2019 11th International Conference on Cyber Conflict:
Silent Battle
T. Minárik, S. Alatalu, S. Biondi,
M. Signoretti, I. Tolga, G. Visky (Eds.)
2019 © NATO CCD COE Publications, Tallinn

Permission to make digital or hard copies of this publication for internal use within NATO and for personal or educational use when for non-profit or non-commercial purposes is granted providing that copies bear this notice and a full citation on the first page. Any other reproduction or transmission requires prior written permission by NATO CCD COE.

Neural Network-Based Technique for Android Smartphone Applications Classification

Roman Graf
Austrian Institute of Technology GmbH
Vienna, Austria
roman.graf@ait.ac.at

L. Aaron Kaplan
CERT.AT
Vienna, Austria
kaplan@cert.at

Ross King
Austrian Institute of Technology GmbH
Vienna, Austria
ross.king@ait.ac.at

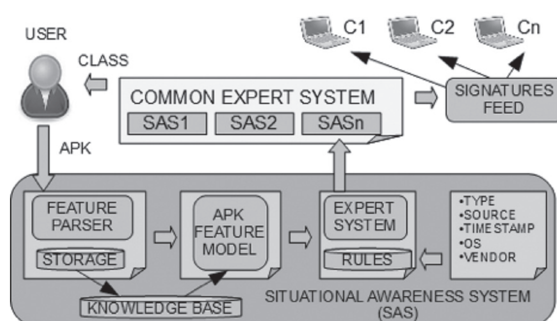
Abstract: With the booming development of smartphone capabilities, these devices are increasingly frequent victims of targeted attacks in the 'silent battle' of cyberspace. Protecting Android smartphones against the increasing number of malware applications has become as crucial as it is complex. To be effective in identifying and defeating malware applications, cyber analysts require novel distributed detection and reaction methodologies based on information security techniques that can automatically analyse new applications and share analysis results between smartphone users. Our goal is to provide a real-time solution that can extract application features and find related correlations within an aggregated knowledge base in a fast and scalable way, and to automate the classification of Android smartphone applications. Our effective and fast application analysis method is based on artificial intelligence and can support smartphone users in malware detection and allow them to quickly adopt suitable countermeasures following malware detection. In this paper, we evaluate a deep neural network supported by word-embedding technology as a system for malware application classification and assess its accuracy and performance. This approach should reduce the number of infected smartphones and increase smartphone security. We demonstrate how the presented techniques can be applied to support smartphone application classification tasks performed by smartphone users.

Keywords: *Cyber security, neural network, AI, smartphone, malware*

1. INTRODUCTION

The Android operating system for tablets, phones and smart devices is by far the most widespread mobile operating system in the world, with millions of active devices. Millions of new malware programs have been released for this platform in recent years. The market share of exploits that target the Android platform makes it the second most targeted platform for running exploit attacks. The goal of this paper is to train a neural network to evaluate the discoverability and explainability of upcoming attack patterns. Classification capabilities of neural networks are heavily reliant on the quality of the underlying datasets, and subsequently even more dependent on the granularity of extracted features. The presented technique (see Figure 1) will apply deep neural networks and supervised learning to evaluate the capabilities of detecting smartphone malware applications in Android. Currently there is a lack of technology supporting an integrated solution of large-scale feature extraction and neural network training. The goal of this approach is to release an open source framework that provides integrated functionality along the required workflow. This workflow comprises application source code extraction, feature composition, neural network training and analysis of results. The components of this system are executed at scale within Hadoop and GPU clusters. The platform supports publishing of the harvested ground-truth dataset, the extracted features and the trained neural network on an open data platform. To visualize the projects results and to raise awareness for malware applications prevention in the general public, a demonstrator was developed that allows live inspection of the trustworthiness of Android applications.

FIGURE 1. THE OVERVIEW OF ESTABLISHING THE CYBER SITUATIONAL AWARENESS USING NEURAL NETWORK FOR APK CLASSIFICATION.



The neural network approach is widely used for different analytical tasks. A machine learning framework based on word-embedding techniques can be used for the classification of text files. Standard machine learning algorithms are incapable of

processing strings or plain text in their raw form; rather, they require numbers as inputs to perform any type of calculations. In the word-embedding approach, words are mapped to numerical vectors. The difference from other language processing methods is that the embedding vector also keeps the context of the word in a sentence or file. This improves the overall accuracy of the prediction model, compared to simple counting of words in a file. Our approach provides a numerical representation of contextual similarities between Android Package (APK) features extracted in text format. Each feature is represented by a real-value vector with tens or hundreds of dimensions. In contrast, other methods, such as a one-hot encoding, employ thousands or millions of dimensions required for sparse word representations.

This paper is structured as follows. Section 2 gives an overview of related work and concepts. Section 3 explains the APK classification workflow including the feature extraction method and neural network training for APK classification. The expert system issues and related rule engine are covered in Section 4. Section 5 presents the experimental setup, applied methods end evaluation. Section 6 presents our conclusions.

2. RELATED WORK

The design of the presented framework is inspired by the DREBIN project (Rieck, 2004; Hoffmann, 2013), which combines a broad static analysis of gathered smartphone application features and applies machine learning for identifying patterns that are indicative of malware. The manifest and decompiled dex (Dalvik Executable) codes are scraped to extract feature sets and DREBIN utilizes a linear SVM algorithm (Shawe-Taylor, 2000), which assumes real-value inputs. The manifest file provides features such as requested hardware components and system granted permissions, declared components such as services or broadcast receivers and filtered intents which are used for inter-process communication. By analysing the disassembled bytecode, additional “hidden” features are gathered, such as restricted API calls, actually used permissions, calls to sensitive resources (e.g. frequently used for obfuscation) and a list of all network addresses. This demonstrated approach provides both effective detection rates and explainable results and was able to outperform related approaches as well as 9 out of 10 popular anti-virus scanners with a detection rate of 94% and a false positive rate of 1%, and reliably detect all malware families except Gappusin. DREBIN showed the importance of the different features sets and that their proper composition can lead to reliable and explainable detection results using neural networks and machine learning. While the methodology is well-documented, and the collected corpus of 120 thousand apps (including 22% malware samples) is published for academic re-use, the corpus itself is outdated (SDK level 12) and the DREBIN

framework and neural network itself are closed-source. Furthermore, DREBIN is highly restricted in its learning-based detection capabilities as the project targeted the smartphone as runtime and detection environment where such a dataset must be heavily maintained and updated. The SVM approach is limited by the choice of the kernel, which is a general weak point of SVM applications. Alternative algorithms employing categorical features and labels are Naive Bayes (Schütze, 2008), Logistic Regression (Cox, 1958) and Random Forests (Ho, 1995). Approaches based on decision trees such as Random Forests are very fast to train, but quite slow to create predictions once trained. A higher degree of accuracy requires additional trees, which means losing performance. Naive Bayes often serves as a robust method for data classification, but the vectors representing incident in Naive Bayes are larger than in word-embedding methods and also Naive Bayes classifiers make a very strong assumption on the shape of the data distribution. Further problems may result due to data scarcity, which can result in probabilities going towards 0 or 1, leading to numerical instabilities and worse detection results. Logistic regression like a Naive Bayes method requires that each feature in an incident is independent from all other features. Logistic regression models are also vulnerable to overconfidence as a result of sampling bias.

A brief overview of related approaches for the detection of Android malware lists some comparable methods for this task. Kirin (McDaniel, 2009) checks application permissions, Stowaway (Wagner, 2011) analyses API calls to detect overprivileged applications and RiskRanker (Jiang, 2012) identifies applications with different security risks. However, none of these approaches includes multiple features sets or features received from reverse-engineering the applications' source code, elements that were proven crucial for the detection results in DEBRIN. Open source tools such as Smali2 and Androguard3 enable dissecting the application's content for subsequent feature extraction and are evaluated for their use within framework's extraction pipeline. The dedicated analysis system DroidScope (Droidscope, 2012), which enables introspection at different layers of the Android platform, allows users to dynamically monitor applications in a protected environment at runtime. Methods of sandboxing try to mimic a real-world environment and aim to discover malicious behaviour but are limited due to sophisticated obfuscation methods used in modern malware. ParanoidAndroid (Bos, 2010) creates a virtual clone of the smartphone that runs in parallel on a dedicated server and synchronizes with the activities of the device. This configuration allows for monitoring the behaviour of applications on the clone without disrupting the functionality of the real device, but the resources required for a large number of devices are often not technically feasible. Dynamic analysis tools, such as DroidRanger (Jiang Y. Z., 2012) are suitable for filtering malicious applications from Android markets.

Dedicated open source frameworks in the domain of malware detection are rare. A prominent but outdated system is MobileSandbox (Hoffmann, 2013), which is designed to automatically analyse Android applications by combining a static and dynamic approach, which for example allowed the analyst to log system calls to native APIs. MobileSandbox provides a highly complex and immature system due to the enormous integration effort and customizations required for the Davlik virtual machine and emulators.

The advantage of the embedding method is that it not only takes into account features such as count and word context, but also learns automatically from examples. The autoencoder (Cheng-hua, 2008) makes use of neural networks, which are already in use by latent semantic analysis for text categorization to reduce dimensionality and to improve performance; but this method has the disadvantage of not using the context of the feature. Another application (Lee, 1999) employs an artificial neural network to improve text classifier scalability.

Classification methods implemented in these threat intelligence tools suffer from large vector sizes and are less effective as the number of features rises. The main drawback of existing text classification methods such as SVM or the Gensim tool is that they require a huge database for training to provide meaningful results. Another common disadvantage of these techniques is the lack of result transparency due to employing vectors containing real-valued numbers. These tools provide results, but it is difficult to explain how the results were calculated. In particular, the SVM approach is limited by the choice of the kernel. Another disadvantage is the inability to handle words that were not previously included in the training vocabulary.

Multiple researchers are developing an automated technology that will support an information classification system. An attempt to classify the relationships between documents and concepts employs principles of ontology. Currently, APKs can be classified based on the features included in the package and in source code. Contrary to this approach, we classify not only by data extracted from APK that can differ from dataset to dataset, but we also employ additional rules implemented in an expert system and take in account APK source, type, timestamp, dataset and other parameters. This technique provides more accurate prediction.

Neural networks with word-embeddings in general also require large training datasets, but for APK classification, taking into account the fact that we have multiple different datasets, we will train multiple models for each dataset and additionally employ a rule engine to produce accurate results compared to the case if we would just train one huge model ignoring intrinsic differences in the datasets. Consequently, for the particular use case of APK classification task, we suggest using the word-embedding neural

network solution that scales well because of the split-models concept and supportive rule engine, while maintaining a high level of accuracy. Our goal is to make a more accurate prediction for a specific dataset, employing the whole aggregated expert knowledge and applying expert rules.

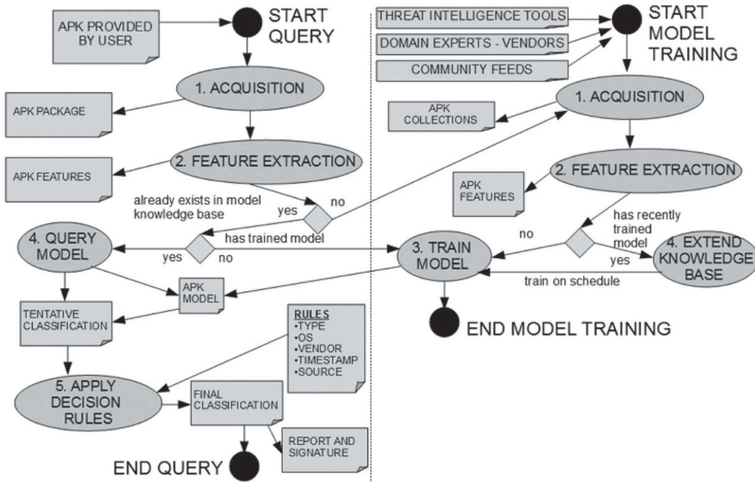
3. APPLICATION CLASSIFICATION METHOD

APK classification employs application features extraction and training of neural network to produce a model for the queries. Deep Learning is employed for learning in neural networks and describes a subset of machine learning algorithms that deal with accurately assigning weights across many neural network layers. Three main types of machine learning can be distinguished: Supervised, Unsupervised and Reinforcement Learning. Supervised learning can solve classification problems. Classification predicts previously defined categories for a given sample. In the case of Android malware these categories are binary: “benign” or “malware”. Supervised learning employs labelled training data to learn mapping functions from a given input (embedding vector in our case) to a desired output value. A supervised learning algorithm analyses the data through weights and activation functions that activate neurons and produces an inferred function, which is then used for mapping new samples or correctly determining classification labels for unseen instances.

A. Application Classification Using Neural Networks

Figure 1 provides an overview of establishing the cyber situational awareness using neural networks for APK classification. This approach is based on a knowledge base containing large number of labelled smartphone applications. This data can be provided by different vendors, collected at different times for particular operating systems, and may be separated by type of application. Therefore, for each use case (Situational Awareness System – SAS) we propose to have a separate expert system and associated decision rules. All such SAS systems are then aggregated in a common expert system, which performs final classification. A user uploads their APK package to the SAS. The system extracts features from this package, stores them for further analysis and queries an APK model that was trained based on knowledge base. The final classification result in the form of a report and signatures is disseminated by means of a signatures feed for subscribed clients C1-Cn.

FIGURE 2. THE WORKFLOW FOR FEATURE EXTRACTION AND CLASSIFICATION OF APK USING A NEURAL NETWORK APPROACH.



To employ the embedding method, features aggregated in text form must be converted into numerical values, since machine learning algorithms and deep learning architectures cannot process plain text. Therefore, each uploaded APK (see Figure 2) is converted into an array of strings, where each string represents a particular feature. Then strings are encoded by indices, and each feature string has a unique index. If this feature repeats in the APKs, we re-use its index. Finally, arrays of indexes are converted in one-hot encoded vectors, meaning that the position of each feature in the original feature set is encoded using “1” if a feature exists in the given place or “0” if not. After defining the number of latent factors expressed in the length of the embedding vector, we convert produced on-hot vectors into embedding vectors, giving an array of float numbers. Therefore, we create a list of embedding sequences for each APK with embedding vector representation of each feature. Embedding vectors are an input to the neural network.

The neural network is composed of an input embedding layer, a flattening layer and two hidden layers, where the model will be trained to classify APKs as either “benign” or “malware.” The flattening layer is required to enable a connection between the dense and embedding layers. We flatten the two-dimensional output matrix of the embedding layer (with one embedding for each feature in the input sequence of features) to a one-dimensional vector used by the dense layer.

B. Application Features Extraction

The workflow process is composed of two parts. One process is a neural network model training, where workflow acquires APK data from different sources such as community feeds, threat intelligence tools and domain experts, which are vendors or anti-malware producers. The model is trained and regularly updated by extended knowledge from new APK collections. The query workflow execution begins with reading a smartphone application package (see step 1 in Figure 2) provided by a user and parsing the extracted content for features extraction in step 2. For the acquisition computation we employ a parsing method developed by researchers who reimplemented the DREBIN parsing method described in the DREBIN paper (Rieck, 2004). By means of extracted features, we obtain an APK vector. If the given APK is not in the model, we additionally extend the model knowledge base for subsequent training. In the next step we train the APK model using a neural network (step 3) or a query trained model in step 4, applying the created feature vector. The model responds with a tentative classification. Finally, we calculate the APK classification employing an expert system and the decision rules in step 5. These rules comprise decision logic and expert profile settings that are specific for an organisation. Factors such as APK type, operating system, vendor, creation time and origin have an impact on the resulting decision. At the end we provide a report accompanied by an APK signature.

4. EXPERT SYSTEM

Table 1 lists the layers that are employed in the neural network, including their type, activation function, size and parameter number.

TABLE 1: DEPENDENCY CHART WITH INTERACTIONS AMONG THE RULES AND ASSOCIATED IMPACT FACTORS.

Rules/Actions	Install	Remove	Ignore	Alarm	Quarantine	Clean	Log
Neural network model classification (benign/malware)	+	+	+	+	+		+
Metadata (has conclusive feature)	+	+	+	+	+	+	+
File size (large/small)	+	+	+				+
File name (known/unknown)	+	+	+	+			+
Malware signature (known/unknown)	+	+	+	+	+	+	+
Operating system (Android/Mac OS/Win)	+	+	+				
Vendor (trusted/not trusted)	+	+	+	+	+		+
Type (game/office...)	+	+	+				
Source (trusted/not trusted)	+	+	+		+		+
Creation time (old/new)	+	+	+				+

To organize the knowledge base (see Figure 1), we must structure the information that has been obtained from the domain experts of APK domain and from conducted experiments.

We aim to achieve the following objectives:

1) Define typical scenarios for smartphone application handling;
2) Identify the parameters used by cyber experts for APK handling;
3) Define the linguistic labels that are used by the experts to classify measured values of each parameter and identify the range of each label when possible; and
4) Define typical scenarios for determining the conditional rules that relate these linguistic labels to specific control actions.

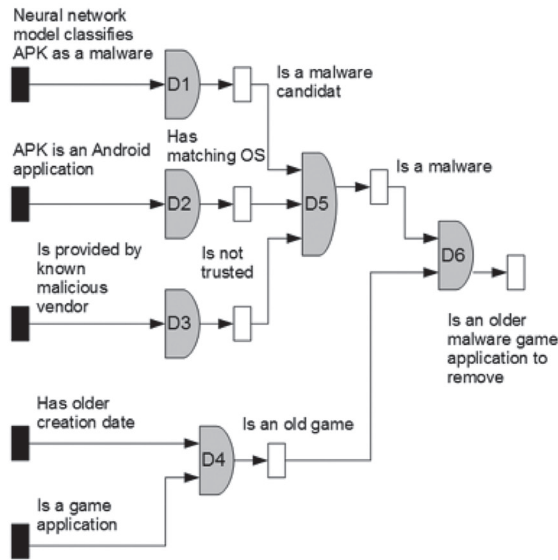
Knowledge acquisition for the knowledge base occurs through the domain expert. In our case, these are cyber analysts and SOC operators who provide the knowledge with typical application use cases, metrics and parameters that characterize the APK analysis processes. Information retrieved from the APK packages is processed by the customized domain model. This model enables structured and maintainable handling of analysed data and its storage in a database for further treatment. Inferred data is processed in an inference engine by rules application in order to provide the rationale for a particular analysis action. A user communicates with the expert system using GUI by sending a request query and receiving an advice in response.

The development of a knowledge base is an iterative process. Knowledge can be encoded, tested, added, updated and removed. Potential problems with rule definition and coverage are redundant rules, conflicting rules, rules that are subsumed by other rules, unreachable rules, inconsistent rules and circular rules chains. In order to avoid the rule-based systems faults described by Arman (2007), we generated a dependency chart that shows the interactions among the rules (Nguyen, 1985). The dependency chart presented in Table 1 gives an overview of the identified rules and associated impact on the knowledge base. The dependency chart helps to find potential rules problems and to keep an overview of the rules.

Among the most important rules (see Figure 3) are those regarding APK issues, like "neural network model classification", "metadata", "file size", "file name" and "malware signature". According to the requirements and circumstances for a particular APK, an expert could leverage these rules; for example, if a file name has a semantic meaning or if file size is of interest for analysis. Sometimes metadata contains important and useful information. The "malware signature" rule becomes significant in the case of known malware signature in an application source code. The

issue of “type” means that APK has significant risk if it belongs to particular type of application e.g. gambling. The issues “vendor” and “source” could have higher severity if the actors are known for producing malicious APKs.

FIGURE 3. AN EXAMPLE SELECTION OF FORWARD RULE CHAINING FOR SMARTPHONE APPLICATIONS CLASSIFICATION.



Rules are associated with related actions. Table 1 gives an overview of these relations. Upon the provided inputs, the rule engine can trigger actions, such as “install”, “remove”, “ignore”, “alarm”, “quarantine”, “clean” or “log” the given APK. The “clean” action is the most challenging and supposes an attempt to remove malware from the APK, which is applicable only by a high value of APK. Other actions are self-descriptive.

The previously defined rules should be organized in order to process input statements (assertions) and to infer appropriate action and conclusions. A process of the forward rule chaining for APK collection is presented in Figure 3. It is a process of moving from the “if” patterns (antecedents) to the “then” patterns (consequents) in a rule-based reaction system. We consider the antecedent as satisfied when “if” pattern matches the assertion. Assertions are depicted in the figure as the black rectangles on the input side and as the white rectangles on the output side. The rules are presented in the form of blue semi-circles (*R1-Rn*). The rule is triggered if all the antecedents are satisfied. A triggered rule is considered as fired if it produces a new assertion or performs an

action as output (white rectangle in the figure). Since our expert system is presently focused on APK collections, we do not need a conflict-resolution procedure to resolve possible rules conflicts. Managing dependencies, as depicted in Table 1, reduces the risk of conflicting rules and lists rules required to distinguish malware from other applications. A variable x acquires value as antecedent pattern is matched to assertion. For example, using information from well-established and reliable "FDroid tool", rule *R3* determines that an application stems from known malicious source:

R3: If ?x is provided by known malicious source
Then ?x is not trusted

The rule-based system starts APK classification with the rule *R1*. Suppose that particular APK was classified by the neural network model as malware. Then if the antecedent pattern "*?x is a malware*" matches that assertion, the value x becomes "*is a malware candidate*" and rule *R1* fires. Because application is an Android APK, rule *R2* fires, establishing that the document "*has matching OS*". Rule *R3* fires with the value "*is not trusted*". If two input assertions match an antecedent pattern, rule *R4* fires. The output assertions of the first three rules become the input assertion for the rule *R5* and if there is a match to the antecedent pattern the rule fires with the value "*is a malware*". Finally, if the input assertions of rule *R6* match, the rule fires with resulting action "*is an older malware game application to remove*".

The output of the rule-based system is a conclusion for a malware classification. The classification of the given APK is calculated based on the features of the associated APK. The inference engine performs conditional rules and classification analysis, infers appropriate action and formulates advice using relation of linguistic labels to specific control actions.

5. EXPERIMENTAL EVALUATION

A. Evaluation Data Set

The experimental dataset with ground-truth labels was provided by firms I and C and processed on an ABC server, which comprises Hadoop and GPU clusters. We split samples into test (5,640), validation (5,076) and training sets (45,676). For feature extraction we employ APK feature extractor described on a research site[1] and reimplemented on GitHub.[2]

B. Experimental Results and Interpretation

Classification of APK samples into benign and malware was evaluated employing techniques described in the previous sections. Features were extracted from APK

1 https://www.sec.cs.tu-bs.de/~danarp/drebin/
2 https://github.com/MLDroid/drebin

packages and converted in embedding vectors. Here is an example selection of loaded features:

- UsedPermissionsList_android.permission.VIBRATE
- UsedPermissionsList_android.permission.ACCESS_NETWORK_STATE
- UsedPermissionsList_android.permission.INTERNET
- BroadcastReceiverList_com.google.android.apps.analytics.AnalyticsReceiver
- SuspiciousApiList_Landroid/content/Context.getSystemService
- SuspiciousApiList_Landroid/app/Activity.getSystemService

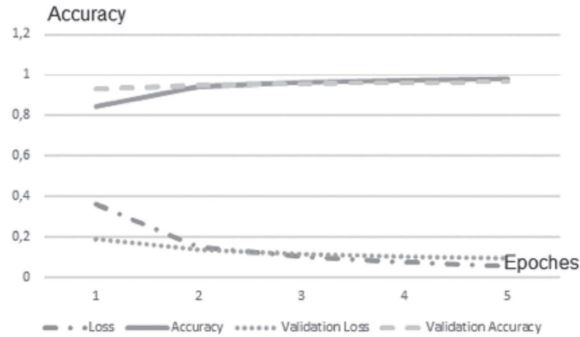
Embedding vectors describing loaded features were used as an input to a neural network. Table 2 lists the layers employed in the neural network including their type, activation function, size and parameter number. The total number of parameters used in the input and hidden layers during the training was 19,546,001. We employed an embedding approach for the input layer and sigmoid activation function for the dense layer. The total training time was 29,578 seconds. The model parameter settings for this particular training is presented in the fifth row in Table 3.

TABLE 2: SUMMARY OF THE NEURAL NETWORK TRAINING PROCESS.

Layer	Type	Activation Function	Size	Parameters #
Input layer	Embedding		200x30	19,245,900
Hidden layer 1	Flatten		6,000	0
Hidden layer 2	Dense	Sigmoid	50	300,050
Hidden layer 3	Dense	Sigmoid	1	51

Figure 4 visualizes the training results. We can see that, in general, the model training accuracy improves with every iteration (epoch) from 0.845 at the beginning to 0.982 at the end, which is sufficiently good; whereas training loss (error) of original information decreases from 0.362 to 0.056. This means that the outputs will be degraded compared to the original inputs, but it is an acceptable rate. Similarly, validation accuracy is in the range between 0.931 and 0.966. Validation loss decreases from 0.191 to 0.096.

FIGURE 4. ACCURACY AND LOSS CHARACTERISTICS BY NEURAL NETWORK TRAINING.



C. Evaluation Effectiveness

Table 3 shows the impact of parameter tuning on the neural network output and accuracy.

TABLE 3: IMPACT OF PARAMETER CHANGING ON NEURAL NETWORK OUTPUT AND ACCURACY.

LR	MVL	EVL	Time	TL	TA	VL	VA	NNA	TP	FP	FN	TN	TPR	FPR
0.001	200	30	1,287	0.0050	0.9989	0.1168	0.9663	99.936	2,880	82	99	2,659	96	2
0.01	200	30	3,231	0.0082	0.9980	0.1398	0.9639	99.875	2,719	163	44	2,714	98	5
0.0001	200	30	31,769	0.0432	0.9866	0.0919	0.9697	98.885	2,761	121	64	2,694	97	4
0.0001	100	30	31,335	0.0497	0.9840	0.0902	0.9675	98.791	2,761	121	64	2,694	97	4
0.0001	50	30	29,578	0.0563	0.9819	0.0961	0.9657	98.640	2,773	109	85	2,673	97	3
0.001	50	30	28,852	0.0047	0.9989	0.1446	0.9547	99.927	2,824	58	147	2,611	95	2
0.01	50	30	5,843	0.0107	0.9973	0.1381	0.9618	99.877	2,783	99	97	2,661	96	3
0.01	100	30	3,836	0.0094	0.9974	0.1465	0.9675	99.840	2,709	173	40	2,718	98	5
0.001	100	30	3,957	0.0047	0.9988	0.1256	0.9667	99.796	2,812	70	114	2,644	96	2
0.001	200	20	16,673	0.0045	0.9988	0.1395	0.9665	99.873	2,764	118	54	2,704	98	4
0.001	200	10	496	0.0055	0.9986	0.1373	0.9565	99.811	2,823	59	156	2,602	94	2
0.01	50	10	559	0.0049	0.9988	0.1638	0.9636	99.859	2,754	128	79	2,679	97	4
0.2	5	5	951	0.1784	0.9533	0.2922	0.9033	95.262	2,316	566	92	2,666	96	17
0.5	5	5	1217	0.2898	0.9156	0.3325	0.8936	91.925	2,138	744	99	2,659	95	21

Multiple factors can impact characteristics of the neural network model; some of them are depicted in Table 3. These factors are optimization algorithm, maximal length of the embedding vector, dense units number, activation functions, number of training

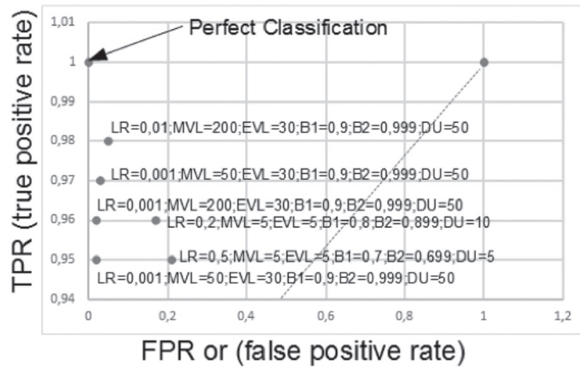
epochs, learning rate (LR), maximal vector length (MVL) and embedding vector length (EVL). The characteristics of the model are training loss (TL), training accuracy (TA), validation loss (VL), validation accuracy (VA), total training accuracy (NNA), time in seconds, number of hidden layer parameters and classification accuracy expressed in receiver operating characteristic (ROC) points. Some well-known and established default settings for tested neural network problems were applied in our evaluation. As an optimization algorithm for the learning model we selected "Adam", which is an extension to stochastic gradient descent that is widely adopted for deep learning applications in natural language processing. This method differs from standard stochastic gradient descent by changing the learning rate during training. This algorithm can be tuned using parameters such as: "alpha", learning rate (0.001); "beta1", the exponential decay rate for the first moment estimates (0.9); "beta2", the exponential decay rate for the second-moment estimates (0.999); "epsilon", a very small number to prevent any division by zero in the implementation (1E-8); and "decay", the learning rate decay over each update (0.0).

During the APK's classification calculation using the neural network, there was a minor fluctuation of accuracy value (between 95.65 and 99.36). This is because the model employs a random weights initialization. Therefore, it is possible that the highest level of accuracy can be achieved with different parameter configurations. In the test scenario, we investigated the provided test APK collection to classify those applications by threat level (malware or benign) without involvement of a human analyst. Due to the large number of possible configurations in Table 3, we describe only the selected configurations, which demonstrate typical cases. LR is presented in the first column. The second column shows the MVL of the extracted features. In the third column, we show the length of embedding vector. Column "time" depicts the time required to train a model with the given parameter settings. The next five columns are related to the model training process and show training and validation accuracy and error. The final six columns show ROC values to assess evaluation accuracy based on labelled training dataset.

The figure shows that the most productive settings for highest accuracy (up to 99.93) are LR=0.001, MVL=200, EVL=30, whereas "LR" and "MVL" are dominating. For a given training collection, the most accurate classification (TPR=97, FPR=3) was achieved by LR=0.0001, MVL=50, EVL=30. The smallest duration for model training was 496 seconds (LR=0.0001, MVL=200, EVL=10) and the longest operation time was 31,769 seconds with settings (LR=0.0001, MVL=200, EVL=30). This difference can be explained by the different embedding vector sizes. The larger the vector, the longer it takes to calculate the model. This evaluation also gives a simple overview of the detected impact of a particular setting, such as "EVL" for calculation speed, "LR" for learning accuracy and "maximal input vector length" for classification accuracy.

Having evaluated the model for different parameter configurations, we can conclude that a smaller LR provides higher accuracy, while employing more time for calculation. The MVL size has limited impact on the presented model, since APKs comprise a relatively small number of significant features, although the longer the EVL the more accurate the result. To prove that the remaining parameters were selected optimally and that their change would reduce the overall quality and accuracy of the model, in last two measurements presented in Table 3 we additionally reduced the "beta1" and "beta2" parameters (0.8, 0.899 and 0.7, 0.699) and the number of dense units of the activation function to 10 and 5 respectively. The reduced accuracy of the last two results confirms our hypothesis that the noted settings provide the best possible result, thus making model optimization easier. Higher accuracy is also related to the number of training parameters in the dense hidden layers of the model, which ranges between 130 and 1,200,200. The number of these parameters is dependent on all the other aforementioned settings.

FIGURE 5. ROC PLOT OF NEURAL NETWORK TRAINING.



The classification effectiveness can be determined in terms of a Relative Operating Characteristic (ROC) using the labelled ground-truth query dataset. The SA analysis makes use of the separation of the provided APK samples into the two groups "benign" and "malware" provided by domain experts. For example, in the first sample in Table 3, the provided algorithm detected 2,880 TP (True Positive), 82 FP (False Positive), 2,659 TN (True Negative) and 99 FN (False Negative) APKs. The primary statistical performance metrics for ROC evaluation are sensitivity (highest is 0.98) or true positive rate and false positive rate (lowest is 0.02). For the first sample, the associated ROC value is represented by the point (0.02, 0.96). The ROC space (see Figure 5) demonstrates that the calculated FPR and TPR values for the evaluated categories are located very close to the so-called "perfect" classification point (0, 1). The distribution

of collection points above the red diagonal demonstrates quite good classification results that could be improved by refining the model settings. Two ROC points with deliberately roughly selected parameters are still situated above the red line, but as expected shifted lower, away from the perfect classification point. The calculation results demonstrate that the calculated classification values for the query APKs are located very close to the labelled classification. These results demonstrate that an automatic approach for APK classification of the method described is very effective and is a significant improvement on manual analysis. Therefore, an analysis method based on neural network technique can be suggested as an effective method for APK classification, and as a supporting method to establish cyber SA. The results of the analysis confirm our hypothesis that an automated approach is able to reliably classify APKs, thus making analysis of a large number of APKs a feasible and affordable process. However, further research is required to improve the decision and accuracy metrics of this method.

6. CONCLUSIONS

In this work we have presented an automated approach to classify Android smartphone applications (APKs) for establishing cyber situational awareness using neural networks. We have combined expertise gathered during the development of methods for application features extraction with the power of the neural network approach and expert system for decision support.

The main contribution of this work is a real-time automatic solution that can classify smartphone applications as either "malware" or "benign" in a fast and effective manner based on a large number of labelled applications, in order to detect malware applications and to secure user devices. The presented method employs a knowledge base collected from domain experts to detect situational awareness risks. Ultimately, our research will lead to the creation of automated security assessment tools with more effective handling of smartphone applications.

REFERENCES

Arman, N. (2007). Fault detection in dynamic rule bases using spanning trees and disjoint sets. *The International Arab Journal of Information Technology, Vol. 4, No. 1*, pp. 67-72. Palestine.

Auria, L. (2008). Support Vector Machines (SVM) as a Technique for Solvency Analysis. *DIW Berlin*.

Bos, H. (2010). Paranoid android: Versatile protection for smartphones. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*.

Cheng-hua, Y. B.-b. (2008). Latent semantic analysis for text categorization using neural. *in Knowledge-Based Systems, 21*, pp. 900-904.

Cox, D. R. (1958). The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, (pp. 215-242). Royal Statistical Society, Wiley.

Droidscope, L.-K. Y. (2012). Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *Proc. of USENIX Security Symposium*, (pp. 393–407).

Ho, T. K. (1995). Random decision forests. *Proceedings of 3rd International Conference on Document Analysis and Recognition*, (pp. 278-282).

Hoffmann, M. S. (2013). MobileSandbox: Looking Deeper into Android Applications. In *Proc. 28th International ACM Symposium on Applied Computing (SAC)*.

Jiang, M. G. (2012). Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, (pp. 281–294).

Jiang, Y. Z. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

Lee, S. L. (1999). Feature reduction for neural network based text categorization. In *Proceedings 6th International Conference on Advanced Systems for Advanced Applications*, (pp. 195-202). Hsinchu.

McDaniel, W. E. (2009). On lightweight mobile phone application certification. In *13 Proc. of ACM Conference on Computer and Communications Security (CCS)*, (pp. 235-245).

Molloy, H. P.-R. (2012). Using probabilistic generative models for ranking risks of android apps. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, (pp. 241–252).

Rieck, D. A. (2004). *Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket*. 21th Annual Network and Distributed System Security Symposium (NDSS).

Schütze, C. D. (2008). Introduction to Information Retrieval. *Cambridge University Press*. New York, USA.

Shawe-Taylor, N. C. (2000). *An introduction to support vector machines and others*. Cambridge University Press.

T. A. Nguyen, W. A. (1985). Checking an Expert System Knowledge Base for Consistency and Completeness. In *Proc of IJCAI-85*, (pp. 375-378).

Wagner, A. P. (2011). Android permissions demystified. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, (pp. 627–638).