Classification of Android Apps and Malware Using Deep Neural Networks

Robin Nix
Division of Computer Science
Louisiana State University
Baton Rouge, Louisiana 70803
Email: rnix3@lsu.edu

Jian Zhang
Division of Computer Science
Louisiana State University
Baton Rouge, Louisiana 70803
Email: zhang@csc.lsu.edu

***Abstract*—Malware targeting mobile devices is a pervasive problem in modern life. The detection of malware is essentially a software classification problem based on information gathered from program analysis. We focus on classification of Android applications using system API-call sequences and investigate the effectiveness of Deep Neural Networks (DNNs) for such purpose. The ability of DNNs to learn complex and flexible features may lead to timely and effective detection of malware. We design a Convolutional Neural Network (CNN) for sequence classification and conduct a set of experiments on malware detection and categorization of software into functionality groups to test and compare our CNN with classifications by recurrent neural network (LSTM) and other n-gram based methods. Both CNN and LSTM significantly outperformed n-gram based methods. Surprisingly, the performance of our CNN is also much better than that of the LSTM, which is considered a natural choice for sequential data.**

I. Introduction

Mobile devices are an essential part of most individuals' everyday lives and a vast majority of mobile devices today run Android, with sales of Android devices averaging 80% of total market share over the past three years [1]. The continuous expansion of the Android operating system has led to a corresponding uptick in malware targeted at Android devices. Since the first Android malware application was detected in 2009, there has been an alarming increase in the number of malware targeting Android devices [2]. Given the large number of Android applications (often referred to as apps) it is simply not possible to apply human effort to reviewing even a fraction of the available apps. (As of June 2016 there are over 2.2 million distinct apps in the GooglePlay store alone [1] and the number is rising fast.)

Identifying malware is essentially a software classification problem. The process of analyzing and classifying software involves two main steps. First, through program analysis, various descriptions of a program are generated and features are extracted from the program descriptions. Then classifiers can be constructed to perform classification based on these features. Due to the sheer number of apps that need to be processed, automated software analysis and classification is necessary. It is desired that little or no human effort be required in either of the steps.

In this paper, we focus on program analysis that examines Android system-API calls made by an app. System-API calls describe how an app interacts with the underlying Android OS. Such interactions are essential for an app to carry out its tasks, hence providing fundamental information on an app's behaviors. We designed and implemented a pseudo-dynamic program analyzer. Each run of the analyzer tracks a possible program execution path and generates a sequence of API calls along the path. After multiple runs of the analyzer, each app is turned into a collection of these sequences.

Traditional methods for sequence classification often transform a sequence into features such as symbol counts or n-grams [3]. For example, we may represent each of our API-call sequences by a bag-of-words (BoW) representation, i.e., using a frequency vector where each entry is the count of the occurrence of the corresponding API call in the sequence. Besides frequencies for individual API calls, one may count a combination of 2-calls (2-grams) or 3-calls (3-grams), etc. However, order information (i.e., the order of the symbols in the sequence) is lost in the bag-of-words representation. The n-gram representation, although maintaining local order information, lacks flexibility. When n gets slightly large, most n-grams will show up only once, in only one of the sequences. This greatly impairs the generalization capability of the classifiers using this type of feature.

We investigate the effectiveness of Deep Neural Networks (DNNs) for API-call based software classification. The DNN's ability to automatically learn features is important for software classification, in particular, for dealing with malware. An approach that requires handcrafted features is slow in nature and therefore cannot provide a rapid response to malware epidemics. DNNs can also produce complex and flexible features that help generalization in the classification [4] [5].

We designed a Convolutional Neural Network (CNN) for sequence classification, which conducts convolution operation along the sequence, learning sequential patterns for each location as the convolution window slides down the sequence. Our CNN architecture also employs multiple CNN layers to construct higher level features from small local features. In addition to the CNN, we also considered classification using

978-1-5090-6182-2/17/$31.00 ©2017 IEEE

Recurrent Neural Networks (RNN), specifically RNNs with Long Short Term Memory (LSTM) [6] since LSTMs are a natural architecture for dealing with sequences. We conducted a set of experiments to test the two types of DNNs and compare them to other methods based on bag-of-words (BoW) and n-gram. The experiments include malware detection, identification of malware families, and categorization of benign software into groups based on functionality. In all the experiments, DNNs significantly outperformed the BoW and the n-gram based approaches. Despite the fact that LSTMs are a natural choice for sequential data, our CNN performed significantly better and had the best performance of all the methods tested.

II. Related Work

Over the past several years a large amount of work has been done with Convolutional Neural Networks (CNNs) in the field of image recognition. ImageNet 2010 marked the first time that a deep learning algorithm, named AlexNet, won the iconic image classification challenge. Since then the competition has been dominated by deep neural networks as opposed to the previous techniques, which focused more on using an ensemble of hand-generated features with other machine learning techniques [7]. Some of the other important recent results from the ImageNet competition include GoogleNet in 2014 [8] and the use of Resnets in 2015 [9]. Fundamentally, all of these networks have at their core a convolutional element and it has been well demonstrated that the spatial locality emphasis inherent to CNNs is well suited to the task of image classification.

CNNs have also been applied to a lesser degree to tasks in other fields such as text document classification and sentiment analysis. Santos and Gatti demonstrated a method to do sentence sentiment classification using convolutional networks applied to word and character level embeddings [10]. Kim used a similar approach to sentence classification but created feature vectors for each word from pre-trained data which was then fed into a convolutional neural network for classification [11]. One limitation of both of these results is that they focus exclusively on very short input strings. Lai et al. combined recurrent and convolutional networks to achieve good classification accuracy on longer document length texts [12]. Johnson and Zhang also showed success of CNNs in sentiment classification of documents [13].

Most malware detection tools use a combination of hand crafted features with a machine learning algorithm such as SVM to make a final classification. Androsimilar and DenDroid are two Android malware analysis tools that create signatures for code chunks and then use those signatures as the input to SVM and other clustering algorithms to make a classification [14] [15]. There are several tools focused on classification of Android malware through analysis of Call Flow Graphs [16] [17]. Compared to these approaches, the main advantage of our method is that the DNNs learn features from data directly and hence eliminate the need for human-crafted features. Furthermore, the DNN features are more flexible and generalizable than those derived from simpler data-mining approaches. The drawback of using DNN is that computation is more demanding.

Unfortunately, research has shown that malware obfuscation techniques can effectively thwart many of these types of detection techniques [18]. Both DroidChameleon and a tool presented by Maiorca et al. automatically modify call flow graph structures, and perform method and class renaming to change the features available for classification [19] [20].

There has been very little published research on the application of deep learning techniques to malware detection. The little that has been published has tended to focus on using relatively small densely connected networks to do classification of features extracted from the APK. For example DroidDetector uses a two-layer fully connected network with only 150 neurons per layer to do malware classification of APK files using 198 binary features defined by the authors [21]. Another work that applied a deep network to malware detection used tri-grams extracted from Windows system calls along with feature reduction to produce a set of 4000 features representing API tri-gram which were then fed into a two-layer fully connected network for classification [3]. The use of hand-generated features as the input to a deep neural network in both of these approaches limits the potential effectiveness of a deep learning algorithm.

III. API-Call Analysis of Android APPs

A. Andriod Apps

All of the necessary files for an Android application are bundled into an APK file archive. Each APK file typically contains the following items: res folder, assets folder, classes.dex file, AndroidManifest.xml file, lib folder, and META-INF folder. The most relevant components for our work are the classes.dex and Androidmanifest.xml files. Native libraries can also be included inside of APK files and are typically stored in the lib folder. The APK file also contains all other resources needed to run the application such as pictures, sound files, etc. in the res and assets folder.

The classes.dex file contains the bytecode for the application and is the main source of executable code, although there are methods for loading external code and code from native library packages in the APK. In this paper, we focus only on code included within the classes.dex file and will not examine code inside native libraries or the contents of the asset files. To date the majority of Android malware utilizes the dex code, although recently there have has been an increase in malware launched from native code libraries [16]. We use the open source tool Androguard to obtain a smali code representation of the classes.dex for our analysis. [22].

B. Psuedo-Dynamic Code Analysis

For the purpose of software classification, program analysts seek to obtain, from the executable file, a good description of the program's functionality. A set of features is then extracted from the description and used to classify the program typically using some form of machine learning

Fig. 1. Sample code for a single method of an Android application

algorithm. The challenge is to select features with the necessary specificity to allow for accurate classification while maintaining sufficient generality.

Program analysis can be broadly divided into two categories: static analysis and dynamic analysis. Static analysis examines code itself without running it while dynamic analysis derives features from execution of the program. For example, as the simplest form of static analysis, one may view the binary executable as a string describing the program. String features, such as counts of symbols, n-grams, and various substrings can then be extracted as features. A classifier can be constructed based on these features to classify the program. Signature-based descriptors and features like these can be easily obfuscated, e.g., by masking (xor) the executable with a key and unmasking at program execution time. In pursuit of a better classification, we aim for descriptions and features that are more fundamental to a program and thus more difficult to obfuscate. To accomplish this goal, we designed an analyzer that performs what we term pseudo-dynamic program flow analysis.

In standard dynamic analysis, the analyst executes the software program, typically in an isolated emulation environment such as a virtual machine, and gathers information during the execution. During execution the program can be provided interactions either manually or through the use of automated tools. A set of program activities such as API calls, memory accesses, or network communications are logged and used as a description of the program.

We focus on logging and tracking Android system-API calls as the features to provide to our classifier. Rather than executing the code decompiled from the .dex file in an emulator/sandbox, our analyzer tracks the code without execution. During parsing, the analyzer follows program instructions in a manner that mirrors a possible execution of a program but does not compute any state information for the program (except the return points of the internal function calls, elaborated below).

In order to thwart dynamic analysis, researchers have demonstrated how malware can employ techniques such as sandbox detection by analyzing things such as sensor testing and delay timers, sometimes lasting weeks, before starting the execution of the malicious code [23] [24]. By generating the program flow analysis through static analysis with our random branching approach, we can obtain more complete code coverage and avoid the possibility of dynamic malware obfuscation techniques.

Suppose while following the code, the analyzer encounters an instruction I. If I is a regular instruction that does not affect the program flow, the analyzer simply steps over it with no state change except the update of the program counter for the length of the instruction. If I is an invocation of an Android system-API call, the analyzer logs the name of the API call to an external file for analysis.

To properly represent the program flow, there are two scenarios the analyzer needs to pay special attention to: 1) I is a branch instruction; 2) I is a call to an (internal) function in the .dex file code (not a call to a system-API function or function in another library). For the first scenario, the analyzer randomly selects a branch direction and follows that branch. In the second scenario, the analyzer steps into the called function and tracks the code there until it reaches the function return. (Since the analyzer may step into functions recursively, it needs to keep a record of the return points of the function calls, allowing it to return to the correct location in the calling function after it finishes tracking code inside the called function.) The analyzer finishes tracking when it reaches the return statement of initial function.

The path the analyzer walks down is a possible execution path of the code. Because our analyzer logs system-API calls along the way while tracking the code, a sequence of API calls is generated for each analyzer run. Fig. 1 shows the code for a single decompiled method from an APK in our dataset. Even in this trivial example, the code is highly nonlinear in the sense that one may go down many different paths when executing the code.

Since we use random selections to determine branch direction, any single execution may skip large sections of the code in the APK file. To increase code coverage, i.e., tracking different possible execution paths of the code, given an individual APK file, we repeatedly run the analyzer on the APK, each run with a different random choice at the branching points. This generates a collection of API-call sequences for each APK file. We use such a collection as a description of the APK and build deep neural network models for classification based on the collection of sequences.

IV. DNN for App Classification

A. Preprocessing and Input to Convolutional Neural Network

Assume the analyzer generates a collection of h API-call sequences for each APK file. We denote the collection for an APK by $\{S^{(1)}, S^{(2)}, \dots\ S^{(h)}\}$ where $S^{(i)} = a_1, a_2, \dots$ and a_i is a particular API call. Because of the random branch selections, the length of the sequences ($S^{(i)}$s), even for those generated from the same APK, can be different. The sequences generated from the APKs in our dataset have lengths varying from under 100 to well over 10,000 API calls, with a majority around thousands.

A review of the API call sequences shows that certain API calls are often called many times in direct succession. For

example, the Java string builder API is often called five or more times in a row. This repetitive calling does not provide meaningful new information and the noise can actually significantly impair classification. Therefore, we reduce the repetitive calls to a single API call. Our experiments showed a significant increase in accuracy from the reduction.

We first investigate CNNs for classification of software. CNNs require fixed sized input. One can pad the shorter sequences so that all sequences have the same length (the length of the longest sequence). However, given that our longest sequence has a length well over 10k and that the correlation between API calls declines greatly over long distances, we take a different approach. (More discussion on the locality of program code is given in the result section.) We select a length n and divide the sequence $S^{(i)}$ into a set of segments $\{S_1^{(i)}, S_2^{(i)} \ldots\}$, each of length n. (Note that the length of $S^{(i)}$ may not be a multiple of n. In this case, the last segment may not have length n. We repeatedly tile the content of this segment until the segment length reaches n.) The optimal length of n was experimentally determined to be 100-200 API calls.

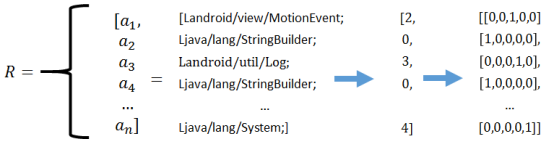

Fig. 2. Mapping API calls to One-hot Vectors

We encode each API call using a one-hot vector. Let A be the set of all API calls we considered and $m = |A|$ be the size of A. A one-hot vector v for the i-th element of A is a vector of length m with $v_i = 1$ and all other elements being zero. Fig. 2 shows a sample conversion of a sequence of API calls to one-hot vectors by first converting the API calls to their (integer) indices in A and then to the corresponding one-hot vectors. In the example shown, it is assumed that there are five API calls (i.e., $m = 5$) in the represented language.

With the one-hot representation, each segment $S_j^{(i)}$ is encoded by a matrix of size $n \times m$, which serves as the input to our CNN.

B. Convolutional Neural Network Architecture and Classifier Construction

Given a set of segments $S_j^{(i)}$ representing an APK file, our task is to construct a classifier to label the APK. We first train a CNN that conducts (soft) classification for each segment. We then average the (soft) classification for each segment derived from the same APK to select the label for that APK.

Our CNN is designed to take a segment and output a soft classification giving the probability that the segment belongs to each class. Formally, let c be the number of classes in our classification task, then the CNN computes a function of the form $R^{n \times m} \mapsto R^c$. Note that in the training set, we only have class labels for the entire APK files and do not have any label for individual segments. To solve this problem, we pass the labels of APKs to the segments. That is, if l is the class

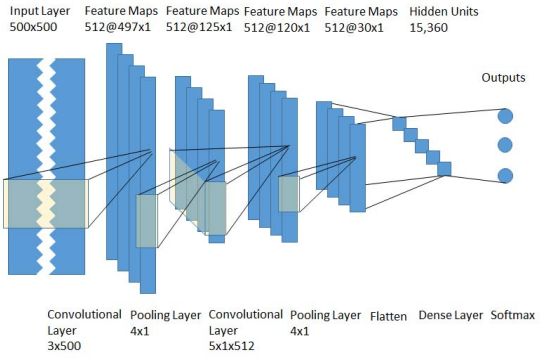

Fig. 3. Architecture of our Convolutional Neural Network

label of an APK and $\{S_j^{(i)}\}$ is the set of segments derived from that APK, then all the segments $S_j^{(i)}$ have the same class label l.

The proposed CNN has the following topology. First there is the input layer, ($n \times m$ matrix), followed by a convolutional layer which collapses the one-hot vector input to a single dimensional collection of feature maps. Next a pooling layer is applied, completing the first level of the CNN. This first level is followed by another convolutional and pooling layer. The final pooling layer is fully connected to a softmax output layer with a node for each class. A diagram of the CNN topology is shown in Fig. 3. The parameters of the network (i.e., number of filters and their size, feature maps size, pooling window size, etc.) are also given in Fig. 3. During training, we also applied dropout layers after each of the convolutional layers and in the dense layer to avoid overfitting and co-dependence between different filters within the convolutional layer. We experimented with many different network architectures (different number and combination of layers and different filter and pooling parameters) and found that the CNN in Fig. 3 gave the best performance.

Let X be the input to a convolutional layer. For the first layer, the input is the one-hot vector representation of the API call segment, i.e., $X \in R^{n \times m}$. The convolution operation is conducted along the dimension of the sequence. Given a window size d, we denote by $X(t)$ the local window at location t, i.e., $X(t) \in R^{d \times m}$ is a part of the matrix X starting from row t and spanning d rows. Let $q(t)$ be the corresponding unit on the feature map. We have

$$q(t) = relu(W \odot X(t) + b)$$

where $relu$ is the rectified linear unit function. $W \in R^{d \times m}$ is the filter parameter and $\odot$ is an operator that takes two matrices U and V and outputs $\sum_{i,j} U_{i,j} \cdot V_{i,j}$. Convolving X with a filter $W \in R^{d \times m}$ without padding the border gives a feature map $q \in R^{n-d+1}$. If r filters are used, the collection of feature maps form a matrix of size $(n - d + 1) \times r$. Next feature maps go through a max-pooling layer. The output of the pooling serves as the input to the next convolutional layer.

Let $CNN(X) \in R^c$ be the output of the last (softmax) layer of our CNN when a segment X is given to the first (input) layer. For each APK file, we have a collection of

segments $\{S_j^{(i)}\}$ and their labels $\{y_j^{(i)}\}$ (the labels of the APK from which the segments are derived). Assume the label $y_j^{(i)}$ is an indicator vector with dimension c (i.e., it has value 1 at index k if the segment belongs to class k and zero at all other locations). We train the CNN using the sum of categorical cross entropy as the loss, i.e., minimizing

$$L = -\sum_{i,j}\sum_{k} y_{jk}^{(i)} \log(CNN(S_j^{(i)})_k)$$

where $y_{jk}^{(i)}$ is the k-th element of the vector $y_j^{(i)}$ and $CNN(X)_k$ is the k-th element of the output vector.

The minimization was performed using adaptive gradient descending (ADAGrad), which reduced over-fitting of the dataset in the final training epochs by dynamically adjusting the learning rate [25]. An additional benefit of ADAGrad is the parameter wise updates of the learning rate, since many of the core API calls are sparsely represented in the data. The CNN and training were implemented using the commonly available Lasagne and Theano packages for Python to allow for easy replication of the results by other researchers [26] [27].

To classify an APK represented by a set of segments $\{S_j^{(i)}\}$, we first apply the CNN to each segment and obtain the output $CNN(S_j^{(i)}) \in R^c$ from the softmax layer, which can be viewed as a soft classification of the segment. We combine the soft classification by summation and determine the label p for the APK by the position in the resultant vector with the largest sum of the soft classifications, i.e.,

$$p = \arg\max_k \sum_{i,j} CNN(S_j^{(i)})_k.$$

C. LSTM Classification

Our software classification is based on sequences of API calls. Long short term memory (LSTM) [6] is a natural neural network architecture designed for dealing with sequences. Therefore, we also investigate LSTM-based classification for our dataset. At each step of the sequence, the input to the LSTM is the one-hot vector representing the API call at that step. We employed a two-layered LSTM network with 256 hidden LSTM units per layer. After the LSTM layers process through the sequence, the output is fed into a dense layer connected to a layer of softmax neurons for (soft) classification, giving a probability for each class. An APK file gives a collection of API-call sequences and the LSTM network outputs soft classification for each sequence. The label for the APK file is again determined by the sum of the soft classifications.

V. Experiment Results

A. Experiment Setting

1) The Datasets: Two sets of android APKs were used for testing the CNN: a benign set and a malware set. The composition for each dataset is described below.

Malware samples were obtained from the Contagio Mobile repository [28]. In particular, for this evaluation, we focus on eight families of malware from the Contagio dataset consisting of a total of 216 individual APK files. All of these samples are recent with the oldest malware having been made available on the site in May of 2016 and should accurately represent currently employed malware techniques. Table I shows a breakdown of the different malware families.

TABLE I
MALWARE FAMILIES AND NUMBER OF SAMPLES USED IN EXPERIMENTS

Malware Family	# of Samples	Malware Family	# of Samples
Godless	8	OverlayLocker	55
Crisis Android	70	AndroidXbot	9
SberbBanker	5	Marcher	6
Hummingbad[1]	42	Xiny	7

We also gathered a collection of benign (non-malware) applications grouped into multiple categories from a variety of third party app stores. Application categories were chosen to match common categories used by the app stores.

In total, 1016 unique APK files were used, spanning eleven categories as shown in Table II. To ensure that no duplicate APK files were included in the dataset, an MD5 hash was run over all APKs and any files that had matching signatures were removed from the dataset.

TABLE II
THE CATEGORIES AND NUMBER OF APPLICATIONS PER CATEGORY FOR THE BENIGN APPLICATIONS USED IN EXPERIMENTS

Category	# of apps	Category	# of apps
System Tools	114	Shopping	98
Communications	110	Food & Drink	85
Books & Reference	103	Puzzle Games	112
Health & Fitness	55	Arcade Games	94
Photos	88	Board Games	79
Themes & Wallpaper	92		

2) Models Compared: We compared our method to other commonly used sequence classification algorithms. N-grams is one of the most popular approaches for sequence classification. For the N-gram based classification, each sequence of API calls was converted into a feature vector listing the number of occurrences for each n-gram. We then applied Term frequency-inverse document frequency (TF-IDF) to normalize the feature vectors. In our experiments, SVM was used for n-gram based classification. We tested different values for n along with different kernels and hyper-parameters and found that 3-grams with RBF kernel gave the best performance. These settings are used in the presented results.

We also tested a Naive Bayes classifier in our experiments. (Input to the Naive Bayes classifier was not n-grams but counts of each type of API calls, similar to the Bag-of-Words representation for documents if we view API calls as words.) We used the SVM and Naive Bayes implementation from the Scikit-Learn Python package [29]. We used 5-fold cross-validation to measure the performance of these methods, as well as for our CNN and LSTM models. The results reported below are all 5-fold CV values.

B. Malware Classification Results

We conducted two classification experiments with the malware dataset. The first involved malware detection where

we need to differentiate between malicious and benign APKs. The second experiment was malware family identification, in which our task is to classify each malware variant into its respective family.

Most real-world app collections (i.e. application marketplaces) consist of a majority of benign applications with only a small percentage of malware samples included. To reflect such a scenario, we constructed two datasets from our collection of malware and benign apps, one with balanced numbers of malware and benign apps and the other with significantly more benign apps. Testing using the second dataset gives a better indication of the possible false positives/negatives a classifier may experience in the real world. We conducted experiments on the two datasets and the results for accuracy, precision, and recall were practically identical.

Table III shows the accuracy, precision and recall rates for the malware detection. The CNN performed well across all measures, in particular giving 100% precision. Also, the CNN performed significantly better than the comparison classifiers in all measures.

TABLE III
PERFORMANCE OF THE MODELS IN MALWARE DETECTION

Model	Accuracy	Precision	Recall
CNN	**99.4%**	**100%**	**98.3%**
LSTM	89.3%	93.8%	55.6%
n-gram SVM	66%	53.3%	34.8%
Nave Bayes	82.0%	47%	54%

Actual Malware Family \ Predicted Malware Family	Crisis	Sberb	Overlay	Godless	Hbad	Xbot	Marcher	Xiny
Crisis Team	69	0	0	1	0	0	0	0
SberbBanker	0	7	0	0	0	0	0	0
OverlayLocker	0	0	55	0	0	0	0	0
Godless	0	0	0	6	2	0	0	0
Hummingbad	0	0	0	0	42	0	0	0
AndoridXbot	0	0	0	0	0	9	0	0
Marcher	0	0	0	0	0	0	5	0
Xiny	0	0	0	1	0	0	0	5

Fig. 4. Confusion matrix for malware family classification using CNN

In the malware family identification experiment, our task is to classify each malware variant into its respective family. Family identification is important to the computer security community because knowing which family a piece of malware belongs can help researchers more quickly analyze its code or assess what damage a malicious app may have caused by referring to previous analyses of its familial counterparts. We tested our CNN for malware family identification using our malware dataset. Results (confusion matrix) from 5-fold cross validation are shown in Fig. 4.

The CNN also delivered excellent results for the family identification task. It is not uncommon for a classifier to have higher accuracy in the malware detection task than in the family identification task. As explained in more detail by Tangil et al. [30], this can be the result of the fact that certain malware families (as classified by malware analysts) may actually be based on the same or similar exploit or basic code fragments blurring the distinction between classes.

C. Benign APK Categorization Results

The preceding results show our CNN model did very well in both the detection and the family identification tasks commonly used as benchmarks in malware analysis. To further demonstrate the capability of the CNN model, we tested it, together with the LSTM and the n-gram methods, on the task of functionality categorization, i.e., categorizing the benign android apps into functionality groups such as System Tools, Puzzle Games, etc. This task can be potentially more challenging than malware detection and family identification, due to the fact that there are a huge variety of apps in the same functionality group (e.g., consider System Tools apps, which include disk cleaners, settings managers, and Wifi apps). Additionally, several of the categories are thematically very similar to one another, such as different categories of games, making the task even more challenging. The diversity in a functionality group can be much larger than that among malware of the same family. With such a high level of diversity, a classifier must generalize of the underlying behavior of the app to ensure accurate classification.

In this experiment, only the benign dataset was used and the functionality group (class label) of an app was determined by the category in which the application was listed in the app store. The results from 5-fold cross validation are reported below. Table IV shows the accuracy of different methods and Fig. 5 shows the confusion matrix for CNN. (Rather than listing number of apps in each entry, we listed the fraction of the apps, with respect to each category/functionality group.)

TABLE IV
ACCURACY IN FUNCTIONALITY GROUP CLASSIFICATION

Model	Accuracy
CNN	**58.2%**
LSTM	27.8%
n-gram SVM	18.8%
Nave Bayes	13.6%

Predicted Category \ Actual Category	Sys Tools	Comms	Books	Health	Photos	Themes	Shopping	Food	Puzzles	Arcade	Board
System Tools	**0.58**	0.13	0.14	0.03	0.02	0.01	0.04	0.04	0	0	0.02
Communications	0.09	**0.67**	0.06	0	0.04	0.05	0.05	0.03	0.02	0	0
Books	0.22	0.06	**0.53**	0.03	0.04	0.01	0.04	0.05	0.02	0	0.01
Health & Fitness	0.12	0.01	0.11	**0.53**	0.09	0	0.04	0.02	0.06	0.01	0
Photos	0.08	0.11	0.08	0.02	**0.54**	0	0.07	0.07	0.01	0.01	0
Themes & Wallpaper	0.04	0.15	0.06	0	0	**0.58**	0.02	0.09	0.02	0.04	0
Shopping	0.05	0.05	0.03	0	0.03	0.01	**0.66**	0.14	0.01	0	0.02
Food & Drink	0.01	0.05	0.01	0	0.04	0.02	0.21	**0.64**	0.01	0	0.01
Puzzle Games	0.05	0.04	0.05	0.01	0.03	0.02	0.03	0.01	**0.63**	0.06	0.08
Arcade Games	0	0.04	0.01	0.02	0.04	0.03	0.03	0.02	0.13	**0.58**	0.09
Board Games	0.04	0.04	0.04	0.01	0.07	0	0.07	0.08	0.18	0.07	**0.41**

Fig. 5. Confusion matrix for benign app categorization using CNN, with heat map for misclassifications (note: the diagonal shows the correct classification)

The CNN model performed much better than that of both the n-gram SVM and Naive Bayes methods. This shows a high capability of the CNN model to generalize the behavior of the application for diverse data. Additionally, if we look at the mispredictions made by the classifier, they are often to thematically similar classes. For example, the great majority

of the incorrect classification for game apps are mistakes for other game categories. Also, Shopping and Food & Drink both have a large amount of sale-based or transactional applications, so it makes sense to see a high amount of crossover in those categories as well. These results serve as further proof that the CNN is obtaining a good understanding of the underlying application behavior.

D. Effect of Training Size

We examined how the size of the training set affects the classification accuracy. We conducted experiments using the task of benign software categorization. In these experiments, the size of the training set was varied but the size and contents of the validation set were held constant.

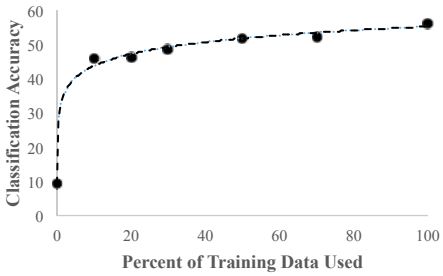

Fig. 6. The accuracy of the CNN classifier v.s. the percentage of the data used during the training (There were 1000 APKs in the complete dataset)

As expected, the accuracy increases when there are more training examples. However, we also observe that the classifier still achieved a respectable level of accuracy even when only 10% of the training data were used to train the model. We hypothesize that creating multiple variations of each APK through our random branching has a similar effect to techniques used for increasing dataset sizes by creating new images through the addition of noise [31]. However, in our case, we have the added benefit that the new sample variations are not created through noise but through exploring different possible execution paths.

E. Effect of Sequence Length

Recall that each of our API-call sequences was divided into segments of a fixed length. In this test we examine the relationship between the length of the segments and the accuracy in categorizing benign software. The results from these experiments were also used to derive the optimal segment length for the other classification tasks.

Fig. 7 shows that the optimal segment length is approximately 200-500 API calls. Intuitively, this makes sense when we consider locality among the API calls. Outside of a certain range, API calls become fairly unrelated. Segments of approximately 200-500 APIs likely represent the maximum size of a logic block in most programs in the dataset and a segment size of 500 generated the results shown in this paper. In the malware detection scenario, locating a small segment that is indicative allows an analyst to focus on particular sections of code and be more efficient in analyzing malware. Even at segments of 100 API calls the accuracy is only

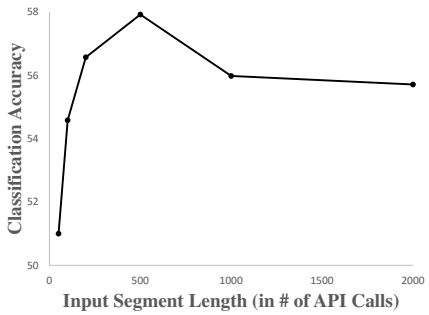

Fig. 7. Accuracy of the CNN classifier with different segment length

3% (relatively) below the peak. This means that small API windows could be used to help narrow down the sections of code in an APK that might be malicious, out of the entire APK.

F. Discussion

Our experimental results from three different classification tasks (malware detection, family identification, functional categorization) all demonstrated that the CNN model is clearly capable of Android APK classification. Furthermore, it outperforms the n-gram based and Naive Bayes models used for comparison by significant margins. One of the major benefits of using a CNN for malware detection as described in this paper is the elimination of any manually generated features and the flexibility of the automatically generated features which are capable of extending to new and novel examples.

To understand why a CNN is well-suited to this task, it is worth considering the fundamental nature of CNNs and an analogy to the field of image processing where CNNs have been widely studied. In analyzing program code, feature locality of API calls is very important. API calls that appear close to each other are much more tightly related than API calls that exist hundreds of lines apart. CNNs naturally account for this by using small local filters to develop features in each layer of the network, which are consecutively passed upward to create more abstract features in the network. This is similar to how human experts approach malware analysis, by looking at small sections of code to determine their function and progressively considering the interactions between different code.

In the image processing realm, CNNs have shown great resiliency to spatial transformations of images such as deformation, noise, and rotation [31]. While code is not deformed in the same way as objects in images, the insertion of extra API calls between segments of code containing key features is very similar to elongation of an object in an image. The ability to tolerate noise and deformations makes it harder for typical code obfuscation techniques to defeat analysis without changing the underlying effect of the code and potentially eliminating its malicious payload.

For comparison, the very insightful and effective work done by Zhou et al. in RiskRanker produced excellent results but relied heavily on manually generated features and

behavioral definitions to aid in detection [32]. As malware evolves, these features need to constantly be updated to keep up with the new techniques of malware writers. The classifier proposed here will automatically update and generate new features after training with new examples reducing the burden on human analysts.

Between our malware and the benign datasets, there are less varieties among malware of the same type than that among benign apps of the same category. The diversity (variety) affects the performance and this applies to all classification methods. For complex apps, the diversity can be large and thus make it more difficult to classify. Still, our CNN performed well (and is much better than the other methods) on the benign app dataset, which includes quite number of complex apps.

VI. Conclusion

In this paper, we have shown the effectiveness of Deep Neural Networks, specifically Convolution Neural Networks, in classification of software. This may lead to effective and efficient malware detection and classification. Our work is one of the first attempts to construct DNNs for Android application/malware classification. Our experimental results showed that DNNs performed much better than the traditional methods based on n-grams. Furthermore, in the software classification scenario, our CNN also significantly outperformed LSTM, which is often the first choice for sequential data.

While our work demonstrated the effectiveness of DNNs for classification of Android apps, there is considerable room for improvement. In particular, we have focused on the API-call sequences from the main code of an application. There are many other information one can extract from an app, e.g., call graphs and code in the attached library files. We believe that by fusing the information from different aspects/components of an app, a better software classifier can be constructed. One of our future directions is to design deep neural networks that take multiple input channels from other sources in the APK file and fuses the information for better classification.

References

[1] Statista, https://www.statista.com/, accessed:2016-24-19.

[2] T. Wyatt, "Lookout security blog," https://blog.lookout.com/blog/2010/08/10/security-alert-first-android-sms-trojan-found-in-the-wild/, accessed: 2016-10-24.

[3] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 3422–3426.

[4] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[5] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[10] C. N. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts." in *COLING*, 2014, pp. 69–78.

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

[12] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification." in *AAAI*, 2015, pp. 2267–2273.

[13] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," *CoRR*, vol. abs/1412.1058, 2014. [Online]. Available: http://arxiv.org/abs/1412.1058

[14] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur, and V. Ganmoor, "Androsimilar: Robust signature for detecting variants of android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66–80, 2015.

[15] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[16] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Semantic-based detection of android native code malware," *arXiv preprint arXiv:1602.04693*, 2016.

[17] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 2014, p. 3.

[18] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[19] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[20] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 329–334.

[21] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[22] A. Desnos *et al.*, "Androguard-reverse engineering, malware and goodware analysis of android applications," *URL code. google. com/p/androguard*, 2013.

[23] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 447–458.

[24] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*. ACM, 2014, p. 5.

[25] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[26] The Theano Development Team, R. Al-Rfou, G. Alain *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[27] Lasagne Development Team, S. Dieleman, J. Schlüter, C. Raffel *et al.*, "Lasagne: First release," *Zenodo: Geneva, Switzerland*, 2015.

[28] M. Parkour, "Contagio mobile." http://contagiominidump.blogspot.com/, accessed: 2016-10-19.

[29] F. Scikit-learn Developlment TeamPedregosa, G. Varoquaux, A. Gramfort *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[30] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "Droidscribe: Classifying android malware based on runtime behavior," *Mobile Security Technologies (MoST 2016)*, vol. 7148, pp. 1–12, 2016.

[31] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis." in *ICDAR*, vol. 3, 2003, pp. 958–962.

[32] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.