987	0.987
J48	0.997	0.002	0.997	0.997	0.997
Random Forest	0.997	0.002	0.997	0.997	0.997
Simple Logistic	0.997	0.002	0.997	0.997	0.997
k-star	0.952	0.028	0.958	0.952	0.953

Figure 7: Testing of Splitting Dataset with All Classifiers [2]

The third approach divides the dataset into percentages that allow to assess of the classification results on a test set that is comprised of the original data. For evaluation, the dataset is split by 66%.

2.3 Deep Android Malware Detection

This approach processes an Android application's raw Dalvik bytecode using a convolutional network. In this methodology, an application's preprocessing involves its disassembly and the extraction of opcode sequences for static malware analysis. An Android application is represented as an apk file, which is a compressed file that includes the application resource files, the AndroidManifest.xml file, and the code files. A code file is a dex file that may be converted into "smali" files, each of which is a representation of a single class with its methods contained within. Every method has instructions, and each instruction is made up of several operands and one opcode.

Using Baksmali, each application is disassembled to produce "smali" files containing the application's human-readable Dalvik bytecode. Then the opcode sequence is extracted from each method, throwing away the operands. Every opcode sequence is received from every application class as a result of the preprocessing. The entire application is then represented by a single sequence of opcodes created by concatenating the opcode sequences from all classes. The network architecture of this methodology consists of the opcode embedding layer, convolutional layers, and classification layers. In the opcode embedding layer, sequences of opcode instructions are transformed into a continuous vector space. A weight matrix is utilized to each opcode instruction into embedding space once it has been encoded as a one-hot vector. This makes a neural network to understand associations between opcodes based on their semantic similarity, learning meaningful representations for every opcode. In the convolutional layers, multiple filters are utilized to embedding matrices that are received as input in order to identify opcode sequences. Activation maps are generated by each layer and stacked to form matrices. A Multi-Layer Perceptron (MLP) with a hidden layer and an output layer makes up the classification layers. High-order associations between features that the convolutional layers extracted for classification are detected by the MLP. Repaired linear activation is used in the hidden layer, and a softmax classifier is used to process the output and estimate the likelihood that the software is malware. Three different datasets are utilized for this methodology. The Android Malware Genome project's malware is included in the first dataset. There are 2123 programs in this dataset, of which 1260 are malware from 49 different malware families and 863 are benign. This dataset is called "Small Dataset". The second dataset comes from the internal Android malware repository of the vendor McAfee Labs. There are 3627 benign samples and 2475 malware samples in this collection. This dataset is called "Large Dataset". A further dataset, which was gathered more recently than the first two, has been given by McAfee Labs and comprises roughly 18,000 Android programs. This dataset is called "V. Large Dataset". After the hyper-parameters were adjusted using the smaller datasets, this was utilized to test the finished system. The collection has 9902 malicious files and 9268 benign ones once short files are removed. 10% of each dataset was set aside for testing, and the other 90% was used for training and validation. The mean of the classification accuracy, precision, recall, and f-score is used to report the results. The results are shown in Figure 8.

Classification System	Feature Types	Benign	Malware	Acc.	Prec.	Recall	F-score
Ours (Small DS)	CNN applied to raw opcodes	863	1260	0.98	0.99	0.95	0.97
Ours (Large DS)	CNN applied to raw opcodes	3627	2475	0.80	0.72	0.85	0.78
Ours (V. Large DS)	CNN applied to raw opcodes	9268	9902	0.87	0.87	0.85	0.86

Figure 8: Malware Classification Results [3]

3 Conclusion

In this report, 3 different researches, which aim to develop an Android malware detection system based on AI or machine learning, are compared and explained. They have different methodologies, features, datasets, and results. The identification and selection of unique features from malware samples is a complex task in malware detection and classification. When a program uses obfuscation techniques to avoid such identification, the task becomes very difficult.

References

[1] V. Sihag, M. Vardhan, P. Singh, G. Choudhary, S. Son, *De-LADY: Deep learning based Android malware detection using Dynamic features* , 2021.

[2] A. Mahindru, P. Singh, *Dynamic Permissions based Android Malware Detection using Machine Learning Techniques* , 2018.

[3] N. McLaughin, J. M. del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupe, G. J. Ahn, *Deep Android Malware Detection*, 2017.