electronics

MDPI

Article

Deep Learning Techniques for Android Botnet Detection

Suleiman Y. Yerima [1,*], Mohammed K. Alzaylaee [2], Annette Shajan [3] and Vinod P [4]

[1] Cyber Technology Institute, De Montfort University, Leicester LE1 9BH, UK
[2] College of Computing in Al-Qunfudhah, Umm Al-Qura University, Mecca 21955, Saudi Arabia; mkzaylaee@uqu.edu.sa
[3] RV College of Engineering, Bengaluru 560059, India; annetteshajan@gmail.com
[4] Department of Computer Applications, Cochin University of Science and Technology, Cochin 682022, India; vinod.p@cusat.ac.in
* Correspondence: syerima@dmu.ac.uk

Abstract: Android is increasingly being targeted by malware since it has become the most popular mobile operating system worldwide. Evasive malware families, such as Chamois, designed to turn Android devices into bots that form part of a larger botnet are becoming prevalent. This calls for more effective methods for detection of Android botnets. Recently, deep learning has gained attention as a machine learning based approach to enhance Android botnet detection. However, studies that extensively investigate the efficacy of various deep learning models for Android botnet detection are currently lacking. Hence, in this paper we present a comparative study of deep learning techniques for Android botnet detection using 6802 Android applications consisting of 1929 botnet applications from the ISCX botnet dataset. We evaluate the performance of several deep learning techniques including: CNN, DNN, LSTM, GRU, CNN-LSTM, and CNN-GRU models using 342 static features derived from the applications. In our experiments, the deep learning models achieved state-of-the-art results based on the ISCX botnet dataset and also outperformed the classical machine learning classifiers.

Keywords: botnet detection; deep learning; Android botnets; convolutional neural networks; dense neural networks; recurrent neural networks; long short-term memory; gated recurrent unit; CNN-LSTM; CNN-GRU

Citation: Yerima, S.Y.; Alzaylaee, M.K.; Shajan, A.; P, V. Deep Learning Techniques for Android Botnet Detection. *Electronics* **2021**, *10*, 519. https://doi.org/10.3390/electronics10040519

Academic Editors: Khaled Elleithy and Ana Rosa Cavalli

Received: 31 December 2020
Accepted: 18 February 2021
Published: 23 February 2021

1. Introduction

The increase in Android's popularity worldwide has made it a continuous target for malware authors. The volume of malware targeting Android has continued to grow in the last few years [1,2]. Android has been attacked by numerous malware families aimed at infecting mobile devices and turning them into bots. These bots become parts of larger botnets that are usually under the control of a malicious user or group of users known as botmasters. The Android botnets may be used to launch various types of attacks such as distributed denial of service (DDoS) attacks, phishing, click fraud, theft of credit card details or other credentials, generation and distribution of spam, etc. Nowadays, malicious Android botnets have become a serious threat. Additionally, their increasing use of sophisticated evasive techniques such as self-protection or multi-staged payload execution [3], calls for more effective approaches to detect them.

The Chamois malware family [3–5], which was discovered on Google Play in August 2016 is one example of the emerging sophisticated Android botnet threats. By March 2018, Chamois had infected over 20 million devices, which were commandeered into a botnet that received instructions from a remote command and control server [5]. The botnet was used to serve malicious advertisements and to direct victims to premiums Short Message Service (SMS) scams. The early version of Chamois disguised as benign apps that tricked users into downloading it on their devices, and this was detected and almost completely eradicated by the Android security team. Later versions of Chamois appeared which were

distributed by tricking developers and device manufacturers into incorporating the botnet code directly into their apps. Chamois was sold to developers as a legitimate software development kit, and to the device manufacturers as a mobile payment solution [5].

The emergence of evasive and technically complex families like Chamois has driven interest in adopting machine learning based techniques as a means to improve existing detection systems. In the past few years, several works have investigated traditional machine learning techniques such as Support Vector Machines (SVM), Random Forest, Decision Trees, etc., for Android botnet detection. Some of the more recent machine learning based Android botnet detection work, such as ref. [6] and ref. [7] have focused on deep learning. Nevertheless, empirical studies that extensively investigate various deep learning techniques to provide insight into their relative performance for Android botnet detection are currently lacking. Hence, in this paper, we present a comparative analysis of deep learning models for Android botnet detection using the publicly available ISCX botnet dataset. Our approach is based on classification of unknown applications into 'clean' or 'botnet' using 342 static features extracted from the apps. We evaluate the performance of several deep learning models on 6802 apps consisting of 1929 botnet apps from the ISCX botnet dataset. The models investigated include Convolutional Neural Networks (CNN), Dense Neural Networks (DNN), Gated Recurrent Units (GRU), Long Short-Term Memory (LSTM), as well as more complex networks like CNN-LSTM and CNN-GRU.

The rest of the paper is organized as follows: Section 2 contains related works. Section 3 gives an overview of the overall system for deep learning-based Android botnet detection, while Section 4 provides brief background discussions of the deep learning models that were built for this study. Section 5 discusses the methodology and experimental approach, while Section 6 presents the results and discussion of results. Finally, the conclusions and future work are outlined in Section 7.

2. Related Work

Kadir et al. in their paper [8], studied several families of Android botnets aiming to gain a better understanding of the botnets and their communication characteristics. They presented a deep analysis of the Command and Control channels and built-in URLs of the Android botnets. They provided insights into each malicious infrastructure underlying the families, and uncovered the relationships between the botnet families by using a combination of static and dynamic analysis with visualization. From their work, the ISCX Android botnet dataset consisting of 1929 samples from 14 Android botnet families emerged. Since then, several works on Android botnet detection have been based on the dataset which is available from ref. [9].

Anwar et al. [10] proposed a mobile botnet detection method based on static features. They combined permissions, MD5 signatures, broadcast receivers, and background services to obtain a comprehensive set of features. They then utilized these features to implement machine learning based classifiers to detect mobile botnets. Having performed experiments using 1400 botnet applications of the ISCX dataset, combined with an extra 1400 benign applications, they recorded an accuracy of 95.1%, a recall of 0.827, and a precision of 0.97 as their best results.

Android Botnet Identification System (ABIS) was proposed in [11] to detect Android botnets. The method is based on static and dynamic features consisting of API calls, permissions, and network traffic. ABIS was evaluated with several machine learning techniques. In the end, Random Forest was found to perform better than the other algorithms by achieving 0.972 precision and 0.96 recall.

In ref. [12], machine learning was used to detect Android botnets using permissions and their protection levels as features. Initially, 138 features were utilized and then increased to 145 after protection levels were added as novel features. In total, four machine learning models (i.e., Random Forest, multilayer perceptron (MLP), Decision Trees, and Naive Bayes) were evaluated on 3270 applications containing 1635 benign and 1635 botnets from the ISCX dataset. Random Forest was found to have the best results yielding 97.3%

accuracy, 0.987 recall, and 0.985 precision. The authors of [13] also utilized only the 'requested permissions' as features and applied Information Gain to reduce the features and select the most significant requested permissions. They evaluated their approach using Decision Trees, Naive Bayes, and Random Forest. In their experiments, Random Forest performed best, with an accuracy of 94.6% and false positive rate of 0.099%.

Karim et al. in [14], proposed DeDroid, a static analysis approach to investigate properties that are specific to botnets that can be used in the detection of mobile botnets. in their approach, 'critical features' were first identified by observing the coding behavior of a few known malware binaries that possess Command and Control features. These 'critical features' were then compared with features of malicious applications from Drebin dataset [15]. The comparison with 'critical features' suggested that 35% of the malicious applications in the Drebin dataset could be classed as botnets. However, according to their study, a closer examination confirmed 90% of the apps as botnets.

Jadhav et al. [16], present a cloud-based Android botnet detection system that leverages dynamic analysis by using a virtual environment with cluster analysis. The toolchain for the dynamic analysis process is composed of strace, netflow, logcat, sysdump, and tcpdump within the botnet detection system. However, in the paper there were no experimental results provided to evaluate the effectiveness of the proposed cloud-based solution. Moreover, the virtual environment can easily be evaded by the botnets using different fingerprinting techniques. In addition, being a dynamic-analysis based approach, the systems effectiveness could be degraded by the lack of complete code coverage [17,18].

In ref. [19], a method was proposed by Bernardeschia et al. to identify Android botnets through model checking. Model checking is an automated technique used in verifying finite state systems. This is achieved by checking whether a structure representing a system satisfies a temporal logic formula describing their expected behavior. In particular, static analysis is used to derive a set of finite state automata from the Java byte code that represents approximate information about the run-time behavior of an app. Afterwards, the botnet malicious behavior is formulated using temporal logic formulae [20]; then by adopting a model checker, it can be automatically checked whether the code is malicious and identify where the botnet code is located within the application. These properties are checked using the CAAL (Concurrency Workbench, Aalborg Edition) [21] formal verification environment. The authors evaluated their approach on 96 samples from the Rootsmart botnet family, 28 samples from the Tigerbot botnet family, in addition to 1000 clean samples. The results obtained on the 1124 app samples showed perfect (100%) accuracy, precision, and recall.

Alothman and Rattadilok [22] proposed a source code mining approach based on reverse engineering and text mining techniques to identify Android botnet applications. Dex2Jar was used to reverse engineer the Android apps to Java source code. Natural Language Processing techniques were applied to the obtained Java source code. They also evaluated a 'source code metrics (SCM)' approach of classifying the apps into 'botnet' or 'clean'. In the SCM approach, statistical measures, such as total number of code lines, code to comment ratio, etc., were extracted from the source code and the metrics were used as features for training machine learning classifiers. The Java source code was extracted from 9 apps from 9 ISCX botnet families, as well as 12 normal apps. The TextToWordVector filter within WEKA (Waikato Environment for Knowledge Analysis), together with TF-IDF, was then applied to the code. They also applied WEKA's StringToWordVector with TF-IDF filter while varying the numbers of the 'words to keep' parameter. SubSetEval feature selection method was used to reduce the features. The features were applied to Naive Bayes, KNN, J48, SVM, and Random Forest algorithms, were KNN obtained the best performance.

In ref. [23], a real-time signature-based detection system is proposed to combat SMS botnets, by first applying pattern-matching detection approaches for incoming and outgoing SMS text messages. In the second step, rule-based techniques are used to label unknown SMS messages as suspicious or normal. Their method was evaluated with over 12,000 test messages, where all 747 malicious SMS messages were detected in the dataset.

However, the system produced some false positives where 349 SMS messages were flagged as suspicious. In ref. [24], a botnet detection technique called 'Logdog' is proposed for mobile devices using log analysis. The approach relies on analyzing the logs of mobile devices to find evidence of botnet activities. Logdog writes logcat messages to a text file in the background while the Android user continues to use their device. The system targets HTTP botnets looking for events or series of events that indicate botnet activities and was tested manually on a botnet and a normal app.

In ref. [6], Android botnet detection based on CNN and using permissions as features was proposed. In the proposed method, apps are represented as images that are constructed based on the co-occurrence of permissions used within the applications. The images were then used to train a CNN-based binary classifier. The binary classifier was evaluated using 5450 apps containing 1800 botnet apps from the ISCX dataset. They obtained an accuracy of 97.2%, with a recall of 0.96, precision of 0.955, and f-measure of 0.957. Similarly, ref. [7] proposes an Android botnet detection approach based on CNN, where not only permissions were used as features but also API calls, Commands, Intents, and Extra Files. Unlike in ref. [6], 1D CNN was used and the model was evaluated with the 1929 ISCX botnet apps and 4873 benign apps resulting in 98.9% accuracy, 0.978 recall, 0.983 precision, and 0.981 F1-score.

Different from the aforementioned earlier works, this paper aims to investigate the performance of several deep learning techniques to gain insight into their effectiveness in detecting Android botnets based on the extraction of 342 static features from the applications. To this end, we implemented CNN, DNN, LSTM, GRU, CNN-LSTM, and CNN-GRU models and evaluated the models using 1929 ISCX botnet apps and 4873 benign apps. The deep learning models developed in the study are discussed in Section 4 and the results of the experiments with the models are presented in Section 6.

3. Deep Learning-Based Android Botnet Detection System

At a fundamental level, our botnet detection system is designed to distinguish between clean apps and botnet apps. As a result, it may sometimes fail to correctly classify an unknown app by mistakenly identifying a benign app as botnet or vice-versa. The various accuracy metrics used in the experiments presented in Section 6 will enable us to capture the extent to which a given deep learning model used as a classifier can be relied upon to correctly predict which category an unknown app should belong to. The classification system is implemented by extracting static features from thousands of applications consisting of both botnet and clean examples. A bespoke tool that we developed in Python for automated reverse engineering of Android Package files (APKs) was utilized in the process. Using the tool, we extracted a total of 342 features from 5 different categories shown in Table 1.

Table 1. The five types of features used in developing the deep learning models.

Feature Types	Number
API calls	135
Permissions	130
Commands	19
Extra executables	5
Intents	53
Total	342

The five feature types include: (1) API calls (2) commands (3) permissions (4) Intents (5) extra (binary or executable) files. Most of the features were from the 'API calls' and 'permissions' category as shown in Table 1. A selection of the features is shown in Table 2. These features are represented as vectors of binary numbers with each feature in the vector represented by a '1' or '0'. Each feature vector (corresponding to one application) is labelled with its class. The feature vectors are loaded into the deep learning model during the

training phase. After training, the model can then be used to predict the class (clean or botnet) of an unknown application using its extracted feature vector. Figure 1 gives a high level overview of the overall botnet detection system.

Table 2. Examples of features extracted for the deep learning models.

Feature Name	Type
TelephonyManager.*getDeviceId	API
TelephonyManager.*getSubscriberId	API
abortBroadcast	API
SEND_SMS	Permission
DELETE_PACKAGES	Permission
PHONE_STATE	Permission
SMS_RECIVED	Permission
Ljava.net.InetSocketAddress	API
READ_SMS	Permission
Android.intent.action.BOOT_COMPLETED	Intent
io.File.*delete(	API
Chown	Command
Chmod	Command
Mount	Command
.apk	Extra File
.zip	Extra File
.dex	Extra File
.jar	Extra file
CAMERA	Permission
ACCESS_FINE_LOCATION	Permission
INSTALL_PACKAGES	Permission
android.intent.action.BATTERY_LOW	Intent
.so	Extra File
android.intent.action.POWER_CONNECTED	Intent
System.*LoadLibrary	API

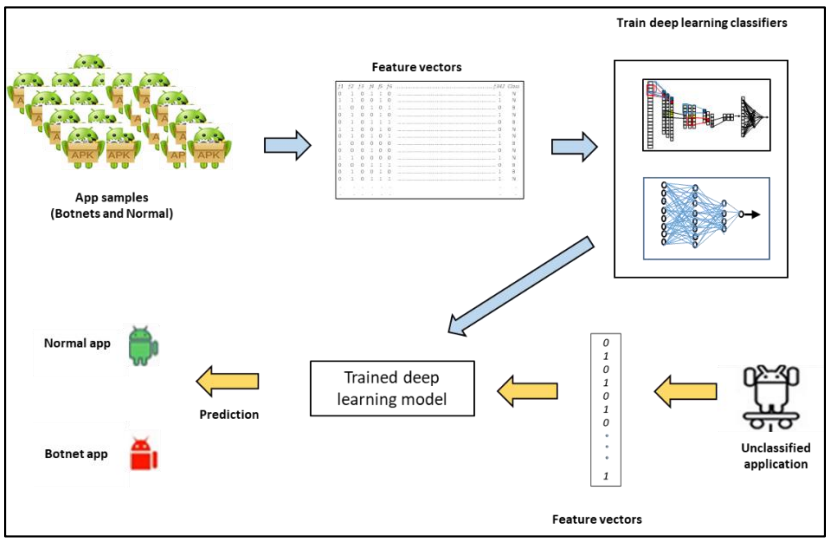

Figure 1. Overview of the deep learning-based detection system for Android botnets.

4. Deep Learning Techniques Applied to Android Botnet Detection

4.1. Convolutional Neural Networks

A CNN is a feedforward neural network whereby the information moves only in the forward direction from the input node, through the hidden nodes to the output nodes with no loops or cycles. Such (feedforward) networks are primarily used for pattern recognition. CNN generally works well for identifying simple patterns in data which will then be used

to form more complex patterns in the higher or deeper layers. CNNs typically consists of convolutional layers and pooling layers. The role of the convolutional layer is to detect local conjunctions of features from the previous layer, while the role of the pooling layer is to merge semantically similar features into one [25]. CNNs combine concepts such as shared weights, local receptive fields and spatial subsampling [26]. They take advantage of many parallel and cascaded convolutional filters to solve high dimensional non-convex problems such as regression, image classification, semantic segmentation, object detection, etc. Due to weight sharing in each layer and by processing limited dimensions, a CNN requires fewer parameters than a traditional neural network and is much easier to train.

Datasets that possess a one-dimensional structure can be processed using a one-dimensional convolutional neural network (1D CNN). A 1D CNN is quite effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall feature set, and where the location of the feature within the segment is not important. The use of 1D CNN can be commonly found in Natural Language Processing (NLP) applications. Similarly, 1D CNN is applicable in problems where vectorized data are used to represent the characteristics of the items whose state or category is being predicted (e.g., an Android application). The 1D CNN could be used to extract potentially more discriminative feature representations that describe any existing patterns or relationships within segments of the vectors characterizing each entity in the dataset. These new features are then fed into a classifier (e.g., LSTM, GRU or a fully connected layer) which will in turn process the derived features to produce a set of outputs that will contribute towards a final classification decision. Hence, CNNs can be employed as feature extraction layers for a given classifier which then eliminates the need to apply separate feature ranking and selection outside of the deep learning model.

Figure 2 depicts a 1D CNN model made up of two convolutional layers and two max pooling layers. The output of last pooling layer is flattened and connected to a dense (fully connected) layer of N units. The N-unit dense layer is in then connected to a final output layer containing a single neuron with a sigmoid activation function which is given by: $S = \frac{1}{1+e^{-x}}$.

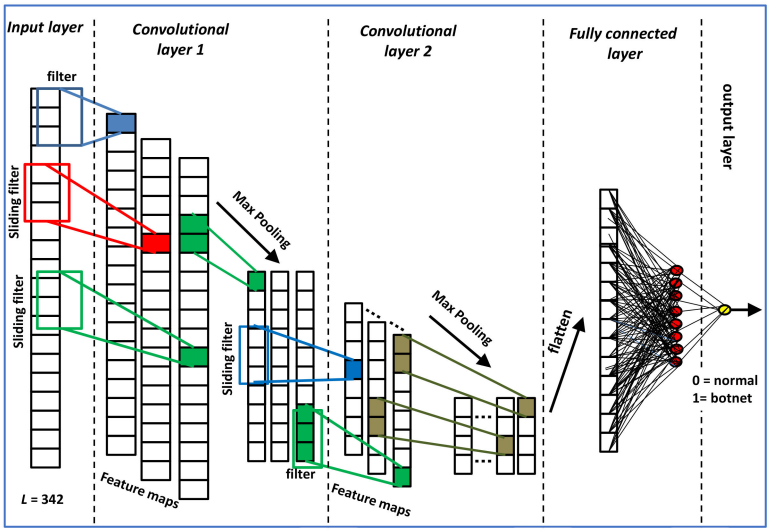

Figure 2. 1D CNN model with 2 convolutional and max pooling layers feeding a dense (fully connected) layer. The model is designed for botnet detection by classifying Android applications into 'normal' or 'botnet'.

The output layer performs the final classification into one of two classes, i.e., 'botnet' or 'normal'. The convolutional layers utilize Rectified Linear Units (ReLU) with activation function given by: $f(x) = max(0, x)$. ReLU helps to mitigate vanishing and exploding gradient issues [27].

4.2. Long-Short Term Memory

LSTM [28,29] is a type of recurrent neural network (RNN) which, unlike feedforward networks, utilizes feedback and is able to 'memorize' parts of the input and use them in making predictions. RNNs are designed to handle sequential data and thus have found popular application in areas such as speech recognition and machine translation. Different from traditional artificial neural networks that fully connect all nodes, or CNN that explore nodes from local to global layer by layer, RNNs use state neurons to explore the relationship in context. Traditional RNNs have a known problem of vanishing gradients which hinders their ability to have long term memory and thus can only make predictions based on the most recent information in the sequence. LSTM solves the vanishing gradient problem and is therefore able to process longer sequences (long term memory). LSTM is a recurrent neural network that can understand contextual information from a sequence of features. It has the ability to add or remove information from the hidden state vector with the aid of a gate function, thereby retaining important information in the hidden layer vectors.

As shown in Figure 3a, LSTM consists of three gate functions. These include: the forget gate, the input gate, and the output gate. The forget gate is used to control the amount of information in C_{t-1} is retained in the process of computing C_t and it (the forget vector) can be expressed as:

$$f_t = \sigma\left(U^f x_t + W^f h_{t-1} + b_f\right) \tag{1}$$

where U^f, W^f, and b_f constitute the parameters of the forget gate and x_t is the input vector in step t, while h_{t-1} is the hidden state vector in the previous step $t-1$. The input gate determines how much information of x_t is added to C_t and can be expressed as:

$$f_t = \sigma\left(U^f x_t + W^f h_{t-1} + b_f\right) \tag{2}$$

where U^i, W^i, and b_i are the parameters of the input gate and hence C_t can be calculated by relying on the forget gate vector f_t as well as the input gate vector i_t as follows:

$$f_t = \sigma\left(U^f x_t + W^f h_{t-1} + b_f\right) \tag{3}$$

where $\check{C}_t = tanh(U^c x_t + W^c h_{t-1} + b_C)$ denotes the information represented in the hidden layer vector. Note that $*$ denotes the Hadamard (element-wise) product. The output gate controls the output in C_t, and we have:

$$o_t = \sigma(U^o x_t + W^o h_{t-1} + b_o)\ ,\ h_t = o_t * tanh(C_t) \tag{4}$$

where U^o, W^o, and b_o are the parameters of the output gate and C_t is the internal state in step t.

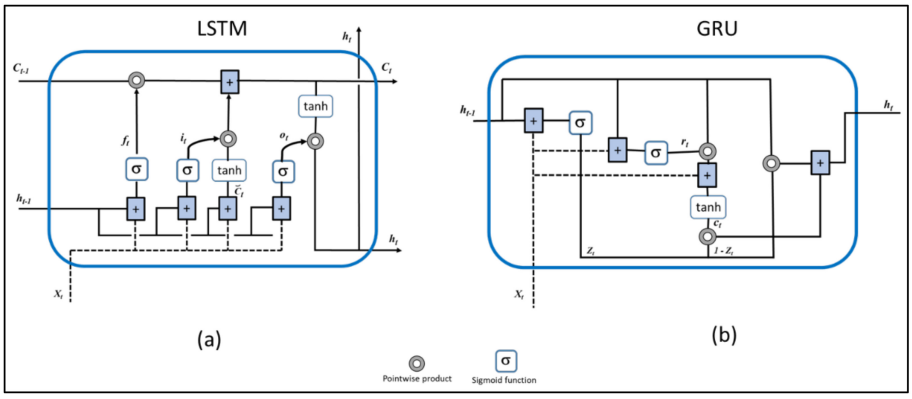

Figure 3. Recurrent neural networks. (**a**) LSTM; (**b**) GRU.

4.3. Gated Recurrent Units

A GRU [30] is also a kind of RNN model and a variant of LSTM. However, unlike LSTM which has three gates, GRU has only two gates, i.e., reset gate and update gate, as shown in Figure 3b. This makes GRU less complicated and therefore faster to train than LSTM. The gates are two vectors that decide which information should be passed to the output. The update gate enables the model to determine how much of the past information needs to be passed along to the future. The update gate Z_t is calculated for step t using the formula given by:

$$z_t = \sigma(U^z x_t + W^z h_{t-1}) \tag{5}$$

where U^z, W^z are the parameters (weights) of the update gate and h_{t-1} holds information for the previous $t-1$ units. The reset gate is used to decide how much of the past information to forget which can be calculated using:

$$r_t = \sigma(U^r x_t + W^r h_{t-1}) \tag{6}$$

where U^r, W^r are the parameters (weights) of the reset gate and h_{t-1} holds information for the previous $t-1$ units. The current memory content will use the reset gate to store the relevant information from the past as follows:

$$c_t = tanh(U^c x_t + r_t * W^c h_{t-1}) \tag{7}$$

where U^c, W^c are the parameters (weights). Note that $*$ denotes the Hadamard (element-wise) product. At the last step, the vector h_t which holds the information for the current unit and passes it down the network will be calculated by:

$$h_t = z_t * h_{t-1} + (1 - z_t) * c_t \tag{8}$$

A GRU network obtains a long spatial (or temporal) sequence with lower computational complexity compared to traditional encoder-decoder architecture. With its gating mechanisms, GRU can overcome the vanishing gradient problem and is therefore capable of processing longer sequences than standard RNN. Both GRU and LSTM can be applied to sequences of spatial features to determine the extent of dependencies or establish context between features that are located several places apart.

4.4. Dense Neural Networks

The DNN model is a regular deeply connected neural network with several layers. In a DNN model, each neuron in a layer receives an input from all the neurons present in the previous layer. The layers are known as the dense layers and constitute the hidden layers of the network. Such neural networks are also known as Multilayer perceptron (MLP). It is composed of an input layer, an output layer that makes a decision or prediction about the input, and an arbitrary number of hidden layers in between. The model is often trained on a set of input–output pairs and learns to model the correlation (or dependencies) between those inputs and outputs. The basic unit (a perceptron) of the model produces a single output based on several real-valued inputs by forming a linear combination using its input weights. The output is typically passed through a non-linear activation function ϑ:

$$y = \vartheta\left(\sum_{1}^{n} w_i x_i + b\right) = \vartheta\left(\boldsymbol{W}^T \boldsymbol{X} + b\right) \tag{9}$$

where $\boldsymbol{W}$ denotes the vector of weights, $\boldsymbol{X}$ is the vector of inputs, b is the bias and ϑ is the non-linear activation function.

The sigmoid or the hyperbolic tangent functions were the non-linear activation functions typically used in the past due to their ability to map complex relationships within data. However, these two non-linear activation functions do not perform well in networks with many layers due to the vanishing gradient problem. Nowadays, rectified linear

activation function ReL (and its variants) is the preferred function used in training dense neural networks. Hence, the neurons in a network employing ReL activation are known as ReLU (Rectified Linear activated Units). ReL is a piecewise linear function given by: $f(x) = max(0, x)$. It will output the input directly if positive, and will output a zero if negative. ReLU overcomes the vanishing gradient problem and enable models to learn faster and perform better. Hence, it is used as the default activation function when developing the DNN and the CNN networks. In our study, we have experimented with different numbers of hidden layers for the DNN, and numbers of units per layer and recorded the performance of each configuration.

4.5. Hybrid Models

In this paper, we refer to hybrid models as those combining different deep learning techniques to leverage the unique capabilities of each of the techniques. For example in CNN-LSTM or CNN-GRU depicted in Figure 4, the model will utilize CNN to extract local n-gram features (where n is set by the length of the filters). The CNN's max pooling layer downsamples the output to reduce the dimensionality, which also contributes to the reduction in overfitting. The LSTM or GRU layers are then used to capture long-range dependencies that may be present within the features encoded by the CNN layers. The vectors output by the LSTM-GRU layer with the context and dependencies information will then be transmitted to dense layers for further processing before the final classification by the sigmoid activated output layer consisting of a single unit.

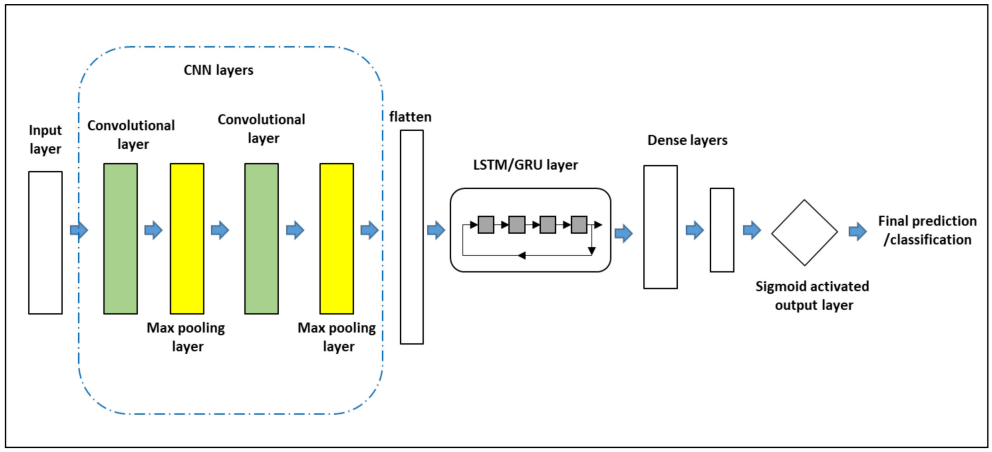

Figure 4. Overview of the CNN-LSTM and CNN-GRU hybrid model architecture.

5. Methodology and Experiments

In this section, we further detail our approach and outline the experiments undertaken to evaluate the deep learning models implemented in this paper. The models were developed in Python using the Keras library with TensorFlow backend. Other libraries utilized include Scikit Learn, Seaborn, Pandas, and Numpy. The experiments were performed on a Ubuntu Linux 16.04 64-bit Machine with 8GB RAM.

5.1. Problem Definition

Let $A = \{a_1, a_2, \ldots\ a_n\}$ be a set of applications where each a_i is represented by a vector containing the values of n features (where $n = 342$). Let $a = \{f_1, f_2, f_3, \ldots\ f_n, cl\}$ where $cl \in \{botnet, normal\}$ is the class label assigned to the app. Thus, A can be used to train a model to learn the behaviors of botnet and normal apps, respectively. The goal of a trained model is then to classify a given unlabeled app $A_{unknown} = \{f_1, f_2, f_3, \ldots\ f_n, ?\}$ by assigning a label cl, where $cl \in \{botnet, normal\}$.

5.2. Dataset Used for the Investigation

As mentioned earlier, the ISCX Android botnet dataset from [9] was utilized for the experiments in this paper. This dataset contains 1929 botnet apps and has been employed in previous works including [6–8,10–13,22]. Table 3 shows the distribution of samples within the 14 different botnet families present in the dataset. To complement the ISCX dataset, we obtained 4873 clean from Google Play store. These apps were cross-checked for maliciousness using Virus Total (https://www.virustotal.com (accessed on 20 December 2020)). Thus, a total of 6802 apps were used in our experiments.

Table 3. Botnet dataset composition.

Botnet Family	Number of Samples
Anserverbot	244
Bmaster	6
Droiddream	363
Geinimi	264
Misosms	100
Nickyspy	199
Notcompatible	76
Pjapps	244
Pletor	85
Rootsmart	28
Sandroid	44
Tigerbot	96
Wroba	100
Zitmo	80
Total	1929

5.3. Experiments to Evaluate the Deep Learning Techniques on the Android Dataset

In order to investigate the performance of the deep learning models, we performed several experiments with different configurations of the models to enable us observe the optimum performance that is possible with each model architecture. The models are designed to exploit the capabilities of the constituent neural network types as discussed in Section 4. The following metrics are used in measuring the performance of the models: Accuracy, precision, recall, and F1-score. Given TP as true positives, FP as false positives, FN as false negatives, and TN as true negatives (all with respect to the botnet class), the metrics are defined as follows (taking the botnet class as positive):

- Accuracy: the ratio between correctly predicted outcomes and the sum of all predictions expressed as: $\frac{TP+TN}{TP+TN+FP+FN}$
- Precision: All true positives divided by all positive predictions, i.e., was the model right when it predicted positive? Expressed as: $\frac{TP}{TP+FP}$
- Recall: All true positives divided by all actual positives. That is, how many positives did the model identify out of all possible positives? Expressed as: $\frac{TP}{TP+FN}$
- F1-score: This is the weighted average of precision and recall, given by: $\frac{2\times Recall\times Precision}{Recall+Precision}$

All the results of the experiments are from 10-fold cross validation where the dataset is divided into 10 equal parts with 10% of the dataset held out for testing, while the models are trained from the remaining 90%. This is repeated until all of the 10 parts have been used for testing. The average of all 10 results is then taken to produce the final result. Additionally, during the training of all the deep learning models (for each fold), 10% of the training set was used for validation.

6. Results and Discussions

This section will present the results of investigating CNN-GRU, CNN-LSTM, CNN, and DNN models. Subsequently, a comparative performance evaluation of the models and how they measure against traditional machine learning models will be discussed.

Finally, we will examine how these models have performed compared to results reported in previous works on Android botnet detection.

6.1. CNN-GRU Model Results

Here, we present the results obtained from CNN-GRU model where the configurations of both the CNN layer and the GRU layer were varied. A summary of the results is presented in Table 4. In the top half of the table, the configuration had 1 convolutional layer and 1 max pooling layer in the CNN part. These models are named as CNN-1-layer-GRU-X where X stands for the number of hidden units in the GRU layer. The Convolutional layer receives input vector of dimension 342 from the input layer, and it consists of 32 filters each of size = 4. The max pooling layer has its parameter set to 2, which means it would reduce the output of the convolutional layer by half. The outputs from the max pooling layer are concatenated into a flat vector before sending to the GRU layer. As depicted in Figure 4, the output from the CNN-GRU layers are forwarded to 2 dense layers. The first dense layer had 128 units, while the second one had 64 units. The 64-unit layer is finally connected to a sigmoid-activated single-unit output layer where the final classification decision into 'clean' or 'botnet' is made. The model can be summarized in the following sequence:

Input [342] -> Conv [32 filters, Size=4] -> max pooling -> flatten -> GRU[X] -> Dense [128, ReLU] -> Dense[64, ReLU]-> Dense[1, Sigmoid] where X is the number of GRU hidden units taken as 5, 10, 25, and 50, respectively.

Table 4. Results from the CNN-GRU models of various configurations using the architecture depicted in Figure 4.

	Accuracy	Precision	Recall	F1-Score	Number of Parameters
CNN-1-layer-GRU-5	0.988	0.982	0.976	0.979	90,442
CNN-1-layer-GRU-10	**0.989**	**0.986**	**0.975**	**0.980**	**172,457**
CNN-1-layer-GRU-25	0.988	0.981	0.978	0.979	419,402
CNN-1-layer-GRU-50	**0.989**	**0.980**	**0.980**	**0.980**	**833,977**
CNN-2-layer-GRU-5	0.988	0.985	0.974	0.979	53,528
CNN-2-layer-GRU-10	0.989	0.986	0.974	0.980	93,993
CNN-2-layer-GRU-25	0.989	0.982	0.978	0.980	217,098
CNN-2-layer-GRU-50	**0.991**	**0.988**	**0.979**	**0.984**	**425,273**

The results of the bottom half of Table 4 are from the same CNN-GRU architecture described above, but with the CNN part having 2 convolutional layers and 2 max pooling layers. The model can be summarized in the following sequence:

Input [342] -> Conv [32 filters, Size=4] -> max pooling -> Conv [32 filters, Size=4] -> max pooling -> flatten -> GRU[X]->Dense [128, ReLU] -> Dense[64, ReLU]->Dense[1, Sigmoid] where X is the number of GRU hidden units taken as 5, 10, 25, and 50, respectively.

Note that a dropout = 0.25 is incorporated between each of the layers in the models to reduce overfitting.

From Table 4, we can see that the model with the 1-layer CNN had higher overall accuracy of 98.9% when the number of GRU hidden units were set at 10 or at 50. The corresponding F1-score were also the highest at 0.980. The recall of the GRU-50 model was 0.980 compared to that of the GRU-10 model, which was 0.975. This means that the GRU-50 model was better at detecting botnet apps than the GRU-10 model in the top half of Table 4. Note that the GRU-5 model from the 1-layer CNN batch (top-half) which had 5 hidden units actually did perform well also by obtaining an overall accuracy of 98.8%, with an F1-score of 0.979 and a botnet detection rate (recall) of 97.6%. It had the least numbers of parameters to train, i.e., 90,422.

From the bottom half of Table 4, the 2-layer CNN models with the best overall accuracy was the one with 50 units in the GRU layer (i.e., CNN-2-GRU-50). It obtained 99.1%

accuracy, and the best F1 score of 0.984. The recall (botnet detection rate) was 97.9% while the precision was 98.8%, the highest in all of the CNN-GRU models. From these set of results, we can conclude the following:

- The best overall performance for the CNN-GRU models was from the model with 2 convolutional layers and 50 hidden units in the GRU layer.
- Very good results can be obtained by CNN-GRU model with only 1 convolutional layer and few hidden units (5) in the GRU layer. The accuracy observed was 98.8%, and the F1 score was 0.979. The lower the number of hidden units, the faster it is to train the model.

6.2. CNN-LSTM Model Results

This section presents the results of the CNN-LSTM models with different configurations in both the CNN layer and the GRU layer. The results are presented in Table 5. Similar to the results of CNN-GRU in Table 4, the top half is for the models with 1 convolutional layer and 1 max pooling layer in the CNN part, while the bottom half (of Table 5) shows the results of the models having 2 convolutional layers and 2 max pooling layers in the CNN part. The models are named with the convention CNN-1-layer-LSTM-X in the top half, or CNN-2-layer-LSTM-X in the bottom half, where X stands for the number of hidden units in the LSTM layer. As depicted in Figure 4, the output from the CNN-LSTM layers are forwarded to 2 dense layers. The first dense layer had 128 units, while the second one had 64 units. The 64-unit layer is finally connected to a single unit sigmoid activated output layer where the final classification decision into 'clean' or 'botnet' is made. The model can be summarized in the following sequence:

Input [342] -> Conv [32 filters, Size=4] -> max pooling -> flatten -> LSTM[X] -> Dense [128, ReLU] -> Dense [64, ReLU]-> Dense[1, Sigmoid]

where X is the number of LSTM hidden units taken as 5, 10, 25, and 50, respectively.

Table 5. Results from the CNN-LSTM models of various configurations using the architecture depicted in Figure 4.

	Accuracy	Precision	Recall	F1-Score	Number of Parameters
CNN-1-layer-LSTM-5	0.990	0.987	0.977	0.982	117,497
CNN-1-layer-LSTM-10	**0.990**	**0.986**	**0.980**	**0.983**	**226,617**
CNN-1-layer-LSTM-25	0.990	0.985	0.979	0.982	555,117
CNN-1-layer-LSTM-50	**0.990**	**0.983**	**0.983**	**0.983**	**1,106,777**
CNN-2-layer-LSTM-5	0.989	0.986	0.973	0.980	66,553
CNN-2-layer-LSTM-10	0.989	0.986	0.974	0.980	120,633
CNN-2-layer-LSTM-25	0.990	0.984	0.979	0.981	284,073
CNN-2-layer-LSTM-50	0.989	0.984	0.976	0.980	560,473

For the bottom half of Table 5, the models can be summarized in the following sequence:

Input [342] -> Conv [32 filters, Size=4] -> max pooling -> Conv [32 filters, Size=4] -> max pooling -> flatten -> LSTM[X]->Dense [128, ReLU] -> Dense [64, ReLU]->Dense [1, Sigmoid]

where X is the number of LSTM hidden units taken as 5, 10, 25, and 50, respectively.

Note that a dropout = 0.25 is incorporated between each of the layers in the models to reduce overfitting.

Table 5 (top half), it can be seen that the all the CNN-LSTM models with only 1 layer in the CNN part had overall accuracy of 99%. The models with 10 and 50 hidden units, respectively, in the LSTM layer obtained identical F1-score of 0.983, compared to the ones with 5 and 25, respectively, which had F1-score of 0.982. The best recall (or botnet detection rate) of 98.3% was recorded with the LSTM-50 model. However, having more

than 1.1 million parameters, the LSTM-50 model will be longer to train than the LSTM-10 model which has only 226,617 parameters.

From the bottom half of Table 5, the 2-layer CNN models with the best overall accuracy was the one with 25 units in the LSTM layer (i.e., CNN-LSTM-25). It obtained 99% accuracy, and the best F1 score of 0.981. The recall (botnet detection rate) was 97.9% while the precision was 98.4%. From the results of Table 5, we can conclude the following:

- The best overall performance for the CNN-LSTM models was from the model with 1 convolutional layer and 25 hidden units in the LSTM layer.
- Very good results can be obtained by CNN-LSTM model with only 1 convolutional layer and few hidden units (5) in the LSTM layer. This is evident from the results of the CNN-1-layer-LSTM-5 where the accuracy observed was 99%, and the F1 score was 0.982, precision was 98.7%, and recall (botnet detection rate) was 97.7%.
- Comparing Tables 4 and 5, the results of CNN-LSTM were generally better than those of CNN-GRU even though a CNN-GRU model obtained the highest F1-score of 0.984 with an overall accuracy of 99.1%.

6.3. CNN Model Results

In this section we discuss the results of the CNN model which is summarized in Table 6. The CNN model consists of 2 convolutional layers and 2 max pooling layers. The resulting vectors are 'flattened' and fed into a dense layer containing 8 units. The model's sequence can be summarized as follows:

Input [342] -> Conv [32 filters, Size=4] -> max pooling -> Conv [32 filters, Size=4] -> max pooling-> flatten -> Dense [8, ReLU] -> Dense [1, Sigmoid]

Table 6. Results from a 2-layer CNN model obtained by varying the number of filters, with length of filters = 4 in both convolutional layers.

	Accuracy	Precision	Recall	F1-Score	Number of Parameters
CNN-2-layer-4-filters	0.986	0.978	0.974	0.976	2,777
CNN-2-layer-8-filters	0.988	0.980	0.977	0.978	5,657
CNN-2-layer-16-filters	0.988	0.980	0.976	0.978	11,801
CNN-2-layer-32-filters	**0.989**	**0.983**	**0.978**	**0.981**	**25,625**
CNN-2-layer-64-filters	0.987	0.980	0.975	0.977	59,419

In our preliminary study presented in [24], this particular configuration of the model has been determined to yield the best performance on the same features extracted from the same app dataset used for the other models presented in this paper. More extensive performance evaluation of the CNN model has been presented in [24], where the effect of varying the other parameters, such as filter length, number of layers, and max pooling parameter has been investigated.

As shown in Table 6, the CNN model with 32 filters yielded the best results with 98.9% overall accuracy, precision = 0.983, recall = 0.978, and F1-score = 0.981. When compared to the results in Tables 4 and 5, it can be observed that most of the CNN-LSTM configurations and some of the CNN-GRU configurations achieved higher results than the CNN-only model. This suggests that the LSTM and GRU were able to capture some dependencies amongst the features thus improving the performance of the model.

6.4. DNN Model Results

The results obtained from the Dense Neural Network model is presented in this section. The naming convention used to describe the models is DNN-Y-layer-N as shown in Table 7, where *Y* stands for the number of hidden layers and *N* is the number of units in the layer. For example, the sequence of the DNN-2-layer-200 model can be summarized as follows:

Input [342] -> Dense [200, ReLU] -> Dense [200, ReLU] -> Dense [1, Sigmoid]

Table 7. Results from the DNN model with various numbers of layers and units per layer.

	Accuracy	Precision	Recall	F1-Score	Number of Parameters
DNN-1-layer-100	0.990	0.984	0.982	0.983	34,401
DNN-2-layer-100	**0.991**	**0.990**	**0.979**	**0.984**	**44,501**
DNN-3-layer-100	**0.991**	**0.988**	**0.980**	**0.984**	**54,601**
DNN-1-layer-200	0.990	0.982	0.982	0.982	68,801
DNN-2-layer-200	0.990	0.981	0.983	0.982	109,001
DNN-3-layer-200	0.989	0.982	0.980	0.981	149,201
DNN-1-layer-300	0.990	0.985	0.981	0.983	103,201
DNN-2-layer-300	0.989	0.984	0.978	0.981	193,501
DNN-3-layer-300	0.989	0.979	0.984	0.981	283,801

Note that a dropout = 0.25 is incorporated between each of the layers in the models to reduce overfitting. Additionally, in all of the DNN models and the previous models in Sections 6.1–6.3, the optimization algorithm used was 'Adam' and 'Binary cross entropy' was used for the loss function. Furthermore, all the models were configured to automatically terminate the training after the validation loss is observed to have not changed for a specific number of K training epochs, where K was set to 20.

From Table 7, it can be observed that the DNN models with a single hidden layer did not result in the best outcomes. Likewise, in most cases, using 3 hidden layers as observed with the DNN-3-layer-200 and DNN-3-layer-300 also did not give the best outcomes. The best performance was obtained from the model with 2 hidden layers and 100 units in each layer, where the overall accuracy is 99.1% and F1-score = 0.984. The model with 3 hidden layers and 100 units in each layer also gave identical results. This shows that increasing the number of units in each layer is unlikely to improve the performance any further.

6.5. Best Deep Learning Results vs. Classical Non-Deep Learning Classifiers

In Table 8, we juxtapose the best results from our investigation of the deep learning classifiers with the results from the classical machine learning techniques. The DNN and the CNN-GRU models achieved the best results as depicted in the table. The highest accuracy achieved by both models were 99.1% which also corresponds to the highest F1-score of 0.984. These results are followed closely by the CNN-LSTM model which achieved 99% overall accuracy and F1-score of 0.983. Next, was the CNN-only model with 98.9% accuracy and F1-score of 0.981. All of these models outperformed the classical machine learning classifiers where the best two were SVM and Random Forest. SVM had 98.7% overall accuracy and F1-score of 0.976, while Random Forest obtained 98.5% accuracy and F1-score of 0.973. These results suggest that with the static based features extracted for detecting Android botnets, the deep learning models will perform beyond the limits of the classical machine learning classifiers.

In the table, the GRU-only model is shown as having the least accuracy results compared to all the other models. This GRU model consisted of 200 hidden units and obtained an overall accuracy of 82.9%. Similarly, with LSTM-only models, overall accuracies below 75% were observed (results not shown in the table). This confirmed our initial expectation that pattern recognition (e.g., with convolutional layers or dense layers) was more important for the type of feature vectors used in the study, rather than context or dependencies. However, the results of Sections 6.1 and 6.2 for the hybrid models suggests that a combination of methods that can capture both characteristics is promising.

Table 8. A summary of the best results of each technique compared to popular non-deep learning classifiers.

	Accuracy	Precision	Recall	F1-Score	Number of Parameters
DNN-2-layer-100	0.991	0.990	0.979	0.984	44,501
DNN-3-layer-100	0.991	0.988	0.980	0.984	54,601
CNN-2-layer-GRU-50	0.991	0.988	0.979	0.984	425,273
CNN-1-layer-LSTM-10	0.990	0.986	0.980	0.983	226,617
CNN-1-layer-LSTM-50	0.990	0.983	0.983	0.983	1,106,777
CNN-2-layer-32-filters	0.989	0.983	0.978	0.981	25,625
SVM	0.987	0.980	0.973	0.976	-
Random Forest	0.985	0.982	0.965	0.973	-
Simple Logistic	0.984	0.983	0.963	0.973	-
Decision Tree (J48)	0.981	0.974	0.958	0.966	-
Naïve Bayes	0.872	0.728	0.874	0.795	-
Bayes Net	0.867	0.736	0.832	0.781	-
GRU-200	0.829	0.677	0.766	0.718	122,001

6.6. Model Training Times

When training the deep learning models, the number of epochs has a major influence on the overall model training time. In our experiments we utilized a stopping criterion based on minimum validation loss rather than specifying a fixed number of training epochs. For this reason, the number of training epochs varied between the different configurations of a given model. Hence, the longest CNN-GRU model to train was the CNN-2-layer-GRU-25 which took 145 s, and the testing time was 0.482 s. Whereas the shortest CNN-GRU model to train was the CNN-1-layer-GRU-10 model which took 84.4 s with a testing time of 0.399 s. The longest CNN-LSTM model to train was the CNN-2-layer-LSTM-5 which took 141 s with a testing time of 0.468 s. The shortest CNN-LSTM model to train was the CNN-1-layer-LSTM-25 model which took 83.6 s with a testing time of 0.419 s. Compared to the other models, the DNN was the fastest to train with training times ranging from 10 to 26 s and an average testing time of 0.15 s.

6.7. Comparison with Previous Works

The results obtained in our study improves the performance beyond the reported results in previous papers that also used the ISCX botnet dataset in their work. This can be observed in Table 9. The second column shows the numbers of the botnet and benign samples used in each of the referenced paper. Note that in some papers, some of the metrics were not reported. Even though the complete datasets and techniques used were different in each of the previous works, Table 9 shows that the models developed in this paper achieved state-of-the-art results with the ISCX botnet dataset compared to the others.

Table 9. Performance comparisons with previous works that utilize ISCX botnets samples.

Paper	Botnets/Benign	ACC (%)	Prec.	Rec.	F1-Score
Hojjatinia et al. [6]	1800/3650	97.2	0.955	0.960	0.957
Tansettanakorn et al. [11]	1926/150	-	0.972	0.969	-
Anwar et al. [10]	1400/1400	95.1	0.970	0.827	-
Abdullah et al. [13]	1505/850	-	0.931	0.946	-
Alqatawna and Faris [12]	1635/1635	97.3	0.987	0.957	-
Yerima and Alzaylaee [7]	1929/4873	98.9	0.983	0.978	0.981
This paper	**1929/4873**	**99.1**	**0.990**	**0.979**	**0.984**

7. Conclusions and Future Work

In this paper, we presented an extensive evaluation of various deep learning techniques for Android botnet detection using 342 static features consisting of 5 different types. The deep learning models investigated include: CNN, DNN, GRU, LSTM, as well as CNN-LSTM and CNN-GRU. The experiments were undertaken using 6802 apps consisting of 1929 botnet apps from the ISCX botnet dataset which has been utilized in several previous works. The outcomes of our experiments showed that with optimum configuration, the deep learning models performed quite well yielding high accuracies that were beyond the limits of the classical machine learning classifiers. DNN showed the best overall performance, while CNN-GRU and CNN-LSTM showed promising results that were much better than GRU-only or LSTM-only models. In future work, we plan to further investigate the performance of the deep learning models for botnet detection using alternative static and dynamic features. Another possible direction is to explore alternative network architectures such as those consisting of parallel rather than purely sequential integrations of the deep learning model components.

Author Contributions: Conceptualization, S.Y.Y.; Data curation, M.K.A.; Investigation, S.Y.Y., M.K.A. and A.S.; Methodology, S.Y.Y.; Resources, V.P.; Supervision, V.P.; Validation, M.K.A., A.S. and V.P.; Writing—original draft, S.Y.Y.; Writing—review & editing, M.K.A. and V.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are publicly available in FigShare at https://doi.org/10.6084/m9.figshare.14079581 (accessed on 10 December 2020).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. McAfee Mobile Threat Report Q1. 2020. Available online: https://www.mcafee.com/en-us/consumer-support/2020-mobile-threat-report.html (accessed on 5 December 2020).
2. Yerima, S.Y.; Khan, S. Longitudinal Performance Analysis of Machine Learning based Android Malware Detectors. In Proceedings of the 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Oxford, UK, 3–4 June 2019.
3. Grill, B.B.; Ruthven, M.; Zhao, X. "Detecting and Eliminating Chamois, a Fraud Botnet on Android" Android Developers Blog. March 2017. Available online: https://android-developers.googleblog.com/2017/03/detecting-and-eliminating-chamois-fraud.html (accessed on 10 December 2020).
4. Chris Brook "Google Eliminates Android Adfraud Botnet Chamois" Threat Post. March 2017. Available online: https://threatpost.com/google-eliminates-android-adfraud-botnet-chamois/124311/ (accessed on 10 December 2020).
5. Fahmida, Y. Rashid "Chamois: The Big Botnet You Didn't Hear about" April 2019 Decipher, by Duo Security. Available online: https://duo.com/decipher/chamois-the-big-botnet-you-didnt-hear-about (accessed on 10 December 2020).
6. Hojjatinia, S.; Hamzenejadi, S.; Mohseni, H. Android Botnet Detection using Convolutional Neural Networks. In Proceedings of the 2020 28th Iranian Conference on Electrical Engineering (ICEE), Tabriz, Iran, 4–6 August 2020.
7. Yerima, S.Y.; Alzaylaee, M.K. Mobile Botnet Detection: A Deep Learning Approach Using Convolutional Neural Networks. In Proceedings of the 2020 International Conference on Cyber Situational Awareness (Cyber SA 2020), Dublin, Ireland, 15–19 June 2020.
8. Kadir, A.F.A.; Stakhanova, N.; Ghorbani, A.A. Android botnets: What urls are telling us. In Proceedings of the International Conference on Network and System Security, New York, NY, USA, 3–5 November 2015; Springer: New York, NY, USA, 2015; pp. 78–91.
9. ISCX Android Botnet Dataset. Available online: https://www.unb.ca/cic/dataset/android-botnet.html (accessed on 23 December 2020).
10. Anwar, S.; Zain, J.M.; Inayat, Z.; Haq, R.U.; Karim, A.; Jabir, A.N. A static approach towards mobile botnet detection. In Proceedings of the 2016 3rd International Conference on Electronic Design (ICED), Phuket, Thailand, 11–12 August 2016; pp. 563–567.
11. Tansettanakorn, C.; Thongprasit, S.; Thamkongka, S.; Visoottiviseth, V. ABIS: A prototype of android botnet identification system. In Proceedings of the 2016 Fifth ICT International Student Project Conference (ICT-ISPC), Nakhon Pathom, Thailand, 27–28 May 2016; pp. 1–5.
12. Alqatawna, J.F.; Faris, H. Toward a Detection Framework for Android Botnet. In Proceedings of the 2017 International Conference on New Trends in Computing Sciences (ICTCS), Amman, Jordan, 11–13 October 2017; pp. 197–202.

13. Abdullah, Z.; Saudi, M.M.; Anuar, N.B. ABC: Android botnet classification using feature selection and classification algorithms. *Adv. Sci. Lett.* **2017**, *23*, 4717–4720. [CrossRef]
14. Karim, A.; Rosli, S.; Syed, S. DeDroid: A Mobile Botnet Detection Approach Based on Static Analysis. In Proceedings of the 7th International Symposium on UbiCom Frontiers—Innovative Research, Systems and Technologies, Beijing, China, 10–14 August 2015. [CrossRef]
15. The Drebin Dataset. Available online: https://www.sec.cs.tu-bs.de/~{}danarp/drebin/index.html (accessed on 22 December 2020).
16. Jadhav, S.; Dutia, S.; Calangutkar, K.; Oh, T.; Kim, Y.H.; Kim, J.N. Cloud-based android botnet malware detection system. In Proceedings of the 2015 17th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Korea, 1–3 July 2015; pp. 347–352.
17. Yerima, S.Y.; Alzaylaee, M.K.; Sezer, S. Machine learning-based dynamic analysis of Android apps with improved code coverage. *EURASIP J. Inf. Secur.* **2019**. [CrossRef]
18. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. Improving dynamic analysis of android apps using hybrid test input generation. In Proceedings of the 2017 International Conference on Cyber Security and Protection Of Digital Services (Cyber Security 2017), London, UK, 19–20 June 2017.
19. Bernardeschia, C.; Mercaldo, F.; Nardonec, V.; Santoned, A. Exploiting Model Checking for Mobile Botnet Detection. *Procedia Comput. Sci.* **2019**, *159*, 963–972. [CrossRef]
20. Clarke, E.; Emerson, E.; Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* **1986**, *8*, 244–263. [CrossRef]
21. Andersen, J.R.; Andersen, N.; Enevoldsen, S.; Hansen, M.M.; Larsen, K.G.; Olesen, S.R.; Srba, J.; Wortmann, J.K. CAAL: Concurrency workbench, aalborg edition. In Proceedings of the Theoretical Aspects of Computing—ICTAC 2015—12th International Colloquium, Cali, Colombia, 29–31 October 2015; Springer: Cham, Switzerland, 2015; pp. 573–582.
22. Alothman, B.; Rattadilok, P. Android botnet detection: An integrated source code mining approach. In Proceedings of the 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, UK, 11–14 December 2017; pp. 111–115.
23. Alzahrani, A.J.; Ghorbani, A.A. Real-time signature-based detection approach for sms botnet. In Proceedings of the 2015 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, Turkey, 21–23 July 2015; pp. 157–164.
24. Girei, D.A.; Shah, M.A.; Shahid, M.B. An enhanced botnet detection technique for mobile devices using log analysis. In Proceedings of the 2016 22nd International Conference on Automation and Computing (ICAC), Colchester, UK, 7–8 September 2016; pp. 450–455.
25. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
26. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
27. Glorot, X.; Bordes, A.; Bengio, Y. Deep sparse rectifier neural networks. In Proceedings of the 4th International Conference on Artificial Intelligence and Statistics, Ft. Lauderdale, FL, USA, 11–13 April 2011; pp. 315–323.
28. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
29. Graves, A. *Long Short-Term Memory*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 37–45.
30. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv* **2014**, arXiv:1412.3555.