Received October 6, 2019, accepted October 31, 2019, date of publication November 6, 2019, date of current version November 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2951751

Identifying Malicious Software Using Deep Residual Long-Short Term Memory

AZIZ ALOTAIBI, (Member, IEEE)
College of Computers and Information Technology, Taif University, Taif 21974, Saudi Arabia
e-mail: azotaibi@tu.edu.sa

ABSTRACT The use of smartphone applications based on the Android OS platform is rapidly growing among smartphone users. However, malicious apps for Android are being developed to perform attacks, such as destroying operating systems, stealing confidential data, gathering personal information, and hijacking or encrypting sensitive data. Several malware detection systems based on machine learning have been developed and deployed to extract a variety of features to prevent such attacks. However, new efficient detection methods are needed to extract complex features and hidden structures from malicious apps to detect malware. This paper proposes a novel framework, namely, MalResLSTM, based on deep residual long short-term memory to identify and classify malware variants. The framework imposes a set of constraints on the deep learning architecture to capture dependencies between the extracted features from the Android package kit (APK) file. These feature sets are mapped to a vector space to process the input sequence using a sequence model based on the residual LSTM network. To evaluate the performance of the proposed framework, several experiments are conducted on the Drebin dataset, which contains 129,013 applications. The results demonstrate that MalResLSTM can achieve a 99.32% detection accuracy and outperforms previous algorithms. An extensive experimental analysis was conducted, which included machine-learning-based algorithms and a variety of deep learning-based algorithms, to evaluate the efficiency and robustness of our proposed framework.

INDEX TERMS Malware Detection, android malware, malware analysis, malware classification, static analysis, deep learning-based algorithms.

I. INTRODUCTION

Malicious software (malware) is an unwanted program that intends to harm the victim's workstations, mobile devices, servers, and gateways [1]. Common malware programs, such as viruses, worms, trojans, horses, spyware, ransomware, scareware, bots, and rootkits, exploit the system vulnerabilities to infect the target. Cybercriminals attack the systems of individuals and organizations with various objectives, such as destroying operating systems, damaging computers or networks, stealing confidential data, gathering personal information, and hijacking or encrypting sensitive data. To protect users' computers, mobiles, servers and gateways, malware detection tools should be developed and deployed to prevent attacks. Many malware tools have been developed by anti-virus companies for scanning vulnerabilities, detecting intrusions, and monitoring and preventing malware. These defense products, such as Semantic, Kaspersky, and McAfee, are constantly updated to maintain their effectiveness against malware attacks. The Android OS platform is rapidly growing and has dominated more than 85% of the smartphone market [2], [3]; as a result, Android malware development is increasing and has reached more than 26 million programs [4]. Thus, an effective detection system is needed for investigating a variety of malware scenarios [5]. Malware issues can be detected via three feature extraction analysis methods [6] [7]: static, dynamic, and hybrid analysis methods. First, static analysis collects the features from the Android app without executing the APK files. The malicious code is hidden in the APK file using a packer tool such as UPX, NsPack, VMprotect, or Andromeda [8]. The APK file includes features such as Windows API calls, the network address, byte n-grams, strings, opcodes, and control flow graphs. However, this method is not effective against dynamic code loading or code obfuscation. Second, dynamic analysis is used with more complicated and complex malware, which

The associate editor coordinating the review of this manuscript and approving it for publication was Kaitai Liang.

TABLE 1. Comparison of static, dynamic and hybrid analysis methods.

METHOD	Advantages	Disadvantages
Static	- Simple and fast to run - Low time complexity	- Obfuscation - Dynamic code loading - Zero-day attack
Dynamic	- Zero-day attack detection - Detects unknown malwares	- High time complexity - Difficulty detecting multipath malware
Hybrid	- High classification accuracy - More efficient - Analyzes polymorphic malware	- High time complexity - Low scalability

cannot be identified via static analysis. Dynamic analysis observes the behavior of the malware during the program execution process. Dynamic detection methods utilize information flow tracking through the system calls, network traffic and user interactions. However, challenges are encountered with this method regarding the duration of the execution, the observation process, the number of programs that are assessed by the malware detection system, and technique that is used to avoid Android virtualization [9]. Third, hybrid analysis is a combination of static and dynamic analysis. Typically, the API call information is extracted through the static analysis, followed by observation of the file execution behavior [10], which typically achieves higher accuracy detection compared to the static and dynamic methods. However, hybrid analysis has a high time complexity and requires a framework for utilizing both static and dynamic features. Table 1 compares the static, dynamic and hybrid analysis methods.

After extracting the features, malware analysis algorithms are applied to detect and classify the malware. Recently, machine learning (ML) algorithms have been intensively used in malware analysis, especially for unknown malware, with a variety of techniques, such as support vector machine (SVM) [11], random forest (RF) [12] [13], neural network (NN) [14], and convolutional neural network (CNN) [15], among others [2], [31]. The learning algorithms are categorized into three domains: supervised, semi-supervised, and unsupervised. Supervised learning utilizes previously labeled samples to enable the model to predict a new sample. Unsupervised learning utilizes unlabeled samples to extract the underlying structure from the hidden features. Semi-supervised learning is a combination of supervised and supervised learning in which a small number of samples are labeled and most of the samples are unlabeled. In this paper, we apply supervised learning since the training samples are labeled. The main contributions of this work are summarized as follows:

1- Introduce a novel malware detection framework that is based on residual deep learning;
2- Map the extracted feature sets to a vector space to capture dependencies and the hidden structure;
3- Propose a specialized residual long short-term memory (LSTM) architecture and constraints for classifying the application file;
4- The proposed approach outperforms all other considered methods on the Drebin dataset;
5- The efficiency and robustness of the proposed framework are evaluated via several experimental analyses.

The remainder of the paper is organized as follows: Section II discusses related work in detail. Section III describes how to construct the feature vector space and presents the overall architecture of proposed Android malware detection framework, namely, MalResLSTM, in detail. The efficiency and robustness of the proposed detection framework and the experimental results of the proposed framework are analyzed and discussed in Section IV. Section V summarizes the proposed framework and discusses future work.

II. RELATED WORK

The use of the Android OS and its applications, such as mobile payment systems, has been rapidly increasing. The Android OS platform has more vulnerabilities because it uses open-source software. This has attracted many attackers to write complex malicious programs for infecting users' devices. Effective malware detection systems are needed for countering such attacks. In this section, previous malware detection analysis algorithms are discussed and analyzed according to their feature extraction and classification/clustering algorithm types. Malware detection methods can be categorized into four categories: signature-based methods, behavior-based methods, intelligence-based methods, and cloud-based methods.

A. SIGNATURE-BASED METHODS

Predefined pattern matching is one of the most popular malware detection methods, which is based on known malwares that are stored in a large database. Methods that use predefined pattern matching are called signature-based methods. Signature-based methods are widely applied by famous antivirus companies, such as Comodo, McAfee, Symantec, and Kingsoft. When a new malware program is released, antimalware software products must create new signatures and update their databases regularly. Signature-based detection utilizes a unique pattern and a sequence of bytes that is extracted from the malware file to identify the malware [16]. Tang et al. [17] proposed a network-based signature generation (NSG) method for constructing a tree structure, namely, PloyTree, for defending against polymorphic worms. The PloyTree algorithm consists of two components for classifying the variant of worms by updating the signature tree construction: a signature tree generator and a signature selector. Fraley et al. [18] detected polymorphic malware by utilizing topological feature extraction with data mining techniques. Alam et al. [19] introduced an Android malware detector system, namely, DroidNative, which analyzes a control-flow pattern to detect malwares in android native code and other variants. DroidAnalytics [20] is an Android malware analytic system that is based on a multi-level signature algorithm for retrieving and associating malicious logic at the opcode level. DroidAnalytics can detect zero-day repackaged malware. Signature-based methods are easy to apply and can quickly identify the malware variants [21]; however, they can be fooled via encryption, obfuscation and polymorphism techniques [22].

B. BEHAVIOR-BASED METHODS

Behavior-based methods monitor the program's behaviors and activities to detect malware in the executable file. The malware detection is based on both the object's code and structure and uses dynamic analysis. The two main features that have been utilized in recent behavior malware studies are system calls and network traffic [7]. In [23], Miao et al. introduced a bilayer behavior abstraction technique that utilizes API sequences, in which behavior features are based on semantic layers. Sun et al. [3] implemented a detection system that monitors the behavior of an Android application based on kernel-level monitoring. Furthermore, Saracino et al. [24] proposed a novel behavior-based android malware detection system, namely, MADAM. MADAM monitors, extracts and analyzes features at four levels, namely, the application, kernel, package, and user levels, to detect misbehavior in system calls. In [25], Jang et al. designed a hybrid malware detection system that is based on behavior profiling, namely, Andro-Profiler, which utilizes the system logs and calls to detect malware. In addition, Monet [26] is a malware detection framework that utilizes the generation of the runtime graph and the system call to form the runtime behavior signature. Monet consists of a client and a backend server module for detecting variant malware. The runtime behavior signature is analyzed by the backend server to detect and defend against malware attacks such as the transformation attack. Behavior-based methods can detect unknown and polymorphic malware; however, the time complexity for detecting variant malware at runtime is regarded as an overhead.

C. INTELLIGENCE-BASED METHODS

Intelligence-based methods use artificial intelligence algorithms to cluster and classify malware apps. With the advancements in deep intelligent methods, intelligence-based methods have been used in most recent malware detection studies, where both data mining and machine learning are used as intelligent models to identify and classify sophisticated malware apps. Yuan et al. [27] implemented an online deep-learning-based malware detector, namely, DroidDetector, for detecting malware applications. The DroidDetector model is based on two phases: deep belief networks (DBNs) followed by stacked restricted Boltzmann machines (RBMs) with a deep neural network, which is available online for automated detection. LI et al. [28] developed a significant permission identification (SigPID) malware detection system that utilized the decision tree model, which is a supervised machine learning technique, to detect and classify malware families. In [29], Bat-Erdene et al. proposed a novel technique for identifying and detecting packing algorithms via symbolic aggregate approximation (SAX) using naive Bayes and SVM classifiers. LI et al. [14] proposed a two-level model and a prediction model that are based on a machine learning framework for identifying and detecting the domain generation algorithm (DGA). First, the DGA domains are distinguished from the normal domain. Then, a clustering method is used to identify the algorithms that form the DGA. Second, a prediction model that is based on the hidden Markov model (HMM) is used to detect the incoming domain features. Karbab et al. [30] proposed the MalDoze framework, which is based on a raw sequence of app's API calls and uses deep learning techniques. In [31], Vinayakumar et al. proposed a deep-learning-based framework, namely, ScaleMalNet, for identifying and classifying zero-day malware. In addition, the authors evaluated a variety of machine-learning-based and deep-learning-based algorithms that utilize static analysis, dynamic and image processing methods for malware detection systems. Moreover, Kang et al. [32] utilized opcodes and API function names to classify malicious files into families using a word2vec model and LSTM networks. In [33], Xiao et al proposed a detection method based on two LSTM models utilizing semantic information to classify the system call sequence. Vinayakumar *et al.* [34] proposed a stacked LSTM network to detect all individual behaviors of malware application.

D. CLOUD-BASED METHODS

The use of cloud-based methods has been rapidly increasing due to the huge number of released malwares and to the low cost; however, cloud-based methods have shortcomings in terms of time consumption and data vulnerability. The detection agents that provide the security services are on the cloud servers. Users can utilize the services by uploading any types of files on the cloud servers and receive the detection result. In [35], Abdelsalam et al. proposed a malware detection system that is based on cloud infrastructures for VMs and uses a deep learning approach. Sun et al. [36] introduced a cloud-based malware detection method, namely, CloudEyes, that relies on a scanning agent on the cloud. Furthermore, Zonouz et al. [37] designed a cloud-based service, namely, Secloud, that emulates a version of the smartphone device inside the cloud and synchronizes the devices inputs and the network connections to detect malware. Mirza et al. [38] introduced a cloud-based scalable service, namely, CloudIntell, that is hosted on Amazon web services (AWS) for detecting malware via machine learning techniques.

Previous malware detection systems have attempted to address the malware detection issues through either signature, behavior, cloud or intelligent based methods. However,

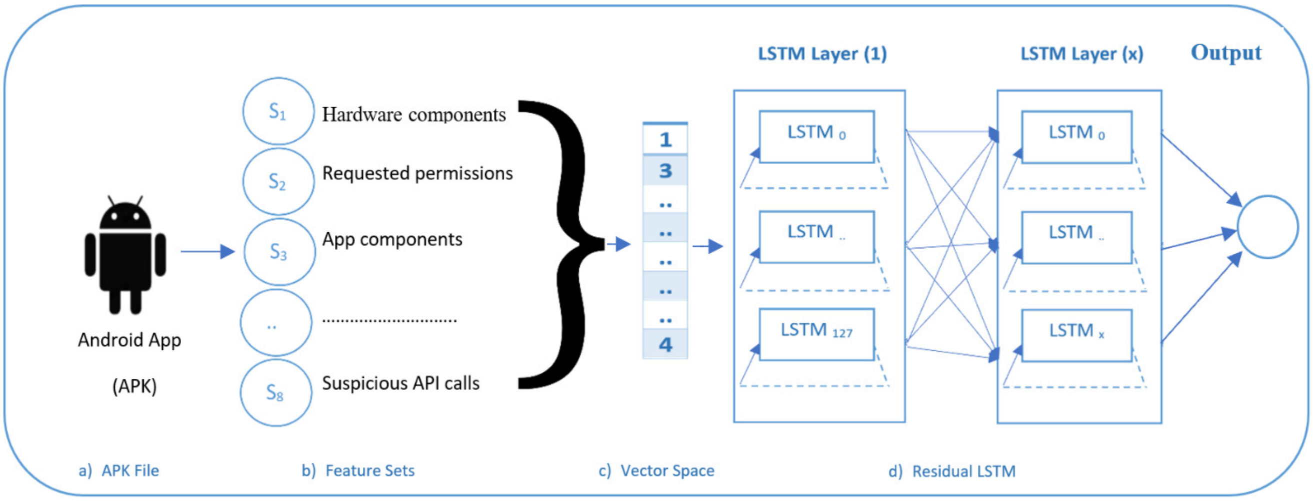

FIGURE 1. **The overview architecture of the proposed deep residual long short-term memory framework.**

TABLE 2. **Feature extraction set.**

Feature Set	Explanation
S_1 Hardware components	A request to access the hardware components
S_2 Requested permissions	App requests permission to access the device resources
S_3 App components	App contains suspicious components
S_4 Filtered intents	App enables the components to register and to receive messages
S_5 Restricted API calls	App requests access after requesting a permission
S_6 Used permissions	App used the granted permission for access
S_7 Suspicious API calls	App requests access through suspicious API calls
S_8 Network addresses	App establishes a connection to retrieve the host data

intelligent methods have been lately proven to outperform other methods in detecting complex malware variants. Recently, deep learning algorithms have been explored and utilized to analyze the feature extraction in order to detect malware variants. However, capturing the dependencies between extracted features has not been fully explored, thus, this paper has explored the dependent relationship between API calls to detect malware variants as explained next.

III. PROPOSED FRAMEWORK

In this section, the MalResLSTM framework architecture and its components are described, as illustrated in Fig. 1. The framework utilizes eight sets of features, which are listed in Table 2: hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses. These sets of features are embedded into a feature vector. Then, the feature vector is fed to our specialized deep learning algorithms for classification/detection. The proposed framework consists of four major phases: static extraction of raw data, feature extraction sets, vector space embedding, and a residual long short-term memory model process. These phases are detailed in the next subsections.

A. STATIC EXTRACTION OF RAW DATA

The first step in extracting features from an Android package kit (APK) file is to access the manifest file and to disassemble the dex file. Both the manifest and the disassembled files are decoded using the Android asset packaging tool to retrieve sets of features as strings, as explained below.

B. FEATURE EXTRACTION SETS

String features are extracted from the APK file and are divided into eight feature types [39] as shown in Fig. 1.:

(S_1) *Hardware components*: This type is based on a request to access hardware components such as the camera and GPS. The requests to access a specified hardware component is declared in the manifest file and has a security implication.

(S_2)*Requested permissions*: This request is granted by the user during installation and configuration, which allows the attacker to access the device resources, such as SEND SMS permission.

(S_3) *App components*: An Android application consists of four components: activities, content providers, broadcast receivers, and services. The name of each component can be utilized in the feature set to identify well-known malware.

(S_4) *Filtered intents*: The communications between components and the applications that enable the components to register and to receive messages are called intents, such as BOOT COMPLETED.

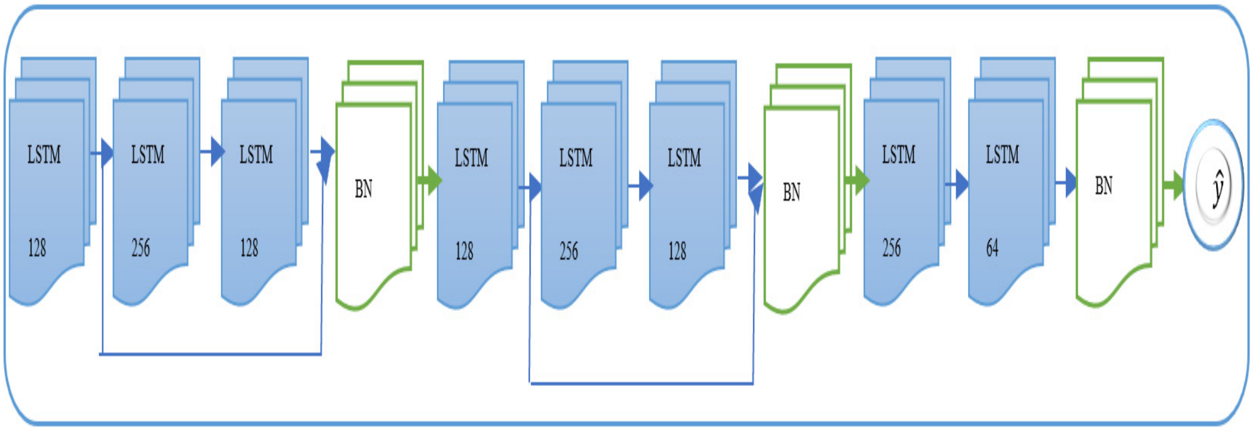

FIGURE 2. Deep residual long short-term memory network.

(S_5) *Restricted API calls*: The Android system has identified a list of restricted API calls that can be accessed only after requesting a permission.

(S_6) *Used permissions*: Matching of API calls with permissions is utilized to monitor the behavior of the application.

(S_7) *Suspicious API calls*: Only a set of API calls are allowed to access sensitive data and specified resources. These API calls sometimes lead to suspicious behavior, such as getDeviceId() and Cipher.getInstance().

(S_8) *Network addresses*: Establishing a connection to retrieve data from the smartphone device is the last step in the proposed framework. The obtained data are the IP address, hostname, and URL, which are found in the disassembled file.

C. EMBEDDING IN A VECTOR SPACE

Extracted features from the previous feature set process are utilized to identify malicious software, for example, sendSMS() is used to access sensitive data in set S7 and SEND SMS() is used to obtain permission in set S1. However, deep learning utilizes numerical computations. Therefore, extracted feature sets are mapped to a vector space to capture the dependencies between these features and to facilitate the detection processes. All eight feature sets are joined to define a vector as shown in Fig. 1.

$$SV := S_1 \cup S_2 \cup S_3 \cup \ldots . S_8 \tag{1}$$

Each feature S that is extracted from an application (A) is set to value (1) in the vector space φ(a) and the other dimension is set to value (0).

For a set of applications A:

$$\varphi : A \rightarrow \{0, 1\}^{|S|}, \quad \varphi(a) \rightarrow I(a, s)_{s \in SV} \tag{2}$$

where $I(a, s)$ is defined as follows:

$$I(a, s) = \begin{cases} 1, & \text{App (a) has feature } (\mathrm{S_x}) \\ 0, & \text{App (a) has no feature } (\mathrm{S_x}) \end{cases} \tag{3}$$

The vector space of φ(a) for a malicious application is:

$$\varphi(a) \rightarrow \left\{\begin{array}{c} 0\\1\\0\\1\\1\\1\\1\\1\\1\\1 \end{array}\right\} \begin{array}{l} \left.\begin{array}{l}\cdots\cdots\cdots\cdots\cdots\cdots\\ \textit{hardware.telephony}\\ \cdots\cdots\cdots\cdots\cdots\cdots\end{array}\right\} S_1 \\ \left.\begin{array}{l}\cdots\cdots\cdots\cdots\cdots\cdots\\ \textit{SendSMS}\\ \cdots\cdots\cdots\cdots\cdots\cdots\end{array}\right\} S_2 \\ \left.\begin{array}{l}\cdots\cdots\cdots\cdots\cdots\cdots\\ \cdots\cdots\cdots\cdots\cdots\cdots\\ \cdots\cdots\cdots\cdots\cdots\cdots\\ \cdots\cdots\cdots\cdots\cdots\cdots\end{array}\right\} S_x \end{array} \tag{4}$$

According to Equation (4), the feature vector of φ(a) is:

$$\varphi(a) = [1, 3, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, 4] \tag{5}$$

The feature vector is used as an input to the residual LSTM network, as explained in the next subsection.

D. RESIDUAL LONG SHORT-TERM MEMORY MODEL

Fig. 2 illustrates the architecture of the residual long short-term memory network. The long short-term memory (LSTM) network is a powerful deep learning method that was proposed by Hochreiter et al. [40] for overcoming the vanishing gradient and exploding gradient in recurrent neural networks (RNNs).

In addition, LSTM can learn long-term dependencies through memory gates based on time series. The entire architecture consists of thirteen layers, including the input and output layers.

The shape of the input layer is a tensor with three parameters (samples, timesteps, and feature) and the batch size at each iteration is 400 samples. The second layer is an LSTM layer with 128 units, which accepts the input tensor as an input. The third layer has 256 LSTM units and the fourth layer has 128 LSTM units. The fifth layer is a residual layer that

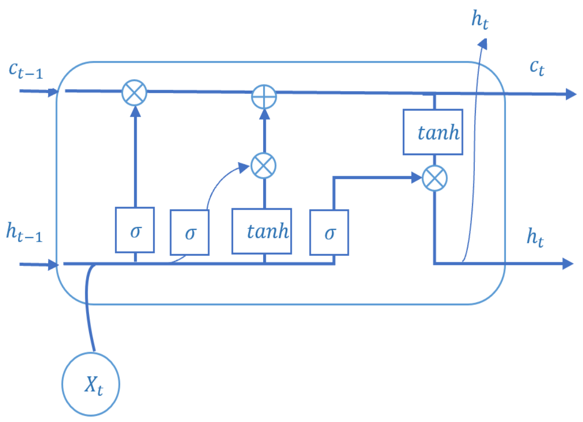

FIGURE 3. Basic structure of a long short-term memory cell.

```
Algorithm 1 Batch Normalization [42]
  Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1.......m}\}$;
         Parameters to be learned: $\gamma, \beta$
  Output: $\{\mathcal{Y}_i = \mathbf{BN}_{\gamma,\beta}(x_i)\}$
  1: $\mu\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ Mini-batch mean
  2: $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu\mathcal{B})^2$ Mini-batch variance
  3: $\hat{x}_i \leftarrow \frac{x_i - \mu\mathcal{B}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ Normalize
  4: $\mathcal{Y}_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathbf{BN}_{\gamma,\beta}(x_i)$ Scale and Shift
```

accepts inputs from the 2nd layer and the 4th layer and passes them through a batch normalization layer. The 6th layer has 128 LSTM units and the 7th layer has 256 LSTM units. The 8th layer is an LSTM layer with 128 units. The 9th layer is a batch normalization layer that accepts residual connections from the 6th and 8th layers. The 10th layer is an LSTM layer with 256 units and the 11th layer has 64 LSTM units, which only outputs one h by setting *return sequence* to *false*. The 12th layer is a batch normalization layer that accepts the output of the 11th layer as an input. The output layer uses a sigmoid activation function as a classifier. The basic structure of the long short-term memory cell is explained next.

E. LSTM

The proposed LSTM framework consists of memory blocks, each of which consists of three gates: an input gate i_t, a forget gate f_t, and an output gate o_t. Fig. 3 illustrates the architecture of the long short-term memory cell. These gates and states are defined as follows:

Input gate:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{6}$$

Forget gate:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{7}$$

Output gate:

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{8}$$

Input transform:

$$c\_im_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{c_{im}}) \tag{9}$$

State update:

$$i_t = f_i \otimes c_{t-1} + i_t \otimes c\_im_t) \tag{10}$$

$$h_t = o_t \otimes \tanh(c_t) \tag{11}$$

where σ is the sigmoid activation function; W and b denote the weights and bias, respectively; x_t denotes the input feature vector at timestep t; $\otimes$ is the element-wise product; and c_t and h_t denote the memory cell and the hidden layer output, respectively.

F. RESIDUAL CONNECTION

Residual networks were introduced in [41] for avoiding vanishing gradients and enhancing the generalization performance. Residual connection is implemented in the proposed MalResLSTM framework by utilizing the skip connections, namely, the shortcut connections, where the identity map and the output are added to the output of the stacked layer, as illustrated in Fig. 2. In the framework, both the output of the hidden states $\hat{h}$ and the stacked hidden layer must have the same LSTM units. Throughout the architecture, each residual layer has 128 LSTM units. Implementing the deep residual connection increases the detection accuracy by reducing the convergence rate of the training error. Each residual connection is followed by a normalization layer, as explained next.

G. BATCH NORMALIZATION

Batch normalization was introduced by Loffe and Szegedy[42] for stabilizing learning and for overcoming the poor initialization problem. Batch normalization is applied to a mini-batch and acts as a regularizer to prevent overfitting. Batch normalization is implemented by setting the mean to zero and the variance to one, as specified in Algorithm 1. However, batch normalization is not applied to all layers in the framework to avoid model instability; it is applied only after the residual connection layers and after the last LSTM layer. Empirically, a mini-batch size of 400 samples was observed to result in a higher detection rate when the mini-batch was normalized in the normalization layer.

H. CONSTRAINTS

While training the framework, it is observed that imposing constraints on the model increased the learning rate and the stability. First, the fully connected layer is eliminated from the top of the residual LSTM. Second, the Adam optimizer [43] is used as an optimization algorithm instead of stochastic gradient descent. Third, the last layer uses a sigmoid function to classify the input application as benign or malicious.

TABLE 3. Performance comparison with the proposed system (from [44]).

Method	Dataset		Classification	Accuracy (%)
	Malware	Benign		
T. Kim [44]	13,075 (V.S.)	19,747	MNN-s	98%
	1,209 (M.P.)	1,300		99%
Z. Yuan [27]	1,760 (M.P.)	20,000	DBN	96.8%
W. Yu [45]	92 (M.P.)	96	DNN/RNN	90%
N. McLaughlin [46]	9,902 (M.P.)	9,268	CNN	87%
H. Fereidooni [47]	18,677 (M.P.)	11,187	XGboost/Adaboost/RF SVM/K-NN/LR/NB/DT/DNN	97%
M. Zhang [48]	2,200 (M.P.)	13,500	NB	93%
D. Arp [39]	5,560 (M.P.)	123,453	SVM	93.9%
S.Y. Yerima [49]	1,000 (M.P.)	1,000	Bayesian-based classifier	92%
S.Y. Yerima [50]	2,925 (Mc.)	3,938	RF/LR/NB/DT	97.5%
Our proposed framework	5,560 (M.P.)	123,453	**Deep ResLSTM**	**99.32%**

**Most of the samples that are used in these systems are from the Malgenome project.
** Abbreviation: M.P. – Malgenome project, V.S. – VirusShare, Mc. – McAfee, MNN-Multimodal Neural Network, DBN-Deep Belief Network, DNN-Deep Neural Network, RNN-Recurrent Neural Network, CNN-Convolution Neural Network, RF-Random Forest, SVM-Support vector machine, K-NN – K-Nearest Neighbor, DT-Decision Tree, LR- Logistic Regression, NB-Naïve Bayes,

IV. DISCUSSION AND PERFORMANCE EVALUATION

A. DATASET

The Drebin dataset [39], [51] is used to evaluate the performance of our method experimentally. Drebin contains 131,611 applications: 96,150 applications were collected from Google play, 19,545 from Chinese markets, 2,810 from Russian markets and 13,106 from other sources. In addition, all samples in the Android malware genome project [52] are included. To identify the malicious and benign applications in the dataset, ten anti-virus scanners are used to inspect the samples: Kaspersky, ClamAV, Sophos, McAfee, Panda, AntiVir, AVG, ESET, Bit-Defender, and F-Secure. An Android app is regarded as malicious if it is detected by at least two of the anti-virus scanners. The final Drebin dataset contains 123,453 benign applications and 5560 malware samples. This dataset is one of the largest malware datasets for evaluating the malware detection system.

B. DISCUSSION AND ANALYSIS

In this subsection, the efficiency of the proposed detection framework is discussed and analyzed in depth based on three aspects: the utilization of captured dependencies, the set of constraints on the architecture, and the framework stability. The proposed framework was compared with previous detection systems on samples from the Malgenome project and it outperformed all the systems, according to Table 3.

First, most previous works do not utilize the dependencies between API calls, which effects the performances of their systems. Deep neural networks (DNNs) are more suitable for identifying the relationships between input and output variables, which cannot explicitly capture dependencies between the input sequences. In contrast, recurrent neural networks can recognize and capture dependencies between sequences. When both deep and recurrent neural networks are tested on the same dataset, DNN achieves a 99.66% detection rate and MalResLSTM 99.32%. Hence, recurrent model can capture the dependencies between API call requests.

Second, imposing constraints, such as eliminating the fully connected layer from the top of the residual LSTM, and using the Adam optimizer enable the framework to learn quickly and increase its stability. The LSTM framework with full connected achieves a 98.60% detection rate, whereas the framework without the fully connected achieves a 99.32% detection rate. Third, both batch normalization and residual connection enable the framework to avoid overfitting and stabilize the learning process. Thus, the LSTM network without residual connection and without batch normalization achieves a 99.68% detection rate, whereas that with both residual connection and batch normalization achieves a 99.32% detection rate.

C. PERFORMANCE EVALUATION

In this subsection, the detection performance of the proposed framework is investigated and evaluated using the Drebin dataset. The dataset is split into a training set (66%) and a test set (33%). The training set is used to train the framework, whereas the test set is used to evaluate the performance. First, the proposed framework is compared with other proposed systems in terms of overall accuracy, where the

TABLE 4. Performance comparison on the Drebin dataset.

Algorithm	Accuracy
Logistic Regression	96%
Support Vector Machine	98%
Random Forest	98.78%
Neural Network	98.66%
Deep Residual LSTM	**99.32%**

TABLE 5. Performance comparison using evaluation metrics.

Symbol	Precision	Recall	F1-score
Logistic Regression	0.63	0.20	0.30
Support Vector Machine	0.81	0.70	0.75
Random Forest	0.98	0.91	0.84
Neural Network	0.87	0.80	0.83
Recurrent Neural Network	0.85	0.77	0.81
Gate Recurrent Unit	0.85	0.79	0.82
Long Short-Term Memory	0.85	0.80	0.82
Residual RNN	0.79	0.79	0.79
Residual GRU	0.86	0.78	0.82
Residual LSTM	**0.92**	**0.91**	**0.92**

Malgenome project is used in most of them, as presented in Table 3. The accuracy of our proposed framework was higher than those of other proposed systems, including most of the deep learning algorithms, such convolutional neural network (CNN) and recurrent neural network (RNN). In addition, the MalResLSTM framework outperforms the other systems on the original dataset that was proposed by D. Arb [39], according to Table 3. The proposed framework and various machine learning algorithms are implemented using Windows 10 with an Intel i7-8750H CPU and 32 GB RAM. Scikit-learning, Tensorflow and the Keras library are utilized as tools to implement the framework. To evaluate the performance of the MalResLSTM framework, various machine learning algorithms are implemented, such as logistic regression, support vector machine, random forest and neural network, on the same dataset as reported in Table 4. For comparison, the following algorithms are tested on the Drebin dataset: logistic regression (LR), support vector machine (SVM), random forest (RF), neural network (NN), recurrent neural network (RNN), gated recurrent unit (GRU), and long short-term memory (LSTM).

The following evaluation metrics are computed to measure the performance of the classifier:

$$\textit{Accuracy} = \frac{TN + TP}{TP + FP + TN + FN} \tag{12}$$

$$\textit{Precision} = \frac{TP}{TP + FP} \tag{13}$$

TABLE 6. Performance comparison of two sequence models on the Drebin dataset.

Algorithm	Accuracy
Deep Residual RNN	99.22%
Deep Residual GRU	99.50%
Deep Residual LSTM	**99.32%**

$$\textit{Recall} = \frac{TP}{TP + FN} \tag{14}$$

$$F1 - \textit{score} = \frac{2 * TP}{2 * TP + FP + FN} \tag{15}$$

where TN and TP denote the numbers of true-negative and true-positive results, respectively, and FN and FP denote the numbers of false-negative and false-positive results, respectively. According to the comparison results of these algorithms using the evaluation metrics, the proposed framework achieves the best result, as presented in Table 5. The precision score of the proposed framework is 0.92 which is the highest score that was obtained.

According to Table 6, deep residual LSTM outperforms sequence models RNN and GRU due to its superior ability to extract the dependence between the features and the model stability.

V. CONCLUSION AND FUTURE WORK

Malware programs have affected many mobile devices by utilizing system vulnerabilities, such as viruses, worms, trojans, horses, spyware, ransomware, scareware, bots, and rootkits, to infect the target. Malware programs are used to achieve various objectives, such as destroying operating systems, damaging computers or networks, stealing confidential data, gathering personal information, and hijacking or encrypting sensitive data. To address these issues and to detect malware programs, this paper proposes a novel framework that is based on a deep learning algorithm for identifying and classifying malware variants. First, Feature sets are extracted from APK **file** and then mapped to a feature vector. Then, the feature vector is utilized **to** extract the hidden features and structural dependencies using a sequence model that is based on the deep residual long short-term memory network, namely, MalResLSTM. A set of constraints is applied to the deep learning architecture to capture dependencies between the extracted features from the Android package kit (APK). The architecture consists of thirteen layers, including the input and output layers. Applying residual connection to the LSTM layers reduces the convergence rate of the training error and improves the detection performance. An extensive experimental analysis was conducted to investigate the efficiency of the proposed framework . The performance of the MalResLSTM framework is evaluated using the Drebin dataset, on which it achieves 99.32% detection accuracy and outperforms previously proposed algorithms. In the future, bidirectional LSTM and the attention mechanism will be further explored and investigated. Furthermore, an adversarial example should be used to evaluate the vulnerability of the framework.

REFERENCES

[1] S. D. Gantz and D. R. Philpott, *FISMA and the Risk Management Framework: The New Practice of Federal Cyber Security*. Newnes, 2012.

[2] N. Chavan, F. Di Troia, and M. Stamp, "A comparative analysis of Android malware," 2019, *arXiv:1904.00735*. [Online]. Available: https://arxiv.org/abs/1904.00735

[3] S. Sun, X. Fu, H. Ruan, X. Du, B. Luo, and M. Guizani, "Real-time behavior analysis and identification for Android application," *IEEE Access*, vol. 6, pp. 38041–38051, 2018.

[4] J. Clement. (2018). *Development of Android Malware Worldwide 2011–2018*. [Online]. Available: https://www.statista.com/statistics/680705/global-android-malware-volume/

[5] E. B. Karbab and M. Debbabi, "MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports," *Digit. Invest.*, vol. 28, pp. S77–S87, Apr. 2019.

[6] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Comput. Secur.*, vol. 81, pp. 123–147, Mar. 2018.

[7] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digit. Invest.*, vol. 13, pp. 22–37, Jun. 2015.

[8] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Comput. Sci. Rev.*, vol. 32, pp. 1–23, May 2019.

[9] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Gener. Comput. Syst.*, vol. 97, pp. 887–909, Aug. 2019.

[10] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," *Comput. Secur.*, vol. 83, pp. 208–233, Jun. 2019.

[11] L. Sun, Z. Li, Q. Yan, W. Srisa-An, and Y. Pan, "SigPID: Significant permission identification for Android malware detection," in *Proc. 11th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2016, pp. 1–8.

[12] A. Bhattacharya and R. T. Goswami, "DMDAM: Data mining based detection of Android malware," in *Proc. 1st Int. Conf. Intell. Comput. Commun.* Singapore: Springer, 2017, pp. 187–194.

[13] L. D. Coronado-De-Alba, A. Rodríguez-Mota, and P. J. Escamilla-Ambrosio, "Feature selection and ensemble of classifiers for Android malware detection," in *Proc. 8th IEEE Latin-Amer. Conf. Commun. (LATINCOM)*, Nov. 2016, pp. 1–6.

[14] Y. Li, K. Xiong, T. Chin, and C. Hu, "A machine learning framework for domain generation algorithm-based malware detection," *IEEE Access*, vol. 7, pp. 32765–32782, 2019.

[15] B. Chen, Z. Ren, C. Yu, I. Hussain, and J. Liu, "Adversarial examples for CNN-based malware detectors," *IEEE Access*, vol. 7, pp. 54360–54371, 2019.

[16] Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, 2017, Art. no. 41.

[17] Y. Tang, B. Xiao, and X. Lu, "Signature tree generation for polymorphic worms," *IEEE Trans. Comput.*, vol. 60, no. 4, pp. 565–579, Apr. 2011.

[18] J. B. Fraley and M. Figueroa, "Polymorphic malware detection using topological feature extraction with data mining," in *Proc. SoutheastCon*, Mar./Apr. 2016, pp. 1–7.

[19] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, Mar. 2016.

[20] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate Android malware," in *Proc. 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jul. 2013, pp. 163–171.

[21] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, no. 1, p. 3, Dec. 2018.

[22] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "DL4MD: A deep learning framework for intelligent malware detection," in *Proc. Int. Conf. Data Mining (DMIN), Steering Committee World Congr. Comput. Sci.*, 2016, pp. 1–7.

[23] Q. Miao, J. Liu, Y. Cao, and J. Song, "Malware detection using bilayer behavior abstraction and improved one-class support vector machines," *Int. J. Inf. Secur.*, vol. 15, no. 4, pp. 361–379, 2016.

[24] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.

[25] J.-W. Jang, J. Yun, A. Mohaisen, J. Woo, and H. K. Kim, "Detecting and classifying method based on similarity matching of Android malware behavior with profile," *SpringerPlus*, vol. 5, no. 1, p. 273, 2016.

[26] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A user-oriented behavior-based malware variants detection system for Android," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1103–1112, May 2016.

[27] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: Android malware characterization and detection using deep learning," *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, Feb. 2016.

[28] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018.

[29] M. Bat-Erdene, H. Park, H. Li, H. Lee, and M.-S. Choi, "Entropy analysis to classify unknown packing algorithms for malware detection," *Int. J. Inf. Secur.*, vol. 16, no. 3, pp. 227–248, Jun. 2017.

[30] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, Mar. 2018.

[31] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019.

[32] J. Kang, S. Jang, S. Li, Y.-S. Jeong, and Y. Sung, "Long short-term memory-based malware classification method for information security," *Comput. Elect. Eng.*, vol. 77, pp. 366–375, Jul. 2019.

[33] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools Appl.*, vol. 78, no. 4, pp. 3979–3999, 2019.

[34] R. Vinayakumar, K. P. Soman, P. Poornachandran, and S. S. Kumar, "Detecting Android malware using long short-term memory (LSTM)," *J. Intell. Fuzzy Syst.*, vol. 34, no. 3, pp. 1277–1288, 2018.

[35] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 162–169.

[36] H. Sun, X. Wang, R. Buyya, and J. Su, "CloudEyes: Cloud-based malware detection with reversible sketch for resource-constrained Internet of Things (IoT) devices," *Softw., Pract. Exper.*, vol. 47, no. 3, pp. 421–441, 2017.

[37] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Comput. Secur.*, vol. 37, pp. 215–227, Sep. 2013.

[38] Q. K. A. Mirza, I. Awan, and M. Younas, "CloudIntell: An intelligent malware detection system," *Future Gener. Comput. Syst.*, vol. 86, pp. 1042–1053, Sep. 2018.

[39] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 1–15.

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

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[42] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: https://arxiv.org/abs/1502.03167

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: https://arxiv.org/abs/1412.6980

[44] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for Android malware detection using various features," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 3, pp. 773–788, Mar. 2019.

[45] W. Yu, L. Ge, G. Xu, and X. Fu, "Towards neural network based malware detection on Android mobile devices," in *Cybersecurity Systems for Human Cognition Augmentation*. Cham, Switzerland: Springer, 2014, pp. 99–117.

[46] N. McLaughlin, J. M. del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. J. Ahn, "Deep Android malware detection," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.

[47] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: Android malware detection using static analysis of applications," in *Proc. 8th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Nov. 2016, pp. 1–5.

[48] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1105–1116.

[49] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new Android malware detection approach using Bayesian classification," in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Mar. 2013, pp. 121–128.

[50] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy Android malware detection using ensemble learning," *IET Inf. Secur.*, vol. 9, no. 6, pp. 313–320, 2015.

[51] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into Android applications," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 1808–1815.

[52] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.

AZIZ ALOTAIBI received the Ph.D. degree in computer science and computer engineering from the University of Bridgeport, Bridgeport, CT, USA. He is currently an Assistant Professor with the Computer Science Department, Taif University. His research interests include machine learning, deep learning, computer vision, cybersecurity, and web services.