Deep Android Malware Detection

Niall McLaughlin[*] , Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer
Centre for Secure Information Technologies (CSIT)
Queen´s University Belfast, UK

Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupe, Gail Joon Ahn
Center for Cybersecurity and Digital Forensics
Arizona State University, USA

ABSTRACT

In this paper, we propose a novel android malware detection system that uses a deep convolutional neural network (CNN). Malware classification is performed based on static analysis of the raw opcode sequence from a disassembled program. Features indicative of malware are automatically learned by the network from the raw opcode sequence thus removing the need for hand-engineered malware features. The training pipeline of our proposed system is much simpler than existing n-gram based malware detection methods, as the network is trained end-to-end to jointly learn appropriate features and to perform classification, thus removing the need to explicitly enumerate millions of n-grams during training. The network design also allows the use of long n-gram like features, not computationally feasible with existing methods. Once trained, the network can be efficiently executed on a GPU, allowing a very large number of files to be scanned quickly.

CCS Concepts

•**Security and privacy** → **Malware and its mitigation;** *Software and application security;* •**Computing methodologies** → *Neural networks;*

Keywords

Malware Detection, Android, Deep Learning

1. INTRODUCTION

Malware detection is a growing problem, especially in mobile platforms. Given the proliferation of mobile devices and their associated app-stores, the volume of new applications is too large to manually examine each application for malicious behavior. Malware detection has traditionally been based on manually examining the behavior and/or de-compiled code of known malware programs in order to design malware signatures by hand. This process does not easily scale to large numbers of applications, especially given the static nature of signature based malware detection, meaning that new malware can be designed to evade existing signatures. Consequently, there has recently been a large volume of work on automatic malware detection using ideas from machine learning. Various methods have been proposed based on examining the dynamic application behavior [18, 21], requested permissions [14, 16, 19] and the n-grams present in the application byte-code [7, 11, 10]. However many of these methods are reliant on expert analysis to design the discriminative features that are passed to the machine learning system used to make the final classification decision.

Recently, convolutional networks have been shown to perform well on a variety of tasks related to natural language processing [12, 26]. In this work we investigate the application of convolutional networks to malware detection by treating the disassembled byte-code of an application as a *text* to be analyzed. This approach has the advantage that features are automatically learned from raw data, and hence removes the need for malware signatures to be designed by hand. Our proposed malware detection method is computationally efficient as training and testing time is linearly proportional to the number of malware examples. The detection network can be run on a GPU, which is now a standard component of many mobile devices, meaning a large number of malware files can be scanned per-second. In addition, we expect that as more training data is provided the accuracy of malware detection will improve because neural networks have been shown to have a very high learning capacity, and hence can benefit from very large training-sets [20].

Our proposed malware detection method takes inspiration from existing n-gram based methods [7, 11, 10], but unlike existing methods there is no need to exhaustively enumerate a large number of n-grams during training. This is because the convolutional network can intrinsically learn to detect n-gram like signatures by learning to detect sequences of opcodes that are indicative of malware. In addition, our proposed method allows very long n-gram type signatures to be discovered, which would be impractical if explicit enumeration of all n-grams was required. The malware signatures found by the proposed method may be complementary to those discovered by hand as the automated system will have different strengths and biases from human analysts, therefore they could be valuable for use in conjunction with conventional malware signatures databases. Once our sys-

[*]Corresponding author: n.mclaughlin@qub.ac.uk

CODASPY'17, March 22 - 24, 2017, Scottsdale, AZ, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4523-1/17/03. . . $15.00

DOI: http://dx.doi.org/10.1145/3029806.3029823

tem has been trained, large numbers of files can be efficiently scanned using a GPU implementation, and given that new malware is constantly appearing, a useful feature of our proposed method is that it can be re-trained with new malware samples to adapt to the changing malware environment.

2. RELATED WORK

2.1 Malware Detection

Learning based approaches using hand-designed features have been applied extensively to both dynamic [18, 21] and static [23, 22, 25] malware detection. A variety of similar approaches to static malware detection have used manually derived features, such as API calls, intents, permissions and commands, with different classifiers such as support vector machine (SVM) [5], Naive Bayes, and k-Nearest Neighbor [19]. Malware detection approaches have also been proposed that use static features derived exclusively from the permissions requested by the application [14, 16].

In contrast with approaches using high-level hand-designed features, n-grams based malware detection uses sequences of low-level opcodes as features. The n-grams features can be used to train a classifier to distinguish between malware and benign software [10]. Perhaps surprisingly, even a 1-gram based feature, which is simply a histogram of the number of times each opcode is used, can distinguish malware from benign software [7]. The length of the n-gram used [10] and number of n-gram sequences used in classification [7] can both have an effect on the accuracy of the classifier. However increasing either parameter can massively increase the computational resources needed [7], which is clearly a disadvantage of standard n-gram based malware detection approaches. N-grams method also require feature selection to reduce the length of the feature-vector, which would otherwise be millions of elements long in the case of long n-grams. In this work we propose a method that allows very long n-grams features to be used, and allows an n-grams classifier to be trained in a much more efficient manner, based on neural networks.

2.2 Neural Networks

Recently, convolutional neural networks (CNNs) have shown state-of-the-art performance for object recognition in images [20] and natural language processing (NLP) [12]. In NLP, local patterns of symbols, known as n-grams, have been used as features for a variety of tasks [27]. It has recently been shown that if sufficient training data is available, very deep CNNs can outperform traditional NLP methods [26] across a range of text classification tasks. We postulate that static malware analysis has much in common with NLP as the analysis of the disassembled source code of a given program can be understood as a form of textual processing. Therefore, techniques such as CNNs have huge potential to be applied in the field of malware detection.

A variety of approaches to malware detection using other neural network architectures have been proposed. Several of the proposed methods are based on learning which sequences of operating system calls or API calls are indicative of malware [15, 9, 8] during dynamic analysis. The existing neural network based approaches to malware detection differ from our proposed method as they make use of a virtual machine to capture dynamic behavioural features [15, 9, 8]. This may prove problematic given that malware is often designed to detect when it is being run in a virtual environment in order to evade detection. Other existing neural network based malware detection methods use hand-designed features, which may not be the optimal way to detect malware [17]. We will attempt to address the limitations of existing neural network based malware detection methods, by using a novel static analysis method based on a CNN architecture that automatically learns an appropriate feature representation from raw data.

In this work we apply convolutional neural networks to the problem of malware detection. The CNN learns to detect patterns in the disassembled byte-code of applications that are indicative of malware. Our approach has several advantages over existing methods of malware detection, such as those based on high-level hand-designed features and those based on detection of n-grams. Scalability and performance are major drawbacks of existing n-gram based approaches, as the length of the feature vector grows rapidly when increasing the n-gram length. In contrast, our approach eliminates the need for counting and storing millions of n-grams during training and can learn longer n-grams than conventional methods used for malware detection. The improved efficiency makes it possible to use our proposed method with larger datasets, where the use of traditional methods would be intractable. Our whole system is jointly optimized to perform feature extraction and classification simultaneously by showing the system a large number of labeled samples. This removes the need for hand-designed features, as features are automatically learned during supervised network training, and removes the need for an ad-hoc pipeline consisting of feature-extraction, feature-selection and classification, as feature extraction and classification are optimized together. The existence of a fully end-to-end system also saves time when the system is presented with new malware to be recognized, as the network can easily be updated by simply increasing the size of the training-set, which may also improve its overall accuracy. Finally, the features discovered by our method may be different from, and complementary to, those discovered by manual analysis.

3. METHOD

In this work we propose a malware detection method that uses a convolutional network to process the raw Dalvik byte-code of an Android application. The overall structure of the malware detection network is shown in Fig. 2. In the following section we will first explain how an Android application is disassembled to give a sequence of raw Dalvik byte-codes, and then explain how this byte-code sequence is processed by the convolutional network.

3.1 Disassembly of Android Application

In our system, the preprocessing of an application consists of disassembling the application and extracting opcode sequences for static malware analysis, as shown in Fig.1. An Android application is an *apk* file, which is a compressed file containing the code files, the *AndroidManifest.xml* file, and the application resource files. A code file is a *dex* file that can be transformed into *smali* files, where each *smali* file represents a single class and contains the methods of such a class. Each method contains instructions and each instruction consists of a single opcode and multiple operands. We disassemble each application using *baksmali* [1] to obtain the *smali* files that contain the human-readable Dalvik byte-code of the application, then extracting the opcode sequence from each method, discarding the operands. As the result of the preprocessing we obtain all the opcode sequences from all the classes of the application. The opcode sequences from all classes are then concatenated to give a single sequence of opcodes representing whole application.

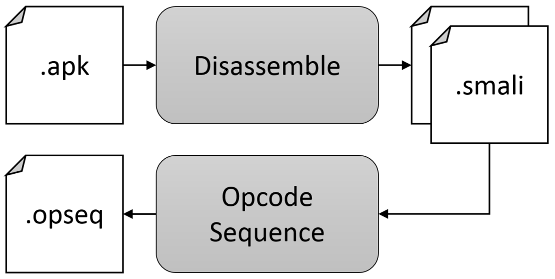

Figure 1: Work-flow of how an Android application is disassembled to produce an opcode sequence.

3.2 Network Architecture

3.2.1 Opcode Embedding Layer

Let $X = \{x_1 ... x_n\}$ be a sequence of opcode instructions encoded as one-hot vectors, where x_n is the one-hot vector for the n'th opcode in the sequence. To form a one-hot vector we associate each opcode with a number in the range 1 to D. In the case of Dalvik, where there are currently 218 defined opcodes, $D = 218$ [2]. The one-hot vector x_n is a vector of zeros, of length D, with a '1' in the position corresponding with the n'th opcode's integer mapping. Any operands associated with the opcodes were discarded during disassembly and preprocessing, meaning malware classification is based only on patterns in the sequence of opcodes.

Opcodes in X are projected into an embedding space by multiplying each one-hot vector by a weight matrix, $W_E \in \mathbb{R}^{D \times k}$, where k is the dimensionality of the embedding-space as follows

$$p_i = x_i W_E \tag{1}$$

projection of all opcodes in X, the program is represented by a matrix, P, of size $n \times k$, where each row, p_i, corresponds to the representation of opcode x_i. The weights in W_E, and hence the representation for each opcode, are initialized randomly at first then updated by back-propagation during training along with the rest of the network's parameters.

The purpose of representing the program as a list of one-hot vectors then projecting into an embedding space, is that it allows the network to learn an appropriate representation for each opcode as a vector in a k-dimensional continuous vector space, $\mathbb{R}^k$ where relationships between opcodes can be represented. The embedding space may encode semantic information for example, during training the network may discover that certain opcodes have similar meanings or perform equivalent operations, and hence should be treated similarly by deeper network layers for classification purposes. This can be achieved by projecting those opcodes to nearby points in the embedding space, while very different opcodes will be projected to distant points. The number of dimensions used in the embedding space may influence the network's ability to perform such semantic mapping, hence using more dimensions may, up to a point, give the network greater flexibility in learning the expected highly non-linear mapping from sequences of opcodes to classification decisions.

3.2.2 Convolutional Layers

In our proposed network we use one or more convolutional layers, numbered from 1 to L, where l refers to the l'th convolutional layer. The first convolutional layer receives the $n \times k$ program embedding matrix P as input, while deeper convolutional layers receive the output of the previous convolutional layer as input. Each convolutional layer has m_l filters, which are of size $s_1 \times k$ in the first layer, and of size $s_l \times m_{l-1}$ in deeper layers. This means filters in the first layer can potentially detect sequences of up to s_1 opcodes. During the forward pass of an example through a convolutional layer, each of the m_l convolutional filters produces an activation map $a_{l,m}$ of size $n \times 1$, which can be stacked together to produce, a matrix, A_l, of size $n \times m_l$. Note that before applying the convolutional filters we zero-pad the start and end of the input by $s_l/2$ to ensure that the length of the output matrix from the convolutional layer is the same as the length of its input. The convolution of the first layer filters with program embedding matrix P can be denoted as follows

$$a_{l,m} = relu(Conv(P)_{W_{l,m}, b_{l,m}}) \tag{2}$$

$$A_l = [a_{l,1} \,|\, a_{l,2} \,|\, ... \,|\, a_{l,m}] \tag{3}$$

where $w_{l,m}$ and $b_{l,m}$ are the respective weight and bias parameters of the m'th convolutional filter of convolution layer l, where $Conv$ represents the mathematical operation of convolution of the filter with the input, and where the rectified linear activation function, $relu(x) = \max\{0, x\}$, is used. In deeper layers the convolution operation is similar, however we replace input matrix P in Eq. 2 by the output matrix from the previous convolutional layer, A_{l-1}. Given output matrix A_L from the final convolutional layer, max-pooling [27] is then used over the program length dimension as follows

$$f = [\max(a_{L,1}) \,|\, \max(a_{L,2}) \,|\, ... \,|\, \max(a_{L,m})] \tag{4}$$

to give a vector f of length m_L, which contains the maximum activation of each convolutional filter over the program length. Using max-pooling over the length of the opcode sequence allows a program of arbitrarily length to be represented by a fixed-length feature vector. Moreover, selecting the maximum activation of each convolutional filter using max-pooling also focuses the attention of the classification layer on parts of the opcode sequence that are most relevant to the classification task.

3.2.3 Classification Layers

Finally, the resulting vector f is passed to a multi-layer perceptron (MLP), which consists of a full-connected hidden layer and a full-connected output layer. The purpose of the MLP is to output the probability that the current example is malware. The use of the MLP with hidden layer allows high-order relationships between the features extracted by the convolutional layer to be detected [6] and used for clas-

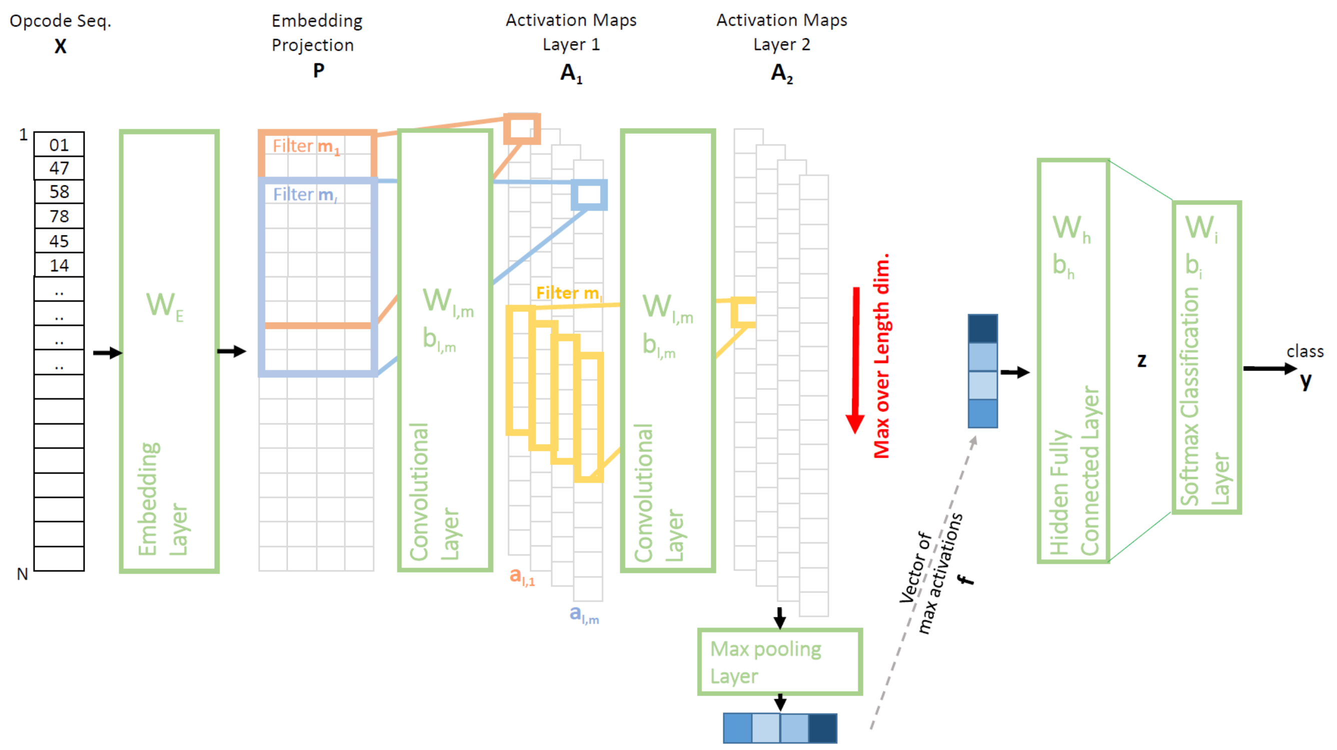

Figure 2: Malware Detection Network Architecture.

sification. We can write the hidden layer as follows

$$z = relu(W_h f + b_h) \tag{5}$$

where W_h, b_h, are the parameters of the fully-connected hidden layer, and where the rectified linear activation function is used. Finally, the output, z, from the MLP is passed to a soft-max classifier function, which gives the probability that program X is malware, denoted as follows

$$p(y = i|z) = \frac{\exp(w_i^T z + b_i)}{\sum_{i'=1}^{I} \exp(w_{i'}^T z + b_{i'})} \tag{6}$$

where w_i and b_i denote the parameters of the classifier for class $i \in I$, and the label y indicates whether the current sample is either malware or benign. The softmax classifier outputs the normalized probability of the current sample belonging to each class. As malware classification is a two class problem (benign/malware) i.e., $I = 2$ and z is a two element vector. Other applications such as the problem of malware family classification, could be targeted by increasing the number of classes, I, to be equal to the number of malware families to be classified.

3.3 Learning Process

Given the above definitions, the cost function to be minimized during training for a batch of b training samples, $\{X^{(1)} \dots X^{(b)}\}$, can be written as follows

$$C = -\frac{1}{b} \sum_{j=1}^{b} \sum_{i=1}^{I} 1\{y^{(j)} = i\} log\, p(y^{(j)} = i|z^{(j)}) \tag{7}$$

where $z^{(j)}$ is the vector output after applying the neural network to example training example $X^{(j)}$, where $y^{(j)}$ is the provided correct label for the example $X^{(j)}$, and where $1\{x\}$ is an indicator function that is 1 if its argument x is true and is 0 otherwise. The cost is dependent on both the parameters of the neural network, Θ, i.e. the weights and bias across all layers -W_E, $w_{l,m}$, $b_{l,m}$,W_h, b_h,w_i, and b_i - and on the current training sample. The objective during training is to update the network's parameters, which are initialized randomly before training begins, to reduce the cost. This update is performed stochastically by computing the gradient of the cost function with respect to the parameters, $\frac{\partial C}{\partial \Theta}$, given the current batch of samples, and using this gradient to update the parameters after every batch to reduce the cost as follows

$$\Theta^{(t+1)} = \Theta^{(t)} - \alpha \frac{\partial C}{\partial \Theta} \tag{8}$$

where α is a small positive real number called the learning rate. During training the network is repeatedly presented with batches of training samples in randomized order until the parameters converge.

To deal with an imbalance in the number of training samples available for the malware and benign classes, the gradients used to update the network parameters are weighted depending on the label of the current training sample. This helps to reduce classifier bias towards predicting the more populous class. Let the number of malware samples in the training-set be M and number of benign samples in the training-set be B. Assuming there are more samples of benign software than malware, the weight for malware samples is $1 - M/(M + B)$ and the weight for benign samples is $M/(M + B)$ i.e. the gradients are weighted in inverse proportion to the number of samples for each class.

Note that a consideration when designing our proposed architectures was to keep the number of parameters relatively low, in order to help prevent over-fitting given the relatively small number of training samples usually available. A typical deep network may have millions of parameters [20], while our malware detection network has only tens of thousands of parameters, which drastically reduces the need for large numbers of training samples.

4. RESULTS

In order to evaluate the performance of our approach a set of experiments was designed. The architecture used in all experiments had only a single convolutional layer. This architecture was used because the available datasets have a relatively small number of training samples which means that networks with large numbers of parameters could be prone to over-fitting. Convolutional networks with only a single convolutional layer have been shown to perform well on natural language text classification tasks [27]. In this architecture, the remaining hyperparameters, such as the dimension of the embedding space and the length and the number of convolutional filters, are set empirically using 10-fold cross validation on the validation-set of the small and large dataset. The resulting values are a 8-dimensional embedding space, 64 convolutional filters of length 8, and 16 neurons in the hidden fully connected layer.

Our experiments were carried out on three different datasets. The first dataset consists of malware from the Android Malware Genome project [28] and has been widely used [10, 11]. This dataset has a total of 2123 applications, of which 863 are benign and 1260 are malware from 49 different malware families. Labels are provided for the malware family of each sample. The benign samples in this dataset were collected from the Google play store and have been checked using virusTotal to ascertain that they were highly probable to be malware free. We refer to this dataset as the 'Small Dataset'.

The second dataset was provided by McAfee Labs (now Intel Security) and comes from the vendor's internal repository of Android malware. After discarding empty files or files that are less than 8 opcodes long, the dataset contains 2475 malware samples and 3627 benign samples. This dataset does not include malware family labels and may include malware and/or benign applications present in the small dataset. Hence to ensuring training hygiene i.e. to ensure we do not train on the testing-set, the network is trained and tested on each dataset separately without cross-contamination. We refer to this dataset as the 'Large Dataset'.

We also have an additional dataset provided by McAfee Labs containing approximately 18,000 android programs, and which was collected more recently than the first two datasets. This was used for testing the final system after setting the hyper-parameters using the smaller datasets. After discarding short files, the dataset contains 9268 benign files and 9902 malware files. We refer to this dataset as the 'V. Large Dataset'.

Each dataset was split into 90% for training and validation and the remaining 10% was held-out for testing. Care was taken to ensure that the ratio of positive to negative samples in the validation and testing sets was the same as in the dataset as a whole.

Results are reported using the mean of the classification accuracy, precision, recall and f-score. The key indicator of performance is f-score, because the number of samples in the malware and benign classes is not equal. In this situation, classification accuracy is too influenced by the number of samples in each class. For example if the majority of samples were of class x, and the classifier simply reported x in all cases, the classification accuracy would be high, although the classifier would not be useful. However, given the same conditions, the f-score, which is based on the precision and recall, would be low.

Our neural network software was developed using the Torch scientific computing environment [4]. During training the network parameters were optimized using RMSProp [3] with a learning rate of 1e-2, for 10 epochs, using a mini-batch size of 16. The network weights were randomly initialized using the default Torch initialization. We used an Nvidia GTX 980 GPU for development of the network, and training the network to perform malware classification takes around 25 minutes on the large dataset (which contains approximately 6000 example programs). Once the network has been trained our implementation can classify approximately 3000 files per-second on the GPU.

4.1 Classification Accuracy

In this experiment, the network's performance is measured in terms of accuracy. The network was trained using the complete training and validation set, then tested on the held-out test-set that was not seen during hyper-parameter tuning. We compare the performance of our proposed system with our own implementation of an n-gram based malware detection method [10]. For both datasets we measured the performance of this system using 1, 2 and 3-gram features. The same training and testing samples were used for both systems in order to allow for direct comparison of their performance. The results for the small and large and v. large datasets are shown in Table 1. We have endeavored to select papers from the literature that use similar Android malware datasets to give as fair a comparison as possible.

In the small dataset our proposed method clearly achieves state-of-the-art performance, and is comparable to methods such as [10] and [23]. It achieves better performance than our baseline n-gram system with 1-gram features and near identical performance to the baseline with 2 and 3-gram features.

The large dataset is more challenging due to the greater variably of malware present. Our system achieves similar performance to the baseline n-gram system, while having far greater computational efficiency (See Section 4.2). Although other methods have achieved better performance on similar tests, they make use of additional outside information such as the application's requested permissions or API calls [25]. In contrast, our proposed method needs only the raw opcodes, which avoids the need for features manually designed by domain experts. Moreover, our proposed method has the advantage over existing methods of being very computational efficient, as it is capable of classifying approximately 3000 files per-second.

The results on the v. large dataset, which was obtained from the same source as the large dataset and hence likely shares similar characteristics, shows that our system's performance improves as more training data is provided. This phenomenon has been observed when training neural networks in other domains, where performance is highly correlated with the number of training samples. We expect that

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
n-grams (Small DS)	opcode n-grams (n=1)	863	1260	0.95	0.95	0.95	0.95
	opcode n-grams (n=2)	863	1260	0.98	0.98	0.98	0.98
	opcode n-grams (n=3)	863	1260	0.98	0.98	0.98	0.98
n-grams (Large DS)	opcode n-grams (n=1)	3627	2475	0.80	0.81	0.80	0.80
	opcode n-grams (n=2)	3627	2475	0.81	0.83	0.82	0.82
	opcode n-grams (n=3)	3627	2475	0.82	0.83	0.82	0.82
DroidDetective [13]	Perms. combination	741	1260	0.96	0.89	0.96	0.92
Yerima [23]	API calls, Perms., intents, cmnds	1000	1000	0.91	0.94	0.91	0.92
Jerome [10]	opcode n-grams	1260	1246	-	-	-	0.98
Yerima [25] *	API calls, Perms., intents, cmnds	2925	3938	0.97	0.98	0.97	0.97
Yerima (2) [24]*	API calls, Perms., intents, cmnds.	2925	3938	0.96	0.96	0.96	0.96

Table 1: Malware classification results for our system on both the small and large datasets compared with results from the literature. Results from the literature marked with a (*) use malware from the McAfee Labs dataset i.e. our large dataset, while all others use malware sampled from the Android Malware Genome project [28] dataset i.e. our small dataset

these results can be further improved given greater quantities of training data, which will also allow more complex network architectures to be explored. Unfortunately comparisons with the baseline n-gram system on the v. large dataset were not possible due to computational cost associated with the n-gram method.

4.2 Computational Efficiency

In this experiment we compare the computational efficiency of our proposed malware classification system with our implementation of a conventional n-gram based malware classification system [10]. Note that when reporting the results we do not include the time take to disassemble the malware files as this is constant for both systems. The results in Table 2 are presented in terms of both the average time to reach a classification decision for a single malware file, and the corresponding average number of programs that can be classified per second.

System	Time per program (s)	Programs per second
Ours	0.000329	3039.8
1-gram	0.000569	1758.3
2-gram	0.010711	93.4
3-gram	0.172749	5.8

Table 2: Comparing the time taken to reach a classification decision and number of programs that can be classified per second, for our proposed neural network system and a conventional n-gram based system.

It can be seen from Table 2 that our system can produce a much higher number malware classification decisions per second than the n-gram based system. The n-gram based system also experiences exponential slow-down as the length of the n-gram features are increased. This severely limits the use of longer n-grams, which are necessary for improved classification accuracy. Our proposed system is not limited in the same way, and in fact, the features extracted by the first layer of the CNN can be thought of as n-grams where $n = 8$. Use of such features with a conventional n-gram based system would be much too computationally expensive. Our proposed neural network system is implemented on a desktop GPU, specifically an Nvidia GTX-980, however it could easily be moved to the GPU of a mobile device, allowing for fast and efficient malware classification of Android applications.

Finally, the memory usage required to execute the trained neural network is constant. Increasing the length or number of convolutional filters, or increasing the number of training examples linearly increases memory usage. Whereas with n-gram based systems, increasing the training-set size dramatically increases the number of unique n-grams and hence memory usage. For instance, with the small dataset there are 213 unique 1-grams, 1891 unique 2-grams, and 286471 unique 3-grams. This means our proposed neural network based system also more efficient in terms of memory usage during training.

4.3 Learning Curves

In this experiment we aim to understand the system's performance as a function of the quantity of training data, with the aim of predicting how its performance is likely to change if more training data were to be made available.

This experiment was performed on the V. Large dataset. As in previous experiments, the dataset is split into training and validation sets. Throughout the experiment the validation-set remains fixed. An artificially reduced size training-set is constructed by randomly sub-sampling from the complete set of training examples. The network is then trained from scratch on this reduced size training-set, and the system's performance measured on both the training and validation sets. This process is repeated for several different sizes of training-set, ranging from a small number of examples up to the complete set of all training-examples. The system's performance on the validation-set and training-set are then plotted as a function of the training-set size. Performance is recorded in terms of $1 -$ f-score, meaning that perfect performance would produce a value of zero.

In figure 3, we can see that when only a small number of training-examples are provided, training-set performance is perfect, while validation-set performance is very poor. This is to be expected as with such a small number of training-examples the system will over-fit to the training-set and the learned parameters will not generalize to the unseen validation-set. However, as more training-examples are provided the validation-set error decreases, showing that the system has learned to generalize from the training-set. We can predict from the learning curves in figure 3 that if more training-examples were to be provided, the validation-set error would continue to decrease.

These results suggest that our system benefits from larger quantities of training-data as expected with neural networks [20]. They also show that the poor performance on the 'Large Dataset', which was obtained from the same source as the 'V. Large dataset' and hence shares similar characteristics, is caused by lack of data. This is indicated by the gap between the validation and testing-set errors when only approximately 6000 training examples are provided.

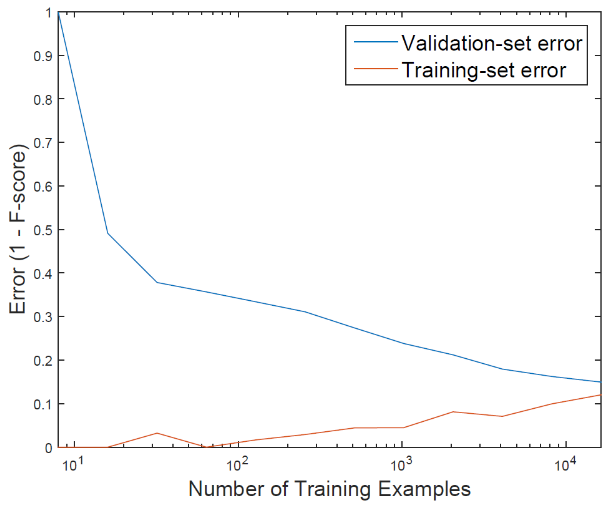

Figure 3: Learning curves for the Validation-set and Training-set as the number of training examples is varied. Note the log-scale on the x-axis.

4.4 Realistic Testing

In order to assess the potential of our proposed classification technique in realistic environments we apply our trained network to a completely new dataset. This allows us to demonstrate the real-world potential of our classification technique when applied to an unknown and realistic dataset at a bigger scale. The network used in this experiment was trained on the V. Large dataset, introduced in Section 4.

Our new dataset consists of 96,412 benign apps and 24,103 malware apps. The benign apps were randomly selected from the Google Play store, and were collected during July and August 2016. To represent a distinct set of malicious apps, we used another dataset containing known malware apps, including those from the Android Malware Genome project [28], but removing the ones overlapping with the training set of the network.

Approximately 1 TB of APKs were used in this experiment. The APKs were converted to opcode sequences using a cloud architecture consisting of 29 machines running in parallel, in a process which took around 11 hours. Classification of the opcode sequences was performed using an Nvidia GTX 1080 GPU, and took an hour to complete.

Note that for this experiment we assume that all APKs in the Google Play dataset are benign, and all the APKs in the malicious dataset are malicious. Of course, this may be a naive assumption, as it is possible for malicious apps to exist on Google Play.

Cross validation testing was performed on our new dataset. In each cross validation fold approximately 24,000 malware applications and 24,000 benign application were used. Therefore, in order to present all applications to the network four-fold cross validation was used. The results of this experiment are reported in Table 3.

Classification System	Acc.	Prec.	Recall	F-score
Ours	0.69	0.67	0.74	0.71

Table 3: Malware classification results of our system tested on an independent dataset of benign and malware Android applications.

We can see from the results in Table 3 that although the f-score is lower than previous experiments, our system has the potential to work in realistic environments. This is because our new testing dataset is much larger than the one used for training the network and contains greater variability of applications. The results of this experiment show that the network has learned features with the ability to generalise to realistic data. In future work we hope to take advantage of our new dataset to explore more complex network architectures that can be learned given more training data.

5. CONCLUSIONS

In this paper we have presented a novel Android malware detection system based on deep neural networks. This innovative application of deep learning to the field of malware analysis has shown good performance and potential in comparison with other state-of-art techniques, and has been validated in four different Android malware datasets. Our system is capable of simultaneously learning to perform feature extraction and malware classification given only the raw opcode sequences of a large number of labeled malware samples. The main advantages of our system are that it removes the need for hand-engineered malware features, it is much more computationally efficient than existing n-gram based malware classification systems, and can be implemented to run on the GPU of mobile devices.

As future work, we would like to extend our methodology to both dynamic and static malware analysis in different platforms. Our proposed method is general enough that it could be applied to other types of malware analysis with only minor changes to the network architecture. For instance, the network could process sequences of instructions produced by dynamic analysis software. Similarly, by changing the disassembly preprocessing step the same network architecture could be applied to malware analysis on different platforms.

Another open problem for malware classification, which may allow networks with more parameters, and hence greater discriminative power, to be used, is data augmentation. Data augmentation is a way to artificially increase the size of the training-set, by slightly modifying existing training-examples.

The transformations used in data augmentation are usually chosen to simulate variations that occur in real world data, but which may not be extensively covered by the available training-set. We would like to investigate the design of data-augmentation schemes appropriate to malware detection.

6. ACKNOWLEDGEMENTS

This work was partially supported by the grants from Global Research Laboratory Project through National Research Foundation (NRF-2014K1A1A2043029) and the Center for Cybersecurity and Digital Forensics at Arizona State University. This work was also partially supported by Engineering and Physical Sciences Research Council (EPSRC) grant EP/N508664/1.

7. REFERENCES

[1] Baksmali. https://github.com/JesusFreke/smali. Accessed: 2015-02-15.

[2] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. Accessed: 2015-02-01.

[3] RMSProp. www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Slide 29.

[4] Torch. http://torch.ch/.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[6] C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[7] G. Canfora, F. Mercaldo, and C. A. Visaggio. Mobile malware detection using op-code frequency histograms. In *Proc.of Int. Conf. on Security and Cryptography (SECRYPT)*, 2015.

[8] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE Int. Conf. on*, pages 3422–3426, 2013.

[9] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *Neural Networks (IJCNN), 2015 Int. Joint Conf. on*, pages 1–8, 2015.

[10] Q. Jerome, K. Allix, R. State, and T. Engel. Using opcode-sequences to detect malicious android applications. In *Communications (ICC), 2014 IEEE Int. Conf. on*, pages 914–919, 2014.

[11] B. Kang, B. Kang, J. Kim, and E. G. Im. Android malware classification method: Dalvik bytecode frequency analysis. In *Proc. of the 2013 Research in Adaptive and Convergent Systems*, pages 349–350, 2013.

[12] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[13] S. Liang and X. Du. Permission-combination-based scheme for android mobile malware detection. In *Communications (ICC), 2014 IEEE Int. Conf. on*, pages 2301–2306, 2014.

[14] X. Liu and J. Liu. A two-layered permission-based android malware detection scheme. In *Mobile Cloud Computing, Services and Engineering (MobileCloud), 2014 2nd IEEE Int. Conf. on*, pages 142–148, 2014.

[15] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE Int. Conf. on*, pages 1916–1920, 2015.

[16] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez. Puma: Permission usage to detect malware in android. In *Int. Joint Conf. CISIS´12-ICEUTE´12-SOCO´12*, pages 289–298, 2013.

[17] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20, Oct 2015.

[18] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. " andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[19] A. Sharma and S. K. Dash. Mining api calls and permissions for android malware detection. In *Cryptology and Network Security*, pages 191–205. 2014.

[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[21] X. Su, M. C. Chuah, and G. Tan. Smartphone dual defense protection framework: Detecting malicious applications in android markets. In *Mobile Ad-hoc and Sensor Networks (MSN), 2012 Eighth Int. Conf. on*, pages 153–160, 2012.

[22] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 7th Asia Joint Conf. on*, pages 62–69, 2012.

[23] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th Int.l Conf. on*, pages 121–128, 2013.

[24] S. Y. Yerima, S. Sezer, and I. Muttik. Android malware detection: An eigenspace analysis approach. In *Science and Information Conference (SAI), 2015*, pages 1236–1242, 2015.

[25] S. Y. Yerima, S. Sezer, and I. Muttik. High accuracy android malware detection using ensemble learning. *Information Security, IET*, 9(6):313–320, 2015.

[26] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems*, pages 649–657, 2015.

[27] Y. Zhang and B. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

[28] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symp. on*, pages 95–109, 2012.