Hindawi
Applied Computational Intelligence and Soft Computing
Volume 2023, Article ID 2029064, 12 pages
https://doi.org/10.1155/2023/2029064

Research Article

An Energy Efficient, Robust, Sustainable, and Low Computational Cost Method for Mobile Malware Detection

Rohan Chopra, Saket Acharya, Umashankar Rawat, and Roheet Bhatnagar

Manipal University Jaipur, Jaipur, India

Correspondence should be addressed to Umashankar Rawat; umashankar.rawat@jaipur.manipal.edu

Received 30 September 2022; Revised 25 January 2023; Accepted 28 January 2023; Published 25 February 2023

Academic Editor: Aniello Minutolo

Android malware has been rising alongside the popularity of the Android operating system. Attackers are developing malicious malware that undermines the ability of malware detecting systems and circumvents such systems by obfuscating their disposition. Several machine learning and deep learning techniques have been proposed to retaliate to such problems; nevertheless, they demand high computational power and are not energy efficient. Hence, this article presents an approach to distinguish between benign and malicious malware, which is robust, cost-efficient, and energy-saving by characterizing CNN-based architectures such as the traditional CNN, AlexNet, ResNet, and LeNet-5 and using transfer learning to determine the most efficient framework. The OAT (of-ahead time) files created during the installation of an application on Android are examined and transformed into images to train the datasets. The Hilbert space-filling curve is then used to transfer instructions into pixel locations of the 2-D image. To determine the most ideal model, we have performed several experiments on Android applications containing several benign and malicious samples. We used distinct datasets to test the performance of the models against distinct study questions. We have compared the performance of the aforementioned CNN-based architectures and found that the transfer learning model was the most efficacious and computationally inexpensive one. The proposed framework when used with a transfer learning approach provides better results in comparison to other state-of-the-art techniques.

1. Introduction

The Android operating system gained popularity for smartphones and tablets from 2012 onwards. With giant companies such as Samsung, Motorola, Xiaomi, and LG adopting the Android operating system for their smartphones, it quickly became the most popular OS, hitting over 2.8 billion active users in 2021 and capturing an approximate 80% market share [1]. It is currently the biggest competitor of the Apple's iOS operating system. The availability of different application markets such as the official store, Google Play, and third-party application markets make Android devices a popular target to not only legitimate developers but also malware developers. Android is characterized by its flexible policy and allows the development of good ware as well as malware applications by imposing minimum or no restrictions on the developers. In addition, it allows the installation of third-party apps that are not present in the official Google Play Store. By taking advantage of these permissions, cybercriminals publish malware applications over the official store and target user privacy data. Moreover, it is reported that the development of malware applications has doubled each year, reaching the current count of 25 million samples and making them an imminent threat to user privacy [2].

Being an open-source operating system and widely adopted, it brings insurmountable challenges in detecting malware applications. Cybercriminals disguise malware as new or popular applications, text messages, and emails with infected hyperlinks and via malvertising that can download malware in mobile phones through advertisements. Such applications can perform actions without user acknowledgment and may steal vital information and confidential data from the user's smartphone, thus breaching their privacy. According to the reports of Kaspersky antivirus (https://www.kaspersky.co.in/blog/mobile-malware-2021/23989/) (2021), there were approximately 3.5

million malicious applications present in the Google Play Store, and the number of attacks on mobile users' muscles was 5.3 million. To mitigate the spread of malware through applications, Google introduced a mechanism called Google Bouncer to identify potential apps containing malware. The bouncer tests a submitted application in a sandbox environment for over 5 minutes to identify any suspicious behaviors; nevertheless, it is not hard to bypass this security check. Later, Google developed Google Play Protect which scans the application in real-time and stops malware applications from being published over its official store. However, Google Play Protect also failed in determining the malware applications.

These facts indicate that there is a need for additional research into efficient methods to detect zero-day Android malware in the wild to overcome these difficult challenges. The vulnerability lies in the underlying permissions of the application; hence, a user should be aware of the permissions being granted to the application. Currently, various approaches have been proposed that make use of artificial intelligence, machine learning, and deep learning algorithms to detect malware in Android smartphones.

The approaches include static, dynamic, or hybrid analysis. Static analysis extracts a large number of features such as API calls and permissions statically from the various applications. With the help of machine learning-based detection frameworks, we aim to achieve high-performance detection and an acceptable false-positive rate. Besides, to assess the solution, a reliable, large-scale, malware dataset in terms of diversity and number of malware applications is used. By contrast, dynamic analysis is a detection technique aimed at evaluating malware by executing the application in a real runtime environment. The main advantage of this technique is it helps in classifying the malware and good ware applications by recording the frequency of system calls and API calls [3].

The behavior-based/anomaly-based detection method is the most common machine learning approach, using an approach that extracts features from the dataset and creates a confusion matrix. Finally, a comparison is made between malware detection models built using different ML approaches such as least square support vector machine (LSSVM), decision trees (DT), and support vector machines (SVM), and the best result yielding model is selected [4].

Deep learning, a subfield of machine learning, has gained much attention over the years and is re-emerging as a popular practice of AI being applied in this field. The most widely used deep learning techniques for malware detection in Android systems are convolutional neural networks (CNNs) and recurring neural networks (RNNs). These techniques help in classifying the malware and good ware applications efficiently by extracting API calls and permissions and performing analysis on them [5]. Most of the studies conducted on Android malware detection evaluated the detection performance based on two parameters: accuracy and false-positive rates. The detection rates of various malwares for the years 2021 and 2022 are compared in Figure 1. However, the performance degradation of these detectors and their robustness against obfuscated malware were not considered much. Moreover,

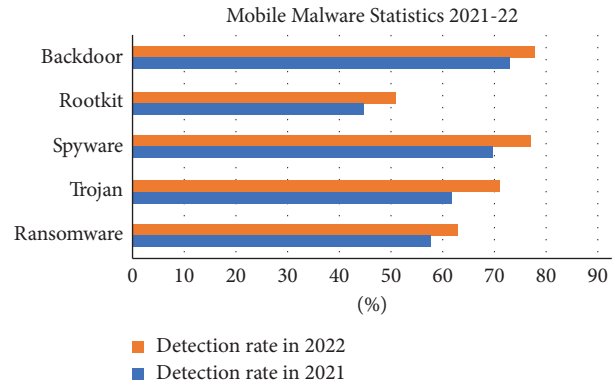

Figure 1: Mobile malware statistics for 2021-22 [6].

the key parameters such as computational cost and energy consumption were not considered while evaluating the overall performance of malware detectors. Our study fills these gaps and provides a robust, sustainable, energy-efficient, and computationally inexpensive approach for detecting malware. In a nutshell, this article makes the following contributions:

(i) Proposes a novel Android malware detection approach that satisfies the following key objectives:

(ii) Robustness: Efficiently detects zero-day and obfuscated malware that changes its behavior frequently to escape the detection process. For this purpose, we considered the PRAGuard dataset [7] which is made by obfuscating the Drebin and Malgenome datasets, respectively.

(iii) Low computational cost: Reduces the high computational costs required during the training phase for complex and big-sized datasets with the help of transfer learning.

(iv) Energy efficient solution: Reduces the energy consumption required in training, testing, and validating the dataset.

(v) Sustainability: Analyzes the overall performance of the proposed method by considering the deterioration rate of the model for the next five years.

(vi) Characterizes popular CNN-based architectures such as the traditional CNN, Alexnet, ResNet, and LeNet-5 and compares the detection accuracy of these state-of-the-art techniques with the proposed approach.

(vii) Evaluates the performance of the proposed approach in terms of robustness, computational cost, energy consumption, and sustainability.

The rest of the article is structured as follows: Section 2 presents the related work. Section 3 provides the architecture and working of the proposed method. Section 4 discusses the research questions and performance evaluation of the proposed method against them. Lastly, Section 5 provides the conclusion of the article and future directions.

2. Related Work

This section provides the study of research works conducted in the field of Android malware detection. Since the studies vary according to detection methodology, we have categorized the research works into four major segments, i.e., static and dynamic detection, learning-based detection, detection based on robustness, and detection based on sustainability.

Several strategies and solutions for detecting Android malware have been proposed by the researchers. One of the most well-known ways for evaluating static source code features to detect Android malware is the static analysis method. Among the proposed methods of detection based on static analysis methodology are those given in [8, 9]. Although this method is quick, it does not detect obfuscated malware [10]. Various solutions based on the dynamic analysis methodology have been proposed to address this issue. Instead of the static analysis method, the dynamic analysis approach runs apps in virtual emulators or on real devices to watch suspicious behavior in real-time. Bhatia and Kaushal provided a method for detecting Android malware in applications using runtime behavior analysis [11]. The authors gathered and retrieved system call traces, as well as the frequency of feature sets. To calculate the accuracy, they used the Random Forest algorithms and J48 Decision tree. Although the results were promising, their model was only trained on a smaller sample size of 50 apps.

Wong and Lie proposed an input generator, IntelliDroid, for analyzing Android applications [3]. The suggested generator offers insights for dynamic analysis tools and can be used in conjunction with them. The authors tested several inputs and were able to successfully identify malicious behavior and extract dangerous pathways.

Due to the challenges of manually detecting malicious programs, researchers have shifted to learning-based approaches to speed up and automate the detection process. Machine learning methods are used in popular research projects to find and categorize zero-day Android vulnerabilities. Deep learning techniques, unlike machine learning, identify relevant features automatically to efficiently train the detection model. Hence, there is no need for high-level domain information or human feature selection. As a result, the researchers have proposed several Android malware detection tools and methodologies based on deep learning techniques.

The authors in [12] have tried constructing a malware detection system that can help determine and examine giving applications unnecessary rights. The LSSVM (least square support vector machine) is used to create the model connected to three different kernel functions (linear, radial, and polynomial). The detection rate of the LSSVM with radial kernel was around 98.8%. The experimental setup entailed collecting features from the dataset and normalizing them using a linear transformation min-max technique. The features were trained using various machine learning algorithms in the third stage, forming a confusion matrix and calculating performance parameters such as accuracy and F-measure. Finally, multiple malware detection models were compared, and the best one was chosen.

The authors in [13] introduced a new approach called the gated recurrent unit (GRU), a recurrent neural network (RNN). They used the CICAndMal2017 dataset, extracting API calls and permissions from Android applications, and tested their approach. They outperformed many other models with a 98.2% accuracy rate. The authors used Python scripts to extract API calls and DEX permissions from malware and APKs from a public dataset. These characteristics are subsequently saved in a CSV file, which serves as a data frame for training. Finally, they ran all of the classifiers (SVM, DT, KNN, RF, and NB) to get the best results.

Almahmoud et al. [14] combined permissions, monitoring system events, API calls, and permission rates in their study to create a unique model based on a combination of four static properties. The methodology has three steps: the extraction of static features, the selection of features, and the evaluation of different classic machine learning classifiers. Their research resulted in a new architecture for a recurrent neural network (RNN) with a 98.58 percent accuracy.

The authors presented Droid-NNet, an automatic malware detection system. It has three layers: an inner layer, a hidden layer, and an outside layer. The outer layer is subjected to a threshold to evaluate whether the program is malicious or benign. For computing error and updating parameters, the authors employed a loss function such as binary cross-entropy and the adaptive moment estimation (Adam) optimizer.

Cai [15] performed research to develop a structured platform for examining the environment of mobile software on a continuous basis. The author emphasized behavioral advancement and performed large-scale research on ecosystem characterization. In addition, an ecological interplay was determined between three attributes: mobile platform, user app platform, and application users. The outcomes ensured long-term application development and security.

Suarez-Tangil and Stringhini [16] undertook a detailed investigation of Android dangerous app behavior, analyzing over 1.2 million malware samples from 1.28 K families over the past seven years. The authors used differential analysis to investigate the growth of repackaged malware, separating nonmalware components and analyzing the behavior of harmful riders. The samples were from a variety of antivirus companies.

Cai and Ryder [17] presented a longitudinal characterization study on Android applications to recognize and examine the model and execution nature of the applications. A lightweight static technique was used by researchers to observe the execution code of more than 17,000 apps over eight years. Moreover, they discovered a correlation between the functions of apps and system design. The functions heavily relied on the design of the Android system, and this dependency is increasing over time.

DroidEvolver was proposed by Xu et al. [18] to identify Android malware that can update itself without human participation. Because the model does not need to be retrained and is updated by learning techniques, the high computational cost is not required. Over six years, the authors tested 33,294 benign apps and 34,722 malicious

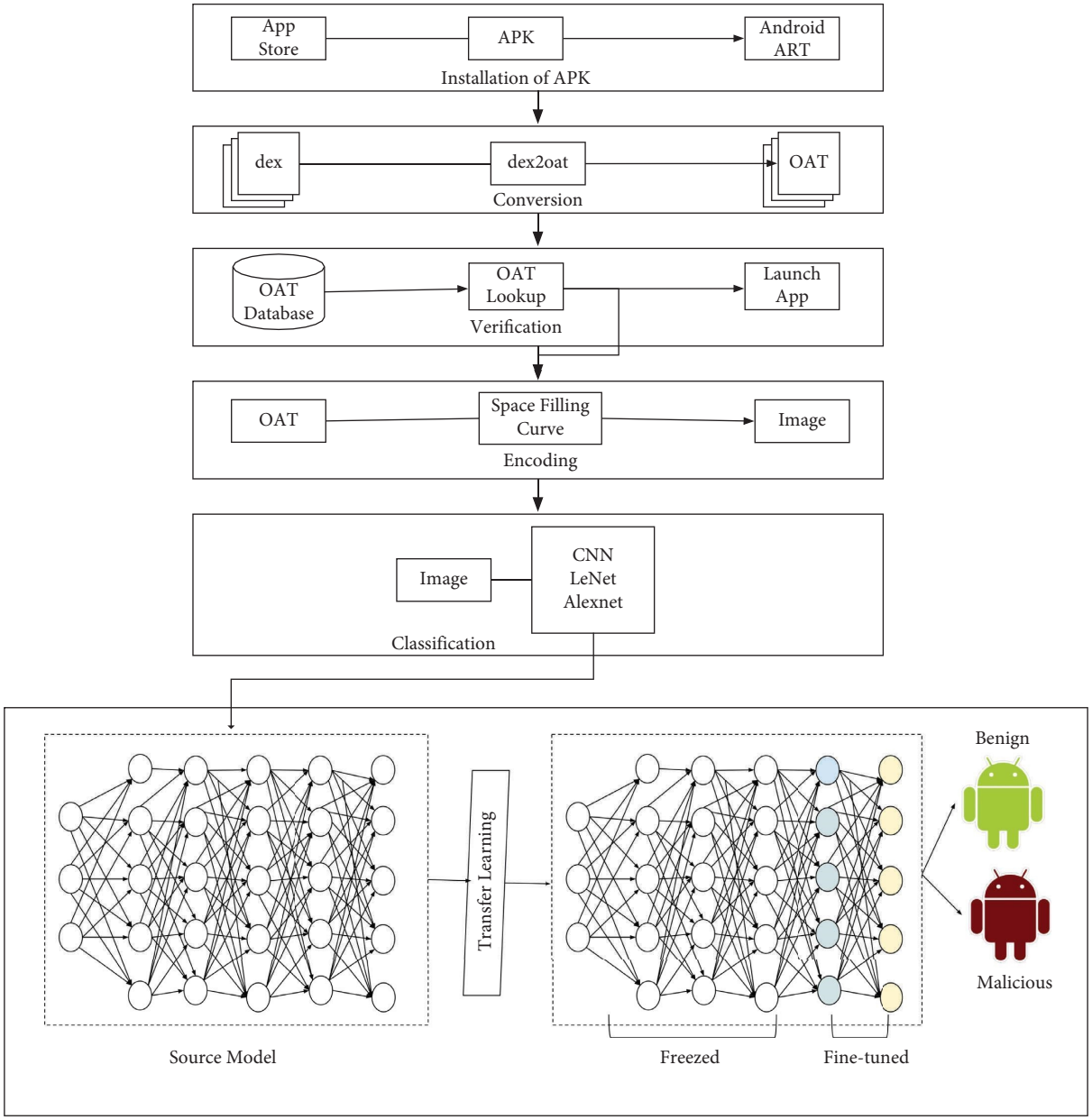

Figure 2: The proposed Android malware detection model.

apps. The authors also compared the model against the state-of-the-art MamaDroid model, surpassing the average *F*1 measure [19].

Cai [20] introduced a revolutionary classification approach called DroidSpan, a new behavioral profile for Android apps that captures the distribution of sensitive access from quick profiling. In terms of sustainability at acceptable costs, DroidSpan exceeded all baselines by 6%–32% for same-period detection and 21%–37% for over-time detection. The major conclusion, which also explains why DroidSpan is better, is that using traits that reliably distinguish malicious apps from good ones over time is crucial for long-term learning-based malware detection and that these features can be discovered through research on app evolution.

Fu and Cai [21] examined the degradation difficulties in Android malware detectors. The scientists examined four revolutionary detectors and summarised how the current solution's performance decreases over time. In addition, the authors developed a revolutionary method based on a longitudinal characterization study of application runtime characteristics. A comparison was drawn between the proposed strategy and four state-of-the-art techniques to examine the deteriorating problem. The authors in [22] introduced a new dynamic app classification technique called DroidCat to supplement existing approaches. It outperforms static and dynamic ways with 97% *F*1-measure accuracy, relying on system calls in terms of robustness by employing a variety of dynamic features focused on function calls and

(i) **Data:** Load APK(x)
(ii) X ← x
(iii) *DEX* ← *Convert*(*X*)
(iv) *OAT* ← *Convert*(*DEX*)
(v) *Install X*
(vi) *Space Filling Image* ← *Convert*(*OAT*)
(vii) *y* ← *Space Filling Image*
(viii) Classifiers ← y
(ix) **if** *condition* = *Malicious App* **then**
(x) *Generate Alert*
(xi) else
(xii) **if** *condition* = *Not Malicious App* **then**
(xiii) *Update Database for Benign App*
(xiv) end
(xv) end

Algorithm 1: Image generation.

Table 1: Datasets.

Dataset	Benign apps		Malicious apps	
	Source	#samples	#samples	#families
Drebin	[24]	9476	5560	179
Andro-AutoPsy	[25]	109000	1000	100
CCCS-CIC-AndMal-2020	[26]	200000	200000	191
Andro PRAGuard	[7]	NIL	10479	50

intercomponent communication (ICC) without needing permission or system calls while fully handling reflection.

Suarez-Tangil et al. [23] proposed DroidSieve, a static analysis-based Android malware classifier that is quick, precise, and resistant to obfuscation. It first determines whether a particular app is malicious and, if it is, categorizes it as being a member of a family of related malware. It takes advantage of features and artifacts introduced by obfuscation algorithms employed in malware that are obfuscation-invariant. The authors achieved up to 99.82% accuracy in malware detection with no false positives and 99.26% accuracy in obfuscated malware family identification.

3. Proposed Method

This section discusses the design and working of the proposed method. The primary objective of our proposed method is to provide a lightweight (in terms of computational complexity), robust, sustainable, and energy-efficient solution for detecting Android malware. To achieve this, we categorized the proposed approach into the following important phases: (a) design and detection methodology, (b) applying transfer learning, (c) energy consumption analysis, and (d) performance analysis for sustainability.

3.1. Design and Detection Methodology. The design of our proposed methodology considers native instruction execution and call traces to detect benign and malicious applications. We have considered OAT (of-ahead time) files that are created during the installation of an application. The OAT files significantly increase the execution speed and allow applications to load faster. Whenever an Android application gets installed, the Android runtime system (ART) converts the Dalvik executable (dex) file containing bytecode into an OAT file. Our method utilizes these OAT files and converts them into images for training the models. Figure 2 shows the design and working of the proposed method. For Android app execution, the ART depends on ahead-of-time compilation techniques. The ART archives each given APK by converting a dex file into an OAT file (ahead-of-time file) via one of its modules. In comparison to the JIT (just-in-time) compilation strategy, this approach is intended to enhance application performance.

3.2. Image Generation from OAT Files. The Hilbert space-filling curve is used in our study to transfer instructions into the location of pixels present within a 2-D image depending on geographical areas (https://github.com/arthursw/space-filling-curves). The Hilbert space-filling curve is a function that maps multidimensional hypercubes into sets of data. It has the virtue of traversing through all points in a given space while only stopping at each one once. As a result, it can maintain spatial information while imposing points in a linear order in a multidimensional space. This attribute is vital to our study since it retains the order of instructions in applications after they have been converted to images. We

Table 2: SQ1 evaluation results: the Drebin dataset.

Technique	Accuracy (%)	TPR	FPR
CNN	97.25	0.972	0.035
LeNet	97.63	0.977	0.027
AlexNet	98.26	0.982	0.018
Transfer learning	98.89	0.989	0.012

Table 3: SQ1 evaluation results: the Andro-AutoPsy dataset.

Technique	Accuracy (%)	TPR	FPR
CNN	97.25	0.972	0.035
LeNet	97.63	0.977	0.027
AlexNet	98.26	0.982	0.018
Transfer learning	98.89	0.989	0.012

Table 4: SQ1 evaluation results: CCCS-CIC-AndMal-2020.

Technique	Accuracy (%)	TPR	FPR
CNN	93.97	0.938	0.021
LeNet	94.73	0.948	0.019
AlexNet	95.43	0.954	0.014
Transfer learning	97.19	0.971	0.011

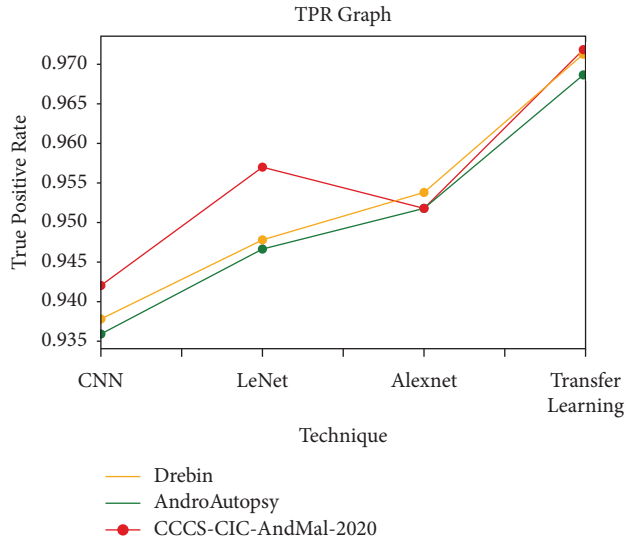

Figure 3: TPR: Drebin vs. Andro-AutoPsy vs. CCCS-CIC-AndMal-2020.

have created two subsets of data images, each with a different color scheme. For the first technique, the Hilbert curve is used to generate a fine-grained RGB palette. This guarantees that pixel values for comparable instructions are assigned based on the RGB. The second technique is more granular, relying on conjunction of entropy with the RGB red and blue components. The intensity of the aforementioned components is defined over a window of size n using the Shannon entropy. Although not as granular, entropy visualization has the advantage of highlighting encrypted content, which is

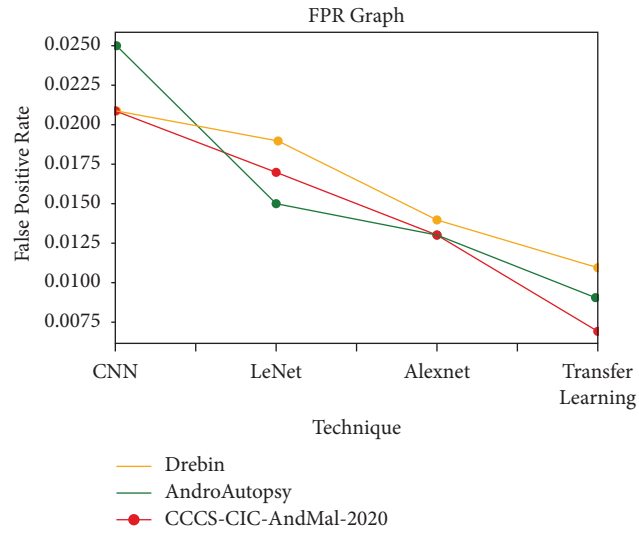

Figure 4: FPR: Drebin vs. Andro-AutoPsy vs. CCCS-CIC-AndMal-2020.

frequently an indicator of harmful content. The image generation algorithm is depicted in Algorithm 1. It can be observed that after converting the dex files to OAT files during installation, image creation becomes easy as these OAT files contain sufficient feature information, especially dynamic behaviors.

3.3. Threat Modelling. We assume that malicious applications are installed on a device using OS-supported procedures, such as downloading an APK from an application store. We assume that the device has not been subjected to any privilege assaults and that the operating system has not been hacked. Because our OAT database is local to the system, we do not consider standard database assaults in our model.

Finally, we believe that malware can impersonate a legitimate program and use delayed updates to download dangerous files. To prevent such assaults, our design double-checks the OAT files before each launch because it might be possible that the threat does not get executed during installation and may get executed later. Moreover, the OAT files are generated and executed in a batch so that the processing time gets reduced. The overall training time will increase if the OAT file processing takes high amount of time.

4. Evaluation

In this section, we will go over datasets and assessment settings. The next step is to assess our proposed strategy against the following study questions (SQs):

(i) SQ1: can the proposed approach accurately detect malware samples? (malicious application detection)

(ii) SQ2: using the proposed method, is it possible to classify malware samples with high TPR and low FPR? (familial categorization)

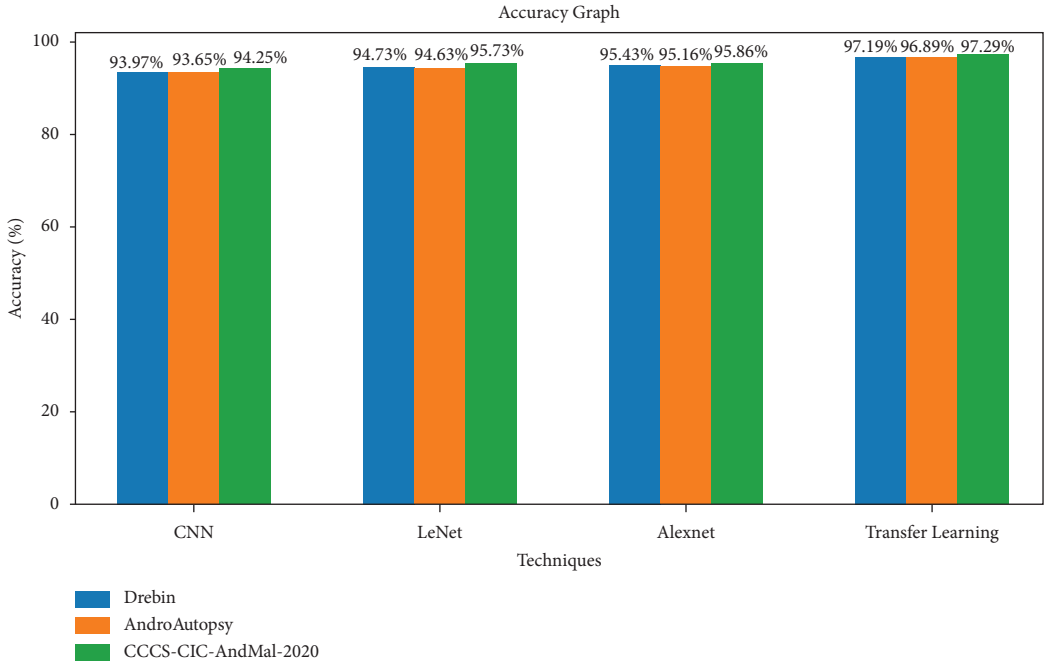

Figure 5: Accuracy: Drebin vs. Andro-AutoPsy vs. CCCS-CIC-AndMal-2020.

Table 5: SQ2 evaluation results: the Andro-AutoPsy dataset.

Technique	Accuracy (%)	TPR	FPR
CNN	93.65	0.936	0.025
LeNet	94.63	0.947	0.015
AlexNet	95.16	0.952	0.013
Transfer learning	96.89	0.969	0.009

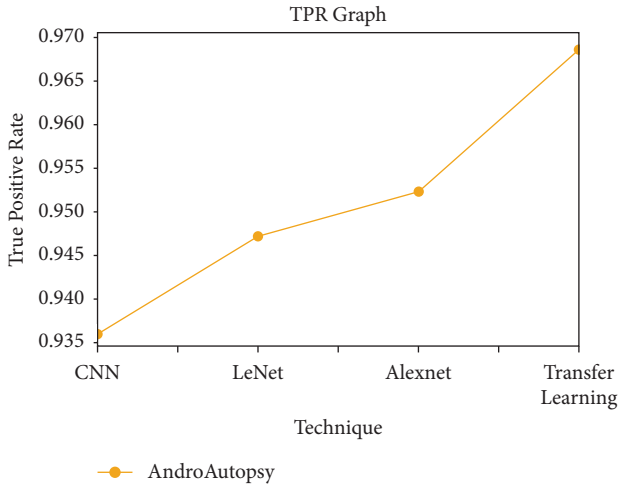

Figure 6: TPR: Andro-AutoPsy.

(iii) SQ3: is the proposed method capable of detecting malware while consuming less computational power? (inexpensive computational cost)

(iv) SQ4: is the proposed approach robust, sustainable, and energy-efficient? (robustness and sustainability)

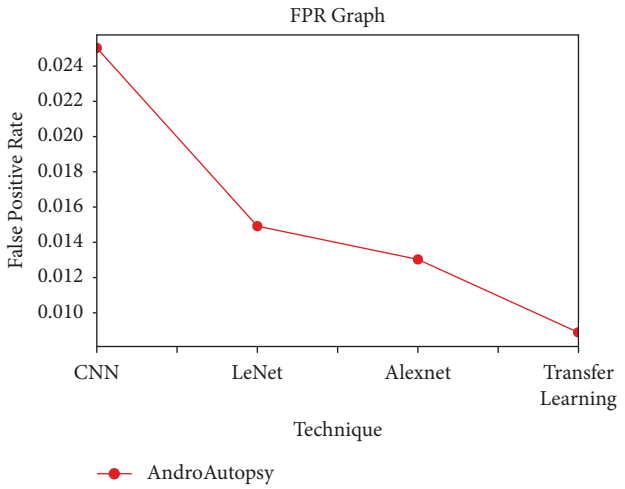

Figure 7: FPR: Andro-AutoPsy.

4.1. Datasets. To address the above-mentioned study questions, we test the proposed method against a variety of benchmark datasets. We chose four Android app datasets: Drebin, Andro-AutoPsy, Andro PRAGuard, and CCCS-CIC-AndMal-2020, respectively. Table 1 describes the datasets.

The Drebin dataset consists of 9476 benign apps and 5560 malicious apps in 179 families. There are 109193 benign and 9990 malware samples in the Andro-AutoPsy dataset, which are divided into 30 families.

With 200,000 benign and 200,000 malicious samples, the CCCS-CIC-AndMal-2020 dataset is among the most recent datasets for Android malware detection. The malicious samples are from 191 different malware families.

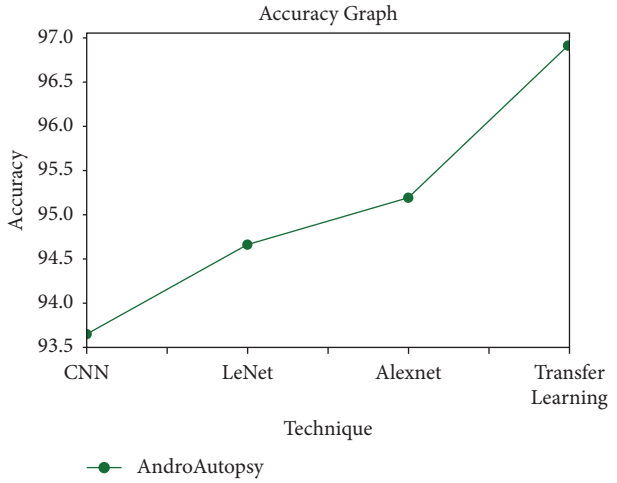

Figure 8: Accuracy: Andro-AutoPsy.

Table 6: SQ3 evaluation results.

Technique	Accuracy (%)	TPR	FPR
AlexNet	95.86	0.952	0.013
Transfer learning	91.29	0.972	0.007

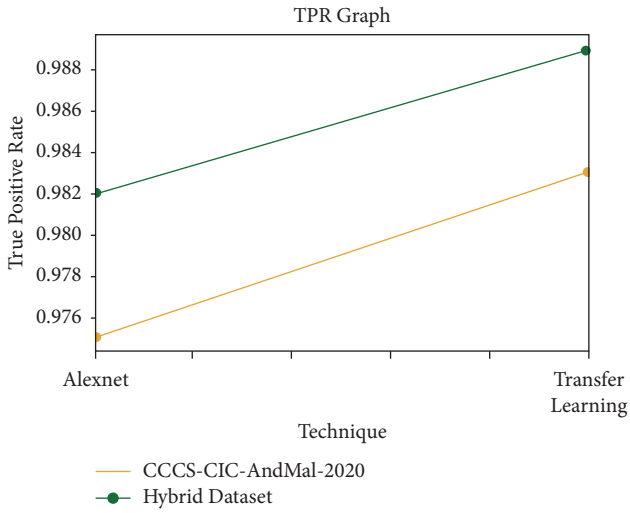

Figure 9: TPR: CCCS-CIC-AndMal-2020 vs. the hybrid dataset.

We have used a collection of obfuscated malware samples, the Android PRAGuard dataset, to test the obfuscation robustness of our proposed technique. It comprises 10479 malware samples that have been obfuscated using various obfuscation methods on Contagio and Malgenome MiniDump. It made use of obfuscation methods such as basic obfuscation, string encryption, class encryption obfuscation, and reflection as well as their permutations. Malgenome-generated obfuscated malware in Android PRAGuard is divided into 23 family labels. We have used Android PRAGuard to respond to SQs on obfuscation resilience and obfuscated malicious app classification.

4.2. Evaluation Parameters and Metrics. To assess the methodology, we employed four well-known metrics: *F score*, *accuracy*, *recall rate*, and *precision*. These parameters are defined using the formulas below:

$$\text{Precision}(a,i) = \frac{N_{ai}}{N_i},$$

$$\text{Recall}(a,i) = \frac{N_{ai}}{N_a}, \quad (1)$$

$$F_{\text{score}}(a,i) = 2\frac{\text{Precision}(a,i) * \text{Recall}(a,i)}{\text{Precision}(a,i) + \text{Recall}(a,i)}.$$

4.3. SQ1: Can the Proposed Approach Accurately Detect Malware Samples? The malware detection problem requires distinguishing between harmful and benign samples. The malware detection performance of our proposed method was initially evaluated for the techniques CNN, LeNet, and AlexNet models using the Drebin, AndroAutopsy, and CCCS-CIC-AndMal-2020 datasets, respectively. Further, we applied the transfer learning approach for malware detection. Tables below show the evaluation outcomes of the proposed method against the evaluation parameters precision, recall, F score, and accuracy, respectively.

It is clear from Tables 2–4 that the transfer learning approach outperformed other state-of-the-art models with a better detection accuracy of 98.8%. In addition, we can refer to Figures 3–5 to conclude that the transfer learning approach was more effective than other techniques.

SQ1 Answer: the proposed method is proficient in detecting the malware samples accurately.

4.4. SQ2: Using the Proposed Method, Is It Possible to Classify Malware Samples with High TPR and Low FPR? Malware is classified into various types based on its behavior, such as adware, rogueware, spyware, and trojans. We have used the Andro-AutoPsy dataset to see how effective the proposed method is at detecting malware types. Andro-AutoPsy divides its malware samples into six categories: trojan, spyware, rootkit, adware, downloader, and riskware. The evaluation results of various models are shown in Table 5. It is clear that the transfer learning model performed better than the CNN, LeNet, and AlexNet models in classifying the malicious samples with high TPR and low FPR. Also, from the Figures 6–8, we can notice significant improvement with the transfer learning approach over the Andro-AutoPsy dataset.

SQ2 Answer: The proposed method is capable of distinguishing between malicious samples of various kinds.

4.5. SQ3: Is the Proposed Method Capable of Detecting Malware While Consuming Less Computational Power? Most of the learning-based models require high-end GPUs to efficiently train the classifiers. The transfer learning approach used in our study solves this problem of requiring high computational cost for detecting malicious samples. As

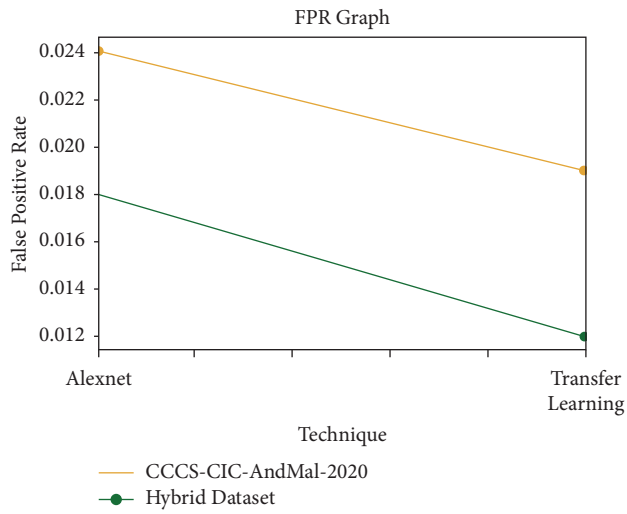

Figure 10: FPR: CCCS-CIC-AndMal-2020 vs. the hybrid dataset.

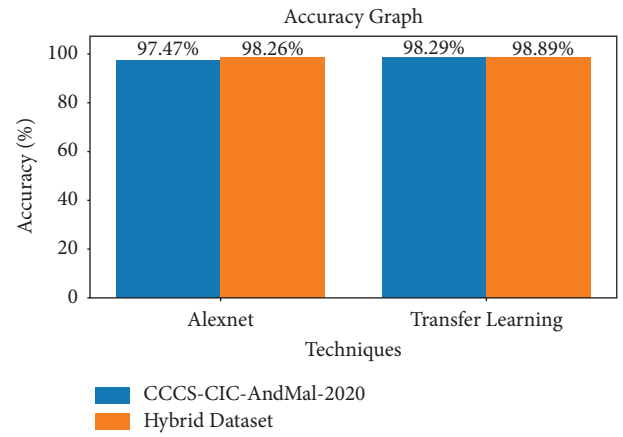

Figure 11: Accuracy: CCCS-CIC-AndMal-2020 vs. the hybrid dataset.

mentioned earlier, the transfer learning approach trains a novel target model by transferring the well-known and critical features from the source model. We have selected the source as the AlexNet model and the target as the transfer learning model. To reduce the training time and computational cost, we have frozen some layers of the AlexNet model and fine-tuned the layers which need to be transferred. Finally, we trained the AlexNet model on the CCCS-CIC-AndMal-2020 dataset and applied transfer learning to the novel model. We trained the novel transfer model on a hybrid dataset containing a subset of samples from the Drebin and Andro-AutoPsy datasets. By applying transfer learning, the model training time and high computational cost were drastically reduced as very few layers of the novel transfer model needed to be trained. The evaluation results of AlexNet and the transfer learning technique are shown in Table 6. As shown in Figures 9 and 10, transfer learning outperforms AlexNet with a low false positive rate and a high true positive rate as compared to AlexNet. From Figure 11, we can see that transfer learning performs better than AlexNet over CCCS-CIC-AndMal-2020 vs. the hybrid dataset.

4.6. SQ4: Is the Proposed Approach Robust, Sustainable, and Energy Efficient? We put our proposed solution to the robustness test by handling obfuscated samples. In this section, we observed whether our method can distinguish between malware samples that have been obfuscated using various approaches. We have used the Android PRAGuard dataset containing malware samples from the Contagio and Malgenome datasets to achieve this. The dataset has been obfuscated using a variety of techniques including class encryption, basic string encryption, obfuscation, reflection, and their permutations. To conduct a thorough study, we extracted subdatasets from Android PRAGuard. The PRAGuard Malgenome (Tm, Sm, Rm, and Cm) and PRAGuard Contagio (Tc, Sc, Rc, and Cc) datasets contain samples that have been obfuscated using trivial, string, reflection, or class encryption. The energy and runtime consumption of different CNN models with respect to LeNet have been provided in Table 7 and demonstrated in Figure 12, respectively. In general, it was observed that AlexNet (8-layered architecture) had the second-minimal effect on performance in terms of runtime and was the second-most efficient model with reference to energy consumption. The energy and runtime are with respect to the LeNet model range between 3x and 7x. Hence, the results provide support for LeNet to be the most appropriate model for mobile systems.

Table 7: SQ4 energy and runtime of different models.

Technique	Relative energy
LeNet	1
AlexNet	2.86
Other CNN models	6.95
Technique	Relative runtime
LeNet	1
AlexNet	2.86
Other CNN models	6.95

Our system comprises three relevant sources of energy and runtime running costs. These costs for performing phases include verification, encoding, and classification. The distribution of the above-mentioned overheads has been given in Table 8 and shown in Figure 13. In the design of our system, the verification phase is performed consistently when an application is launched and a common occurrence is observed. The energy and runtime consumption costs of this step were 370 ms and 1.2 J, respectively. This phase allows us to keep the extra costs minimal by pre-empting the following steps unless the application is installed or updated. It is relevant to note that this cost is constricted to the start-up cost of the application. The second step of running costs is an encoding which transforms the OAT file of an application into an image. This phase results in the highest additional costs in our design. While the

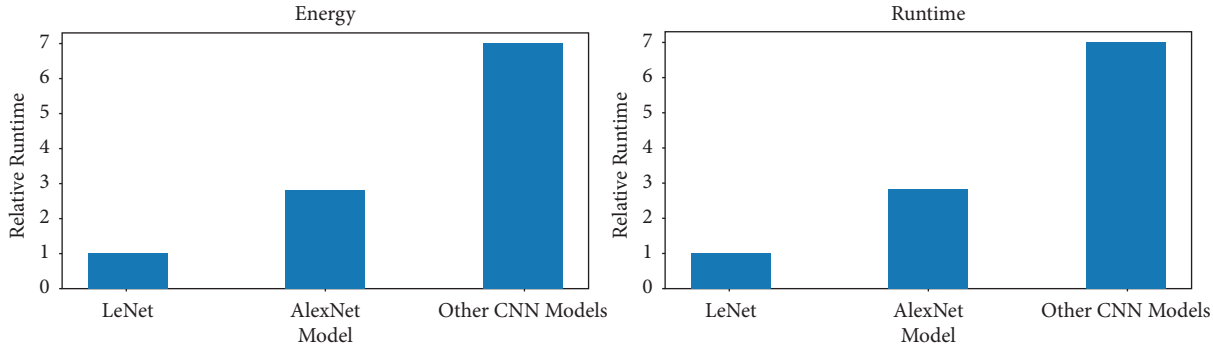

Figure 12: SQ4 graph of energy and runtime of different models.

Table 8: SQ4 energy and runtime overhead breakdown in different phases.

Phase	Energy (Joules)
Verification	1.2
Encoding	8
Classification	0.8
Phase	Runtime (seconds)
Verification	0.370
Encoding	2.5
Classification	0.335

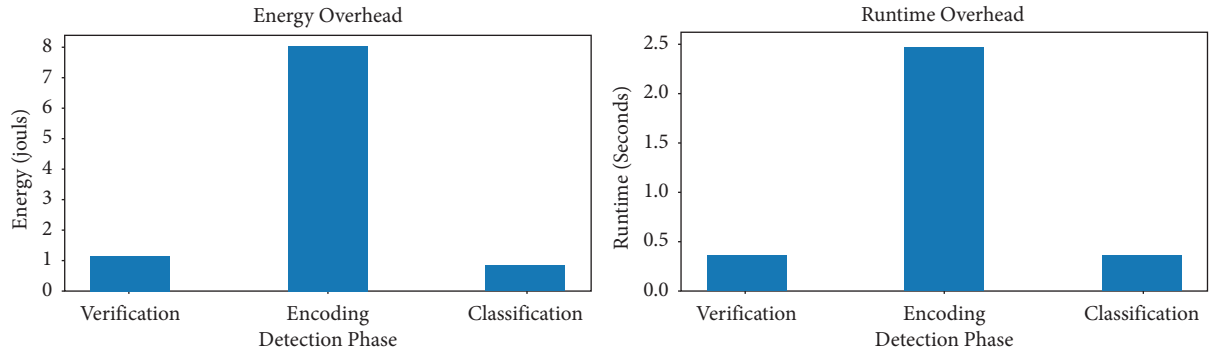

Figure 13: SQ4 graph of energy and runtime overhead breakdown in different phases.

scenario of Hilbert-based images revealed 2.5 s of runtime and 8 J of energy consumption costs, the entropy-based images required only 1.9 s of runtime and 5.6 J of energy. The latter corresponds to a 25% improvement in both energy consumption and runtime with respect to Hilbert-based images. The cost ranges between 5x and7x and is greater than the verification phase. Nevertheless, we expect this cost to incur only when new apps are installed or when applications undergo an update. The final overhead costs occur in the classification phase. The images obtained from the encoding step are classified using the CNN, or in our case, the LeNet model, as it gives the best results. We observed energy and runtime costs of 335 ms and 0.8 J, respectively. Its occurrence is similar to that of the encoding phase. Therefore, the LeNet-E is the most secure and efficient in terms of energy and runtime.

5. Conclusion and Future Work

Malware has been a part of smartphones since their inception. Applications containing malware continue to succeed in eluding security models as the popularity of Android grows. We explored how to detect and categorize Android malware using the classic CNN and transfer learning approaches in this article. The application of the CNN to malware images has become essential due to the widespread use of the convolutional neural network in image processing. A two-stage method for converting Android APKs into binary grayscale images was suggested in this article. The standard CNN model fed the OAT images as input. Further, the transfer learning strategy is applied to the trained CNN, AlexNet, and LeNet models by freezing the first few layers of these models and adjusting the learning rates accordingly to avoid the difficulties of over-fitting,

complexity, and computing expenses. The research answers various study questions based on robustness, sustainability, computational cost, and energy efficiency. It has been observed that the low computational cost transfer learning method works better than the state-of-the-art models in terms of accuracy and false-positive rates. Moreover, less number of computing resources are required as compared to the traditional models. In the upcoming study, the focus will be on combining the dynamic execution of the applications with the transfer learning method to monitor and analyze the obfuscated control flow paths of the malicious samples.

Data Availability

The data supporting the current study are available from the corresponding author upon request.The data sets used in this research are available publicly at the following links: https://www.sec.tu-bs.de/~danarp/drebin/download.html, https://ocslab.hksecurity.net/andro-autopsy/download-autopsy, https://www.unb.ca/cic/datasets/andmal2020.html, and https://www.impactcybertrust.org/dataset_view?idDataset=1339.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] T. Sharma and D. Rattan, "Malicious application detection in android—a systematic literature review," *Computer Science Review*, vol. 40, Article ID 100373, 2021.

[2] M. A. Ashawa and S. Morris, "Analysis of android malware detection techniques: a systematic review," *International Journal of Cyber-Security and Digital Forensics*, vol. 8, 2019.

[3] M. Y. Wong and D. Lie, *Intellidroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware*Vol. 16, NDSS, San Diego, CA, USA, 2016.

[4] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[5] A. Naway and Y. Li, "A review on the use of deep learning in android malware detection," 2018, https://arxiv.org/abs/1812.10360.

[6] S. Acharya, U. Rawat, and R. Bhatnagar, "A low computational cost method for mobile malware detection using transfer learning and familial classification using topic modelling," *Applied Computational Intelligence and Soft Computing*, vol. 2022, Article ID 4119500, 22 pages, 2022.

[7] V. Sihag, M. Vardhan, and P. Singh, "Blade: robust malware detection against obfuscation in android," *Forensic Science International: Digital Investigation*, vol. 38, Article ID 301176, 2021.

[8] A. Kapratwar, F. Di Troia, and M. Stamp, "Static and Dynamic Analysis of Android Malware Detection," pp. 653–662, ICISSP, Lisbon, Portugal, 2017, Master of Science.

[9] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[10] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "Dandroid: a multi-view discriminative adversarial network for obfuscated android malware detection," in *Proceedings of the tenth ACM conference on data and application security and privacy*, pp. 353–364, New Orleans, LA, USA, March 2020.

[11] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *Proceedings of the 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pp. 1–6, IEEE, London, UK, June 2017.

[12] A. Mahindru and A. Sangal, "Fsdroid:-a feature selection technique to detect malware from android using machine learning techniques," *Multimedia Tools and Applications*, vol. 80, no. 9, Article ID 13271, 2021.

[13] O. N. Elayan and A. M. Mustafa, "Android malware detection using deep learning," *Procedia Computer Science*, vol. 184, pp. 847–852, 2021.

[14] M. Almahmoud, D. Alzu'bi, and Q. Yaseen, "Redroiddet: android malware detection based on recurrent neural network," *Procedia Computer Science*, vol. 184, pp. 841–846, 2021.

[15] H. Cai, "Embracing mobile app evolution via continuous ecosystem mining and characterization," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 31–35, Seoul, Korea, July 2020.

[16] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned," 2018, https://arxiv.org/abs/1801.08115.

[17] H. Cai and B. Ryder, "A longitudinal study of application structure and behaviors in android," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2934–2955, 2021.

[18] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: self-evolving android malware detection system," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, IEEE, Stockholm, Sweden, June 2019.

[19] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: detecting android malware by building Markov chains of behavioral models," 2016, https://arxiv.org/pdf/1612.04433.pdf.

[20] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, pp. 1–28, 2020.

[21] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for android," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 272-273, IEEE, Montreal, QC, Canada, May 2019.

[22] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.

[23] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: fast and accurate classification of obfuscated android malware," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 309–320, Scottsdale, AZ, USA, March 2017.

[24] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: effective and explainable detection of android malware in your pocket," *Ndss*, vol. 14, pp. 23–26, 2014.

[25] J.-w. Jang, H. Kang, J. Woo, A. Mohaisen, and H. K. Kim, "Andro-autopsy: anti-malware system based on similarity matching of malware and malware creator-centric information," *Digital Investigation*, vol. 14, pp. 17–35, 2015.

[26] D. S. Keyes, B. Li, G. Kaur, A. H. Lashkari, F. Gagnon, and F. Massicotte, "Entroplyzer: android malware classification and characterization using entropy analysis of dynamic characteristics," in *Proceedings of the 2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, pp. 1–12, IEEE, Hamilton, ON, Canada, May 2021.