A Deep Learning Approach to Android Malware Feature Learning and Detection

Xin Su*, Dafang Zhang†, Wenjia Li‡, Kai Zhao†
*Hunan Provincial Key Laboratory of Network Investigational Technology, Hunan Police Academy, Changsha, China
Email: suxin@hnu.edu.cn
†College of Computer Science and Electronics Engineering, Hunan University, Changsha, China
Email: {dfzhang, kzhao}@hnu.edu.cn
‡Department of Computer Sciences, New York Institute of Technology, New York, NY, USA
Email: wli20@nyit.edu

***Abstract*—The growing amount and diversity of Android malware has significantly weakened the effectiveness of the conventional defense mechanisms, and thus Android platform often remains unprotected from new and unknown malware. To address these limitations, we propose *DroidDeep*, a malware detection approach for the Android platform based on the deep learning model. Deep learning emerges as a new area of machine learning research that has attracted increasing attention in artificial intelligence. To implement this, we first extract five types of features from the static analysis of Android apps. Then, we build the deep learning model to learn features from Android apps. Finally, the learned features are used to detect unknown Android malware. In an experiment with 3,986 benign apps and 3,986 malware, *DroidDeep* outperforms several existing malware detection approaches and achieves a 99.4% detection accuracy. Moreover, *DroidDeep* can achieve a remarkable runtime efficiency which makes it very easy to adapt to a lager scale of real-world Android malware detection.**

***Keywords*-Android malware, Deep learning, Security, Static analysis**

I. Introduction

Smartphone applications (apps) have become the predominant means of accessing personalized computing services such as email, banking, etc. However, this rapid deployment and extensive availability of mobile apps has made them attractive targets for various malware. Malware authors generally take advantage of the update mechanism of mobile apps to infect existing mobile apps with malicious code and thus compromise the security of the smartphone. Recent statistical data show that malware based on Android platform accounts for 97% of mobile malware [1]. The private data of the users, such as IMEI, contacts list, and other user specific data are the primary target for the attackers, which is a serious threat for the security and privacy of mobile users. Consequently, there is an urgent need to identify and cope with the malware for the Android platform.

A large number of research works have thus been studied for analyzing and detecting Android malware based on static or dynamic analysis [1], [2]. These approaches discover static or dynamic behavior patterns to distinguish Android malware, which are effective solutions for known Android malware detection. However, this kind of approaches will become less effective when detecting unknown Android malware. Therefore, some researchers used machine learning techniques to detect unknown Android malware [3], [4], [5], [6]. These works extract features (e.g., permission, API, etc.) from known Android benign apps and malware, then use machine learning algorithms (e.g., decision tree, etc.) to learn these features in order to detect unknown Android malware. For example, DroidAPIMiner [4], APKAuditor [5], SherlockDroid [6] focus on extracting single level of features from Android malware for classification purposes. But single level of features cannot reflect the overall characteristics of an Android malware. Therefore, Drebin [3] is proposed to classify Android malware based on several types of features. However, the significant increase to number of features would cause extra overhead of the classification process, such as time, computational resources, etc.

In this paper, we present *DroidDeep*, a deep learning approach for Android malware detection which considers multiple levels of features to address the limitations of the aforementioned research works. *DroidDeep* first considers the static information including permissions, API calls, and deployment of components for characterizing the behavioral pattern of Android apps, and extracts a multi-level feature set from Android apps which contains over 30,000 features. Then, we feed these extracted features into a deep learning model to learn typical features for classification. Finally, we put the learned features into a detector based on *Support Vector Machine* algorithm (SVM) for detecting Android malware.

To implement our approach, there are a main challenges that need to be solved. High dimensions of features would cause some features overlap between benign app and malware, which makes it hard to detect malware in a large amount of real world Android apps. Moreover, the large number of features would lead to high computational overhead. Therefore, we should learn the most typical and important features from the 30K+ features which are extracted directly from Android apps. To address this issue, we use Deep Belief Network (DBN) [7], a fast, greedy learning algorithm which is able to learn typical

[1]Mobile Malware. http://www.forbes.com/sites/gordonkelly-/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/.

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.69

features and reduce the number of extracted features to save computational resources.

In summary, we make the following contributions to the detection of Android malware in this paper:

- We design *DroidDeep* which combines both static analysis and deep learning that is capable of detecting Android malware with a high accuracy and very few false alarm, and it does not rely on manually crafted detection patterns.
- We extract more than 30,000 multi-level features from Android apps, which could cover almost all of the static behaviors of Android apps. Moreover, we use the DBN algorithm to learn the typical features which can effectively reduce the number of features to be used in classification.
- We conduct extensive experiments on real-world Android benign apps and malware, and compare with previous well-known Android malware detection approaches. The results show that our approach outperforms the existing machine learning approaches, and achieve high accuracy and low false alarm rate.

The rest of this paper is organized as follows. In Section II, we introduce the design of *DroidDeep*. The experiments we conducted on large-scale app sets are described in Section III. Finally, we present the literature review in Section IV and draw our conclusions in Section V.

II. Design of DroidDeep

Traditional machine learning models that have less than three layers of computational units are considered to have shallow-structured architectures. These architectures typically contain at most two layers of nonlinear feature transformations. For instance, *SVM* uses a shallow linear pattern separation model with one or zero feature transformation layer when kernel trick is used or otherwise. Shallow architectures have been shown to be effective in solving many simple or well-constrained problems, but their limited modeling and representational power can cause difficulties when dealing with more complicated real-world applications, such as Android malware detection.

Moreover, traditional feature selection algorithms, such as *Chi-Square* [2], select features based on calculating and sorting features. These algorithms would lead to Distribution Bias and Long Tail Effect, which cannot select typical features for classification. Zhao et al [8] proposed a feature selection algorithm called *FrequenSel* to select features from Android malware, which selects features by finding the difference between malware and benign apps in terms of the feature frequencies.

In this section, we design *DroidDeep*, a novel Android malware detection approach based on deep learning model. The architecture of *DroidDeep* is shown in Figure 1, which consists of four components. *Feature Extractor* is responsible for extracting features from Android apps transforming the extracted features into a multi-dimensional vector in an automatic manner which has been described in Section II-A. The third component is the deep learning model which takes the multi-dimensional vector as input and learn unique features for Android malware detection (which is shown in Section II-B). The fourth component is the classification function based on the learning features (which is described in Section II-C).

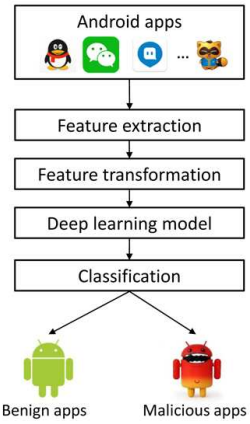


Fig. 1. Architecture of DroidDeep

A. Feature Extraction

To detect Android malware on smartphones, our approach requires a comprehensive yet unique representation of Android apps that helps determine the typical indications of malicious activity. To achieve this goal, our method employs a wide variety of static analysis that extracts feature sets from different sources, such as *Manifest.xml*, API calls, etc. Our goal is to systematically characterize the Andorid apps into five types: requested permission, used permission, sensitive API calls, action, app components(e.g., Activity, Service).

In terms of static analysis, all we need is the .apk file of an Android app. After decompressing the apk file with the *apktool*[3], we mainly focus on parsing two files named *AndroidManifest.xml* and *classes.dex* respectively. The above five kinds of features can be extracted from the two files. For example, by parsing the *AndroidManifest.xml* file with the tool *AXMLPrinter2* and the parser *TinyXml*, we can know what permissions an app requires, e.g. permission *android.permission.call_phone* stands for permitting an app to make a phone call. Moreover, by parsing *classes.dex* file with the disassembler *baksmali*, we can know what kind of sensitive API will be called, e.g., function *sendTextMessage()* stands for a sensitive API and might be used for sending text messages.

We totally extract 32,247 features from each Android app based on the five given types of features. We will explain each type of feature as follows.

- **Requested permission:** One of the most important security mechanisms introduced in Android is the permission system. Permissions are actively granted by the user at installation time, which allow an application to access security-related resources. Previous works [9] show that

[2]Chi-Square. https://en.wikipedia.org/wiki/chi-squared_distribution/.

[3]Apktool. http://ibotpeaches.github.io/Apktool/

Android malware tends to request certain permissions more often than benign apps. For example, a high percentage of current malware sends premium SMS messages and thus requests for the SEND_SMS permission. We thus gather all permissions listed in the *manifest.xml* in a feature set.

- **Used permission:** Whenever an API call is invoked during the execution of an application, the Android platform will verify if the API call is permission-protected before proceeding to execute the call; such permissions are referred to as used permissions.
- **Sensitive API call:** This kind of feature consists of two parts, namely restricted API call and suspicious API call. First, the Android permission system restricts access to a series of critical API calls. Our method searches for the occurrence of these calls in the disassembled code in order to gain a deeper understanding of the functionality of an application. Second, certain API calls allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. As these calls can particularly lead to malicious behavior.
- **Action:** Android apps can register actions they are concerned about in *AndroidManifest.xml* or code, and this mechanism allows Android malware monitors some system events. For example, we found that *android.intent-.action.SIG_STR* and *android.intent.action.BATTERY_-CHANGED_ACTION* are more frequently used in Android malware than benign app. The two actions represent when battery of smartphone runs out sharply, Android malware would suspend their process to avoid detection.
- **App Component:** There exist four different types of components in an Android app, each of which defines a different interfaces to the system: *activities*, *services*, *content providers* and *broadcast receivers*. Every application can declare several components of each type in the manifest. The names of these components are also collected in a feature set, as the names may help to identify well-known components of malware. For example, several variants of the so-called DroidKungFu family share the same name of particular services.

After features extraction, we transform feature set into vector $V = \{0, 1, 0, 0, ...\}$, in which 1 indicates that the feature is contained in this app, whereas 0 indicates not. However, V is often a sparse vector, in order to reduce the storage overhead, we transform V to a compressed format V^*. Assuming that the features are arranged in a fixed order, then we can index a feature by its position, and V^* is defined as $V^* = \{2, 5, 8, ...\}$. The positions of non-zero elements in V are stored in V^*, which saves a great amount of memory space.

B. Deep Learning Model

A deep learning model can be constructed with different architectures [10], e.g., Deep Belief Networks (DBN) and convolutional neural networks. However, deep neural networks (e.g., neural networks that have many hidden layers) are difficult or impossible to train using gradient descent algorithm [7]. The DBN circumvents this problem by performing a greedy layer-wise unsupervised pre-training. It has been shown that this unsupervised pre-training can build a representation from which it is possible to perform a successful supervised learning by "fine-tuning" the resulting weights using gradient descent learning. In other words, the unsupervised stage sets the weights of the network to be closer to a better solution than random initialization, thus avoiding local minima when using supervised gradient descent learning. For this study, we chose DBN architecture to construct our deep learning model and then classify Android apps.

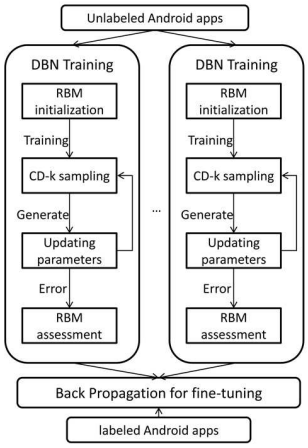


Fig. 2. Deep learning model constructed with DBN

The Deep Belief Network (DBN) is a neural network constructed from many layers of Restricted Boltzmann Machines (RBMs) [7]. As shown in Figure 2, the construction of a deep learning model has two phases, an unsupervised pre-training phase and a supervised back-propagation phase. In the pre-training phase, the DBN is hierarchically built by stacking a number of Restricted Boltzmann Machines (RBM), with the deep neural network regarded as a latent variable model, which is beneficial for gradually evolving high-level representations.

RBM initialization. A RBM is an Markov random field (MRF) [4] associated with a bipartite undirected graph as shown in Figure 3, which is structured as two layers of neurons: a visible layer and a hidden layer. The visible layer consists of m visible units $V = \{V_1, ..., V_m\}$ to represent observable variables (e.g., features from Android apps), and the hidden layer consists of n hidden units $H = \{H_1, ..., H_n\}$ to capture dependencies between observed variables. Each unit is fully connected to the units of the other layer, but there is no connection between units of the same layer.

Based on section II-B, the extracted features have already been converted into a binary vector. Therefore, the RBMs take binary vectors as input, and the random variables (V, H) take values $(v, h) \in \{0, 1\}^{m+n}$ and the joint probability distribution under the model is given by the Gibbs distribution [11] $p(v, h) = \frac{1}{Z} e^{-E(v,h)}$ with the energy function, where

[4]https://en.wikipedia.org/wiki/Markov_random_field

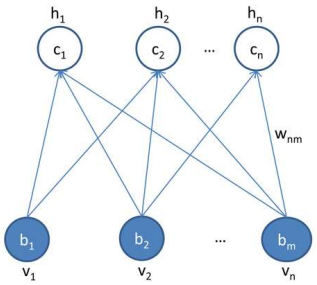


Fig. 3. The undirected graph of an RBM with n hidden and m visible variables

$Z = \sum_{v,h} e^{-\varepsilon(v,h)}$:

$$E(v,h) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij}h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \quad (1)$$

For all $i \in \{1,...,n\}$ and $j \in \{1,...,m\}$, w_{ij} is a real valued weight associated with the edge between units V_j and H_i and b_j and c_i are real valued bias terms associated with the $j-$th visible and the $i-$th hidden variable, respectively.

Differentiating the log-likelihood $\ell(\theta|v_l)$ of the model parameters θ given one training example v_l with respect to θ yields:

$$\begin{aligned}\frac{\partial}{\partial\theta}\ell(\theta|v_l) &= -\sum_h p(h|v_l)\frac{\partial\varepsilon(v_l,h)}{\partial\theta} \\ &+ \sum_v p(v)\sum_h p(h|v)\frac{\partial\varepsilon(v,h)}{\partial\theta}\end{aligned} \quad (2)$$

Computing the first term on the right side of the equation is straightforward because it factorizes. The computation of the second term is intractable for regular sized RBMs because its complexity is exponential to the size of the smallest layer. Therefore, the expectation over p(v) can be approximated by alternating Gibbs sampling [12]. But since the sampling chain needs to be long to get almost unbiased samples of the distribution modeled by the RBM, the computational effort is still too large.

CD-k sampling. The goal of RBM training is to make marginal probability distribution $p(v)$ fit probability distribution of training samples based on justifying the parameters of model. To achieve this, we use k-steps contrastive divergence learning algorithm (CD-k) [13] to train RBMs which is a standard way to train RBMs. The idea of CD-k is quite simple: the chain is run for only k steps, starting from an example $v^{(0)}$ of the training set and yielding the sample $v^{(k)}$. Each step t consists of sampling $h^{(t)}$ from $p(h|v^{(t)})$ and sampling $v^{(t+1)}$ from $p(v|h^{(t)})$ subsequently.

The gradient in equation 2 with respect to θ of the log-likelihood for one training pattern $v^{(0)}$ is then approximated by equation 3,

$$\begin{aligned}CD_k(\theta, v^{(0)}) &= -\sum_h p(h|v^{(0)})\frac{\partial\varepsilon(v^{(0)},h)}{\partial\theta} \\ &+ \sum_h p(h|v^{(k)})\frac{\partial\varepsilon(v^{(k)},h)}{\partial\theta}\end{aligned} \quad (3)$$

In the following, we restrict our considerations to RBMs with binary units for which $E_{p(h_i|v)}[h_i] = sigmoid(c_i + \sum_{j=1}^{m} w_{ij}v_j)$ with $sigmoid(x) = (1 + exp(-x))^{-1}$.

The expectation $E_{p(v^{(k)}|v^{(0)})}[CD_k(\theta, v^{(0)})]$ is denoted by $CD_k^*(\theta, v^{(0)})$. Further, we denote the average of $CD_k(\theta, v^{(0)})$ over a training set by $C\bar{D}_k(\theta)$ and its expectation by $C\bar{D}_k^*(\theta)$. The expectations are considered for theoretical reasons. They lead to deterministic updates, but are computable only for small models.

Updating parameters. This step includes updating parameters from both visible and hidden layer. In the visible layer, the visible units are binary, the way to update the parameters of visible layers when generating a reconstruction is to stochastically pick a 1 or 0 with a probability determined by the total top-down input:

$$p(v_i = 1) = \sigma(c_i + \sum_j h_j w_{ij}) \quad (4)$$

However, it is common to use the probability, p_i, instead of sampling a binary value. This is not nearly as problematic as using probabilities for the data-driven parameters of hidden layers and it reduces sampling noise thus allowing faster learning.

In the hidden layer, the hidden units are binary and we use CD-k sampling in the proposed deep learning model. The hidden units should have stochastic binary states when they are being driven by a data-vector. The probability of turning on a hidden unit, j, is computed by applying the logistic function $\sigma(x) = \frac{1}{1+exp(-x)}$ to its total input:

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij}) \quad (5)$$

and the hidden unit turns on if this probability is greater than a random number uniformly distributed between 0 and 1.

RBM assessment. After training RBMs, we need to assess the trained RBMs. Obviously, the simplest metric to assess RBMs is the likelihood of RBMs in a certain training samples. However, computing the likelihood relevant to a normalizing constant, and the value of such constant is hard to obtain directly. Therefore, we use approximation method for RBM assessment.

Annealed Importance Sampling algorithm (AIS) [14] is a popular approach for RBM assessment. The idea of this algorithm is quiet simple, which approximates likelihood of data from RBMs based on Monte Carlo approach [15]. The idea of Monte Carlo approach is described as follows. Assume that we want to calculate an normalizing constant Z_A of a certain distribution $P_A(x)$. To achieve this, we can introduce another easier sampling distribution $P_B(x)$ with the same state space and an normalizing constant Z_B which is known in advance. Then, we can calculate Z_A based on calculating value of $\frac{Z_A}{Z_B}$. We set $Z_A = \sum_x f(x)$, $Z_B = \sum_x g(x)$, and $\frac{Z_A}{Z_B}$ can be calculated as equation 6.

$$\frac{Z_A}{Z_B} = \frac{\sum_x f(x)}{\sum_x g(x)} = \sum_x \frac{g(x)}{\sum_x g(x)}\frac{f(x)}{g(x)} = \langle\frac{f(x)}{g(x)}\rangle_{P_B} \quad (6)$$

the above equation shows that $\frac{Z_A}{Z_B}$ equals mean value of distribution P_B which is derived by function $\frac{f(x)}{g(x)}$.

TABLE I
CLASSIFICATION RESULTS WITH DIFFERENT DEEP LEARNING MODEL CONSTRUCTIONS. (B REPRESENTS BENIGN APPS, M REPRESENTS MALWARE)

# of layers(# of neurons)	Precision(B)	Recall(B)	F-measure(B)	Precision(M)	Recall(M)	F-measure(M)	Overall acuracy
3([5000,5000,500])	0.966	0.977	0.971	0.976	0.966	0.971	0.971
3([5000,5000,1000])	0.967	0.984	0.976	0.984	0.967	0.975	0.975
3([5000,6000,500])	0.969	0.976	0.973	0.976	0.969	0972	0.973
3([6000,6000,1000])	0.967	0.983	0.975	0.983	0.967	0.975	0.975
4([5000,5000,5000,1000])	0.966	0.948	0.957	0.949	0.967	0.958	0.957
5([5000,5000,5000,5000,1000])	0.964	0.982	0.973	0.982	0.963	0.972	0.972

In RBMs assessment, we first introduce a simple RBM and calculate its normalizing constant Z_{simple}. Then, we use AIS to approximate normalizing constant ratio of two RBMs. After that, we can obtain normalizing constant of assessed RBM Z_{rbm} by result of calculating the ratio multiply Z_{simple}. Finally, we can approximate the training data likelihood of RBM and finish RBMs assessment.

Back-propagation. In the back-propagation phase, the pre-trained DBN is fine-tuned with labeled samples in a supervised manner. In this phase ,we use back-propagation (BP) algorithm to classify feature vectors extracted by RBMs, and fine-tuned the DBN. This process compares the output from pre-trained DBN with the labeled Android apps, and obtain the error. Then, we back propagate the error from output to input to modify parameters of DBN. The deep learning model uses the same app set in both phases of the training process. In this way, the deep learning model is completely built.

C. Classification

After the deep learning stage, we obtain the abstract features of Android apps, and then use them to build the classification model to classify the Android apps into benign apps or malware. For that, we have applied the *Support Vector Machine (SVM)* algorithm. SVM consists of two phases: training and testing. Given positive and negative samples in the training phase, an SVM finds a hyperplane which is specified by the normal vector ω and perpendicular distance b to the origin that separates the two classes with the largest margin γ. Figure 4 shows a schematic depiction of an SVM.

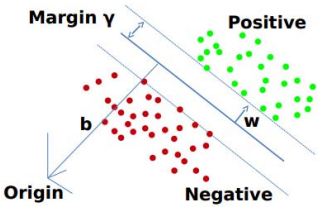


Fig. 4. an illustration of the SVM method. ω is the normal vector and b is the perpendicular distance to the origin

During the testing phase, the samples are classified by the SVM prediction model and assigned either a positive or negative label. The decision function f of the linear SVM is given by

$$f(x) = \langle \omega, x \rangle + b \tag{7}$$

where x is a feature vector representing the sample. It is classified as positive if $f(x) > 0$ and negative otherwise. In the training phase, $\langle \omega, x \rangle$ are computed as the SVM prediction model from the training data. In the testing phase, the samples are classified using equation 7 with ω and b from the prediction model.

III. EVALUATION

After presenting DroidDeep in details, we now proceed to an empirical evaluation regarding its effectiveness and efficiency. In particular, we conduct the following three experiments:

- **Detection performance.** First, we evaluate the detection performance of DroidDeep on a dataset of 3,986 malware samples and 3,986 benign applications (Section III-B).
- **Results comparison.** In the second experiment, we compare the detection results of the *DroidDeep* approach with two well-known Android malware detection approaches, namely DroidAPIMiner [4], Fest [8] (Section III-C).
- **Run-time performance.** Finally, we evaluate the run-time performance of DroidDeep. For this experiment we conduct different run-time measurements using GPU environment (Section III-D).

A. Data sets

For all experiments, we consider two datasets of real Android apps and one dataset of real malware. The first real Android app set was crawled from the Google Play Store, which contains the most popular apps from each category (e.g., game, social) in the Google Play Store. To implement this, we designed a tool for downloading Android apps from Google Play based on a java library named androidmarketapi[5], and this library allows access to the official Android market servers. The second real Android app set was downloaded from several popular third-party market in China, such as Anzhi. We consider the apps from this dataset as unknown apps. The real malware dataset is collected from Drebin [3], Android Malware Genome Project [16], and the Contagio Community [6].

To determine malware and benign apps, we send each sample from the Google Play and malware dataset to the VirusTotal service [7] and inspect the output of ten common anti-virus scanners (AntiVir, AVG, BitDe-fender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos). We flag an app as malware if it is detected by at least two of the scanners as malicious. This procedure ensures that our app data is (almost)

[5]Android-market-api. Available from: https://code.google.com/p/android-market-api/.

[6]Contagio. http://contagiominidump.blogspot.com/.

[7]VirusTotal. https://www.virustotal.com/.

correctly divide into benign and malware samples even if one of the ten scanners falsely labels a benign app as malware. Moreover, this procedure was conducted in an off-line manner, and each app needs several seconds to scan by VirusTotal.

Finally, we remove samples labeled as adware from our dataset, as this type of software is in a twilight zone between malware and benign apps. The final dataset contains 3,986 benign applications and 3,986 malware samples.

B. Detection Performance

In our first experiment, we evaluate the detection performance of DroidDeep. For this experiment, we randomly select 3,500 benign apps and 3,500 malware from our Android app datasets to evaluate the classification results of different deep learning model constructions. We also vary several parameters (e.g., number of layers, etc.) when building the deep learning model. The detailed results are shown in Table I.

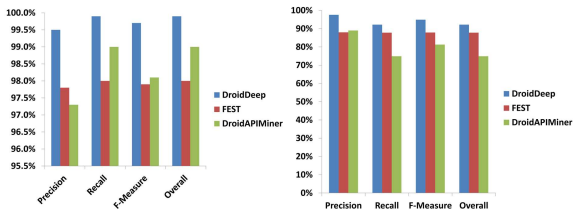
Table I demonstrates that the classification results vary according to the different number of layers and also different number of neurons at each layer. Traditional machine learning models that have less than three layers of computation units are considered to have shallow architectures. Therefore, we set the number of layers for the deep learning model from three to five, and compare the classification results to find which one can achieve the best results for training the DBN model. We can see that DBN can achieve 97.5% accuracy when setting number of layer to three and number of neurons to $[5000, 5000, 1000]$, $[6000, 6000, 1000]$ respectively. We also can see that the average accuracy under different model constructions is higher than 96%.

In real-world situations, the ratio between malware and benign apps may not be 1:1, so we conducted experiments with various ratios of malware to benign apps, including 1:1, 1:2, 1:5, 1:10, 1:100. In this experiment, we select malware and benign apps from the two dataset randomly. We also set the number of layer to three and the number of neurons to $[5000, 6000, 500]$ from layer one to three, respectively. The detail classification results of mixes of malware and benign apps are shown in Table II.

From Table II, we can see that more training data leads to a better accuracy when the ratio of malware to benign apps is 1:1. Particularly, *DroidDeep* can achieve a high 97.5% detection accuracy when the training data size of both classes reach 3500. Moreover, we can see that although the precision, recall and F-measure of malware class fluctuates to some extent, the overall accuracy rises to more than 99% with a higher ratio of benign apps. We believe that this fluctuation of malware class is reasonable since the ratio of malware to benign apps is so lopsided, and there is generally too few malware for deep learning. However, we believe that the classification accuracy of malware class could be further improved by training with more malware samples even under this lopsided ratio.

C. Results Comparison

In this experiment, we compare the detection results of DroidDeep against several related static approaches for detection of Android malware. In particularly, we consider the



(a) Detection results of benign apps (b) Detection results of malware

Fig. 5. Compare detection results among DroidDeep, FEST and DroidAPIMiner

works of *FEST* [8] and *DroidAPIMiner* [4]. The *FEST* is a static feature (e.g., API, permission) extraction and selection approach, and it uses the SVM model to classify Android malware. The *DroidAPIMiner* uses the KNN model to mine API-level features for Android malware detection. First, we use the same training dataset from Section III-A to train classification model of *FEST* and *DroidAPIMiner* in this experiment. Then, to compare detection results on unknown Android apps of the three approaches, we use a testing dataset which contains 1,515 Android apps downloaded from third-party markets and treat them as unknown apps. The comparison results are shown in Figure 5.

The final classification results of each approach are evaluated by *VirusTotal* which was described in Section III-A. Among the four experiment metrics, we can see that *DroidDeep* performances better than other two approaches in classifying benign apps and malware, respectively. In particularly, *DroidDeep* can achieve 92.2% accuracy when detecting malware, and both of *FEST* and *DroidAPIMiner* can only achieve 87.8% and 74.9%, respectively. In terms of the classification accuracy of classify a mixed dataset of benign apps and malware, *DroidDeep* achieves 99.4% which is higher than *FEST* (97.9%) and *DroidAPIMiner* (96.6%). These results clearly show that *DroidDeep* is able to detect malware with better results than the existing approaches that are based on traditional machine learning algorithms.

D. Run-time performance

To analyze the run-time performance of *DroidDeep*, we implement a deep learning model using *Theano* [8]. *Theano* is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently, which can build and run *DroidDeep* on Graphic Processing Units (GPU) easily. This implementation can reduce processing time of model building in order to analyze a large scale of Android apps. In this experiment, we implement *DroidDeep* on a PC, which is equipped with a Geforce GTX 750Ti graphics card with 2 GB dedicated memory, and 4GB RAM. Figure 6 shows the deep learning model building time for different numbers of Android apps.

In this figure, the processing time includes two parts, namely feature extraction (static analysis), feature learning (deep learning). The processing time of classification is too

[8]Theano. http://deeplearning.net/software/theano/

TABLE II
DEEP-LEARNING-BASED MALWARE DETECTION WITH DIFFERENT RATIOS OF MALWARE TO BENIGN APPS. (B REPRESENTS BENIGN APPS, M REPRESENTS MALWARE)

Ratio	Malware/Benign	Precision(B)	Recall(B)	F-measure(B)	Precision(M)	Recall(M)	F-measure(M)	Overall acuracy
1:1	100/100	0.938	0.900	0.918	0.904	0.940	0.922	0.920
1:1	1000/1000	0.944	0.961	0.952	0.960	0.943	0.952	0.952
1:1	3500/3500	0.967	0.984	0.976	0.984	0.967	0975	0.975
1:2	1750/3500	0.974	0.988	0.981	0.976	0.947	0.961	0.975
1:5	700/3500	0.977	0.993	0.985	0.964	0.883	0.922	0.975
1:10	350/3500	0.986	0.995	0.990	0.963	0.869	0.909	0.983
1:100	35/3500	0.996	0.998	0.998	0.886	0.762	0.805	0.995

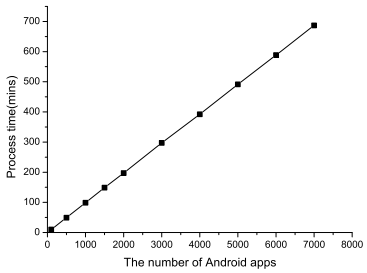


Fig. 6. The process time of build deep learning model with different number of Android apps

small compared to feature extraction and feature learning, and thus we do not take this time into account. We can see that the processing time increases linearly when the number of Android apps increases. *DroidDeep* can achieve analysis performance of almost 6 seconds per Android app, which enables analyzing 14,400 Android apps within a day. Given that we extract more than 30,000 features per Android app, and deep learning model is built offline, we consider that the run-time performance of *DroidDeep* is acceptable and can be extended to analyze lager number of Android apps in real-world for Android malware detection purpose.

IV. RELATED WORK

A. Detection using static and dynamic analysis

The first category of approaches for detecting Android malware was inspired by static program analysis. Several methods have been proposed that statically inspect Android apps and disassemble their code. For example, static approaches include analyzing permission requests for app installation [17], signature-based detection [18].

Enck et al. [19] study popular apps by decompiling them back into their source code and then searching for unsafe coding vulnerabilities. Yang et al. [20] propose AppContext, a static program analysis approach to classify benign and malicious apps. AppContext classifies apps based on the contexts that trigger security-sensitive behaviors.

Our method *DroidDeep* is somewhat related to these approaches, but we extract far more comprehensive features for identifying malware, such as permissions, Action, App Component, and API calls. Moreover, signature-based works can only detect known malware effectively, whereas our work can detect unknown malware based on learning features from known malware.

Dynamic analysis techniques consist of running apps in a sandbox environment or on real devices in order to gather information about the app behavior. DroidScope [21] allows dynamically monitoring apps in a protected environment. Dini et al. [22] propose a framework (MADAM) for Android malware detection which monitors apps at the kernel and user level. MADAM detects system calls at the kernel level and user activity/idleness at the user level to capture the app behavior.

This kind of works can capture dynamic behavior of Android malware, which is complement for static analysis. However, some works (e.g., *TaintDroid*) need to modify Android OS to implement on smartphone, which is technically not feasible. Moreover, dynamic analysis would induce a run-time overhead that may be problematic for directly protecting smartphones.

B. Detection using feature learning

Feature learning is important for Android malware detection, which is able to reduce dimension of features and improve detection accuracy. Zhao et al [8] present a feature extraction and selection tools, which is a feature-based machine learning approach for malware detection for Android platform. This tool selects feature based on feature frequency in Android apps, and uses selected features to detect Android malware. MAST [23] statically extracts 182 features. MAST relies on attribute-based selection, that is, features extracted independently from other features, and on subset-based extraction that takes into account dependencies between features.

C. Detection using machine learning

The difficulty of manually crafting and updating detection patterns for Android malware has motivated the application of machine learning to the malware detection problem. Several methods have been proposed that analyze apps automatically using learning methods (e.g., [4], [8]). As an example, *DroidAPIMiner* [4] analyzes API-level features that are statically extracted from Android apps using several machine learning techniques, such as *C4.5*. *FEST* is a feature extraction and selection tool for Android malware detection [8]. The authors first proposed a feature selection algorithm based on frequency of feature occur in Android malware. Then, SVM algorithm is used to classify Android benign apps and malware based on the selected features. There are also other

recent research efforts which used Support Vector Machine (SVM) algorithm to distinguish malware from benign apps for Android platform [24], [25].

The existing works use traditional machine learning algorithms which have shallow architectures. In contrast, our work uses Deep learning algorithm which can learn high-level representations by associating features obtained from static analysis, which makes it possible to better characterize Android malware for the purpose of detection.

Moreover, to the best of our knowledge, there is one related work which attempted to apply deep learning to malware detection for Android [26]. However, we have used far more features (over 30,000 features) than their approach (192 features). Besides, our training and testing datasets are both much larger and more comprehensive than their approach.

V. Conclusion

In this paper, we introduce *DroidDeep*, a deep learning method for detection of Android malware. *DroidDeep* combines concepts from static analysis and deep learning, which makes it easier to keep up with malware evolution. In this work, we first extracted a total of 32,247 features from five main static feature types of Android apps. Then, we design *DroidDeep* which use a DBN-based deep learning model to learning features from Android benign app and malware, and classify Android benign apps and malware using SVM algorithm. Our evaluation results depict the potential of this approach, where *DroidDeep* outperforms other related approaches and it can identify malware with very high accuracy. Moreover, we also evaluate the run-time performance of *DroidDeep*, which shows that *DroidDeep* only takes a small amount of time to process each Android app on off-line mode, and thus it is easy to extend to a larger scale of Android apps.

VI. Acknowledgement

This work is supported by the Open Research Fund of Hunan Provincial Key Laboratory of Network Investigational Technology, Grant NO.2016WLZC008, the National Basic Research Program of China (973) under Grant 2013CB315805, the National Science Foundation of China under Grant 61472130, 61471169, the Academy and PSB Cooperation Program of Hunan Police Academy 2015, Grant No.2015YJHZ06, the Open Innovation Platform of the Education Department of Hunan Province under Grant 15K037.

References

[1] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[2] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX*, 2010.

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

[4] Y. Aafer, W.L. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *SecureComm*. 2013.

[5] K.A. Talha, D.I. Alper, and C. Aydin. Apk auditor: Permission-based android malware detection system. *Digital Investigation*, 13:1–14, 2015.

[6] L. Apvrille and A. Apvrille. Identifying unknown android malware with feature extractions and classification techniques. In *Trustcom*, 2015.

[7] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19*, pages 153–160, 2007.

[8] K. Zhao, D.F. Zhang, X. Su, and W.J. Li. Fest: A feature extraction and selection tool for android malware detection. In *ISCC*, 2015.

[9] B.P. Sarma, N.H. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: A perspective combining risks and benefits. SACMAT, 2012.

[10] Y. Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[11] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov 1984.

[12] D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. Connectionist models and their implications: Readings from cognitive science. chapter A Learning Algorithm for Boltzmann Machines, pages 285–307. 1988.

[13] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, August 2002.

[14] R.M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2):125–139, April 2001.

[15] Jun S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer Publishing Company, Incorporated, 2008.

[16] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.

[17] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. ICSE, 2014.

[18] F. Yu, A. Saswat, D. Isil, and A. Alex. Apposcopy: Semantics-based detection of android malware through static analysis. FSE, 2014.

[19] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.

[20] W. Yang, X.S. Xiao, B. Andow, S.H. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. ICSE, 2015.

[21] L.K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. Security, 2012.

[22] D. Gianluca, M. Fabio, S. Andrea, and S. Daniele. Madam: A multi-level anomaly detector for android malware. MMM-ACNS, 2012.

[23] C. Saurabh, R. Bradley, T. Patrick, and William Enck. Mast: Triage for market-scale mobile malware analysis. WiSec, 2013.

[24] W.J. Li, J.G. Ge, and G.Q. Dai. Detecting malware for android platform: An svm based approach. In *IEEE CSCloud*. IEEE, 2015.

[25] G.Q. Dai, J.G. Ge, M.H. Cai, D.Q. Xu, and W.J. Li. Svm-based malware detection for android applications. In *Wisec*. ACM, 2015.

[26] Z.L. Yuan, Y.Q. Lu, Z.G Wang, and Y.B Xue. Droid-sec: Deep learning in android malware detection. *SIGCOMM Comput. Commun. Rev.*, 44(4):371–372, August 2014.