2019 International Conference on Computing, Networking and Communications (ICNC): Communications and Information Security Symposium

Towards Multimodal Learning for Android Malware Detection

Josh McGiff*, William G. Hatcher*, James Nguyen†, Wei Yu*, Erik Blasch‡, and Chao Lu*
*Department of Computer and Information Systems, Towson University, MD, USA
Emails: {jmcgif1, whatch2}@students.towson.edu, {wyu, clu}@towson.edu
†US Army Communications-Electronics Research, Development, and Engineering Center (CERDEC), MD, USA
Email: james.huy.nguyen@gmail.com
‡US Air Force Office Scientific Research, VA, USA
Email: erik.blasch@gmail.com

***Abstract*—As the worldwide adoption of smartphones nears complete saturation, the complexity and volume of mobile malware continue to progress, subverting what has become the largest set of targets with the most valuable rewards. At the same time, deep learning has become a powerful tool for data analysis and prediction, demonstrating significant efficacy in the detection of new and unforeseen malicious software through supervised learning. Despite these advances, the proper application of deep learning to malware detection could be further improved through additional data and model construction. In this work, we consider the detection of Android malware using static analysis techniques on multiple extractable data classes. Particularly, both Permission and Hardware Feature data are applied in various multimodal input scenarios and deep network shapes. Through experimental analysis, we demonstrate that the combination of both sets of data could improve overall performance, achieving as high as 94.5 % classification accuracy. In addition, using only a limited grid search, we observe that the largest multimodal network requires the lowest time to train while achieving equivalent or greater accuracy compared with our other models.**

***Index Terms*—Deep Learning, Malware Detection, Performance Tuning**

I. Introduction

The Android operating system continues to play an important role in the global smartphone marketplace. It is more important than ever that accurate and efficient mechanisms for detecting malware that target the Android OS are developed and widely deployed. Moreover, as techniques for applying machine learning such as deep neural networks (DNN) become powerful [1], [2], their use in malware detection is of significant interest, having been proven effective in a variety of detection and classification tasks [3], [4], [5].

As applied specifically to the case of Android malware, a number of research efforts have sought to analyze and detect intrusions and malicious applications through both static and dynamic analyses, along with other analyses [6], [7], [8], [9], [10]. Yet, while previous experiments have proven successful, questions still remain as to how best to improve performance in terms of prediction accuracy and resource use. To this end, the application of ensemble networks and multimodal data analysis show promising results. More specifically, the treatment of data by an ensemble of classifiers can improve final classification results, as the consideration of different modalities can be designed as separate subnetworks prior to final classification in a collective network.

In this work, we consider static analysis of Android Manifest data from application installation files implemented in a moultimodal manner. As the most rapid and easily accessible form of analysis, static analysis is powerful in malware prediction. Of particular interest, various shapes can be considered from application Permissions and Hardware Features as both single and separate modalities. Thus, there is a need to explore how to incorporate these two types of data for deep learning analysis.

In the application of deep learning for Android malware detection and classification, our contributions are as follows: (i) Static malware detection of Android applications extracts and exploits Permissions and Hardware Features from the Android application manifest. The paper compares the performance of identical structures of neural networks when making classifications based on permission data only, hardware feature data only, and hardware feature and permission data in combination. (ii) A multi-input neural network is implemented via Keras. Using multiple inputs, we direct permission and feature data into the network separately to achieve a higher degree of accuracy by initially processing each type of data independently. After passing through the initial processing layers, the two sides of the network are concatenated prior to final classification at the output layer. Two multi-input neural network models are compared against one another, and with the three single-input models. (ii) Extensive experiments demonstrate that the combination of both Permissions and Hardware features can improve overall performance, achieving as high as 94.5 % classification accuracy. Additionally, by leveraging a limited grid search, the results show that the largest multimodal network requires the lowest time to train while achieving equivalent or greater accuracy in comparison with the other models.

The remainder of this paper is as follows: Section II details the testing environment. Section III outlines our multimodal approach, system workflow, and test scenarios and parameters. Section IV describes the performance evaluation. Finally, in Section V, we provide come concluding remarks.

978-1-5386-9223-3/19/$31.00 ©2019 IEEE

II. Malware Testbed and Dataset

This section outlines our testing environment and the malware dataset utilized in the experiment.

A. Test Environment

The underlying deep learning framework implemented for our experiment is Google's TensorFlow [11], with Keras [12] as a front-end wrapper. In addition, several python libraries were utilized to facilitate the evaluation. In particular, Scikit-learn [13] is an open source machine learning library for Python, which was used to parse and vectorize extracted Permission and Hardware Feature data.

B. Dataset

The data for the experiment is drawn from a combined collection of 58,884 benign and malicious Android Packages (APKs). Specifically, this dataset consists of 19,273 malicious APKs compiled from a number of sources, including the Android Malware Genome Project [14] and the Contagio [15] malware repository, with the vast majority coming from the VirusShare [16] malware repository. The benign portion of the dataset consists of 38,941 APK samples downloaded from the Google Play Store [17], as well as from APKpure [18] and APKmirror [19], two popular third party Android marketplace mirrors. From the XML manifest of each application in this dataset, we extracted Permissions and Hardware Features as strings for classification, preprocessing the output of each APK for input into the neural network.

Permissions: Permissions exist to protect the data and privacy of Android users by restricting an application's access to sensitive data and system features [20]. Applications must request Permission to access these resources which may be granted or denied by the system, or prompt the user for the decision. Permissions are classified into either normal or dangerous permissions, and those permissions classified as dangerous will prompt the user to allow or deny them, either upon install, or during runtime. Additionally, permissions can also be classified as standard or custom, being either of the defined set native to the OS, or being intentionally created by the app developer for a specific purpose.

Hardware Features: A special class of permissions exists to restrict access to hardware features on a phone in the Android OS. These permissions, denoted as Hardware features, are identified in the manifest by the <uses-feature> tag, and may be either optional or required. More specifically, these do not share the same tag as the Permissions noted above (which instead have the tags <permission> and <uses-permission>), and are used to declare a single external hardware or software entity. They can also be designated as "required" for the app to run. Examples of Hardware feature requests would be to access resources such as a phone's camera or microphone.

III. Multimodal DNN Approach

In this section, we introduce a multimodal DNN approach to Android malware analysis and classification. This section discusses several essential concepts, the system workflow, and finally, the specific parameters and scenarios tested.

A. System Workflow

Feature extraction is the first step in analyzing the APKs. A series of scripts are run over the APKs to first dump information from the XML Manifests, and then format the data for input into the designed networks. Information is extracted from the APK Manifest using the Android Asset Packaging Tool (AAPT). AAPT is first used to create a raw dump of information from the APK Manifest, including software development kit (SDK) version, permissions, and hardware features, among other data.

Next, using a series of bash scripts and regular expressions, the data is parsed to extract the requested features and permissions for each application. The data is then reformatted into a single combined text file, featuring data for a single application per line. The result of this process is two long text files, one for data extracted from the benign applications, and the other from the malicious applications.

Once the data is formatted, it is then passed to the analysis script, where several machine learning libraries are invoked, namely Keras, Scikit-Learn, and Tensorflow (as previously noted). Tensorflow is not utilized directly, but rather incorporated as the back-end to be interfaced through Keras. Note that TensorFlow could be substituted with another framework (CNTK, Theano), however, we have chosen TensorFlow because it demonstrates better performance on our server-grade CPU-only hardware.

The analysis process in the classification script is as follows: First, the benign and malicious input files are passed to a Scikit-Learn tokenizer, which tokenizes each data file into a feature matrix based on a regular expression. Here, three regular expressions are defined, one to tokenize only the permission strings, one for only the Hardware Feature strings, and one that accepts both.

As the system processes the data, each tokenizer creates a feature matrix based on the tokens that it finds. Each unique token matching our expression creates a new column in the feature matrix, while each new line in the input file creates a new row. The process of tokenizing the data yields a vocabulary for each tokenizer, which is the set of unique tokens that it identified while processing the data. The vocabulary size for each tokenizer is shown in Table I.

TABLE I. Regex used

Data	Tokenizing Expression	Size
Hardware Features	\regex(?:\w|\.)+(?:hardware).(?:\w|\.)+	122
Permissions	\regex(?:\w|\.)+(?:permission).(?:\w|\.)+	16003
Combined	\regex(?:\w|\.)+(?:hardware|permission).(?:\w|\.)+	16125

The resulting matrix is $M \times N$, where M is the total number of APKs and N is the total number of unique features discovered. The resulting matrices are of dimension $77,884 \times 16,003$ for Permissions only and $77,884 \times 122$ for Hardware Features only, and the matrix for both combined is $77,884 \times 16,125$.

Finally, using supervised learning, an array of labels is created to match each feature matrix. These labels identify

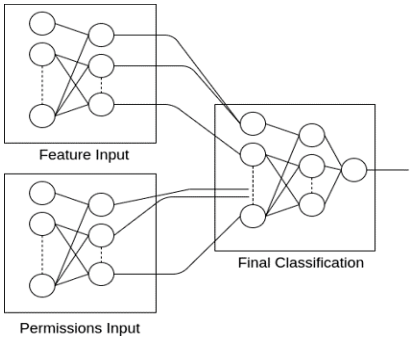

Fig. 1. Example Multimodal Network Input Configuration

whether a particular row of the feature matrix corresponds to a malicious or benign application. When the networks are trained, they will be passed a subset of the total dataset along with the corresponding labels. Also, when testing the network, predictions will be made with no knowledge of the labels, which will afterwards be measured for accuracy. With the feature matrix and corresponding labels prepared, the networks can then be trained and tested.

B. System Parameters and Scenarios

This section describes the parameters and scenarios tested. The differences between the various models tested are detailed, followed by descriptions of each of the hyper-parameters that were tuned.

1) Neural Network Models: In our tests, we set up several scenarios for comparison with three distinct network topologies. The first network type is a single-input network, meaning that there is only one input layer, with a single hidden layer of the base width (32 neurons). This network is tested in three data arrangements: *Permissions Only*, *Hardware Features Only*, and *Combined*. The second network is a small dual-input network, with two independent input layers, one each for the Hardware Feature and Permission data. Each input layer connects to a second dense network of the same width before they are concatenated together and connected to the output layer. The third network model maintains the same structure as the second but with an additional dense and dropout layer before and after the concatenated layer. In all, the result is three single-input networks, one small dual-input network, and one large dual-input network, or five networks in total.

Notice that the two dual-input network types are designed to be multimodal, treating the data individually and combining the two distinct modalities in the subsequent final classification network. An example of this network shape is shown in Fig. 1, the design of which allows for independent networks to be individually tuned for the dataset provided.

2) Model Parameters: An important component in producing effective neural networks is tuning their hyperparameters. Hyperparameters are variables that are set with the creation of the network that dictate many of the aspects of how it functions. Identifying the best hyperparameter values for a model is a difficult task. Thus, to obtain the best performance, several hyperparameters will be fixed and held constant across each model, while several others will be optimized via the grid search method.

TABLE II. Grid Search Parameters

Batch Size	Epochs	Input Ratio
8	4	.125
16	8	.25
32	16	
64	32	
128		

The fixed parameters are the Loss Function, Activation Function, Optimizer, Base Layer Width, and Dropout Rate, where applicable. All models use binary cross entropy as their loss function, ReLU as their activation function, and the Nesterov Adam optimizer (Nadam optimizer). In addition, the dropout rate is fixed at 10 % on models featuring dropout layers. A base layer width of 32 Neurons is also shared across each model. Notice that these values were all fixed because they demonstrated the best performance in preliminary testing, but overall the contributions of each were minor.

The parameters to be optimized via grid search are the Number of Epochs, Batch Size, and a third parameter which we define as the Input Ratio. For the dual-input models the input ratio dictates the base width of the Hardware Feature input layer, and subsequent dense layers proportional to the base width of the Permissions input layer. Within the data, there exists many more data points for software Permissions than Hardware features, and so we surmise that limiting the size of the Hardware feature layers in the model may improve performance.

3) Parameter Tuning: Using the grid search, a range of values for each parameter is defined and tested at every combination of values to find the optimal performance to be selected. This can be an immensely time consuming process. Thus, we implement a limited grid search over only select hyperparameters, while holding the others constant (as noted above). For our experiment, we chose to grid search for optimal values of batch size, epochs, and input ratio as they demonstrated the greatest effect on the results. Table II represents the grid search parameter used for each model.

Based on the results of the grid search, final hyperparameter values (batch size, epochs, and input ratio) are individually selected for each model. The grid search was performed as a preliminary to obtain the best performing values for evaluation. The final values of each model are shown in Table III. After the grid search is completed, the final networks could be constructed and their performance evaluated.

TABLE III. Final Grid Search Values for All DNN Models

Model	Batch Size	Epochs	Input Ratio
Small Dual-Input	16	32	.125
Large Dual-Input	128	32	.125
Permissions Single-Input	32	16	
Hardware Features Single-Input	16	32	
Combined Single-Input	32	32	

IV. Performance Evaluation

We now describe our performance evaluation in detail. First, we outline the performance metrics evaluated, and then provide the experimental results and interpretation thereof.

A. Evaluation Metrics

To compare and evaluate the performance of each network, we varied the training ratios at 20 %, 40 %, 60 %, and 80 % training versus testing. Cross validation was performed using a five-split stratified shuffle split. Accuracy and F1 Scores were calculated for each model at each training ratio, and averaged across the five splits. Each model was run with the top performing hyperparameter values detailed in Table III. To measure the relative performance of each model, we define their raw classification accuracy, F1 scores, and training time as metrics for comparison, as follows:

Accuracy is measured simply as the sum of all correct predictions across all five validation splits, divided by the sum of total predictions across all of the splits.

F1 Score is defined as the harmonic average of precision and recall, expressed as,

$$F1 = \frac{2 * (\text{Precision} * \text{Recall})}{\text{Precision} + \text{Recall}}, \tag{1}$$

where Precision is defined as the ratio of true positive predictions over total positive predictions as,

$$\text{Precision} = \frac{TP}{FP + TP}. \tag{2}$$

Likewise, Recall is defined as the ratio of true positives predictions over the sum of true positives and false negatives:

$$\text{Recall} = \frac{TP}{FN + TP}. \tag{3}$$

Also notice that TP represents True Positives (total positive cases that have been correctly classified as positive), FP is False Positives (total negative cases incorrectly classified as positive), and FN is False Negatives (total positive cases incorrectly identified as negative).

Training Time is simply time necessary to train a model. Notice that inference of a model on any individual application takes negligible time, but training the network involves consideration of the entire training dataset.

B. Evaluation Results

Figs. 2-4 show the experimental results. Specifically, Fig. 2 presents the accuracy score of each model at each training ratio, Fig. 3presents the F1 scores of each model, and Fig. 4 presents the training times. Notice that the green and red curves represent the single input networks of Hardware Features Only and Permissions Only datasets, which are clearly distinct from the other curves, especially in Figs. 2 and 3. More difficult to discern are the Dual-Input Small (blue), Dual-Input Large (orange), and single-input Combined (purple). Explicitly, the blue curve of Dual-Input Small is just below the other two in Figs. 2 and 3, while the purple Combined curve outperforms the Dual-Input Large for the first two training ratios of 20 % and 40 % before being overtaken in the final two training ratios of 60 % and 80 %. Regarding Fig. 4, all of the curves are clearly distinct.

Accuracy: Examining the classification accuracy of each model in Fig. 2, we see that the Hardware Features Only single-input model performs the worst, achieving a maximum of 80.7 % accuracy at a training ratio of 80%. This model improved very little with more training data, only by 0.7 % over the initial accuracy score of 80.0 % at a training ratio of 20 %. The next worst model is the Permissions Only single-input model, achieving a maximum accuracy of 93.44 % at a training ratio of 80 %. The Permissions Only model saw a greater improvement as the training ratio increased compared to the Hardware Features Only model, improving from 92.16 % to 93.44 % with a training ratio increase from 20 % to 80 %.

The top performers were the two Dual-Input models along with the Combined Data single-input model, which is the logical result. Each of these three models featured nearly identical performance in both accuracy and F1 scores. The overall best performer, in terms of accuracy, was the Large Dual-Input model with an accuracy of 94.62 % at a training ratio of 80%. The lowest accuracy of the top three performers was the Small Dual-Input model, whose accuracy was only 0.125 % lower than the Large Dual-Input model.

F1 Score: Concerning the F1 scores, as seen in Fig. 3, these largely mirrored that of the classification accuracies, with Permissions Only and Hardware Features Only models lagging behind the top cluster of combined data models. There is, however, a noticeably larger gap between the F1 score performance of the Hardware Feature Only single-layer model and the other models, compared to the difference in accuracy scores. Notice that a higher F1 score indicates both a better fit of the model to the trained data as well as better generalization in classifying new data. We can consider the low F1 score of the Hardware Feature Only model to indicate the limitation of the dataset and the resulting model, which is reasonable given the small feature of token size of the Hardware Feature dataset in general.

Training Time: Looking at the average training and testing time for each model in Fig. 4, there are several interesting observations. First, there is a largely linear trend in training time as the training ratio increases. Predictably, the single-input Hardware Features Only model is the quickest to train and make predictions, owing to its small size and considerably reduced feature vocabulary. Surprisingly, the next fastest performer was the Large Dual-Input network, which features not only the most layers and neurons, but is operating with the largest possible feature vocabulary. At an 80 % training ratio, the Large Dual-Input model took an average of 621 seconds to complete its training. In contrast, the Small Dual-Input model was the worst performer, averaging 1,921 seconds to complete training at the same training ratio. The reason for this can be attributed to the combination of batch sizes and epochs, which were selected for optimal performance for each model, as well as the size of the input vocabulary in each case.

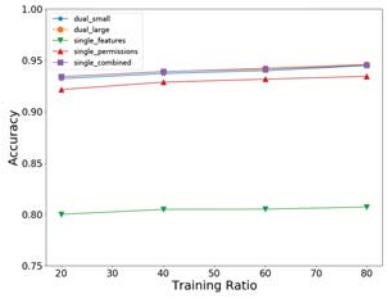

Fig. 2. Training Ratio vs. Accuracy

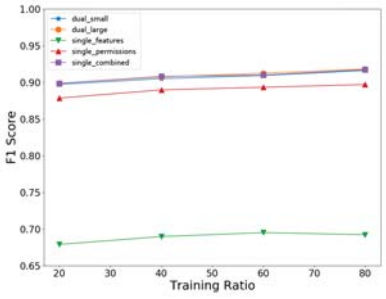

Fig. 3. Training Ratio vs. F1 Score

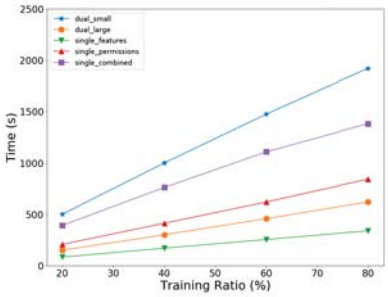

Fig. 4. Training Ratio vs. Training Time

The single-input Permissions Only network finished in the middle of the five scenarios, averaging 843 seconds to train at an 80 % training ratio, beaten only by the Large Dual-Input network and the single-input Hardware Features Only network. The single-input Combined network beat the Permissions Only network in F1 and accuracy scores, but trains considerably slower, averaging 1,382 seconds at a training ratio of 80 %.

The best performing model with consideration to both training time and performance metrics is the *Large Dual-Input model*. Though the differences in performance scores between the two Dual-Input models and the single-input Combined model are almost negligible, the large Dual-Input model is able to train and predict considerably faster. A possible reason is that the grid search performance results, in which it performed best at the larger batch size of 128. In comparison to the batch size of 16 on the Small Dual-Input network, and 32 on the one layer Combined model, the Large Dual-Input model views more data concurrently and so is able to process the data much faster, though all three models use 32 training epochs.

V. Final Remarks

This paper addresses the issue of detecting Android malware using static analysis on multiple extractable data classes. A novel multimodal DNN classifier is designed using Permission and Hardware feature data. Through our experiments, we conclude that the inclusion of both Hardware Feature and Permission data in our deep learning models will increase performance. In terms of classification accuracy and F1 score, the manner by which the two types of data are combined (single combined input or separate) has no significant effect. When additionally accounting for training time, larger and more complicated models may be able to outperform simpler models by virtue of a more favorable set of hyperparameters.

Acknowledgement

This work was supported in part by the US National Science Foundation (NSF) under grants: CNS 1350145, and the University System of Maryland (USM) Endowed Wilson H. Elkins Professorship Award Fund. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the agencies.

References

[1] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24 411–24 432, 2018.
[2] R. I. Hammoud, C. S. Sahin, E. P. Blasch, B. J. Rhodes, and T. Wang, "Automatic association of chats and video tracks for activity learning and recognition in aerial video surveillance," *Sensors*, vol. 14, pp. 19 843–19 860, 2014.
[3] W. Yu, H. Zhang, L. Ge, and R. Hardy, "On behavior-based detection of malware on android platform," in *2013 IEEE Global Communications Conference (GLOBECOM)*, Dec 2013, pp. 814–819.
[4] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen, and C. Lu, "Tuning deep learning performance for android malware detection," in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, June 2018, pp. 140–145.
[5] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, and V. C. M. Leung, "A survey on security threats and defensive techniques of machine learning: A data driven view," *IEEE Access*, vol. 6, pp. 12 103–12 117, 2018.
[6] H. Zhang, Y. Cole, L. Ge, S. Wei, W. Yu, C. Lu, G. Chen, D. Shen, E. Blasch, and K. D. Pham, "ScanMe Mobile: a cloud-based android malware analysis service," *SIGAPP Appl. Comput. Rev.*, vol. 16, no. 1, pp. 36–49, Apr. 2016.
[7] X. Su, D. Zhang, W. Li, and K. Zhao, "A deep learning approach to android malware feature learning and detection," in *2016 IEEE Trustcom/BigDataSE/ISPA*, Aug 2016, pp. 244–251.
[8] W. Yu, L. Ge, G. Xu, and X. Fu, "Towards neural network based malware detection on android mobile devices," *Pino R., Kott A., Shevenell M. (eds) Cybersecurity Systems for Human Cognition Augmentation. Advances in Information Security*, vol. 61, 2014.
[9] X. Wang, W. Yu, A. Champion, X. Fu, and D. Xuan, "Detecting worms via mining dynamic program execution," in *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007*, Sept 2007, pp. 412–421.
[10] W. Yu, G. Xu, Z. Chen, and P. Moulema, "A cloud computing based architecture for cyber security situation awareness," in *2013 IEEE Conference on Communications and Network Security (CNS)*, Oct 2013, pp. 488–492.
[11] "TensorFlow: An open source machine learning library everyone," 2017, https://www.tensorflow.org/.
[12] "Keras: The Python Deep Learning library," 2017, https://keras.io/.
[13] "scikit-learn – Machine Learning in Python," 2018, http://scikit-learn.org/stable/.
[14] "Android Malware Genome Project," 2011, http://www.malgenomeproject.org/.
[15] "Contagio Mobile," 2017, https://contagiominidump.blogspot.com/.
[16] "VirusShare," 2017, https://virusshare.com/.
[17] "Google Play," 2015, https://play.google.com/.
[18] "APKPure," 2017, https://apkpure.com/.
[19] "APKMirror," 2017, https://www.apkmirror.com/.
[20] Google Developers, "Permissions overview," 2017, https://developer.android.com/guide/topics/permissions/overview.