Computers, Materials & Continua
DOI:10.32604/cmc.2022.021271
Article

Tech Science Press

Using Capsule Networks for Android Malware Detection Through Orientation-Based Features

Sohail Khan[1,*], Mohammad Nauman[2], Suleiman Ali Alsaif[1], Toqeer Ali Syed[3] and Hassan Ahmad Eleraky[1]

[1]Deanship of Preparatory Year and Supporting Studies, Computer Science Department, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia
[2]National University of Computer and Emerging Sciences, Pakistan
[3]Department of Computer Science, Islamic University of Medina, Medina, Saudi Arabia
[*]Corresponding Author: Sohail Khan. Email: sokhan@iau.edu.sa
Received: 28 June 2021; Accepted: 30 July 2021

Abstract: Mobile phones are an essential part of modern life. The two popular mobile phone platforms, Android and iPhone Operating System (iOS), have an immense impact on the lives of millions of people. Among these two, Android currently boasts more than 84% market share. Thus, any personal data put on it are at great risk if not properly protected. On the other hand, more than a million pieces of malware have been reported on Android in just 2021 till date. Detecting and mitigating all this malware is extremely difficult for any set of human experts. Due to this reason, machine learning–and specifically deep learning–has been utilized in the recent past to resolve this issue. However, deep learning models have primarily been designed for image analysis. While this line of research has shown promising results, it has been difficult to really understand what the features extracted by deep learning models are in the domain of malware. Moreover, due to the translation invariance property of popular models based on Convolutional Neural Network (CNN), the true potential of deep learning for malware analysis is yet to be realized. To resolve this issue, we envision the use of Capsule Networks (CapsNets), a state-of-the-art model in deep learning. We argue that since CapsNets are orientation-based in terms of images, they can potentially be used to capture spatial relationships between different features at different locations within a sequence of opcodes. We design a deep learning-based architecture that efficiently and effectively handles very large scale malware datasets to detect Android malware without resorting to very deep networks. This leads to much faster detection as well as increased accuracy. We achieve state-of-the-art F1 score of 0.987 with an FPR of just 0.002 for three very large, real-world malware datasets. Our code is made available as open source and can be used to further enhance our work with minimal effort.

Keywords: Malware; security; Android; deep learning; capsule networks

1 Introduction

In little more than a decade, Android has become one of the most dominant software stacks for touch screen devices like smartphones and tablets. It has retained the top position in market share for the past few years. According to a report by IDC [1] in the first quarter of 2021, more than 83% of all smartphones sold to end users were powered by Android operating system. Statcounter Global Stats in their recent release of market share statistics [2] has also stated that Android maintained the number one global smartphone operating system spot by further extending its lead over Apple's iOS, with a 72.72 percent market share to 26.46 percent share for iOS.

One of the most significant differences of Android software stack with its competitors is the unprecedented business model realized by open source development and distribution. Android software stack is kept completely open source that includes an operating system, middleware, and some built-in applications (cf. Fig. 1). The open source nature of Android has attracted a large number of developers and contributors, working towards improvement of the architecture. Unlike the previous open source mobile phone approaches, Android has targeted the consumer market and thus become the first open source mobile phone software stack that can be bought and sold off-the-shelf.

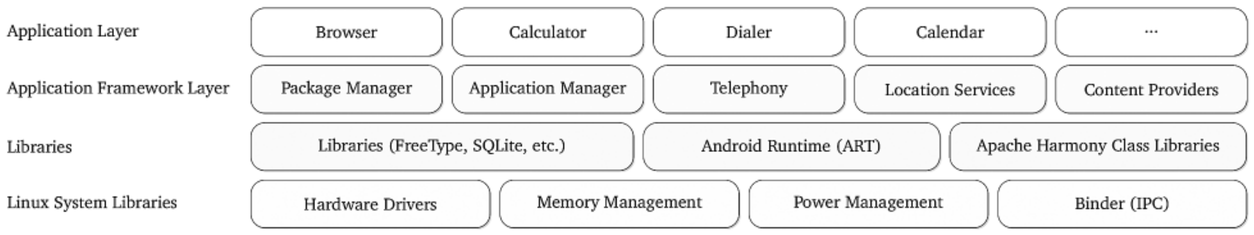

Figure 1: Android software stack

Being open source implies that it is an undeniable choice for all emerging smartphone manufacturers at the time when smartphone markets were taking off. Some prime examples include Samsung, Huawei and the fastest-growing brands such as OnePlus and Xiaomi. From the developers' perspective, it is much easier to build applications for Android as compared to others. On the flip side, this same open nature of Android also attracts a huge number of malware applications that aim to steal users' personal information. Smart devices in general and Android-based devices in particular, are an appealing target of malware writers due to the fact that they are not only popular, open source in nature and have a unique application market [3], but also because of the presence of enormous amount of users' private and sensitive data on these devices [4]. For instance, these devices are equipped with a number of sensors that can easily track users' physical activities and current location. Similarly, embedded cameras and microphones can be used to record videos and take pictures, even without the users' consent and awareness [5,6].

According to McAfee [7], until 2009 no more than 1000 mobile malware were known but since then, malware growth and diversity has been explosive, with the expansion of new technology in the smart device market. F-Secure, in one of their reports [8], show an interesting correlation between the rise in mobile malware detection and the distribution of Android-based smartphones in the market. Accordingly, between 2004 and 2007, only about 400 mobile malware infections were reported, while this number increased by 3350% in the last quarter of 2011 for Android alone [9]. Recently, for Android, Kaspersky Lab detected around 1,245,894 malicious installation packages, of which 38,951 are Trojans targeting banking applications and 3,805 are mobile ransomware Trojans [10]. They also concluded that most of these infected applications are distributed through the official app stores, which makes it really difficult for common users to identify them. Kaspersky Lab data also indicates that mobile malware writers make good use of the current trends to distribute their malicious code. For instance, an application called Gustuff including a malware–Trojan-Banker.AndroidOS.Gustuff.a–appeared on Google Play in 2019 and was one of the most autonomous and massive spreading distributions targeting more than 125 banking and cryptocurrency applications. This makes these threats even more frightening from users' perspective. According to the same report [10], in recent times mobile banking Trojans and Wireless Application Protocol (WAP) Trojans are on the rise in Android devices. The Mobile banking Trojans are not only targeting applications provided by financial institutions but rather getting control over applications providing online hotel and taxi bookings. WAP Trojans, on the other hand, are stealing money from the users by visiting WAP subscription websites without even the consent and knowledge of the users [11].

We note that malware detection has always been a race against the clock, with security experts trying to detect the newly emerging 0-days and malicious software writers cooking up new ways of evading the detection algorithms. However, in the recent past, the use of deep learning has given an advantage to the detection side over malicious software. Due to the fact that learning can be performed from samples automatically, machine learning in general and deep learning models in particular have had great success in mitigating the malware issue on Android [12–14].

Almost all the deep learning models applied to-date for malware detection on Android have been based on the core concept of a Convolutional Neural Network (CNN) [15]. This type of neural network is based on the convolution operation, which is inherently translation invariant. This means that if a feature is found in one location, it is the same as being found in another location and the relationship between different features is not important. An easy to understand example is that of a car–for a CNN, an image with a hood, four tires and a couple of doors is a car regardless of whether they are all in the *correct position relative to each other*. We note that in the case of real world images, it is unlikely that we will see different parts of an image that do not 'fit together' and therefore CNNs have immense success in the image domain.

However, for the case of malware (and opcode sequences in any software in general), this is not a feature one would consider useful. For instance, the order in which different sequences or subsequences of operations are carried out is of immense importance. This has been shown by the extremely crafty technique of Return-Oriented Programming [16] to make benign software act maliciously simply by changing the order in which different operations are carried out.

In this paper, we thus argue that translation invariance is not a property we would like to have in our models and moreover, the spatial relationship between different features is also quite important. For instance, the orientation in images gives different latent properties of the objects within the image such as their geospatial relation with the coordinate system as well as the different objects within that coordinate system. Similarly, the concept of orientation in malware data can capture the relationship between the baseline functioning of executables and their behavior towards other executables on the system. Existing malware detection techniques based on CNN cannot capture such data because of the inherent limitations of CNNs [17]. For instance, in case of image-based datasets, the features learned by different convolutional layers could be visualized and information such as edges, lines, and curves in images could easily be depicted. However, in case of malware datasets it is extremely difficult to visualize or even comprehend the learned features. Moreover, by design CNNs are rotation and translation invariant, which means that a feature found in one location is as good as if it is found in any other location regardless of their spatial relationship as mentioned in our car example above.

We therefore need to forsake CNN and turn to a model that handles these two issues properly. In the recent past, Capsule Networks [17] have been shown to handle both these issues with great success in the domain of image analysis. We therefore utilize Capsule Networks to model malware and thus remove both these limitations from malware detection. Concretely, our contributions in this paper are:

(1) We propose and implement a deep learning-based model for malware analysis that is centered around the concept of Capsule Networks as opposed to CNNs. This helps us learn "orientation-based" relationships between different subsequences in an application's code that captures the behavior in a much more succinct manner.
(2) We evaluate this model on three very large scale datasets to demonstrate its efficacy in detecting real-world malware.
(3) We make our source for the whole pipeline available as open source so that any further modifications to our work can be made easily to further enhance the results.

The rest of the paper is organized as follows: We first describe the background related to deep learning in brief. We then turn to Capsule Networks in particular and show how it is different from the traditional CNN-based models. Section 3 shows the details of our model and network design as well as the details of the dataset. Our detailed experiments, results, limitations, and future directions are discussed in Section 4. Related work is detailed in Section 5 and the paper is concluded at the end.

2 Background

Malware detection and its analysis over various platform has remained a top priority of researchers working in the domain of security. A number of approaches are presented for the purpose, each having its own weaknesses and strengths. The presented solution can be broadly classified into traditional approaches i.e., use of Anti-virus software programs along with various hardware devices. However, it suffers issues like availability of signatures from prior set of families of malware. Machine learning approach regarding malware analysis automates the analysis mechanism. However, traditional approaches are unable to learn application feature over a large scale. Deep architectures have proven quite effective in the tasks of classification, and it facilitates in terms of reducing human involvement and increasing classification accuracy. Deep architectures can learn malign traits in automated fashion. Malware developers write applications in a manner that malware changes its appearance to avoid being detected. Handcrafted features are ultimately used for labeling through use of algorithms. However, the newly developed malware are rapidly increasing and these cannot be detected efficiently and at scale through hand-crafted features.

The conception of Deep learning satiates end-to-end learning, which refers to machine learning that can automate every step starting from input to the architecture, extracting features from the input and then performing various operations for its representation for classifying the input accordingly. The Deep learning architectures include neural networks wherein the nodes are organized into layers and are densely connected. The concept of deep learning is proven quite effective in domains of artificial intelligence [18,19]. For the purpose of clarity, a number of deep learning models are presented below:

Typically, in machine learning a dataset is classified into training and test set. A process is trained on the training set to generate another model that receives samples from test set as input, performs some computation based on theta values and predicts the ground truth. To minimize the energy function, the theta values are modified to optimize the model. Regression model is fitted to obtain an overall conception of how far the expected values are from the ground truth for individual data items. In case errors are high, the gradient descend mechanism is adopted to minimize the error. However, the typical machine learning approaches require human intervention in terms of defining the input for training purpose. Also, the traditional machine learning approaches are not efficient computationally [20]. To overcome the difficulties of traditional machine learning, deep neural network architectures are introduced. In the following subsections, we provide a comprehensive details of various Deep Learning models.

Each Deep learning model essentially includes neural networks. It provides an approximation to the ground truth in a specific domain similar to a human brain. After training, input from the test set is provided to neural networks so that it can extract features and based on learned features it can predict the label as result. A neural network (cf. Fig. 2) has an input layer containing data in the form of text, image etc. The inputs are obtained from a dataset, which can be either in processed or unprocessed form. The processed data contains rows defining various input data points and columns, which specifies associated feature with the data points. These features are then fed to the input layer for Neural Network training one by one [21]. The dataset is preprocessed if it is not in conformance with format design of the Deep learning model.

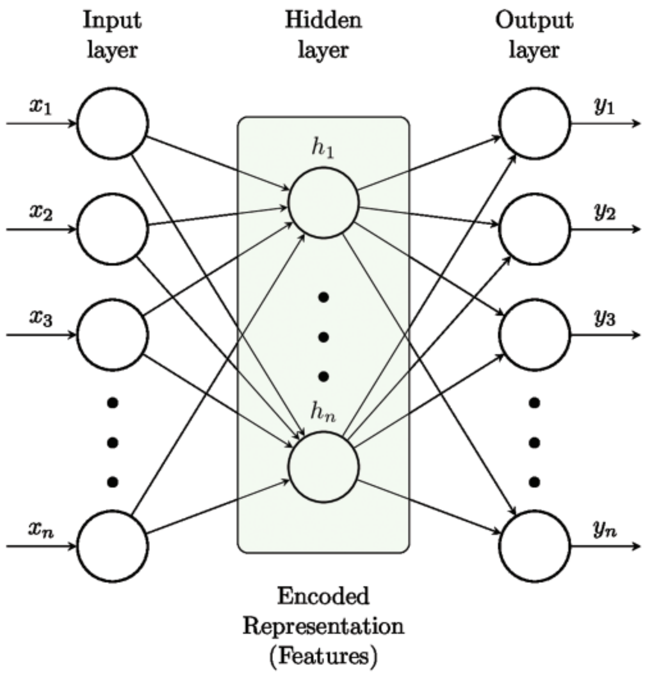

Figure 2: Fully connected neural networks

2.1 Hidden Layers of the Network and its Various Components

Hidden layer includes neurons with activation functions inside them. These receive input previous layer in form of $input{*}weights{+}bias$ and uses activation functions to present the input in a range [22]. Similarly, it provides output to the next layer. Hidden layers are not included in a simple neural network. It is only included for the deep learning model.

2.2 Learning Through Activation Functions

Likewise, activation functions are utilized inside each neuron of hidden and output layer to decide whether the coming input from the previous layer is significant. The activation functions are of various kinds including the linear and sigmoid function. The linear activation function puts linearity in the network and has the following as general function $F(x) = ax + b$ The sigmoid activation function is also a most well-known function that outputs values between 0 or 1. However, it suffers from vanishing gradient problem because it produces small values which diverges the learning process as well [23]. Another activation function is called Tanh function that produce output between -1 and 1. Tanh functions also face vanishing gradient issue as well at some stage of neural training. A general solution repeatedly used to remedy this issue is the ReLu activation function and its several variants [24].

2.3 Fully Connected Neural Network (FCNN)

Fully connected neural network contains layers of nodes that are connected fully to each other. A fully connected layer is a function from R^m to R^n. Wherein its output dimension depends on the input dimension. One of the forms of fully connected layer is represented pictorially in Fig. 2.

Since all nodes are connected and there may be a number of other layers, the fully connected network suffers from vanishing gradient issue [25]. Another issue is that of huge number of parameters which is handled through CNN. Similar to ordinary neural networks, CNNs are made up of neurons that are sparsely connected instead of fully connected nodes. The neurons receive some inputs, perform a dot product followed with a non-linearity. It repeatedly applies a number of filters on images which results in translation invariance [26]. For a detailed analysis of these and other neural network based methods, we refer the readers to [14].

2.4 Machine Learning and Capsnets

Traditional machine learning employed the age old concept of statistically modelling the correlation between input features and outputs. As mentioned before, this was extremely difficult and did not scale well. Fully connected deep neural networks were introduced to automatically extract features from large datasets.

However, there are two major limitations with CNN first, CNN were developed with image-based datasets in mind. This has its upsides–learned features could be visualized and depicted information such as edges, lines, curves, and high and low contrast areas. However, for datasets of a different nature–such as malware, the learned features are extremely difficult to visualize or even comprehend.

The second and more important limitation is more rooted in the way CNNs are designed. By their very nature, CNNs are rotation and translation invariant. This means that if a feature is found in one location, it is just as good as if it was found in any other location. While this makes CNNs more scalable and a lot more computationally feasible, it has its drawbacks. Consider, for example, a case where there is an image that contains several tires strewn around a workshop.

There is also a hood of a car and a few doors in the image. As far as the CNN is concerned, all the features that define a "car" are present in the image and therefore, the image represents a car–which is definitely not the case.

To tackle this issue, Sabour et al. [17] devised a new architecture of deep learning called Capsule Networks (or CapsNet). The idea is that the different features extracted from a data point not only have to be present, but they also need to have an interaction with each other before they can be considered to form a larger whole. In the example above, the tires and the hood need to be in a certain position relative to *each other* before they can be considered parts of a car. The way through which this is achieved is described briefly as follows.

Capsule networks are very similar to CNNs with a few minor changes. The basic architecture of a CapsNet is essentially that of a few filters connected sparsely with each node on the next layer. The output produced are through the typical convolution operation.

However, after the convolution operation has been applied, the activation has three possible outcomes:

- If the activation is *not* carried out but the feature does not contribute correctly to the overall label, it is penalized by a factor of β_u
- If the activation is carried out and the feature does not contribute to the final label, it is penalized by β_u and an additional penalty of β_a
- Finally, if we feature did contribute to the correct label, it is not penalized and in essence carried more weight in the final label.

This seemingly simple algorithm is termed as routing-by-agreement since the different features combine together and agree on what the final label should be. An example of how parts of an image can contribute to the correct labeling of the overall image can be seen in Fig. 3. The two images contain the correct parts but the orientation of the triangle and its spatial position relative to the rectangle dictates whether the image should be labeled a house or a boat.

Moreover, the cost function is slightly modified as well. Briefly, it is defined in Eq. (1) as:

$$cost\frac{h}{j} = \sum - r_{ij} \ln(P^h_{i|j})$$

$$= \frac{\sum_i r_{ij} \left(V^h_{ij} - \mu^h_j\right)^2}{2\left(\sigma^h_j\right)^2} + \left(\ln(\sigma^h_j) + \frac{\ln(2\pi)}{2}\right) \sum_i r_{ij}$$

$$= \left(\ln(\sigma^h_j) + \frac{1}{2} + \frac{\ln(2\pi)}{2}\right) \sum_i r_{ij} \qquad (1)$$

Let us explain that briefly: i denotes a data point and j denotes a single capsule. $P^h_{i|j}$is the probability density of the vote dimension h for the given data point and capsule. Finally, $V^h_{i|j}$ is the "vote" for the capsule j for data point i along dimension h. This may be, for the sake of understanding, considered as the activation along this dimension.

An easier way to understand this is to think of the activation of the output of a single node as not being a single value but a vector of h dimensions.

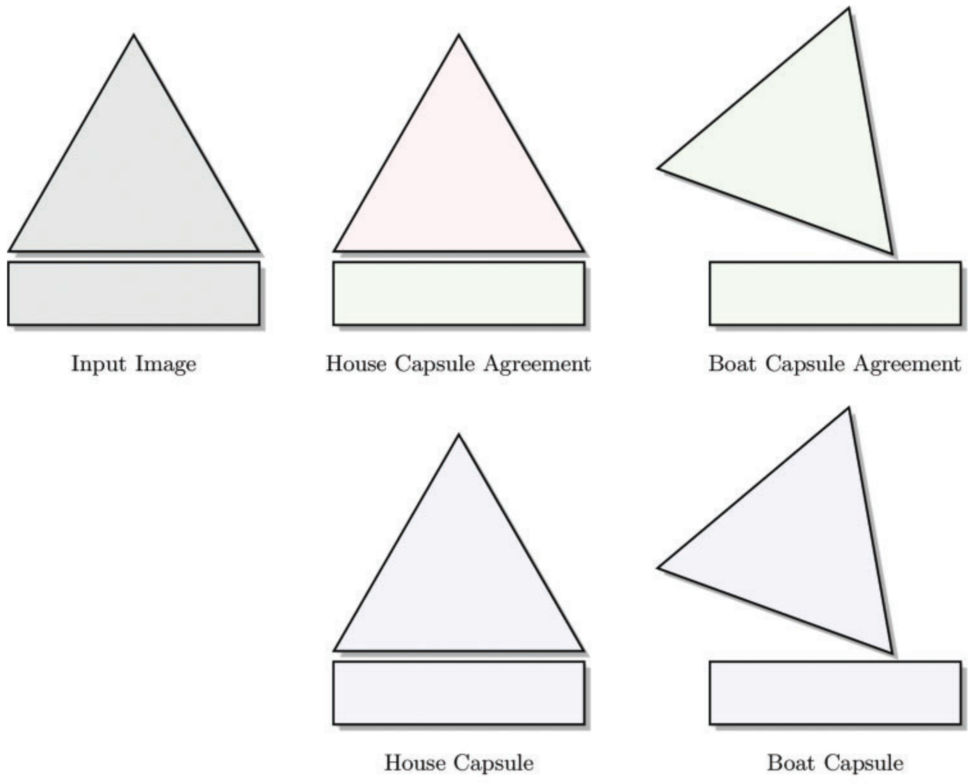

Figure 3: Capsule network routing-by-agreement example

Based on the above, the final activation for capsule j in the output layer is defined in Eq. (2) as:

$$a_j = logistic\left(\lambda\left(\beta_a - \beta_u \sum_i r_{ij} - \sum_h cost_j^h\right)\right) \tag{2}$$

This routing-by-agreement algorithm ensures that only those features are contributing to the final label which actually have an agreement about the different *properties* of the final label. These properties are, in the case of images, orientation of the different parts and their composition. For instance, the orientation of a triangle in Fig. 3. For further details about this algorithm as well as the effectiveness of this method, we refer the interested reader to [17].

In our use case of malware analysis, these different properties will be completely different. We discuss how CapsNet can be used to improve malware analysis and make it much more semantics-aware in the section below.

3 CapsDroid

As discussed before, the limitations of CNN have been catered to effectively in the domain of image classification through the use of Capsule Networks. In this section, we describe how we envision the same model as being applied to malware analysis and the changes required to model the problem in this domain.

3.1 Network Architecture

We base our malware analysis model on the seminal work by Sabour et al. [17]. First, we describe the architecture that achieved the best results and then discuss some of design choices and hyperparameter values we tried along the way to our top result.

Our top-performing model in terms of accuracy of classification is shown in Fig. 4. It is essentially a two-layer learning process. The primary difference between the way capsules are used in image classification and our work is the input. Images are two dimensional by nature and have a horizontal and vertical correlation between neighboring pixels.

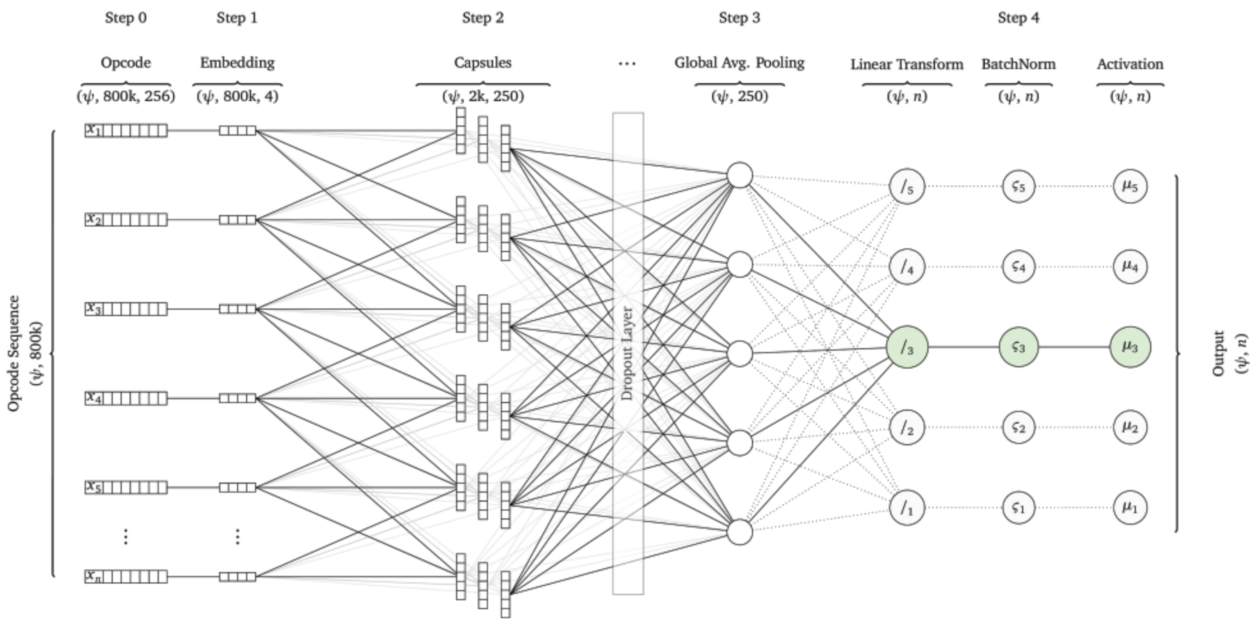

Figure 4: CapsNet architecture for predicting families of malware

3.1.1 Embeddings for Opcodes

The complete architecture of our CapsNet is shown at an abstract level in Fig. 4. Malware analysis is typically done using sequences of opcodes (or bytecode in our case). This is a one-dimensional structure. Moreover, the dependence can be not only among neighboring operators but also among opcodes farther apart from each other. Finally, since the opcode values are discrete and do not have an inherent partial or full order to them, the numerical values assigned to opcodes are purely arbitrary.

In order to break the partial ordering between the numeric values assigned to them, we first convert the numeric values to one-hot vector encoding. This results in each opcode in the sequence to be represented using a vector of length 256–the total number of possible opcodes. However, this is a very large vector (especially considering the fact that typical length of the sequences of these opcodes is more than 80,000). If we use this encoding, the number of nodes in our input layer for the CapsNet would be more than 20 million–which is obviously infeasible.

To cater to this problem, we first use a different neural network to learn the embeddings for each of the opcodes. This can be seen in Step 0 and Step 1 of our full architecture in Fig. 4. To achieve this goal, we create a four-layer neural network which tries to classify the data points into malware and benign applications. While this dataset is nothing new and does not achieve good results, it does help us learn embeddings for the opcodes in its first layer. Once this network reached an accuracy of 80%, we stopped the training and used the first layer of the network as the embeddings for the opcodes. The weights of the nodes for the first layer were used to learn vector embeddings for opcodes. This vector was only of length 4 which significantly reduces the amount of computation and storage required in the CapsNet network.

3.1.2 CapsNet

We then re-encoded our complete dataset to represent each of the opcodes using the embeddings learned in the first step. This was a one-time process and any changes to the CapsNet architecture did not affect our embeddings. The output of this layer was therefore in the shape (ψ,800000,4), where ψ was the batch size. These embeddings were then fed to the capsule layer. This is where the core of our learning happens. We kept the number of capsules to quite small. Our best results were achieved using the number of capsules set to 250. (We discuss alternate results later.) In order to avoid overfitting over the training data, we utilized a separate dropout layer after the capsule layer as it is not clear if embedding the dropout within the capsule layer would be prudent.

Afterwards, in order to reduce the number of parameters, we utilized a global average pooling layer which significantly reduces the number of parameters before passing them on to a fully connected layer. While this has the disadvantage of losing some information, the alternative is simply not feasible as two fully connected layers of more than 500,000 parameters would lead to over 250 billion trainable parameters. Finally, we utilize batch norm to avoid the problem of internal co-variate shift which invariable creeps into deep learning model [27]. The output layer of our model depicted in Fig. 4 was used to predict the family of malware to provide fine-grained classification. Our initial experiments however, only predicted whether the sample being fed was malicious or benign.

Some further details of how the analysis was carried out from a practical perspective are given below.

3.2 Dataset Description

As with any deep learning-based analysis, we need to have a large enough dataset to satiate the requirements of the deep model with a large number of parameters. If we do not have sufficient data, we would be significantly overfitting. To handle this issue, we utilized three large scale datasets in tandem. First one is Drebin which is a massively popular, high quality collection of malware collected over a period of two years containing more than five thousand malware samples belonging to 20 families.

The second dataset used in our study was Android Malware Dataset (AMD) which is a slightly newer dataset that has more than 60 gigabytes of malware collected in more than a hundred families.

Finally, we augmented this with some very recent malware samples coming from VirusShare. All three combined, we had a total malware collection of 40,000+ samples. To create a balanced dataset, we downloaded around 50,000 samples of benign applications from the Google Play store (along with some third party stores). To get a slightly better assurance that these samples were indeed benign, they were passed through the VirusTotal online virus detection service. The final result was a benign sample size of 45,000 and a malicious sample size of approx. 40,000.

Since the dataset contained Android packages (apk files), which cannot be fed directly to a neural network, we processed them using the AndroGuard tool which helped us extract the dex files, which contain the actual bytecode of applications. After getting the dex file, we used an in-house script to extract the opcode sequences from the code section of the dex file. The operator part of each instruction was saved, and the operand parts were ignored for the sake of tractability. The resulting data was saved in a *hdf5* file for easier processing in the future. This dataset was then fed to the neural network as described in Section 3.1.2. We describe the physical aspect and some design decisions for our experiment in the section below.

4 Experiments and Results

4.1 Experiment Setup

The experiment was run on an NVidia k80 GPU (mounted on a system with a 32GB main memory and a 8-core processor clocked at 3.5 GHz) using Keras with the Tensorflow backend. The code for our CapsNet module for Keras is available as open source under the GPLv3. Since the sequences in datasets were quite long, we clipped them to a max sequence length of 800,000. This covered more than 92% of the data points fully but led to a significant improvement in performance and learning times.

Our codebase is dependent on Keras as the front-end API to Tensorflow. We utilized *h5py* to perform out-of-memory data allocation since the total dataset was much larger than the total available physical memory. Matplotlib and the excellent seaborn library were used to automatically generate learning curves and result plots simultaneously with the training and validation process.

Our maximum sequence length discussed above was a major hyperparameter. If this hyperparameter is reduced by a lot, the training becomes quite fast but unfortunately turns the whole learning process into an n-gram feature classifier, which has poor accuracy. If, on the other hand, we increase this number by a lot, the resulting accuracy is exceptional but leads to slow and sometimes infeasible learning times.

Another important hyperparameter is the number of layers in the CapsNet architecture described above. Again, the issue is that if we increase the number of layers even by one, the inefficiency of the CapsNet architecture to-date means increasing the time taken by an order of magnitude. This significantly reduces the ability to, at least given the hardware we have access to, increase the number of layers to improve the accuracy further. Hopefully with a cluster of GPUs and utilizing Tensorflow's horizontal parallelization of the network, we should be able to train a much larger network in the future.

Last but not least, the number of capsules in the hidden layer is a major hyperparameter. We tried several alternatives for this hyperparameter. Details of the effect of these experiments are discussed in the section below.

One important point to note is that, typically, hyperparameters are optimized using a better strategy such as grid search. This was not feasible in our case as a single training session takes days to complete even a handful of epochs. This would make grid search infeasible and thus required expert hand-tuning of the hyperparameters discussed above.

4.2 *Results and Discussion*

We summarize the results of all our experiments in Tab. 1. We compare our results to [14,28–31] since these are the best results achieved for malware detection till date. The symbol '-' in Tab. 1 indicates that the corresponding metric is not reported by the source paper. As can be seen, CapsDroid performs quite well in terms of the evaluation metrics. We achieved an F1 score of 0.987. This seems like a minor improvement over LUNA but note that LUNA is an uncertainty based model and it achieved an F1 score of 0.986 at the expense of a loss of coverage of 0.024. It means that while they were able to predict with that F1 score, they were not able to make predictions for *all* of the applications due to the higher amount of uncertainty associated with some applications. This, we feel was a major limitation in that model. CapsDroid, however was able to classify all applications with a minimal False Positive Rate (FPR).

Table 1: Comparison of results with existing techniques

Metric	[28]	[29]	DDefender [32]	IMCFN [33]	LUNA [14]	CapsDroid
Accuracy	0.833	0.858	0.916	0.982	0.989	0.991
Precision	0.714	0.892	0.857	0.982	0.982	0.983
F1 Score	0.833	0.874	0.923	**0.982**	**0.986**	**0.987**
Coverage	1.0	1.0	1.0	-	**0.976**	1.0
FPR	0.01	-	-	-	0.002	0.002
Detec. Rate	0.94	0.969	-	0.973	0.981	0.980
AUC	0.88	-	0.93	-	0.983	**0.984**
Datasets	Drebin	Android Malware Gnome	Drebin	Malimg and IoT-Android	Drebin and VirusShare	Drebin, Android Malware Dataset, and VirusShare
Learning Model	SVM	DNN	DNN	CNN	Multiple Deep Learning Models	Capsule Network

For the rest of the metrics, it can be noted that we achieved the lowest FPR of only 0.2% as well as a very high detection rate of 98% and an Area Under the Curve (AUC) of 0.984. The higher AUC value achieved by CapsDroid implies that it is able to distinguish the benign applications from the malicious samples in the datasets. An important aspect of the results shown in Tab. 1 is that the F1 score and AUC of the model improves significantly with increase in the complexity of the learning models as evident from the performance metrics. The IMFCN approach also reported a higher F1 score of 0.982 with almost the same accuracy rate. However, these results were achieved by using relatively smaller datasets for experiments.

The Receiver Operating Characteristic (ROC) curves of our model along with that of LUNA are shown in Fig. 5. The graph showing the training and validation loss calculated for each epoch of the CapsDroid network is shown in Fig. 6, where the vertical dashed line shows the *early-stopping* point, which is used as a form of regularization to avoid overfitting of the model. As mentioned earlier, the dropout layer embedded after the capsule layer and early-stopping technique helped in avoiding overfitting over the training data and improves our model's performance on the test data.

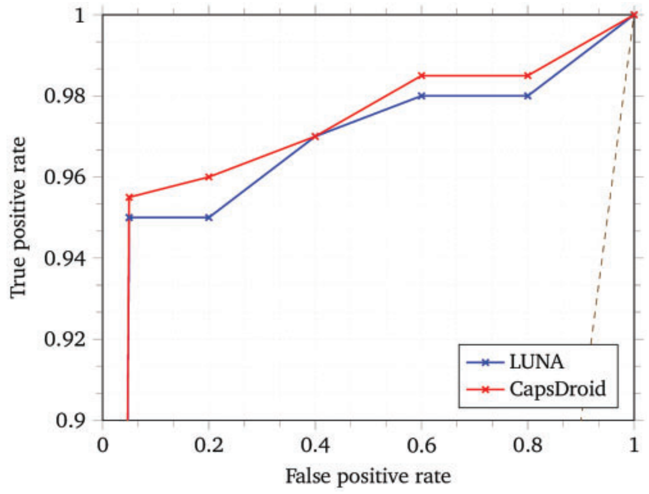

Figure 5: ROC comparison: LUNA *vs.* CapsDroid

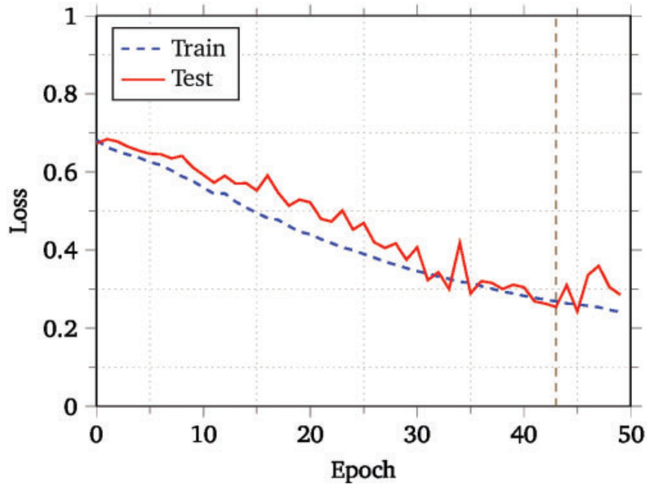

Figure 6: Loss curves for the best model

4.3 Limitations and Future Directions

Although, CapsDroid achieve good results, we do note a limitation of our model here. The time taken to perform the training of our model is quite slow as compared to other deep learning models. This is due to the fact that the capsule module (and its routing-by-agreement algorithm) is quite slow in nature. This is an active area of machine learning research, and we are quite confident that advancements in this area will alleviate this limitation of our work overtime.

Moreover, since this is only an issue at training time, the runtime performance of our model is not an issue at all.

An important research direction to enhance the work presented in this paper is to analyze dynamic behavior of applications such as system calls traces that would classify benign and malicious applications based on the inherent relationship between system calls made by various applications during the course of execution. Another important future research route would be to analyze our model performance over a broader range of applications across platforms other than Android such as iOS and Windows platform. We believe that machine learning has the potential to effectively tackle the over expanding issue of malware on the current mobile ecosystem and our work can contribute to highlight the effectiveness of deep learning models in mitigating these treats.

5 Related Work

Machine learning techniques are extensively used for mitigating cyber security threats in general and are continuously optimized for malware detection and classification in particular [34,35]. Some of the recent machine learning classifiers used for malware detection on mobile devices are presented here.

Bourebaa et al. [36] utilizes multiple CNN-based models for detection and classification of malware on Android-based devices. The authors extracted API calls and application permissions list as features and applied a one-dimensional CNN model to suits low memory usage on handheld devices. Further, they applied Inception and Multichannel CNN-based models to achieve better performance in terms of accuracy rate. Although, the choice for linear model for simplification and lower memory usage is surmounted by the complexity of inception and multichannel models. Similarly, Alshahrani et al. [32] proposed DDefender, an Android malware detection solution that employ both the static and dynamic malware analysis techniques to extract features such as application's components, system calls, requested permissions, network traffics, and system information from the device. DDefender use dynamic analysis to extract features for an inspected application at runtime, then apply static analysis to further extract features from the APK files, and then apply deep learning-based model to detect malicious applications. Though, the hybrid approach of DDefender outperformed some of the best performance models, however significant performance overhead was reported.

McLaughlin et al. [13] have used convolutional neural networks (CNN) for malware classification using static analysis of bytecode from a disassembled application. Malware indicative features are learned automatically by the network from raw bytecode sequence, which eliminates the need for malware signature to be designed manually. Similarly, Karbab et al. [30] proposed MalDozer, an Android malware detection framework based on sequences mining using CNN. MalDozer takes the sequences of API calls as input, which are extracted from Android app packages, and generate sequences of vector for all applications in the same order of API calls. It applies neural network to the vector sequence and classify applications as benign or malicious. Hou et al. [37] have proposed an Android malware detection framework that extracts Linux kernel system calls for each application automatically and construct weighted directed graphs. Further, they apply convolutional neural network framework to classify newly unknown malware. Hasegawa et al. [38] has proposed a one-dimensional CNN (1-D CNN) for malware detection using a small portion of Android Application Package (APK) files. Using the shallow one-dimensional model, their model has achieved a promising accuracy rate. However, the authors only analyze a small portion of APK files without decompression, reduces the chances of capturing key features of malware.

Martinelli et al. [39] has proposed a CNN-based linear model that is applied to system calls occurrences of applications installed on Android device. The authors used a relatively small dataset for experiments and achieved detection accuracy of up to 95%. Similarly, Yakura et al. [40] used a linear CNN model to extract bytecode sequences and combine an attention mechanism to produce attention maps for malware. The bytecode sequences are converted into images and the attention mechanism is applied to highlight important regions for malware classification. However, the extracted sequences from the attention maps are only useful for manual analysis, which could only be performed for bytecode extracted from a smaller number of applications. Another CNN-based approach known as IMCFN, proposed by Vasan et al. [33], converted malware bytecode to colorful images and applied a fine-tuned CNN-based architecture known as ImageNet to perform multiclass malware classification. The used two different malware datasets and achieved an accuracy of 98.82% with one of the datasets used. However, similar to [40], the authors used relatively smaller datasets for experiments.

Fan et al. [31] applied Deep Belief Network (DBN) architecture to classify malware by constructing a deep learning model. The authors have also compared their model detection accuracy with Logistic Regression based models, C4.5, and SVM and concluded that the deep learning model showed better accuracy than the other machine learning models. Ali-Gombe et al. [41], and Canfora et al. [42] proposed dynamic analysis method by identifying behavioral signature of running applications. Although, these and other dynamic analysis methods [43–45] are more resistant to obfuscation but offer less scalability due to incurring additional computational cost [46]. We have based our work on different datasets of significantly larger size. The previous studies mentioned here do not report their findings on these datasets. While we also provide our results based on the datasets they have used, our results are much more comprehensive.

6 Conclusion

Detection of malware on mobile phones–especially on Android-based systems, which form a vast majority of the market share–is an immensely important and constantly changing research landscape. In this paper, we have presented CapsDroid–a Capsule Network-based model–for malware detection. This model presents a new way of detecting features of malware that not only provides better results in terms of detection rates and F1 measure but also output features that are more coherent and capture the intrinsic relationship between opcode sequences in different parts of the code being analyzed. Our model was able to beat state-of-the-art results for malware detection on the Android platform. However, note that our model, in its general form, is not restricted to the Android platform. We envision the use of this technique to analyze a broader range of application code on other platforms as well such as iOS, Windows and MacOS–whichever platform you consider–the code is invariably composed of a sequence of instructions, whether interpreted bytecode or compiled machine code. Our model can be used without too many modifications, and we plan on exploring these venues in much greater in the future.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] International Data Corporation (IDC), Worldwide Quarterly Mobile Phone Tracker, IDC Q1 Report, 2021. [Online]. Available: IDC Mobile Phone Tracker.

[2] Statcounter, GlobalStats, Mobile Operating System Market Share Worldwide Report. Q2 Report, 2021. [Online]. Available: GS Statcounter: OS Market Share Worldwide.

[3] P. Teufl, M. Ferk, A. Fitzek, D. Hein, S. Kraxberger *et al.*, "Malware detection by applying knowledge discovery processes to application metadata on the android market (google play)," *Security and Communication Networks*, vol. 9, no. 5, pp. 389–419, 2016.

[4] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[5] Future of Privacy Forum, Always on: Privacy implications of microphone-enabled devices, Report, 00003-128652, 2016. [Online]. Available: Future of Privacy Forums: Documents: 00003-128652.

[6] J. C. Sipior, B. T. Ward and L. Volonino, "Privacy concerns associated with smartphone use," *Journal of Internet Commerce*, vol. 13, no. 3, pp. 177–193, 2014.

[7] McAfee, McAfee threats report: Second quarter, Q2 Report, 2011. [Online]. Available: McAfee Lab: Threat Reports.

[8] F-Secure, Mobile threat report, Q4 Report, 2012. [Online]. Available: F-Secure: Threat Report: 996508.

[9] Juniper Networks, Report, Mobile threat report, Q4 Report, 2011. [Online]. Available: Juniper Networks: Threat Reports.

[10] Kaspersky lab, Mobile malware evolution, Q2 Report, 2020. [Online]. Available: Kaspersky: Securelist: IT Threat.

[11] L. Gao and K. A. Waechter, "Examining the role of initial trust in user adoption of mobile payment services: An empirical investigation," *Journal of Information Systems Frontiers*, vol. 19, no. 3, pp. 525–548, 2017.

[12] R. Vinayakumar, K. P. Soman and P. Poornachandran, "Deep android malware detection and classification," in *Proc. IEEE Int. Conf. on Advances in Computing, Communications, and Informatics (ICACCI)*, Manipal, India, pp. 1677–1683, 2017.

[13] N. McLaughlin, J. M. Rincon, B. Kang, S. Yerima, P. Miller *et al.*, "Deep android malware detection," in *Proc. Seventh ACM Conf. on Data and Application Security and Privacy (CODASPY)*, Scottsdale, USA, pp. 301–308, 2017.

[14] M. Nauman, T. A. Tanveer, S. Khan and T. A. Syed, "Deep neural architectures for large scale android malware analysis," *Cluster Computing*, vol. 21, no. 1, pp. 569–588, 2018.

[15] 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.

[16] R. Roemer, E. Buchanan, H. Shacham and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15,. no. 1, pp. 1–34, 2012.

[17] S. Sabour, N. Frosst and G. E. Hinton, "Dynamic routing between capsules," in *Proc. Advances in Neural Information Processing Systems*, California, USA, pp. 3856–3866, 2017.

[18] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[19] I. Arel, D. C. Rose and T. P. Karnowski, "Deep machine learning-a new frontier in artificial intelligence research," *IEEE Computational Intelligence Magazine*, vol. 5, no. 4, pp. 13–18, 2010.

[20] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald *et al.*, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, no. 1, pp. 1–21, 2015.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, Boston, MA, USA, pp. 1–9, 2015.

[22] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee *et al.*, "Multimodal deep learning," in *Proc. 28th Int. Conf. on Machine Learning (ICML-11)*, Bellevue, USA, pp. 689–696, 2011.

[23] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, no. 1, pp. 85–117, 2015.

[24] A. Radford, L. Metz and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," preprint, arXiv:1511.06434, 2015.

[25] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, Edinburgh, UK, pp. 770–778, 2016.

[26] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks,", *Advances in Neural Information Processing Systems*, vol. 25, no. 1, pp. 1097–1105, 2012.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. on Machine Learning (ICML)*, Lille, France, pp. 448–456, 2015.

[28] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck *et al.*, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. Annual Symposium on Network and Distributed System Security (NDSS)*, California, USA, pp. 23–26, 2014.

[29] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 1–29, 2018.

[30] E. M. B. Karbab, M. Debbabi, A. Derhab and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[31] C. I. Fan, H. W. Hsiao, C. H. Chou and Y. F. Tseng, "Malware detection systems based on API log data mining," in *Proc. IEEE 39th Annual Computer Software Applications Conf. (COMPSAC)*, Taichung, Taiwan, pp. 255–260, 2015.

[32] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani *et al.*, "DDefender: Android application threat detection using static and dynamic analysis," in *Proc. IEEE Int. Conf. on Consumer Electronics (ICCE)*, Las Vegas, USA, pp. 1–6, 2018.

[33] D. Vasan, M. Alazab, S. Wassan, H. Naeem, B. Safaei *et al.*, "Image-based malware classification using fine-tuned convolutional neural network architecture," *Computer Networks*, vol. 171, no. 1, pp. 107–138, 2020.

[34] K. Shaukat, S. Luo, V. Varadharajan, I. A. Hameed and M. Xu, "A survey on machine learning techniques for cyber security in the last decade," *IEEE Access*, vol. 8, no. 1, pp. 222310–222354, 2020.

[35] K. Shaukat, S. Luo, S. Chen and D. Liu, "Cyber threat detection using machine learning techniques: A performance evaluation perspective," in *Proc. IEEE Int. Conf. on Cyber Warfare and Security (ICCWS)*, Islamabad, Pakistan, pp. 1–6, 2020.

[36] F. Bourebaa and M. Benmohammed, "Android malware detection using convolutional deep neural networks," in *Proc. IEEE Int. Conf. on Advance Aspects of Software Engineering (ICAASE)*, Constantine, Algeria, pp. 1–7, 2020.

[37] S. Hou, A. Saas, L. Chen and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *Proc. IEEE/WIC/ACM Int. Conf. on Web Intelligence Workshops (WIW)*, Omaha, NE, USA, pp. 104–111, 2016.

[38] C. Hasegawa and H. Iyatomi, "One-dimensional convolutional neural networks for android malware detection," in *Proc. Int. Colloquium of Signal Processing and Applications*, Penang, Malaysia, pp. 99–102, 2018.

[39] F. Martinelli, F. Marulli and F. Mercaldo, "Evaluating convolutional neural network for effective mobile malware detection," in *Procedia Computer Science*, vol. 112, no. 1, pp. 2372–2381, 2017.

[40] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama and J. Sakuma, "Malware analysis of imaged binary samples by convolutional neural network with attention mechanism," in *Proc. Eighth ACM Conf. on Data and Application Security and Privacy (CODASPY)*, Tempe, USA, 2018.

[41] A. Ali-Gombe, I. Ahmed, G. G. Richard III and V. Roussev, "Aspectdroid: Android app analysis system," in *Proc. Sixth ACM Conf. on Data and Application Security and Privacy (CODASPY)*, New Orleans, Louisiana, USA, pp. 145–147, 2016.

[42] G. Canfora, E. Medvet, F. Mercaldo, C. Visaggio and A. Corrado, "Acquiring and analyzing app metrics for effective mobile malware detection," in *Proc. ACM Conf. on Int. Workshop on Security and Privacy Analytics (IWSPA)*, New Orleans, Louisiana, USA, pp. 50–57, 2016.

[43] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proc. the 21st USENIX Security Symposium*, Bellevue, USA, pp. 569–584, 2012.

[44] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proc. the Seventh European Workshop on System Security*, Amsterdam, Netherlands, pp. 1–6, 2014.

[45] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, USA, pp. 15–26, 2011.

[46] K. Demertzis and L. Iliadis, "Computational intelligence anti-malware framework for android OS," *Vietnam Journal of Computer Science*, vol. 4, no. 4, pp. 245–259, 2017.