Cloud Computing and Data Science
http://ojs.wiserpub.com/index.php/CCDS/

Research Article

DeepMetaDroid: Real-Time Android Malware Detection Using Deep Learning and Metadata Features

Hashida Haidros Rahima Manzil*, Manohar Naik S

Department of Computer Science, Central University of Kerala, Kerala, India
Email: hashida.pcs071902@cukerala.ac.in

Received: 26 February 2024; **Revised:** 18 April 2024; **Accepted:** 19 April 2024

Abstract: The increasing prevalence of Android malware poses significant risks to mobile devices and user privacy. The traditional detection methods have limitations in keeping up with the evolving landscape of malware attacks, necessitating the development of more effective solutions. In this paper, we present DeepMetaDroid, a real-time detection approach for Android malware that leverages metadata features. By analyzing crucial metadata, including APK size, download size, permissions, certificates, and DEX files, the proposed method enables effective identification of malware and enhances mobile security. Using deep learning techniques, a lightweight Android real-time monitoring system is equipped with the trained model. These methods include long short-term memory (LSTM), gated recurrent units (GRU), convolutional neural networks (CNN), deep neural networks (DNN), and other ensemble models. Utilizing the rectified linear unit (ReLU) as the activation function, the DNN model is constructed with 32 neurons in the input layer. A one-dimensional convolutional layer with 32 neurons and a filter size of three is used as the input layer in the CNN model. The LSTM model is designed with an input layer consisting of 16 neurons. The GRU model with 32 neurons is employed in the input layer. Additionally, ensemble models that combined several architectures were developed. The proposed method offers a faster and more scalable solution for malware detection by consuming fewer resources like memory and CPU. This work ensures device security by providing real-time monitoring on Android devices to prevent users from installing malicious applications and, thus, enhance user privacy and security.

***Keywords*:** Android malware detection, real-time monitoring, metadata features, deep learning, mobile security

1. Introduction

The proliferation of mobile devices and the reliance on smartphone applications have led to an alarming rise in malware threats. In the third quarter of 2022, as reported by the Kaspersky Security Network [1], a significant number of mobile malware attacks, specifically 5623670, were successfully blocked. As the most widely used mobile operating system, Android has become popular with both legitimate developers and malicious actors. As per Statista [2], in the first quarter of 2023, Android continued to dominate as the world's leading mobile operating system, holding a commanding market share of 71.4 percent. The expansive market share and the open-source nature of the Android operating system provide hackers with both the necessary tools and enticing potential rewards to specifically target Android systems [3]. The presence of malware on Android devices poses severe risks, including unauthorized access to sensitive data, financial fraud, and privacy breaches. Cybercriminals typically upload malicious programs to third-party

DOI: https://doi.org/10.37256/ccds.5220244503

sources and disseminate them via malicious links because the Google Play Store offers automatic app scanning to assure security. Malware typically starts alerting users to fresh app installations or system upgrades once it has been installed on their Android smartphone. Consequently, malware will infect the device, opening the door to other cyberattacks such as data theft, data modification, destruction of sensitive data, cyber impersonation, and so forth.

General approaches to Android malware detection have traditionally relied on several methods, including signature-based techniques [4-5], static analysis approaches [6-8], and dynamic analysis-based methods [9-10]. Static analysis is a behavior-based approach used to detect Android malware that examines how Android apps behave without running them on a real device or emulator. The simplicity of this method, which drastically cuts down on implementation complexity and processing time, makes it the preferred choice. The static features are extracted using reverse engineering tools such as APKTool [11], JADX [12], Dex2Jar [13], Android Studio [14], and Androguard [15]. Conversely, the dynamic analysis methodology runs an application on an emulator or a real device to replicate its real-world behavior. This makes it possible to watch and record the runtime operations of the application, as well as its interactions with the operating system, networks, and possible malicious activity. Certain studies [16-20] used a hybrid analysis approach in which they designed malware detection models using both static and dynamic data. However, the landscape of Android malware is constantly evolving, with attackers employing sophisticated techniques to evade detection. Consequently, there is an urgent need for advanced and real-time detection mechanisms to combat this ever-growing threat effectively.

This research paper aims to address this pressing need by proposing a real-time detection framework for Android malware that leverages metadata features. Metadata, encompassing APK size, download size, permissions, certificates, and classes.dex file provides valuable insights into an application's behavior and potential malicious intent. The primary objective of this study is to develop an effective real-time monitoring system integrated with deep learning-based approaches to classify Android applications as either malicious or benign using metadata features. The proposed framework can accurately identify and flag potentially malicious applications before they are installed or executed on the user's device. The significance of this research lies in its potential to enhance the security of Android devices, safeguard user privacy, and mitigate the risks associated with the constantly evolving landscape of Android malware. By providing a real-time detection mechanism, users can be alerted to the presence of malware before it can cause harm. The proposed system will block the corresponding malware installation and isolate the malware from the device to maintain a secure mobile environment. Thus, the proposed framework proactively mitigates Android malware threats to ensure a safer and more secure mobile ecosystem for users.

1.1 *Background*

The Android Operating System (AOS) is a popular open-source platform developed by Google for mobile devices. It is based on the Linux kernel and is designed primarily for smartphones, tablets, and other touch-enabled devices. AOS provides a robust and flexible environment for running applications and offers a wide range of features and services.

1.1.1 *Android application structure*

The architecture of an Android app is based on the Android Package (APK) file format, which serves as the container for the app's components and resources. The APK file follows a specific structure that includes:

1. AndroidManifest.xml: This XML file is a vital component of the APK and provides essential information about the application, such as its package name, version number, permissions required, and the main components (activities, services, etc.) that comprise the application.

2. Classes.dex: This file contains the compiled bytecode of the application's Java or Kotlin code. The bytecode is executed by the Android Runtime (ART) or the Dalvik Virtual Machine (DVM), depending on the AOS version.

3. Resources.arsc: This binary file contains compiled resources such as strings, layouts, images, and other assets used by the application. It helps optimize the retrieval and usage of resources during runtime.

4. lib/: This directory contains native libraries specific to different CPU architectures (e.g., ARM, x86). It allows applications to include native code for enhanced performance or integration with existing libraries.

5. Assets/: This directory holds additional application-specific files that are not compiled but are bundled with the APK. These files can be accessed programmatically by the application at runtime.

6. Res/: This directory contains the source files for various resources used by the application, including layouts,

strings, styles, and images. These files are compiled into the Resources.arsc file.

1.1.2 *Security mechanisms*

To ensure the security of Android apps, the Android operating system incorporates several built-in security mechanisms. These mechanisms aim to protect users' devices and data from malicious activities. Some key security mechanisms in Android include:

1. Permissions: Android apps must declare the permissions they require to access certain system resources or user data. Users are prompted to grant or deny these permissions during the app installation process, allowing them to control the app's access to sensitive information.

2. Sandbox Environment: Android apps run in a sandboxed environment, which means they are isolated from each other and the underlying operating system. This isolation prevents malicious apps from interfering with other apps or the system itself.

3. Application Signing: Android apps are required to be digitally signed with a certificate. The certificate ensures the integrity and authenticity of the app and helps in verifying the app's source. This mechanism helps users to identify trusted apps and protects against tampering or unauthorized modifications.

4. Google Play Protect: Google Play Protect is a built-in security feature provided by Google Play Store. It scans apps for malware and other potentially harmful behavior before they are installed on users' devices. It also periodically checks installed apps for any security risks.

5. Runtime Permissions: Starting from Android 6.0 (Marshmallow), Android introduced a runtime permission model. It allows users to grant or deny permissions to apps at runtime, giving them more control over their privacy and security.

1.2 *Contributions*

The main objective of this paper is to develop a proactive real-time solution for detecting malware threats in Android that ensures impressive classification performance while yielding reduced resource consumption.

The Main contributions of the paper as described below:

• This research proposes a cutting-edge real-time monitoring system that integrates deep learning models to effectively detect and block Android malware.

• The proposed method leverages metadata features which enable a lightweight approach for the malware detection.

• This lightweight approach offers an effective and scalable solution for detecting Android malware due to the consumption of fewer resources like memory and CPU.

• To evaluate the effectiveness of the proposed method, extensive experiments are conducted using various deep-learning techniques and ensemble models.

The subsequent sections of this paper are organized as follows: In Section 2, the literature review is described. Section 3 provides the materials and methods of the paper. Section 4 offers a detailed explanation of the proposed methodology. Section 5 focuses on the experiments conducted and provides a comprehensive discussion of the corresponding results obtained. Finally, Section 6 concludes the paper, summarizing the contributions and highlighting the future research directions.

2. Literature review

Numerous research studies have focused on Android malware detection, predominantly employing supervised machine-learning techniques. For instance, Lê et al. [21] propose a machine learning-based detection technique by utilizing permissions and API features. However, this approach lacks deploying the framework on Android devices.

In another study, Islam et al. [22] presents a machine learning-based ensemble model for Android malware detection using a dynamic feature analysis method. However, their study did not guarantee the detection of malware in real time. Similarly, Bhat et al. [10] utilized dynamic behavior analysis with system call-centric features and employed a stacking ensemble technique for malware detection.

On the other hand, reinforcement learning has emerged as an accepted approach for Android malware detection. For example, Rathore et al. [23] developed different evasion techniques and a defense strategy to counter evasion attacks on malware detection models. Similarly, the studies including [24-28] have utilized the strength of machine learning deep learning techniques for malware detection. The deep learning techniques often provide superior performance over machine learning-based solutions when there is a large dataset. Mbunge et al. [29] suggest that there is a need to increase malware dataset sizes as well as improve the accessibility of malware datasets to the public.

While Ko et al. [30] developed a framework for real-time detection of unknown ransomware, their study primarily focused on discussing the higher-level framework without conducting any experiments. In contrast, our work establishes a real-time monitoring system integrated with deep learning techniques and evaluates its efficiency using various metrics through multiple experiments. Also, we adopt a meta-data feature fusion technique in addition to feature selection to get an optimized feature vector. By utilizing metadata features, the computational overhead associated with dynamic analysis approaches is significantly reduced. Moreover, the use of metadata features makes the proposed system lightweight and efficient, without sacrificing detection accuracy.

Most related studies incorporate system call-based approaches, as seen in [10] and [31]. In [31], *Y*aniv Agman and Danny Hendler present a dynamic analysis framework that continuously monitors running events. This study relies on the eBPF (extended Berkeley Packet Filter) technology of the Linux Kernel to monitor events from the internal kernel, system calls, or native library functions. However, executing eBPF programs introduces performance overhead, making it less suitable for extremely latency-sensitive or high-performance applications.

The authors Iqbal et al. [32] introduced SpyDroid, a real-time malware detection framework that incorporates multiple malware detectors. SpyDroid is designed to operate within the operating system (OS) and facilitates the exchange of valuable information with its sub-detectors. It acts as an intermediary between the OS and other detectors or anti-virus applications. However, this technique relies entirely on the outcomes generated by other auxiliary malware detectors. A summary of the related works is provided in Table 1.

Table 1. Literature review summary

Study	Approach	Real-time Detection	Deployment on Android Devices	Main Contribution
[21]	Machine learning using permissions and API features	No	No	Proposal of ML-based detection technique
[22]	Ensemble model with dynamic feature analysis	No	No	Utilization of ensemble model for malware detection
[10]	Dynamic behavior analysis with system call-centric features	No	No	Stacking ensemble technique for malware detection
[23]	Reinforcement learning techniques and defence strategy	No	No	Development of evasion techniques and defence strategy
[29]	Utilization of machine learning and deep learning techniques	No	No	Call for increasing malware dataset sizes and accessibility
[30]	Framework for real-time detection of unknown ransomware	*Yes*	No	Proposal of real-time detection framework for ransomware
[31]	Dynamic analysis framework using eBPF technology	*Yes*	No	Utilization of eBPF technology for dynamic analysis
[32]	Real-time malware detection framework (SpyDroid)	*Yes*	*Yes*	Introduction of SpyDroid framework for real-time detection

3. Materials and methods

The proposed methodology achieves effective Android malware detection by combining real-time analysis with deep learning-based malware detection through the use of metadata features. It guarantees quick detection and defense against changing malware threats by concentrating on real-time monitoring and feature extraction, improving the security of Android devices and user data. TensorFlow and other well-known deep learning frameworks were used to implement the produced models. The implementation was optimized for efficiency and scalability, allowing for real-time monitoring and detection of Android malware on mobile devices.

3.1 *Data source and tools*

The dataset used in this study was collected from a variety of sources, including Drebin [33], Virus Total [34], Contagio Mini Dump [35], and Google Play Store [36]. To ensure diversity and representativeness, the dataset includes both benign and malicious Android applications. The metadata features used for Android malware detection were extracted from the collected dataset. The data source and tools utilized are further described in Table 2.

Table 2. Data source and tools used

Approach	Data Source	Tool	Features
Real time	1. Drebin	Development: Android Studio, TensorFlow	• APK Size
	2. VirusTotal		• Download Size
	3. ContagioMiniDump		• Certificates
	4. Google Play Store		• Permissions

3.2 *Experimental setup*

The experiments are conducted on a specific machine with the following hardware and software specifications:

• **Hardware:** The machine used an Intel (R) Core (TM) i5-4210U CPU processor running at 1.70 GHz, 2.40 GHz, and had 4 GB of RAM.

• **Software:** Ubuntu 14 Linux operating system, Python 3.3 with the Sci-kit Learn packages, TensorFlow and Android Studio.

4. Proposed system

The proposed methodology incorporates deep learning-based malware detection with real-time analysis by using metadata features to achieve efficient Android malware detection. By focusing on real-time monitoring and feature extraction, it ensures prompt identification and protection against evolving malware threats, enhancing the security of Android devices and user data. The proposed system primarily contains two phases, such as development and deployment (as seen in Figure 1).

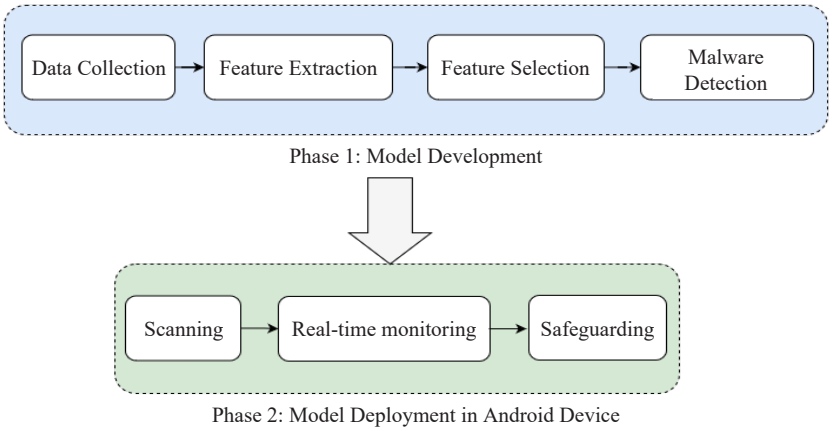


Figure 1. The phases of the proposed method

4.1 *Phase 1: model development*

The development phase is where the actual malware detection model is established. The dataset is trained using deep learning techniques and the model which results in superior accuracy will be opted as the final detection model. The overall processes of the development phase are illustrated in Figure 2. Once this deep learning model is established with better performance, the model is deployed as a real-time monitoring system installed in the Android device in the second phase (deployment phase).

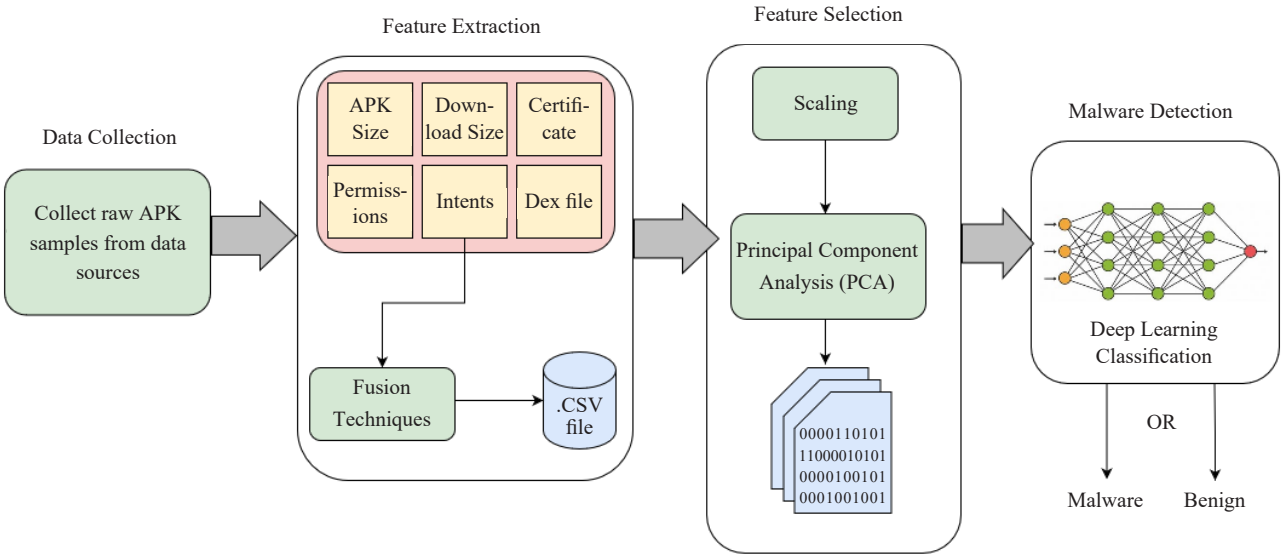


Figure 2. The overall processes in the development phase of the proposed method

4.1.1 *Data collection*

This stage consists of collecting raw APK samples from various data sources. In addition to Drebin [33], Android malware data samples are collected from reputable repositories such as VirusTotal [34] and Contagio Mini Dump [35].

This diverse collection ensures a comprehensive representation of different types of malware. The benign samples were obtained from reliable sources like Google Play Store [36], ensuring their authenticity and relevance. Thus, a total of 2,000 samples are collected in each class consists of 1,000 samples providing a balanced distribution for training.

4.1.2 *Feature extraction*

Algorithm: Extract Metadata Features from Raw APK Files

Input: List of raw APK files

Output: Metadata features for each APK

1. For each raw APK file in the input list, do the following:
   A. Extract metadata features for the APK:
      i. Extract APK size:
         ⊳ Obtain the size of the APK file in bytes.
         ⊳ (apkSize ← get_apk_size (raw_apk))
      ii. Extract download size:
         ⊳ Compute the size of resources to be downloaded when installing the APK.
         ⊳ (downloadSize ← get_download_size (raw_apk))
      iii. Extract permissions:
         ⊳ Retrieve the list of permissions required by the APK.
         ⊳ (permissions ← get_permissions (raw_apk))
      iv. Extract certificates:
         ⊳ Obtain the digital certificates used to sign the APK.
         ⊳ (certificates ← get_certificates (raw_apk))
      v. Extract intents:
         ⊳ Retrieve the list of intents declared in the APK's manifest file.
         ⊳ (intents ← get_intents (raw_apk))
      vi. Extract DEX file:
         ⊳ Extract the Dalvik Executable (DEX) file from the APK.
         ⊳ (dexFile ← extract_dex_file (raw_apk))
   B. Store the extracted metadata features in a data structure:
      ⊳ Combine all extracted features into a metadata structure.
      ⊳ (metadataFeatures ← {apkSize, downloadSize, permissions, certificates, intents, dexFile})
   C. Save the metadata features to a file or database:
      ⊳ Store the metadata features associated in a suitable storage medium.
      ⊳ (save_metadata_features (metadataFeatures))

Figure 3. Pseudocode for extracting metadata features from apps

To extract pertinent metadata features from the collected raw APK samples, as mentioned in Section 4.1.1, different reverse engineering tools are utilized, including Android Studio [14] and Androguard [15]. Different metadata features, encompassing permissions, intents, certificates, Dex files, download sizes, and APK file sizes, are extracted from these APK files. Figure 3 depicts the pseudocode for extracting metadata features from newly installed or updated apps. These features provide valuable insights into the characteristics and potential risks associated with the analyzed apps and can be utilized to uncover patterns that have not yet been identified in prior research. Such metadata serves as a solid foundation for static malware detection, which offers a rapid and efficient approach to identifying Android malware. Moreover, the utilization of meta-data features has reduced resource consumption, like CPU or memory, to make the system lighter. Martín et al. [37] have utilized these kinds of meta-data features integrated with machine learning techniques for malware characterization. However, their study didn't provide detection in real time.

4.1.2.1 *Fusion techniques*

Once the meta-data features are extracted from APK samples, a weighted average fusion technique is applied to them. The fusion techniques refer to the concatenation of individual features for deriving more comprehensive and informative representations of the analyzed apps. These techniques ensure reliability in features, by obtaining a compact set of prominent features that can improve detection accuracy. Hence, by fusing multiple metadata features, we aim to uncover hidden patterns that may not be apparent when considering features in isolation. This fusion process enhances the quality of the input data to be fed into the detection model. The fusion technique is performed using statistical techniques, such as weight-based feature concatenation. By this technique, the quality and robustness of the features can be improvised, enabling more effective detection of Android malware. Thus, after this stage, a total of 250 features are obtained.

The common approach for performing feature fusion is to use a weighted average. The fused feature vector is calculated using the following equation (1):

$$Fused\ Feature = w_1 \times Feature_1 + w_2 \times Feature_2 + \ldots + w_n \times Feature_n \tag{1}$$

Where $Feature_1$, $Feature_2$, ..., $Feature_n$ are the individual metadata features and w_1, w_1, ..., w_n are the weights assigned to each meta-data feature. The weights are assigned using the correlation technique as described in equation (2):

$$w = \frac{\left(\Sigma\left(\left(X - X_{mean}\right) \times \left(Y - Y_{mean}\right)\right)\right)}{\left(\sqrt{\left(\Sigma\left(X - X_{mean}\right)^2\right)} \times \sqrt{\left(\Sigma\left(Y - Y_{mean}\right)^2\right)}\right)} \tag{2}$$

Where X and Y represent the variables (metadata features and the target variable, respectively), X_{mean} and Y_{mean} are the means of X and Y, respectively and Σ denotes the sum of the corresponding terms. The correlation coefficient (w) ranges from -1 to 1, where 1 indicates a strong positive correlation, -1 indicates a strong negative correlation, and 0 indicates no linear correlation between the variables.

4.1.3 *Feature selection*

The feature selection plays a significant role in providing better detection performance. Selected features are fed into the final detection model during this phase. The extracted metadata features are cleansed and pre-processed to obtain data in a well format using scaling then, Principal Component Analysis (PCA) has been applied for dimensionality reduction.

4.1.3.1 *Scaling*

Since data pre-processing accounts for 50%-80% of the time spent on data analytics [38], data transformation is required solely for the purpose of enhancing the performance of a machine learning algorithm [39]. Therefore, a

standard scaling technique is applied to ensure the compatibility of the features. Standard scaling, also known as z-score normalization, transforms the features using the following equation (3):

$$X_{scaled} = \frac{X - \mu}{\sigma} \tag{3}$$

Where X represents the original feature values, μ denotes the mean of the feature values, and σ represents the standard deviation. Applying standard scaling eliminates the potential bias introduced by the different scales and magnitudes of the individual features. This process brings all features to a similar scale, facilitating more meaningful and accurate comparisons between the feature values. The scaled dataset enables us to effectively capture the relative importance and relationships between the metadata features.

4.1.3.2 *Principal Component Analysis (PCA)*

In addition to fusion techniques and scaling, the Principal Component Analysis (PCA) is also applied to the dataset as a feature dimensionality reduction technique to obtain a more optimized feature vector. PCA is a widely used method that transforms the original features into a new set of uncorrelated variables called principal components. These principal components are ordered in terms of their explained variance, with the first component capturing the highest amount of variance in the data. By selecting a subset of the principal components that explain the majority of the variance, the dimensionality of the feature space can be reduced while preserving the essential information [40]. In the proposed approach, the number of components is set to 32. This reduction step helps alleviate the curse of dimensionality, mitigate computational complexity, and enhance model interpretability. Moreover, the use of PCA facilitates the identification of latent structures and patterns within the data and thus obtains an optimized feature set that serves as a foundation for further analysis and modeling.

4.1.4 *Malware detection*

In this phase, deep learning classification has been applied to detect malware. Deep learning techniques are adversarial methods that have been efficiently used in many recent security problems. For instance, the authors [41] utilized deep learning techniques to enhance the security of medical images. The neural networks imitate the features of biological neurons to process the information. Each neuron with weight and threshold adjusted with learning will send a signal if the output is over the threshold; otherwise, there is no message transmitted to the next layer of the networks [42]. To construct an efficient detection model, several experiments are implemented with different deep learning techniques, including deep neural networks (DNNs), convolutional neural networks (CNNs), long short-term memory (LSTM), gated recurrent units (GRU), and ensemble models. The models were trained using popular optimization algorithms like stochastic gradient descent (SGD) and Adam. The regularization technique, like Lasso L2, has been applied to mitigate overfitting issues and employed early stopping techniques to prevent model convergence. The models' performance was evaluated using metrics such as accuracy, loss, precision, recall, F1 score, and area under the precision-recall curve (AUC-PR). The experiment with a deep neural network (DNN) gave the highest detection accuracy of 97.32%.

4.2 *Phase 2: model deployment in android device*

This is the second phase of the proposed methodology, in which the detection model developed in the first phase is deployed in this phase. As per Section 4.1, the deep learning model, which provides better performance, is implemented on an Android device as a real-time monitoring system. Therefore, the proposed real-time application detects new installations on the device through ongoing monitoring. Whenever a new installation or update happens, the system captures its basic meta-data features and applies them to the deep learning detection model that is integrated within the system. The model then classifies it either as benign or malicious. If the model identifies it as malware, the system will alert the user and quickly take appropriate safety precautions. On the other hand, if the detection model results in the app being a legitimate one, the proposed system will ignore the current installation and continue the scanning process. Hence, the proposed system provides early detection of malware and prevents users from installing it. The overall

activities carried out in this stage are presented in the diagram shown in Figure 4.

Briefly, the main processes in DeepMetDroid include scanning, real-time monitoring, and safeguarding, as illustrated in Figure 5. These processes are further explained in subsections 4.2.1 and 4.2.2.

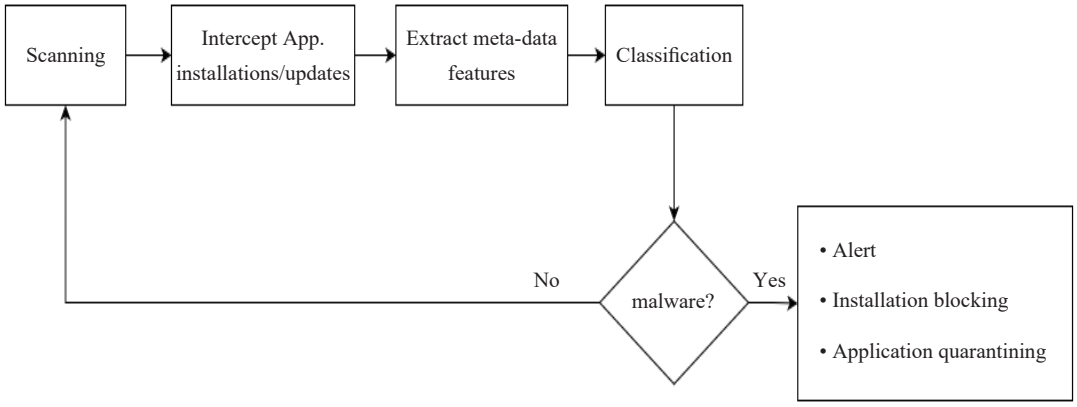


Figure 4. The summary of proposed real-time malware detection

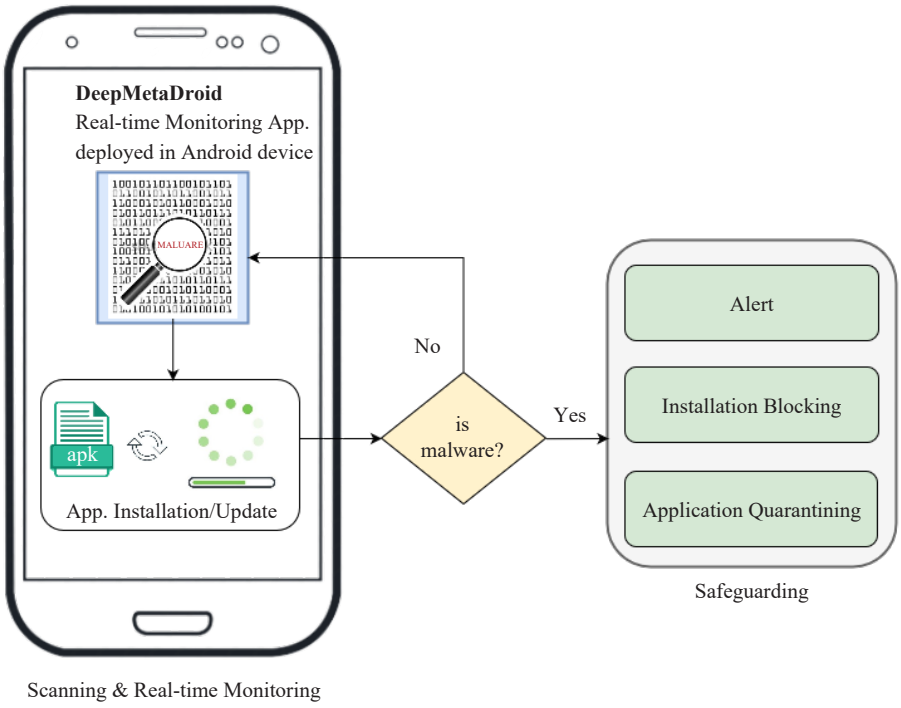


Figure 5. The overall processes in the deployment phase of the proposed method

4.2.1 *Scanning and real-time monitoring*

The proposed system continuously scans the Android device for newly installed or updated apps. Android's BroadcastReceiver functionality enables this by listening to the ACTION_PACKAGE_ADDED broadcast, which will be triggered whenever a new application is installed on the device. Thereby, the system effectively intercepts new

application installations or updates, triggering the feature extraction process on the fly. The extracted features are then fed into the trained deep-learning model for recognizing malware.

4.2.2 *Safeguarding*

As shown in Figure 5, the DeepMetaDroid monitoring system promptly triggers an alert if an application is identified as malware. This alert serves as an early warning, indicating the presence of a potentially harmful application within the device. To safeguard against further damage or compromise, the monitoring system swiftly takes action by blocking the installation of the flagged application or quarantining, which proactively prevents potential security risks. The quarantining process involves placing the identified malware in an isolated environment. This effectively restricts its access to sensitive resources and prevents its interaction with other apps and system components. By confining the malware, the monitoring system significantly reduces its ability to cause harm, thus safeguarding the integrity and security of the Android device.

5. Implementation and results

In this research, a series of experiments are performed using various deep-learning techniques. These techniques include deep neural networks (DNN), convolutional neural networks (CNN), long short-term memory (LSTM), gated recurrent units (GRU), and other ensemble models. Each experiment was aimed at exploring the effectiveness and performance of these techniques. Each model is evaluated with metrics like accuracy, loss, precision, recall, F1-score, and area under the precision-recall curve (AUC-PR). In the following sections, the details of each experiment are discussed along with the corresponding results. The corresponding performance results of all experiments are given in Table 3.

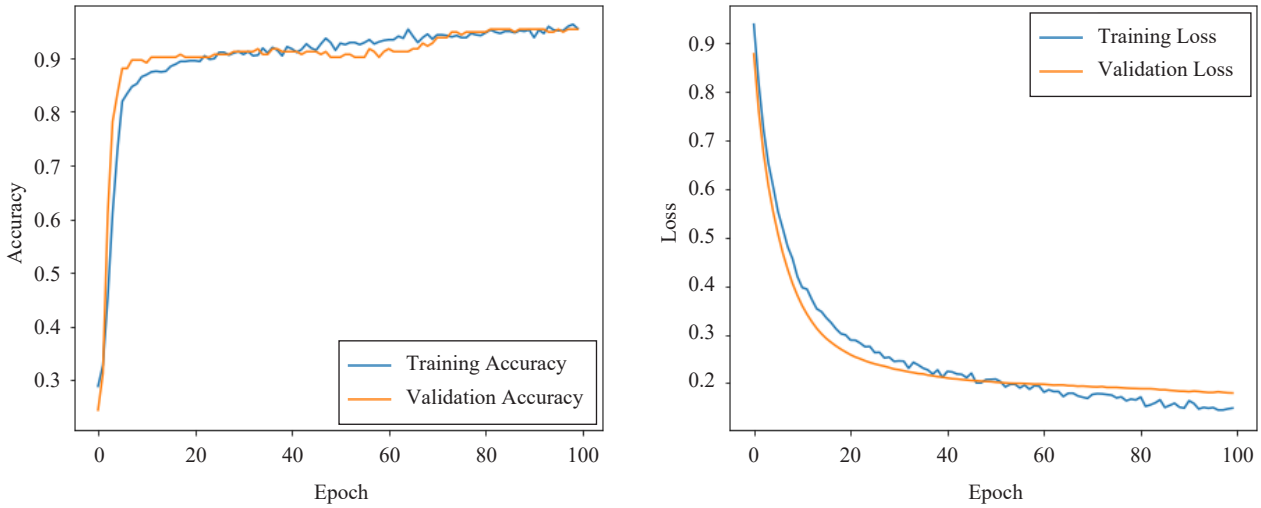
Table 3. The results of all deep learning experiments

Model used	Accuracy	Loss	Precision	Recall	F1-Score	AUC-PR
DNN	**0.9732**	0.1934	0.90243	0.8409	**0.9069**	**0.92**
CNN	0.9479	0.1800	0.9257	0.8409	0.8809	0.90
LSTM	0.92708	0.1885	0.8372	0.8181	0.8275	0.85
GRU	0.8958	0.2628	0.8757	0.6363	0.7365	0.80
Ensemble DNNs	0.9414	0.1776	0.8837	**0.8636**	0.8735	0.89
DNN-LSTM	0.9531	0.1725	**0.9487**	0.8409	0.8915	0.91
CNN-LSTM	0.9531	**0.1674**	**0.9487**	0.8409	0.8915	0.91

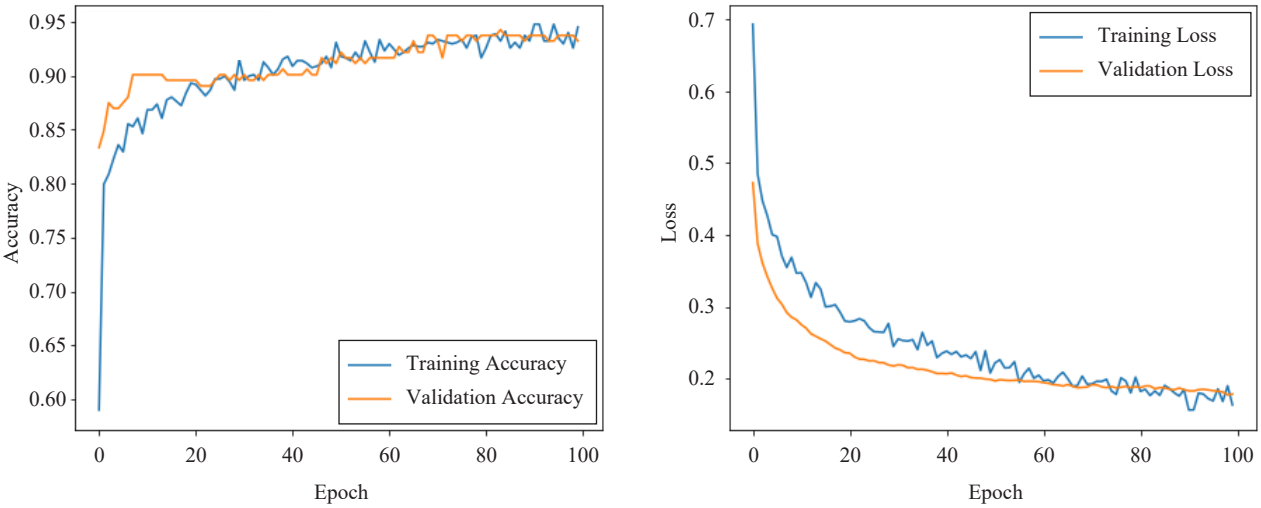
5.1 *Deep Neural Network (DNN)*

In this research, a deep neural network (DNN) is developed with 32 neurons in the input layer and utilizes the rectified linear unit (ReLU) as the activation function. The DNN consisted of multiple interconnected layers that processed the data, allowing it to learn and make predictions based on high-level abstractions in the data. To prevent overfitting, the Lasso L2 regularization technique is applied. This regularization technique helps reduce the complexity of the model by adding a penalty term to the loss function, encouraging smaller and more generalizable weights.

Furthermore, two different optimizers, namely Adam and Stochastic Gradient Descent (SGD), are applied. These optimizers determine how the weights and biases in the DNN are adjusted to minimize the loss and improve the model's performance. The results are presented in Figures 6(a) and 6(b). The model trained with the Adam optimizer achieved a higher accuracy of 0.9732, outperforming the model trained with SGD, which had an accuracy of 0.9322. These findings highlight the effectiveness of the designed DNN model for malware detection.



(a) The accuracy and loss curve of the DNN model (optimizer = 'Adam')



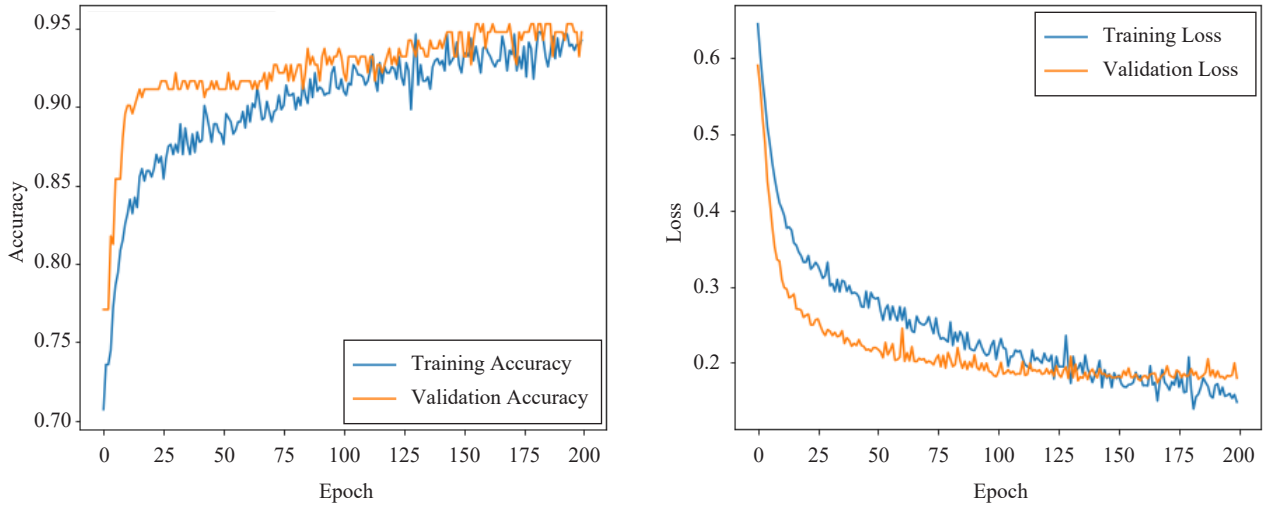
(b) The accuracy and loss curve of the DNN model (optimizer = 'SGD')

Figure 6. The accuracy and loss curves of the DNN model when the optimizer is Adam (a) and when the optimizer is SGD (b)

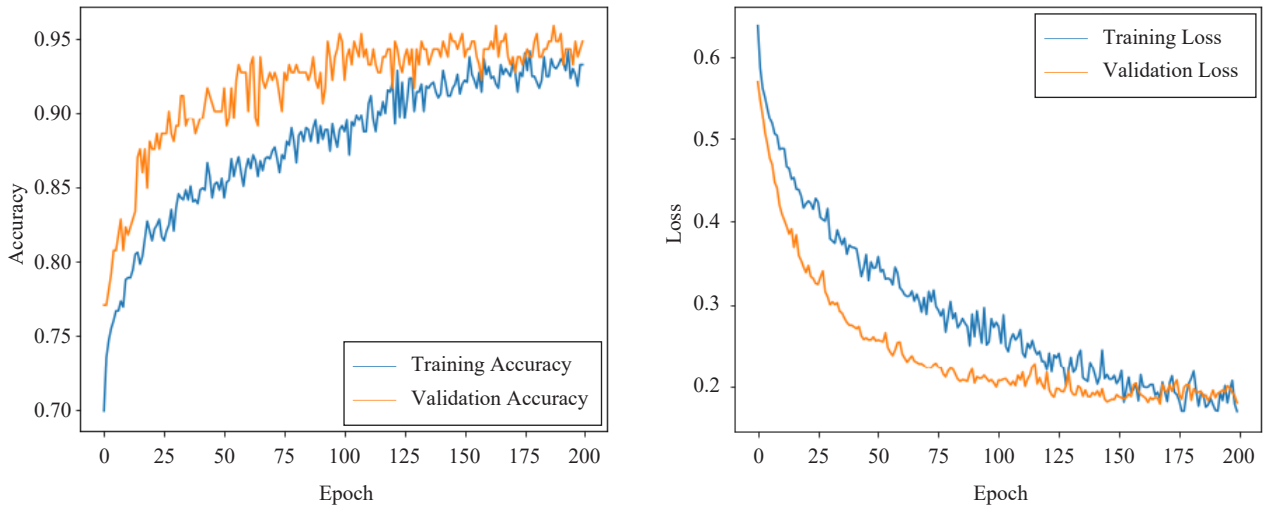
5.2 *Convolutional Neural Network (CNN)*

In this experiment, the convolutional neural network (CNN) is explored in the context of real-time detection of Android malware. CNNs are known for their ability to automatically learn and extract hierarchical representations from data. Generally, CNN models are used for image classification problems like face recognition or biometric

authentication. For instance, the authors of the study [43] discuss biometric watermarking for user authentication. In such situations, CNN models can be efficiently applied. They include convolutional layers that apply learnable filters or kernels to inputs, allowing for feature extraction. Pooling layers and fully connected layers are also incorporated in CNNs. Pooling layers reduce spatial dimensions while preserving essential information in feature maps, while fully connected layers make final predictions based on the extracted features. During the training process, CNNs optimize their weights using backpropagation, adjusting weights based on predicted and true labels. This iterative process enhances the network's capability to recognize and classify patterns of application behavior.



(a) The accuracy and loss curve of the CNN model (optimizer = 'Adam')



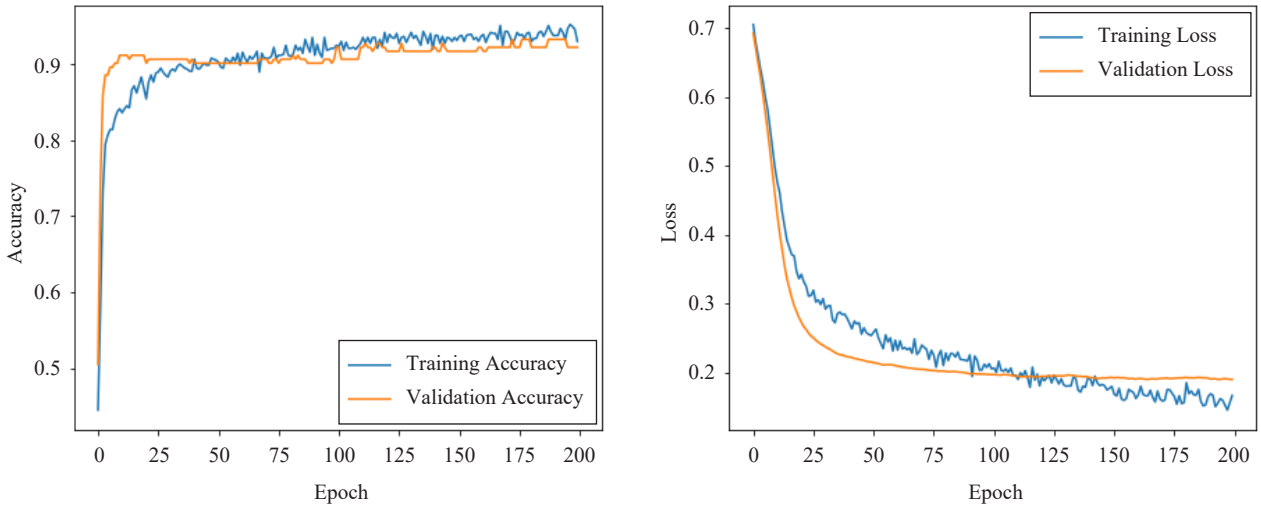
(b) The accuracy and loss curve of the CNN model (optimizer = 'SGD')

Figure 7. The accuracy and loss curves of the CNN model when the optimizer is Adam (a) and when the optimizer is SGD (b).

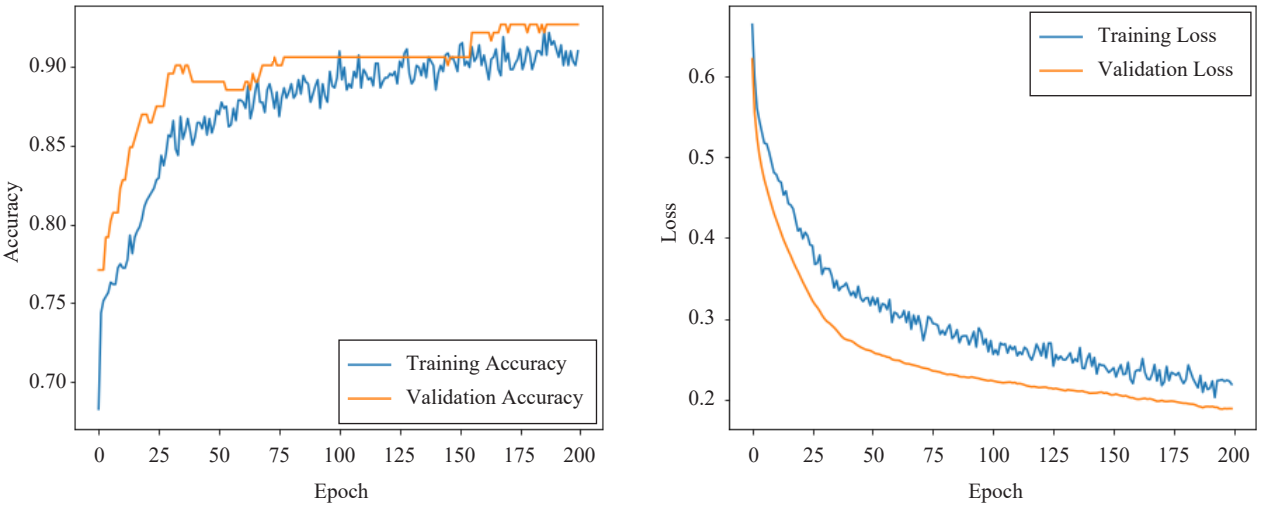
In this experiment, a CNN model was designed that featured a one-dimensional convolutional layer as the input layer, comprising 32 neurons and a filter size of 3. Then max-pooling is used with a dropout rate of 0.2. The model included two hidden layers. The performance of the CNN model is evaluated using two different optimizers, namely

Adam and SGD, as depicted in Figures 7(a) and 7(b), respectively. In particular, the SGD optimizer is implemented with a learning rate of 0.01 and a momentum of 0.9. The graph demonstrates similar performance results for the CNN model with both optimizers, yielding an accuracy of 0.9479 and a loss value of 0.1800. These results suggest that the choice of optimizer did not significantly impact the model's performance.

5.3 *Long Short-Term Memory (LSTM)*



(a) The accuracy and loss curve of the LSTM model (optimizer = 'Adam')



(b) The accuracy and loss curve of the LSTM model (optimizer = 'SGD')

Figure 8. The accuracy and loss curves of the LSTM model when the optimizer is Adam (a) and when the optimizer is SGD (b)

The Long Short-Term Memory (LSTM) model is a popular variant of recurrent neural networks (RNNs) extensively used for analyzing sequential data. It effectively handles the vanishing gradient problem and is capable of capturing long-term dependencies. LSTMs employ memory cells and gating mechanisms such as input, forget, and output gates to regulate the information flow within the model. The LSTM model learns to update and utilize memory

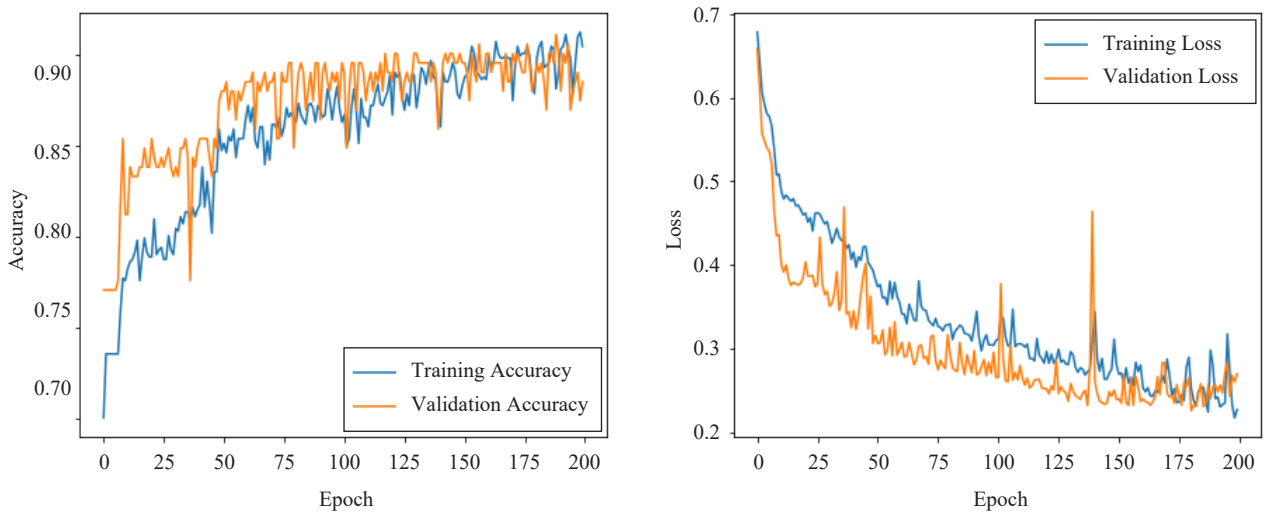
cells during the training process. In this experiment, an LSTM model is designed with an input layer consisting of 16 neurons and utilizes the rectified linear unit (ReLU) activation function. However, this model yields relatively inferior results compared to other deep-learning models employed in this study. To investigate the impact of different optimization techniques on the model's performance, both Adam and stochastic gradient descent (SGD) optimizers are applied. The corresponding accuracy and loss values of the LSTM model are presented in Figures 8(a) and 8(b). With the Adam, the LSTM achieved an accuracy of 0.921875 and a loss value of 0.19067. Similarly, when using the SGD, the model results in an accuracy of 0.92708 and a loss value of 0.1885, which is slightly higher. These findings indicate that the performance of LSTM may vary with different optimization techniques. However, it is important to acknowledge that the achieved results were comparatively lower than those obtained by other deep learning models employed in this research.

5.4 *Gated Recurrent Unit (GRU)*

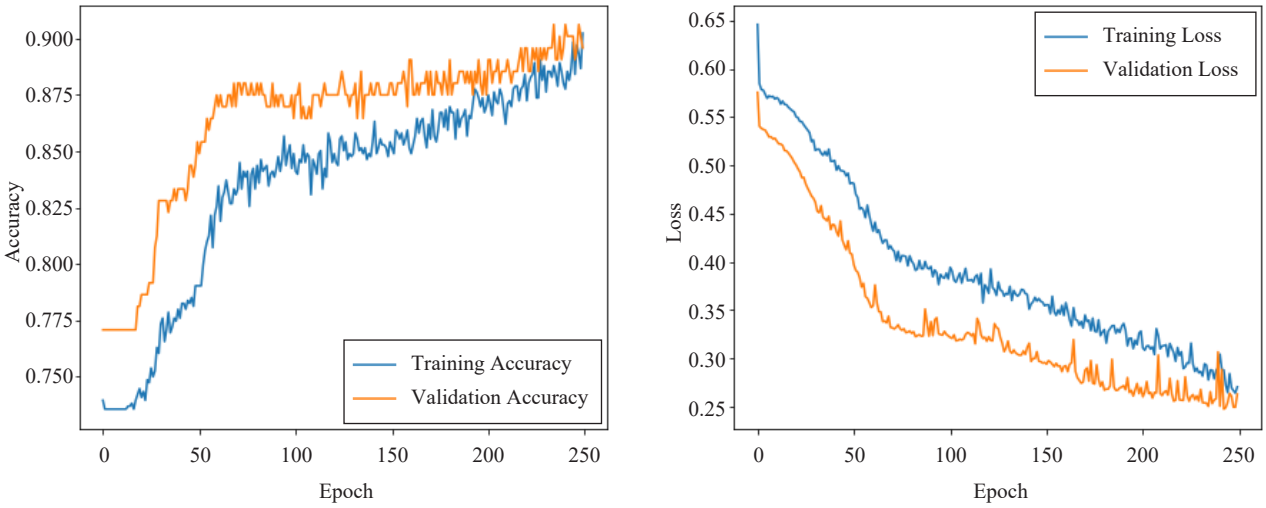
The Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture that addresses the vanishing gradient problem and allows for the efficient learning of long-term dependencies in sequential data. The key innovation of the GRU lies in its gating mechanism, which controls the flow of information within the network. Unlike the standard RNN, the GRU has two gates: the update gate and the reset gate. These gates enable the model to selectively update or reset the hidden state based on the input at each time step.

The update gate determines how much of the previous hidden state should be retained and combined with the new input, allowing the network to capture relevant information from the past. On the other hand, the reset gate controls how much of the previous hidden state should be ignored, enabling the network to forget irrelevant or outdated information. By adaptively updating and resetting the hidden state, the GRU can effectively capture long-term dependencies while mitigating the vanishing gradient problem.

In this work, a GRU model with 32 neurons is employed in the input layer. However, it exhibits lower performance compared to previous models. The model shows an accuracy of 0.8854 and a loss of 0.2696 with the Adam optimizer (Figure 9(a)). Similarly, when using the SGD optimizer, the model results in an accuracy of 0.8958 and a loss value of 0.2628 (Figure 9(b)).



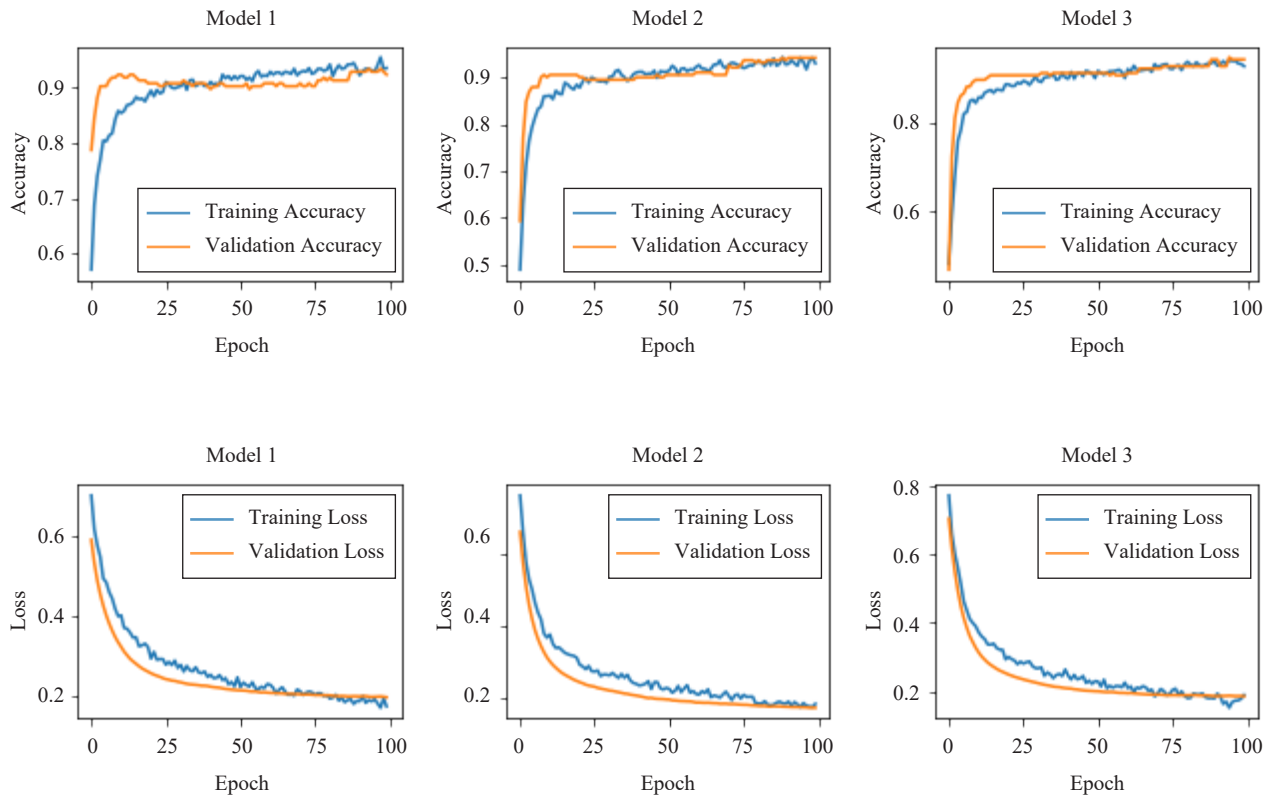
(a) The accuracy and loss curve of the GRU model (optimizer = 'Adam')



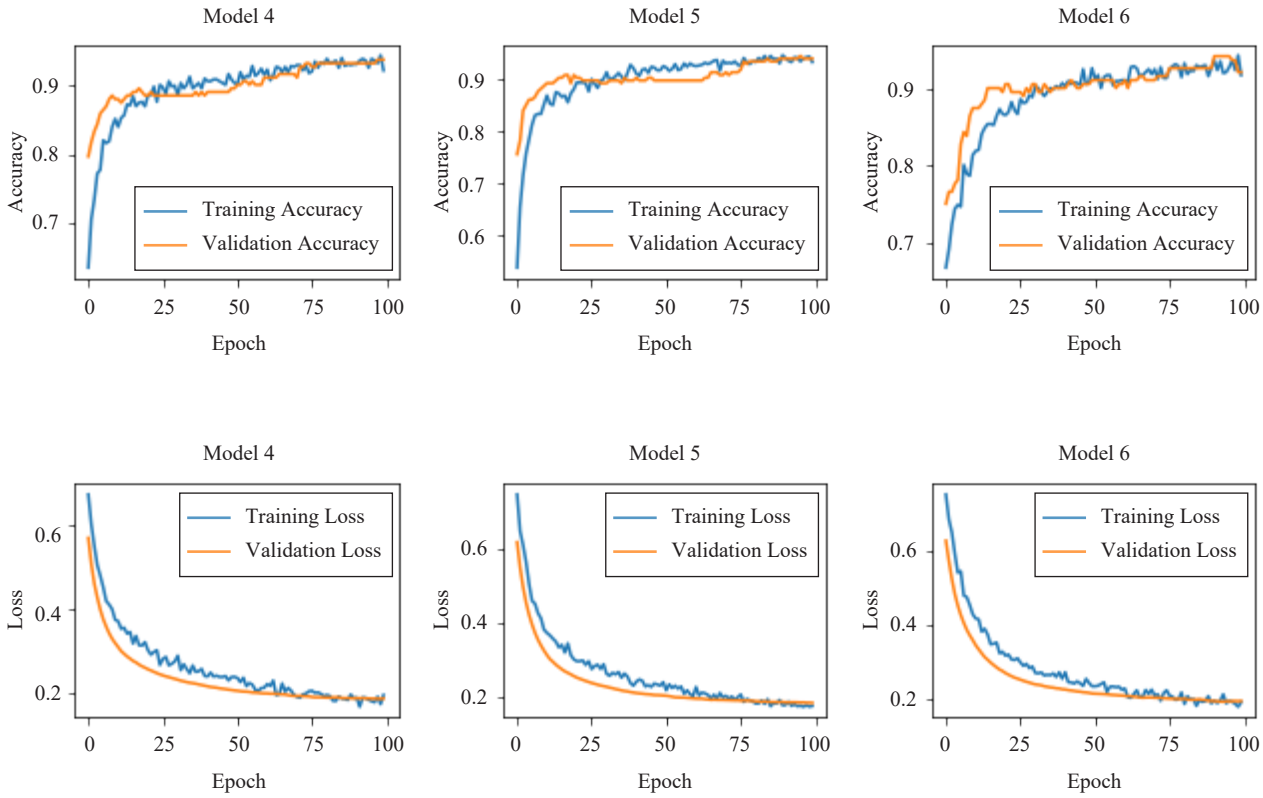
(b) The accuracy and loss curve of the GRU model (optimizer = 'SGD')

Figure 9. The accuracy and loss curves of the GRU model when optimizer is adam (a) & when optimizer is SGD (b)

5.5 *Ensemble model of DNNs*



(a) The accuracy and loss curves of the ensemble of DNNs (first three models)



(b) The accuracy and loss curves of the ensemble of DNNs (last three models)

Figure 10. The accuracy and loss curves of the ensemble of DNNs (first three DNN models (a) and last three DNN models (b))

In this study, an ensemble approach is incorporated, which consists of six deep neural network (DNN) models, to conduct a comprehensive analysis of the results. The ensemble's performance was assessed by evaluating the accuracy and loss values of each individual model, as depicted in Figures 10(a) and 10(b). The results demonstrate an average accuracy of 0.941406 and an average loss value of 0.177646.

5.6 *Hybrid model (DNN with LSTM)*

In this experiment, a hybrid model is established by combining a deep neural network (DNN) and a long short-term memory (LSTM) network. The experimental results revealed that this hybrid model achieves a remarkable accuracy of 0.953125. Furthermore, the loss value obtained was 0.1725. Comparing this DNN-LSTM fusion model with an individual DNN model, we observed significant progress in terms of the loss value. Figure 11 displays the corresponding accuracy and loss curves of the DNN-LSTM fusion model.

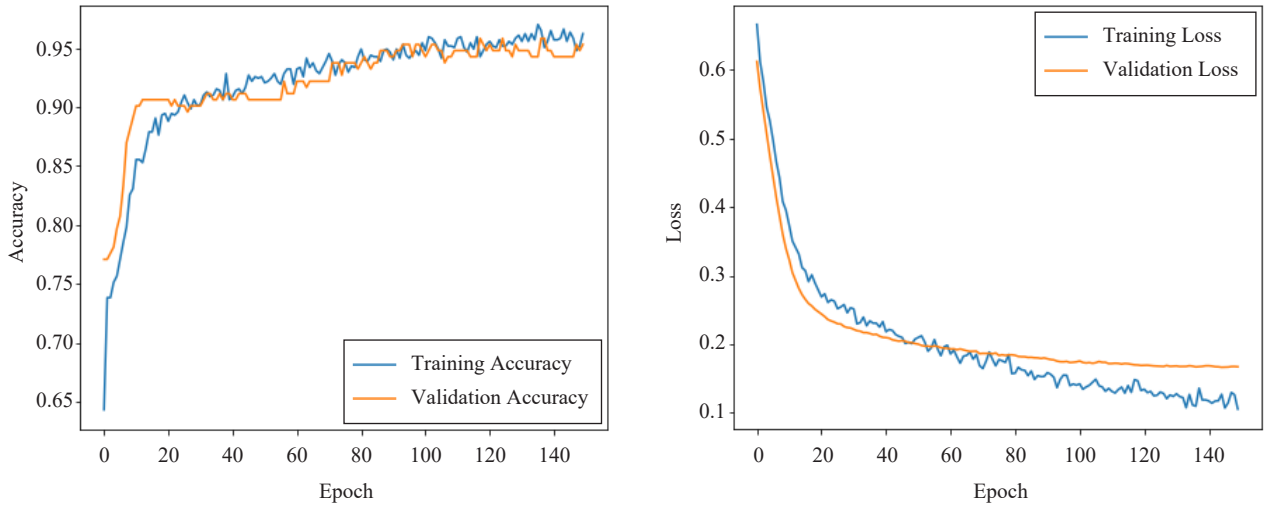


Figure 11. The accuracy and loss curves of DNN-LSTM fusion model

5.7 *Hybrid model (CNN with LSTM)*

A hybrid model that combines the capabilities of a convolutional neural network (CNN) and a long-short-term memory (LSTM) network is constructed in this experiment. The objective is to leverage the strengths of both architectures to improve performance. The experimental results demonstrate an accuracy of 0.953125 and a loss value of 0.1674121, surpassing the performance of the individual CNN model. Comparative analysis between the CNN-LSTM fusion model and the individual CNN model reveals significant progress in terms of both loss value and accuracy. Figure 12 presents the accuracy and loss curves of the CNN-LSTM fusion model.

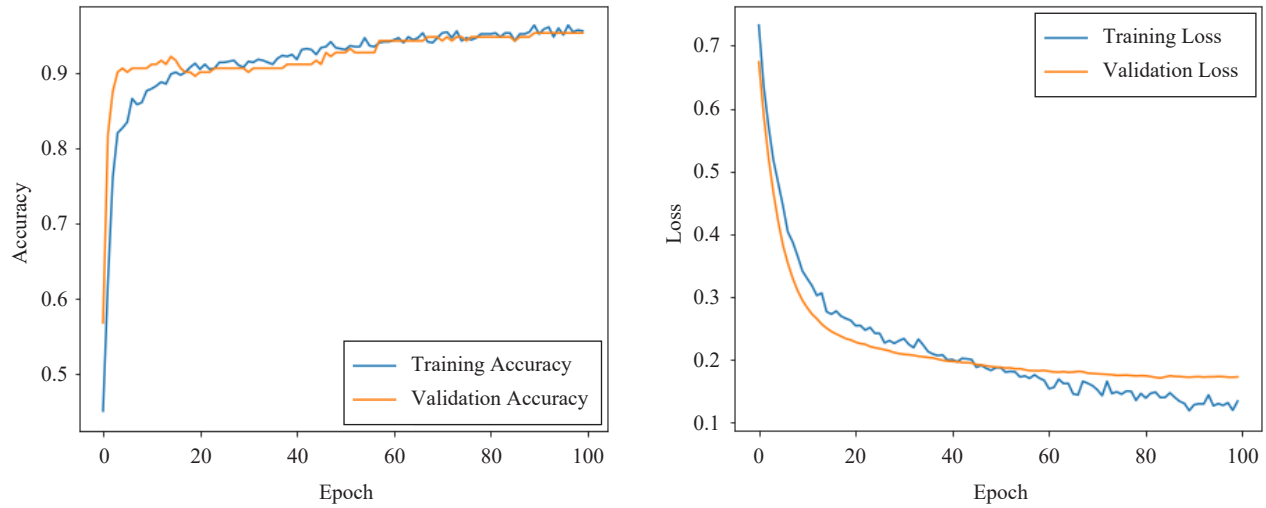


Figure 12. The accuracy and loss curves of CNN-LSTM fusion model

• On-device Deployment Testing

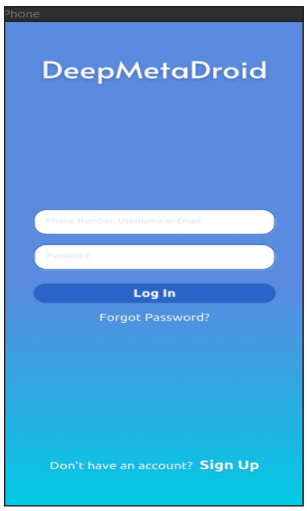
The proposed model is deployed on a real device, and its performance has been evaluated against parameters like run time, memory consumption, and CPU usage. The model is installed on a Xiaomi Mi A2 device with the Android

8.1 version, and real-time monitoring is performed by continuously intercepting any new updates or installations on the device. The meta-data features are extracted from those applications and supplied to the trained model to detect whether they are malicious or not. It is obtained that DeepMetaDroid runs at 15 ms, which is faster compared to some of the existing works. The run time refers to the clock time from initial scanning to final detection, whether an app. is malicious or not, and post-detection activities like application blocking and quarantining.

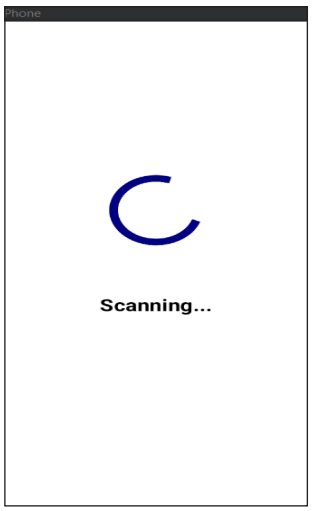
Lightweight systems typically have lower resource requirements in terms of processing power, memory, and storage. This efficiency can result in faster performance, reduced hardware costs, and lower energy consumption. For instance, in a recent work [44], authors have proposed a lightweight authentication system for IoT devices. The proposed system consumes fewer resources, like memory and CPU. Since the utilization of meta-data features makes it a lightweight approach, the model consumes only 4% of the total device CPU resources. Similarly, memory usage plays a critical role in evaluating a real-time malware detection system; the less memory is utilized, the faster the app will run on an Android device. Thus, the proposed system consumes only 99 MB of memory. Both of these parameters play a significant role in assessing the scalability of a real-time malware detection solution. As indicated by Table 4, the proposed real-time detection system, DeepMetaDroid, outperforms some of the existing works in terms of both accuracy and resource consumption.

Table 4. Comparison of proposed method with existing works

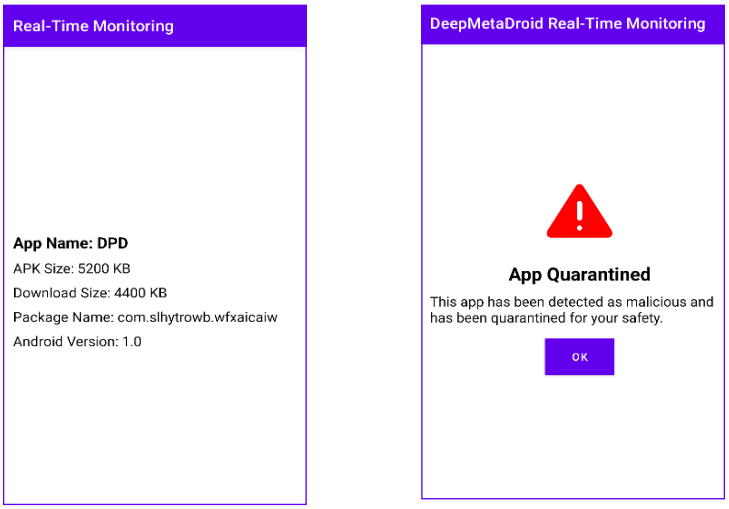
Method	Accuracy (%)	Run-time (ms)	Memory (mb)	CPU (%)	Dataset	Algorithm	Features
[45]	96.4	357.12	123	4	CICInvesAndMal2019 [46]	CNN	Permissions, Intents
[47]	94.3	NA	NA	NA	AndroZoo [48]	DT	Permissions
[49]	94.2	19	NA	NA	Drebin [33], Genome [50]	CNN *YOLO* V5	Permissions
[51]	93.4	NA	NA	NA	CICInvesAndMal2019 [46]	Deep ANN	Permissions, Intents, API calls, log files
[52]	95.3	NA	NA	NA	CICInvesAnd mal2017 [53]	RF	Permissions, Intents, API calls, Network flow data
DeepMetaDroid	**97.32**	**15**	**99**	**4**	Drebin [33], Contagio [35], Google Play Store [36]	DNN	Meta-data features



(a) Home screen



(b) Scanning page



(c) Display of app information after scanning

(d) The page shows that application is quarantined if it is malware

Figure 13. The user interfaces of the DeepMetaDroid real-time monitoring system (home screen (a), scanning page (b), display of app information (c), and alert page (d))

As part of this research, the proposed real-time monitoring system is integrated into the Android device platform. This integration involved incorporating the system's functionalities and features into an Android application. The application serves as a user interface for the real-time detection system, providing users with access to its capabilities and features. Figure 13((a), (b), (c), and (d)) showcases some of the interfaces of the proposed DeepMetaDroid real-time monitoring system.

6. Conclusion

In this research, we present a new approach for real-time Android malware detection: DeepMetaDroid. It uses metadata features and deep-learning methods. The methodology comprises two phases: development and deployment. In the development phase, we collected raw APK samples, extracted features, selected relevant features, and classified malware using various deep-learning models. Experimental results demonstrated the efficacy of our approach, with notable variations in model performance. With an accuracy of 0.9732, the DNN model excelled, while the GRU model had the lowest accuracy of 0.8958. Moving to the deployment phase, we integrated the trained model into a real-time monitoring system deployed on Android devices. Evaluation of the deployed model highlighted its superior performance in terms of runtime, memory, and CPU consumption compared to existing solutions while maintaining high detection accuracy. The effectiveness of DeepMetaDroid is derived from its capacity to achieve high accuracy with fewer resources. Future work will concentrate on improving the suggested methodology by adding more dynamic features and real-time scenarios to the metadata features. As a result, DeepMetaDroid will be even more powerful and versatile in thwarting new and emerging Android malware threats.

Data availability

The datasets generated during and/or analyzed during the current study are available from the corresponding author upon reasonable request.

Conflict of interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Securelist. *IT threat evolution in Q3 2022. Mobile statistics*. 2021. Available from: https://securelist.com/it-threat-evolution-in-q3-2022-mobile-statistics/107978/[Accessed 1st June 2023].

[2] Statista. *Mobile OS market share worldwide 2009-2023*. 2023. Available from: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ [Accessed 1st June 2023].

[3] Kaspersky. *Mobile security: Android vs IOS-Which one is safer?* 2023. Available from: https://www.kaspersky.co.in/resource-center/threats/android-vs-iphone-mobile-security [Accessed 1st June 2023].

[4] Tchakounté F, Ngassi RC, Kamla VC, Udagepola KP. LimonDroid: A system coupling three signature-based schemes for profiling Android malware. *Iran Journal of Computer Science*. 2021; 4: 95-114.

[5] Martín I, Hernández JA, De Los Santos S. Machine-learning based analysis and classification of Android malware signatures. *Future Generation Computer Systems*. 2019; 97: 295-305. Available from: https://doi.org/10.1016/j.future.2019.03.006.

[6] Sandeep HR. Static analysis of android malware detection using deep learning. *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*. Madurai, India: IEEE; 2019. p.841-845. Available from: https://doi.org/10.1109/ICCS45141.2019.9065765.

[7] Wang Z, Li K, Hu Y, Fukuda A, Kong W. Multilevel permission extraction in android applications for malware detection. *2019 International Conference on Computer, Information and Telecommunication Systems (CITS)*. Beijing, China: IEEE; 2019. p.1-5. Available from: https://doi.org/10.1109/CITS.2019.8862060.

[8] Ullah F, Ullah S, Srivastava G, Lin JC. Droid-MCFG: Android malware detection system using manifest and control flow traces with multi-head temporal convolutional network. *Physical Communication*. 2023; 57: 101975.

[9] Mahindru A, Sangal AL. MLDroid-framework for Android malware detection using machine learning techniques. *Neural Computing and Applications*. 2021; 33(10): 5183-5240.

[10] Bhat P, Behal S, Dutta K. A system call-based android malware detection approach with homogeneous & heterogeneous ensemble machine learning. *Computers & Security*. 2023; 130: 103277.

[11] Apktool: Apktool. 2023. Available from: https://ibotpeaches.github.io/Apktool/ [Accessed 5th June 2023].

[12] Skylot. *Skylot/jadx: Dex to java decompiler*. 2023. Available from: https://github.com/skylot/jadx [Accessed 9th June 2023].

[13] Dex2jar. 2023. Available from: https://sourceforge.net/projects/dex2jar/ [Accessed 8th June 2023].

[14] Android Studio. *Logcat command-line tool*. 2023. Available from: https://developer.android.com/studio/command-line/logcat [Accessed 10th June 2023].

[15] Androguard. *Androguard/androguard: Reverse engineering and Pentesting for Android Applications*. 2023. Available from: https://github.com/androguard/androguard [Accessed 10th June 2023].

[16] Amer E, El-Sappagh S. Robust deep learning early alarm prediction model based on the behavioral smell for android malware. *Computers & Security*. 2022; 116: 102670.

[17] Pitolli G, Laurenza G, Aniello L, Querzoni L, Baldoni R. MalFamAware: Automatic family identification and malware classification through online clustering. *International Journal of Information Security*. 2021; 20: 371-386.

[18] Maryam A, Ahmed U, Aleem M, Lin JC, Arshad IM, Iqbal MA. Chybridroid: A machine learning-based hybrid technique for securing the edge computing. *Security and Communication Networks*. 2020; 2020: 1-4.

[19] Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*. 2019; 22(2): 1-34.

[20] Rehman ZU, Khan SN, Muhammad K, Lee JW, Lv Z, Baik SW, et al. Machine learning-assisted signature and heuristic-based detection of malwares in Android devices. *Computers & Electrical Engineering*. 2018; 69: 828-841.

[21] Lê NC, Nguyen TM, Truong T, Nguyen ND, Ngô T. A machine learning approach for real time android malware detection. *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)*. Ho Chi Minh City, Vietnam: IEEE; 2020. p.1-6. Available from: https://doi.org/10.1109/RIVF48685.2020.9140771.

[22] Islam R, Sayed MI, Saha S, Hossain MJ, Masud MA. Android malware classification using optimum feature selection and ensemble machine learning. *Internet of Things and Cyber-Physical Systems*. 2023; 3: 100-111.
[23] Rathore H, Nandanwar A, Sahay SK, Sewak M. Adversarial superiority in android malware detection: Lessons from reinforcement learning based evasion attacks and defenses. *Forensic Science International: Digital Investigation*. 2023; 44: 301511.
[24] Li Y, Xiong K, Chin T, Hu C. A machine learning framework for domain generation algorithm-based malware detection. *IEEE Access*. 2019; 7: 32765-32782.
[25] Syrris V, Geneiatakis D. On machine learning effectiveness for malware detection in Android OS using static analysis data. *Journal of Information Security and Applications*. 2021; 59: 102794.
[26] Elayan ON, Mustafa AM. Android malware detection using deep learning. *Procedia Computer Science*. 2021; 184: 847-852.
[27] Karbab EB, Debbabi M, Derhab A, Mouheb D. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*. 2018; 24: S48-S59.
[28] Pektaş A, Acarman T. Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing*. 2020; 24: 1027-1043.
[29] Mbunge E, Muchemwa B, Batani J, Mbuyisa N. A review of deep learning models to detect malware in Android applications. *Cyber Security and Applications*. 2023; 1: 100014.
[30] Ko JS, Jo JS, Kim DH, Choi SK, Kwak J. Real time android ransomware detection by analyzed android applications. *2019 International Conference on Electronics, Information, and Communication (ICEIC).* Auckland, New Zealand: IEEE; 2019. p.1-5. Available from: https://doi.org/10.23919/ELINFOCOM.2019.8706349.
[31] Agman Y, Hendler D. *BPFroid: Robust real time Android malware detection framework*. arXiv preprint arXiv: 2105.14344. 2021. Available from: https://doi.org/10.48550/arXiv.2105.14344.
[32] Iqbal S, Zulkernine M. SpyDroid: A framework for employing multiple real-time malware detectors on Android. *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. Nantucket, MA, USA: IEEE; 2018. p.1-8. Available from: https://doi.org/10.1109/MALWARE.2018.8659365.
[33] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens CE. Drebin: Effective and explainable detection of android malware in your pocket. *InNdss*. 2014; 14: 23-26.
[34] Virus Total. 2023. Available from: https://www.virustotal.com/gui/home/upload [Accessed 11th June 2023].
[35] Contagio Mobile. 2023. Available from: http://contagiominidump.blogspot.com/ [Accessed 2ed June 2023].
[36] *Android apps on Google Play*. Google; 2023. Available from: https://play.google.com/store/ [Accessed 1st June 2023].
[37] Martín I, Hernández JA, Muñoz A, Guzmán A. Android malware characterization using metadata and machine learning techniques. *Security and Communication Networks*. 2018; 2018: 5749481. Available from: https://doi.org/10.1155/2018/5749481.
[38] Munson MA. A study on the importance of and time spent on different modeling steps. *ACM SIGKDD Explorations Newsletter*. 2012; 13(2): 65-71.
[39] Chu X, Ilyas IF, Krishnan S, Wang J. Data cleaning: Overview and emerging challenges. *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, United States; 2016. p.2201-2206. Available from: https://doi.org/10.1145/2882903.2912574.
[40] Ganapathi P, Dhathathri S, Arumugam R. Evaluation of principal component analysis variants to assess their suitability for mobile malware detection. *Advances in Principal Component Analysis*. London: IntechOpen; 2022. p.47.
[41] Kumar P, Rahman M, Namasudra S, Moparthi NR. Enhancing security of medical images using deep learning, chaotic map, and hash table. *Mobile Networks and Applications*. 2023; 1-5. Available from: https://doi.org/10.1007/s11036-023-02158-y.
[42] Li Y, Yan K. Prediction of barrier option price based on antithetic monte carlo and machine learning methods. *Cloud Computing and Data Science*. 2023; 4(1): 77-86. Available from: https://doi.org/10.37256/ccds.4120232110.
[43] Taj T, Sarkar M. A survey on embedding iris biometric watermarking for user authentication. *Cloud Computing and Data Science.* 2023; 4(2): 203-211. Available from: https://doi.org/10.37256/ccds.4220233051.
[44] Das S, Namasudra S, Deb S, Ger PM, Crespo RG. Securing IoT-based smart healthcare systems by using advanced lightweight privacy-preserving authentication scheme. *IEEE Internet of Things Journal*. 2023; 10(21): 18486-18494. Available from: https://doi.org/10.1109/JIOT.2023.3283347.
[45] Chaudhary M, Masood A. RealMalSol: Real-time optimized model for Android malware detection using efficient neural networks and model quantization. *Neural Computing and Applications*. 2023; 35(15): 11373-11388.
[46] Investigation of the android malware (CIC-InvesAndMal2019). University of New Brunswick; 2023. Available

from: https://www.unb.ca/cic/datasets/invesandmal2019.html [Accessed 7th June 2023].
[47] Thiyagarajan J, Akash A, Murugan B. Improved real-time permission based malware detection and clustering approach using model independent pruning. *IET Information Security*. 2020; 14(5): 531-541.
[48] AndroZoo. University of Luxembourg; 2023. Available from: https://androzoo.uni.lu/ [Accessed 5th June 2023].
[49] Tasyurek M, Arslan RS. RT-Droid: A novel approach for real-time android application analysis with transfer learning-based CNN models. *Journal of Real-Time Image Processing*. 2023; 20(3): 55.
[50] Zhou Y, Jiang X. Dissecting android malware: Characterization and evolution. *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE; 2012. p.95-109. Available from: https://doi.org/10.1109/SP.2012.16.
[51] Imtiaz SI, ur Rehman S, Javed AR, Jalil Z, Liu X, Alnumay WS. DeepAMD: Detection and identification of Android malware using high-efficient Deep Artificial Neural Network. *Future Generation Computer Systems*. 2021; 115: 844-856.
[52] Taheri L, Kadir AF, Lashkari AH. Extensible android malware detection and family classification using network-flows and API-calls. *2019 International Carnahan Conference on Security Technology (ICCST)*. Chennai, India: IEEE; 2019. p.1-8. Available from: https://doi.org/10.1109/CCST.2019.8888430.
[53] Lashkari AH, Kadir AF, Taheri L, Ghorbani AA. Toward developing a systematic approach to generate benchmark android malware datasets and classification. *2018 International Carnahan Conference on Security Technology (ICCST)*. Montreal, QC, Canada: IEEE; 2018. p.1-7. Available from: https://doi.org/10.1109/CCST.2018.8585560.