Computers and Electrical Engineering 77 (2019) 12–26

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

A novel parallel classifier scheme for vulnerability detection in Android ☆

Shivi Garg [a,b], Niyati Baliyan [a,*]

[a] Department of Information Technology, IGDTUW, Delhi, 110006, India
[b] Department of Computer Engineering, J.C. Bose University of Science and Technology YMCA, Faridabad, 121006, India

ARTICLE INFO

Article history:
Received 29 August 2018
Revised 24 April 2019
Accepted 24 April 2019
Available online 4 May 2019

Keywords:
Android
Hybrid analysis
Machine learning
Malware
Parallel classifiers

ABSTRACT

Android is one of the most commonly used mobile operating systems; however, its open-source nature and flexibility of usage attract a lot of attention from cybercriminals. In recent years, the rapid increase in malware has become a major cause of concern amongst Android users. The cybercriminals either aim to exploit confidential information from users or try to corrupt their systems by infecting them with malicious code. In order to make Android systems more secure, several malware detection techniques using static, dynamic, and hybrid analysis have been introduced in recent times; however, these techniques are inaccurate and have low efficiency. The paper not only explains how distinctive parallel classifiers can be used for detecting zero-day android malware but also addresses the oncoming highly elusive vulnerabilities. The proposed methodology combines characteristics from various parallel classifiers using expectation maximization to achieve 98.27% accuracy.

1. Introduction

Internet of Things (IoT) has emerged as a strong communication link that interconnects a large number of devices like home-appliances, smartphones, etc., with people, processes, and data, thereby allowing them to communicate seamlessly. The smartphone-enabling technologies, such as built-in sensors, Bluetooth, Radio-Frequency Identification (RFID) tracking, etc. allow it to be an integral part of the IoT world [1]. However, its use imposes severe security and privacy threats since smartphones usually contain and communicate sensitive private information. With an accelerated evolution in the field of mobile computing, smartphones have become more sophisticated in recent years. These days, smartphones are not only being used for personal conversations, but also for shopping, flight booking, Internet banking, etc., which makes users store their confidential information within smartphone applications. Therefore, a large number of new malware is being developed and introduced to steal sensitive data from users. This malware can steal personal information, encode or encrypt private data, and install fraudulent programs [2], as mentioned in Fig. 1.

There were more than 2.3 billion smartphone users globally in 2017, and by 2019, this number is likely to increase to nearly 2.7 billion [3]. Various mobile operating systems (OS) are available globally; however, Android and iOS are the most used [3]. Android scores over other mobile OS since it is open-source in nature, has a Linux-based kernel, and is a

☆ This paper is for regular issues of CAEE. Reviews processed and recommended for publication to the Editor-in-Chief by Area Editor Dr. G. Martinez Perez.

* Corresponding author.
E-mail address: niyatibaliyan@igdtuw.ac.in (N. Baliyan).

https://doi.org/10.1016/j.compeleceng.2019.04.019
0045-7906/

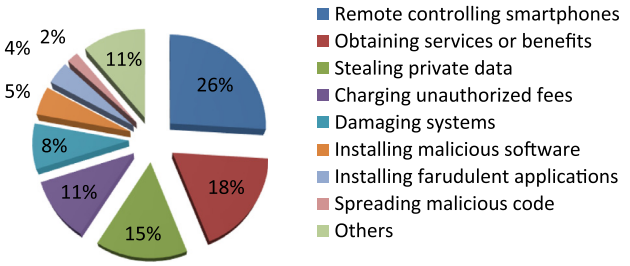

Fig. 1. Percentage distribution of various cyber crimes committed in the year 2017.

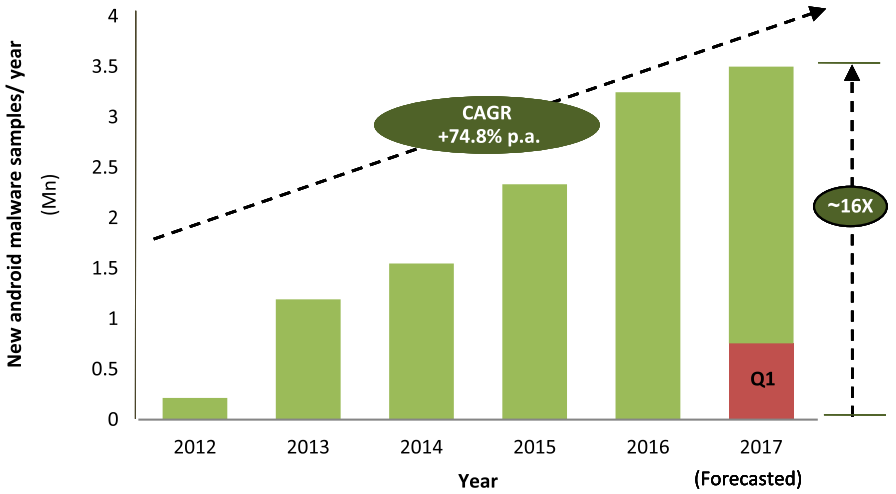

Fig. 2. Growth of Android malware.

light-weight cost-effective platform. However, these features make Android susceptible to attacks. Google Play store and third-party unreliable resources majorly contribute to the expansion of malicious applications on Android [4]. Google's 2015 report[1] claimed that almost 30 percent of all active Android phones and tablets (nearly 420 million out of 1.4 billion at that point of time) below 4.4.4 version did not receive patches. Also, 2016's report[2] mentioned that over 100 security researchers made public contribution to Android for a total of $1 million. In the year 2017, 50 malicious apps entered Google Play store that led to 4.2 million downloads[3] of the rogue program. Fig. 2 explains the growth of Android malware from the year 2013 to 2017, where the count is in millions and Compound Annual Growth Rate (CAGR) is in percentage [5].

Security in Cyber-Physical Systems (CPS) is a critical element. If proper security and privacy rules are not followed, the deployment of CPS may be hindered [6]. Therefore, various security models provide effective security for the IoT network to protect it from both external and internal threats and attacks.

1.1. Related work

With an exponential increase in Android malware in recent years, several approaches are introduced to detect the inherent vulnerabilities present in the Android OS. The first Android malware detection technique used static analysis, where individual sections of the code are analyzed without actually executing the application on an external device or an Android Emulator. This is a cost-friendly, quick and resource efficient method for finding vulnerabilities. In static analysis, two main approaches are reviewed:

(i) Signature-based detection: It is the foundation of AndroSimilar [7], which is used to detect only the samples of known malware. DroidAnalytics [8] predominately carries out collection, extraction, and analysis of malware affected Android APKs.

[1] Google, "Android security 2015 year in Review," Feb. 27, 2018. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf.

[2] Google, "Android security 2016 year in Review," Feb. 27, 2018. [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf.

[3] Liam Tung, "Android malware in Google Play racked up 4.2 M downloads: Are you a victim?," zdnet.com.[Online]. Available:https://www.zdnet.com/article/android-malware-in-google-play-racked-up-4-2-million-downloads-so-are-you-a-victim/. [Accessed: March. 15, 2018].

(ii) Permission-based detection: This method can detect unknown malware samples that remained undetected in Signature-based technique. Several tools are based on permission-based detection method, of which three were reviewed in detail. The first tool, Stowaway [9], can identify malware by tracking Application Programming Interface (API) calls. The second tool, proposed by R. Sato [10], is able to analyze AndroidManifest.xml file and calculate malignancy score. The third tool, PUMA [11] is able to take the extracted permissions into consideration to detect malware.

However, the technique that uses static analysis has a drawback, i.e., code obfuscation makes matching pattern a major problem in detecting malicious behaviour of applications, also it does not allow dynamic loading of code. The similarity score in signature-based detection technique sometimes misclassifies non-harmful apps as malicious and thus, cannot detect unknown malware types. Permission-based detection technique neither analyzes reflective calls nor identifies adware and thereby gives a high false positive rate.

The second technique uses dynamic analysis, which aims to overcome the limitations of the static analysis. In this technique, the behaviour of the application in a real-time environment is monitored. This approach checks the performance of the application in the running state, which successfully resolves the issue of dynamic code loading in static analysis. In dynamic analysis, there are three key approaches:

(i) Anomaly-based detection: It is used in various mechanisms like CrowDroid [12], Andromaly [13], and AntiMalDroid [14], etc. CrowDroid identifies malware by initiating and executing system calls in the client-server architecture. Andromaly detects malware by focusing on and observing the behavior of events in the Android application. AntiMalDroid detects malware during run-time. It generates signatures by recording inputs and corresponding outputs based on the application features.
(ii) Taint Analysis: It is particularly used by TaintDroid [15], which is based on the scientific approach known as dynamic taint analysis. This tool monitors the movement of the tainted information and performs an automatic tagging of sensitive data, such as GPS or microphone.
(iii) Emulation-based detection: It is used by DroidScope [16] which has functionalities that depend on the inspection of the OS, Dalvik Semantics and AASandbox [17]. It performs both static and dynamic analysis in the sandbox by working on the class.dex file. It breaks the file into an easily understandable form using the Monkey tool [18].

However, dynamic analysis is not feasible for battery operated devices since they consume a lot of resources (e.g., time, memory, etc.). Anomaly-based detection techniques consume time and power, and fabricate incorrect results when a legitimate app invokes more system calls. Taint analysis does not track control flow, while emulation-based detection provides limited coverage and neglects new malware. The third technique combines both static and dynamic analysis methods and is known as hybrid analysis. It extracts information and executes application simultaneously. The hybrid analysis provides better results as compared to the results obtained from static and dynamic analysis; however, it increases the time complexity of the system. The paper reviews different tools used in the hybrid analysis:

(i) Mobile Sandbox [19] works on the principle of signature-based and permission-based methods, where it examines the manifest file, user permissions, and anti-virus to identify any malicious code and then checks the behaviour of the application in the running state.
(ii) Andrubis [20] feeds the result of a static analysis, which is obtained by analyzing byte-code and AndroidManifest.xml file, as an input to the dynamic analysis which further performs taint tracing, method tracing, and system level analysis while executing the application.
(iii) SAMADroid [21], a 3-level hybrid malware detection model for Android, is an accurate and efficient solution for malware detection. However, it depends on server communication and therefore, the malicious behavior of Android APKs is detected at the remote site.
(iv) DroidRanger [22] is a heuristics scheme that detects known malware sample through permission-based filtering and behavioral foot-printing. However, it focuses only on 10 permissions for each malware family.
(v) DroidDolphin [23] obtains static and dynamic features through APIMonitor and APE_BOX, and uses SVM for classification. Its drawback is that it is easily evaded by anti-emulator techniques.

These techniques have failed to deliver efficient malware detection as both static and dynamic analysis have managed to achieve a maximum accuracy of 93%, while hybrid analysis has achieved 94% accuracy [24]. Techniques using hybrid analysis have their own limitations that inhibit them from conveying an impeccable result. Previously used techniques deploying hybrid analysis produced results that were worse than the fully static case for all the families.

The fourth technique, Machine Learning (ML), is non-conventional. ML is based on Artificial Intelligence, which allows the system to learn and adapt from experience without the need to be programmed explicitly. ML has proved to be efficient not only in computer science but also in diverse applications like – electromagnetics, circuit theory, atomic physics [25], etc. In this technique, several features from Android Package Kit (APKs) are extracted with which the dataset is trained and APKs are classified. The classifiers are programmed to detect malicious APKs from the testing dataset. ML algorithms are perceived to be more efficient than other available techniques. The paper aims to surpass the best accuracy achieved by ML to date, with the set of algorithms defined in Section 3. Previous ML approaches for malware detection in Android were not efficient and scalable [26]. The paper proposes a novel approach to provide better results and to overcome the limitations of previous ML approaches. Proposed methodology detects malware at an early stage by deploying parallel ML classifiers and their features. Table 1 provides a comparative study of existing Android malware detection techniques.

Table 1
Comparative study of Android malware detection techniques.

Approach	Technique used	Tools	Description	Limitations
Static	Signature-based	AndroSimilar [7]	Used to detect only the samples of known malware	- Classifies non-harmful apps as malicious - Cannot detect unknown malware types - Produces a high false positive rate - Does not analyse reflective calls and adware samples
		DroidAnalytics [8]	Collects, analyses, extracts the association of malware affected android APKs	
	Permission-based	Stowaway [9]	Identifies using API call track down	
		Sato [10]	Analyses the manifest file of an application and calculates the malignancy score	
		PUMA [11]	Performs malware detection by taking the extracted permissions into consideration	
Dynamic	Anomaly-based	CrowDroid [12]	Identifies vulnerabilities by initiating system calls and executing in a client-server architecture	- Consumes time and battery and fabricate incorrect results - Does not track control flow - Provides limited coverage and neglects new malware
		Andromaly [13]	Focuses on behavior and event of the application	
		AntiMalDroid [14]	Records the I/O based on the features performed in the application for the generation of a signature	
	Taint Analysis	TaintDroid [15]	Monitors the movement of data and its automatic tagging	
	Emulation-based	DroidScope [16]	Functionality depends on OS, Dalvik Semantic, and AASandbox	
Hybrid	NA	Mobile sandbox [19]	Examines permissions and AV to identify malicious code to check apps behavior in running state	- Produces worse results than the fully static case for all families - Primarily dependent on server communications - High resource consumption - Malware detection at the remote site instead of localhost - Focuses on 10 permissions for each malware family -It was easily evaded by anti-emulator techniques
	NA	Andrubis [20]	Static analysis results fed to dynamic analysis, to perform taint tracing, method tracing, and system level analysis	
	NA	SAMADroid [21]	Provides high malware detection accuracy since it combines three techniques: -Static and Dynamic Analysis; -Local and Remote Host; -Machine Learning Intelligence.	
	NA	DroidRanger [22]	Heuristics scheme that detects known malware sample based on permission-based filtering and behavioral foot-printing.	
	NA	DroidDolphin [23]	Obtains static and dynamic features through APIMonitor and APE_BOX and used SVM for classification. It could only achieve 86% efficiency.	

1.2. Innovative contributions

The approach proposed in the paper significantly differs from the methodology proposed by Suleiman, Sezer, and Muttik [27]. Their approach is limited to only static features like API calls, commands, and permissions. However, the current work extracts dynamic features along with additional static features, as described in Section 4. The paper also proposes the best combination of the most efficient ML algorithms. Consequently, upgrading the precision and recall. Hence, the accuracy of detecting Android malware is improved considerably.

Following are the innovative contributions of the paper:

- A thorough investigation of Android malware detection techniques with strengths and limitations
- A novel and effective approach for vulnerability detection in Android that combines multiple classifiers
- Relevant APK dataset collection, which is comprehensive and aligned to the scope of the research. The data is taken from Google play store,[4] Wandoujia[5] (third party), Android Malware Detection (AMD) community[6] and Androzoo[7]

[4] Google play store apps. Available: https://play.google.com/store/apps [Accessed: April. 15, 2018].
[5] Wandoujia apps. Available: http://www.wandoujia.com/apps [Accessed: April. 15, 2018].
[6] Wei, Fengguo, Yuping Li, Sankardas Roy, XinmingOu, and Wu Zhou. "Deep ground truth analysis of current android malware." In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 252-276. Springer, Cham, 2017.
[7] Allix, Kevin, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "Androzoo: Collecting millions of android apps for the research community." In Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on, pp. 468-471. IEEE, 2016.

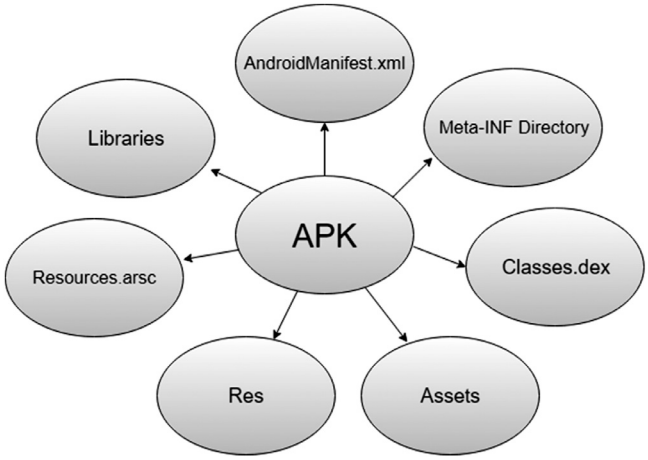

Fig. 3. De-compilation of an APK.

1.3. Organization

The rest of the paper is structured as follows: Section 2 describes the structure of Android application and explains how different application features are extracted. Section 3 presents and explains parallel classifiers in ML paradigm that are used to build malware detection model. Section 4 describes the proposed methodology and presents the performance metrics that are used to evaluate ML algorithms. Section 5 explains the experimental setup and system configuration. Section 6 lists all the results obtained from individual and parallel classifiers. Finally, Section 7 draws out the conclusion.

2. Android architecture and feature extraction

An android application has four fundamentals components that are responsible for defining its overall behavior. They are known as Activities, Broadcast Receivers, Services, and Content Providers. These components are loosely connected by the AndroidManifest.xml file, which explains the interaction amongst all components. Each component has a unique function and life process and is detailed below.

Activities (invoked via intents) control User Interface (UI) and maneuver user interaction, while Services operate background processing. Broadcast Receivers supervise communication between Android OS and applications. Content providers manage data and deal with issues of database management.

Android applications are written in Java. They are then compiled by the Android Software Development Kit (SDK) tools with data and resource files, and are archived as a file (with an extension .APK) named 'Android package'. This file is then used by an Android device to install the application. Fig. 3 shows several components of an APK.

In the paper, the methodology used to detect malware, extracts certain features from APKs, which are then used to classify an application as malicious or benign. Thus, the classification model must be effectively trained to provide correct results. These features, which are further used to train the model, are extracted using Python script from the dataset of 25,450 malware applications obtained from Google play store, Wandoujia APKs, AMD Community, and 60,000 benign applications from Androzoo.

Static features, which are extracted and used in the ML algorithms to determine the presence or absence of the malware in the Android applications, are: Permissions, API Calls, Version, Broadcast Receivers and Services, and Libraries Used. These features are illustrated as follows:

- **Permissions** – Applications in Android request permissions before they can use necessary resources and system data. Based on the application features, the permission is granted by the system or by the user.
- **API Calls** – Android applications use API Calls to interact with the functionality within various devices.
- **Version** – Version decides whether the app is capable of running on installed OS or on its lower versions.
- **Services** – They perform long-running operations running in the background with no need for user interaction.
- **Libraries Used** – Android Libraries encompass specific Java-based libraries, including application framework libraries, and those that promote graphic drawing, UI, and database access.
- **Broadcast Receivers** – Defined earlier

A sample of the features named and explained above is shown in Table 2.

Table 2
Extracted features in a CSV.

APK features	Extracted format
Name	['0d02b9d5539893efc674fbacae14944a.APK']
Permission	['android.permission.INTERNET', 'android.permission.RECEIVE_SMS', ... 'android.permission.CALL_PHONE', 'android.permission. SEND_SMS']
Version	[u'3.0']
Services	[['*my.app.client.Client*']]
Broadcast Receivers	[['*my.app.client.BootReceiver*', '*my.app.client.AlarmListener*']]
Libraries	[['*android.test.runner*']]

3. Machine learning models used with parallel classifiers

A classifier algorithm in ML maps the input data to a category. It generally produces class labels as output using a set of certain extracted features. Over many years these classification algorithms have contributed to the growth of Artificial Intelligence in multiple areas because of which they are now widely preferred to detect malware on almost all electronic mechanisms. ML can be implemented in many ways, e.g., Supervised, Unsupervised, Reinforced, and Semi-supervised learning.

The paper focuses on Supervised Learning. The training dataset has data that is labeled with either of the two classes-malicious or benign. The class for each APK is already known. This becomes the basis for constructing the classification model.

Our methodology incorporates ML algorithms like Pruning Rule-Based Classification Tree (PART), Ripple Down Rule Learner (RIDOR), Support Vector Machine (SVM) and Multilayer Perceptron (MLP). We have used these four algorithms in particular as only a few of the previous studies had used them. Since SVM has a decent accuracy as compared to other ML classifiers, it has been used a few times in the past research projects. In addition, the combination of SVM, RIDOR, PART, and MLP has never been used before in malware detection and the paper aims to improve accuracy using these parallel classifiers.

Fig. 4 explains the methodology used in the paper that combines the disparate ML parallel classifiers. One CPU core is assigned to each algorithm and then made to run simultaneously to obtain higher accuracy by producing a single classification judgment. This is done to detect malware at an early stage with the least possible errors. The scheme of parallel classifiers is followed by decision fusion in which the results from different classifiers can be combined to give better quality estimates than those obtained from any of the individual sources alone. In other words, it improves the generalization capability of individual classifiers. Given below is a brief description of the working of the ML classifiers.

3.1. Multi-Layer Perceptron

MLP is often known as a feed-forward neural network that has one or more hidden layers between the input and output layer. In feed-forward, data flows only in the forward direction, i.e., from input to the output layer. A back-propagation algorithm is used to train the network. Problems that are not linearly separable are generally solved using MLP, e.g., pattern recognition, classification, prediction, and approximation.

3.2. Support vector machine

It is a supervised ML algorithm that can solve both classification and regression problems. However, it is commonly used for classification problems. In SVM, each data item is plotted as a point in n-dimensional space, where n denotes the number of features present; and a particular coordinate is the feature value.

3.3. Pruning rule-based classification tree

It combines two basic strategies of rule learning, i.e., separate-and-conquer and divide-and-conquer. It produces decision lists, which are ordered set of rules. PART builds a partial decision tree iteratively and the largest coverage leaf is turned into a rule.

3.4. Ripple down rule learner

RIDOR is a separate-and-conquer rule classifier. It first generates a default rule and finds exceptions with the least error rate. It then finds the best exception for each exception iteratively. This approach is known as incremental reduced error pruning. The exceptions are common rules that predict classes other than the default.

4. Proposed methodology

Various steps involved in the methodology are elaborated as follows:

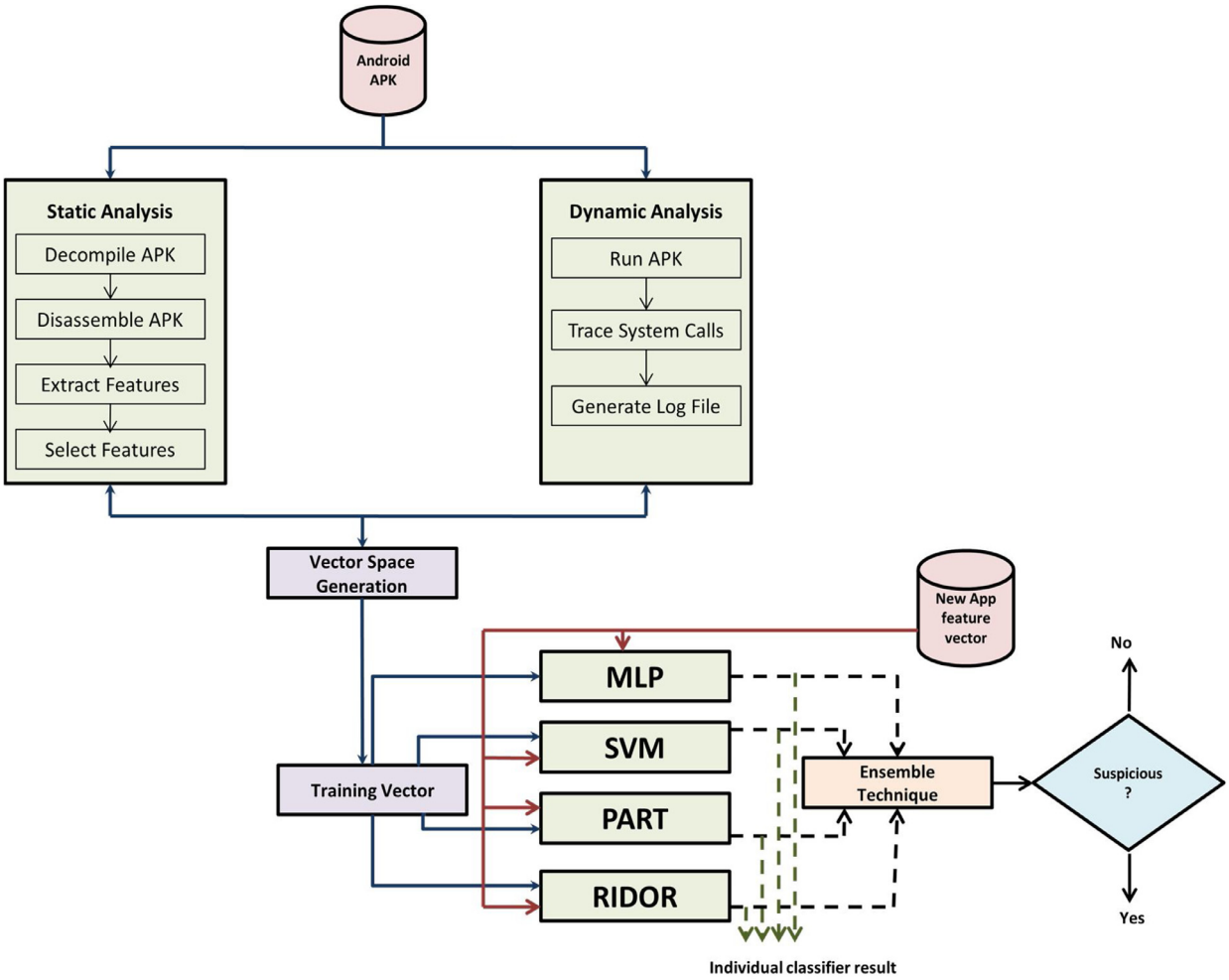

Fig. 4. Schematic representation of parallel classifier.

Table 3
The data source of Android APKs.

Data Source	No. of APKs	Benign/ Malicious
Google Play store,Wandoujia (3rd Party)	800	Benign + Malicious
AMD	24,650	Malicious
Androzoo	60,000	Benign

4.1. Input preprocessing

A dataset of ~85,000 Android applications (benign and malicious) was used from benchmarked datasets as shown in Table 3. With the help of the Apktool, we decompiled the input of APKs. Post de-compilation, we obtained the components of APKs as an outcome in a separate folder as shown in Fig. 3. A further detailed description of the dataset and pseudo codes can be found in [28].

4.2. Feature extraction

For feature extraction, we extracted the static and dynamic features from the collected dataset [29].

4.2.1. Static analysis

Android applications are in the form of an Android package, or APK archive. The APK archive includes the manifest, gradle, resources, and other folders. To extract the features that interest us, we first need to reverse engineer the APK files, which we accomplished using the Apktool. We then designed our own custom Python script to parse the decompiled folders

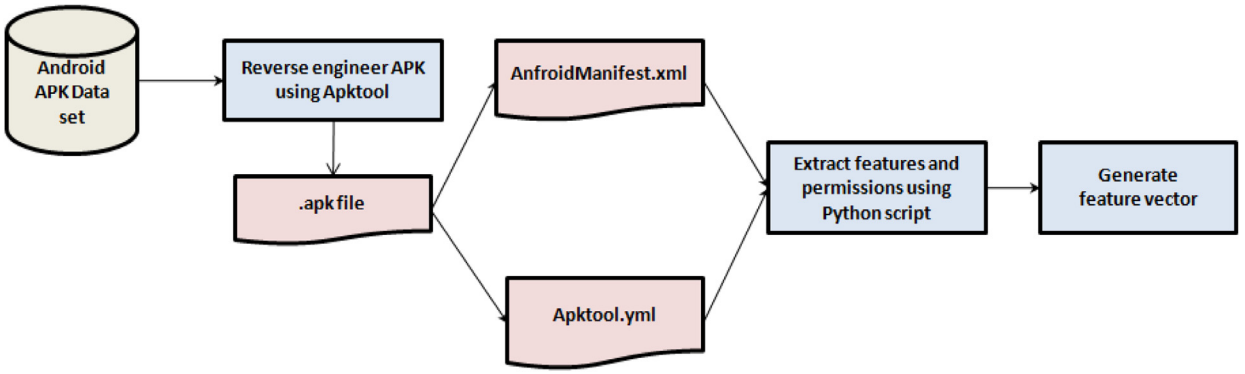

Fig. 5. Schematic diagram for static feature extraction.

and extracted the required permissions, libraries, API calls, version, and broadcast receivers from AndroidManifest.xml and Apktool.yml files.

There is a total of 135 Android permissions. A binary feature vector is constructed from the extracted features. We denote this feature vector as $F = (f_1, f_2, \ldots, f_3)$, where

$$f_i = \begin{cases} 1, & if\ the\ i^{th}\ permission\ is\ present \\ 0, & otherwise \end{cases}$$

The following steps describe the process used for feature extraction as shown in Fig. 5:

- Reverse engineer the Android applications using the Apktool
- Extract permissions and other requested features from the AndroidManifest.xml and Apktool.yml files using Python script
- Generate a binary feature vector, as shown in Eq. (1)
- Finally, create a feature vector dataset for all the APKs in the dataset and store it in .csv file format as shown in Table 1

The binary sequence 1 denotes that the permission is present while binary sequence 0 denotes that the permission is absent. Further, another variable G is introduced where $G \in \{malicious,\ benign\}$ and indicates 1 for benign application and -1 for a malicious application.

For example, the benign and malicious vectors from an APK are represented by Eqs. (1) and (2) respectively. Here, $F_{\alpha\beta\ Benign}$ and $F_{\alpha\beta\ Malicious}$ denote benign and malicious vectors, where α and β denote row and column respectively:

$$F_{\alpha\beta\ Benign} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (1)$$

$$F_{\alpha\beta\ Malicious} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (2)$$

4.2.2. Dynamic analysis

An android application communicates wth the OS using system calls, which we extract using dynamic analysis. Dynamic features like battery charging status, temperature of battery (while charging), memory, network traffic, CPU, SMS, etc., are extracted as shown in Table 4.

For dynamic analysis, we used the in-built emulator in Android studio and connect it with the ADB shell (a command line tool). It generates n number of random events (UI interactions) like clicks, touches, etc., and sends it to the system or an application that captures the system calls. We record system calls using a monitoring tool Strace [30].

Given an Android application, the following steps describe the process that we use to extract dynamic features as shown in Fig. 6:

- Create an emulator instance by opening the AVD manager and run it
- Check whether the ADB is working or not

Table 4
Extracted dynamic features.

Dynamic feature	Feature parameter
BattIsCharging	Battery amount on charging (Percentage)
BattTemperature	Battery temperature while charging (Degree Celsius)
Network traffic	Receive Packet, Transmission Packet (Bytes)
Memory	Allocated, Free, Shared (Bytes)
CPU	Usage of CPU (Percentage)
SMS	Send SMS, Receive SMS (Bytes)

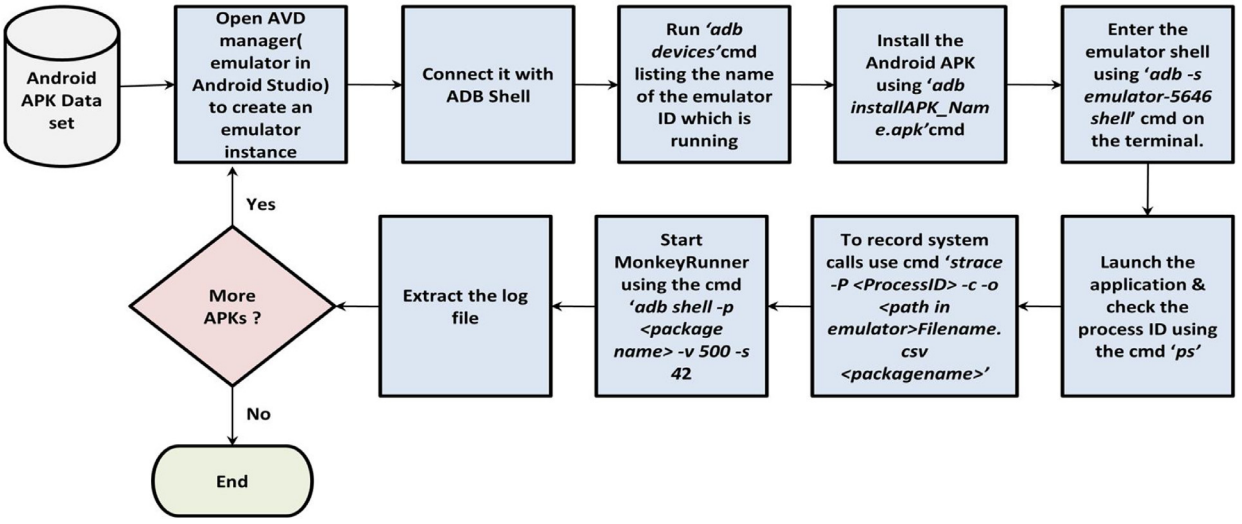

Fig. 6. Schematic diagram for dynamic feature extraction.

- Issue command *adb devices*
- Give the command *adb install APK-name.APK (*assuming the Android application is named *APK-name.APK*)
- Enter *adb -s emulator-5646 shell* command at the terminal
- Launch the application and check the process ID using the command *ps*
- Enter the *Strace* commands
- Start Monkey Runner
- Extract the log file

Let $S = (s_1, s_2, \ldots, s_n)$ be the set of all possible system calls available in the Android OS. Then element j in system call feature vector contains the count for the number of occurrences of system calls s_j. The parameter ξ is an n-length sequence that denotes the frequency of system call captured in a log file for each application.

Let $\xi = (m_1, m_2, \ldots, m_n)$, where $m_i \in S$ is i^{th} observed system call in the log file.

Feature vector obtained above is then passed to the next phase of feature extraction. Here, every attribute in the feature vector denotes the frequency of occurrence of system calls in the strace log. Using this sequence ξ, feature vector $y = [y_1, y_2, \ldots, y_{|s|}]$ is defined, where y_i is the frequency of system call $s_i \in \xi$.

We define a permission vector as a variable G, where $G \in \{malicious, benign\}$ indicates 1 for benign applications, and -1 for the malicious applications. The feature vector for system calls is obtained for benign and malicious application as per Eqs. (3) and (4) respectively.

$$F_{\alpha\beta\ Benign} = \begin{pmatrix} 0 & 0 & 0 & 0 & 2500 & 0 & 0 & 1 & 0 & 0 & 0 & 1500 & 0 & 0 & 0 & 0 & 1 & 0 & 32 \\ 0 & 0 & 36 & 0 & 0 & 0 & 0 & 0 & 753 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 90 & 0 & 56 \\ 0 & 150 & 0 & 0 & 110 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 55 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 87 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 65 & 0 & 0 & 425 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad (3)$$

$$F_{\alpha\beta\ Malicious} = \begin{pmatrix} 0 & 0 & 0 & 0 & 500 & 0 & 0 & 1 & 0 & 0 & 0 & 1500 & 0 & 0 & 0 & 0 & 1 & 0 & 32 \\ 0 & 0 & 86 & 0 & 0 & 0 & 0 & 0 & 653 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 90 & 0 & 56 \\ 0 & 150 & 0 & 0 & 110 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 55 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 887 & 0 & 0 & 0 & 1 & 0 \\ 2238 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 65 & 0 & 0 & 425 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix} \quad (4)$$

Table 5
Confusion matrix.

Predicted				
		Positive	Negative	Total
Observed	Positive	TP (a)	FN (b)	a + b
	Negative	FP (c)	TN (d)	c + d
	Total	a + c	b + d	a + b + c + d

4.3. Model training

In this phase, the pre-processed matrix obtained in the Section 4.2.1 and 4.2.2 of our proposed methodology is divided into two datasets, namely Training dataset and Testing dataset. The Training dataset is modeled using the ML algorithms described in Section 3. Thereafter, the Testing dataset is used to do a test run using the feature vectors from trained models to classify APKs and to capture their respective performance metrics.

4.4. Model evaluation

Ten-fold cross-validation technique is used to calculate the performance of the classifiers in various parallel combination strategies. As the name suggests, the dataset is divided into ten equal parts such that there is no overlapping. They can be termed as part1, part2, part3 up to part10. For the evaluation process, at every level, one portion is taken as the testing dataset and the other nine portions are supposed to administer the training model. The ML model is trained using the cross-validation training dataset and thereafter the accuracy of the model is calculated by validating the predicted results against the validation dataset. Accuracy of the ML model is estimated by averaging the accuracies derived in all the ten cases of cross-validation. The reason behind the selection of this technique is to ensure that our approach also helps identify the unknown harmful applications.

Table 5 lists the terms used in the confusion matrix that are used to express the efficiency and the efficacy of the applied method.

- *True Positive Ratio (TPR)/ Sensitivity:*

The proportion of harmful APKs classified accurately to the total number of harmful APKs present in the dataset.

$$TPR = \frac{a}{a+b}$$

- *True Negative Ratio (TNR)/ Specificity:*

The proportion of non-harmful APKs classified correctly divided by the total number of non-harmful APKs in the dataset.

$$TNR = \frac{d}{c+d}$$

- *False Positive Ratio (FPR):*

The proportion of non-harmful classified incorrectly APKs to the total number of non-harmful APKs in the dataset.

$$FPR = 1 - TNR = 1 - Specificity$$

$$FPR = \frac{c}{c+d}$$

- *False Negative Ratio (FNR):*

The proportion of harmful apps classified incorrectly to the total number of harmful apps in the dataset.

$$FNR = \frac{b}{a+b}$$

- *Accuracy (Acc):*

The proportion of correct predictions to the total number of predictions in the dataset.

$$Acc = \frac{a+d}{(a+b+c+d)}$$

- *Error ratio (Err):*

The proportion of incorrect predictions to the total number of predictions in a dataset.

$$Err = 1 - Acc$$

Table 6
System configuration.

Host machine	
Model	Dell Latitude e5250
Processor	Intel(R) Core™ i5-5300 U CPU @ 2.30 GHz 2.29 GHz
RAM	16.0 GB
System Type	64-bit Operating System
Operating System	Windows 10
Guest Machine	
Operating System Image	Ubuntu 14.04 LTS
Memory	226.0 GB
System type	32-bit Operating System
Android Emulator Configuration	
Platform	Android Studio 1.5.1
Device	Nexus 7
Target	Android 4.2.2 – API Level 17
CPU/ ABI	Intel Atom(x86)
RAM	512 MiB
SD Card	200 MiB

Table 7
Performance metrics of the individual classifiers (in%).

Algorithm	TPR	TNR	FPR	FNR	Acc	Err
MLP	96.11	95.67	4.33	3.89	95.89	4.11
SVM	93.29	94.81	5.19	6.71	94.05	5.95
PART	95.36	96.59	3.41	4.64	95.87	4.03
RIDOR	95.68	94.37	5.63	4.32	95.02	4.98

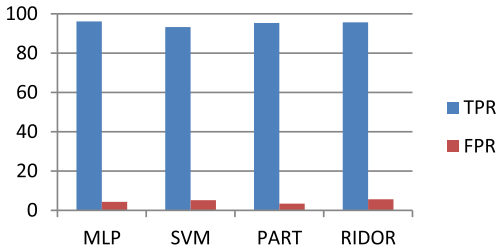

Fig. 7. Comparison of different individual classifiers.

5. Experimental setup and system configuration

Table 6 summarizes the experimental set up used for Android malware detection. The system configuration for host and guest machine (Memory requirements, OS types, etc.) is also shown in Table 6.

6. Results

Sections 6.1 and 6.2 present the summary of results obtained using different ML classifiers.

6.1. Results of individual classifiers

The first part of the research is conducted using individual classifiers that are listed and discussed in the current section. Then, using the same individual classifiers, we create a combined model, known as parallel classifiers to obtain higher accuracy. Table 7 presents the result of the individual classifiers.

It can be clearly seen from Table 7 that MLP is the best classifier amongst the four. TPR or the detection rate of MLP is 96.11% with the least FPR of 4.33% as shown in Fig. 7. The accuracy of MLP is the highest (95.89%) while the accuracy of SVM was the lowest (94.05%) as shown in Fig. 8. ROC curve for MLP (0.9589) is shown in Fig. 9.

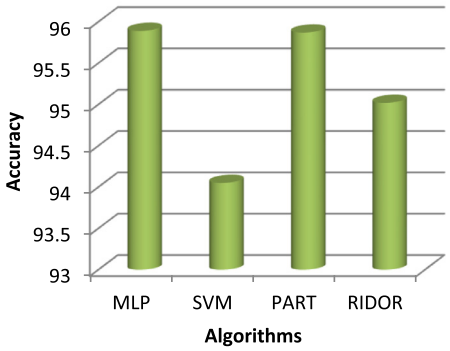

Fig. 8. Accuracy for different algorithms.

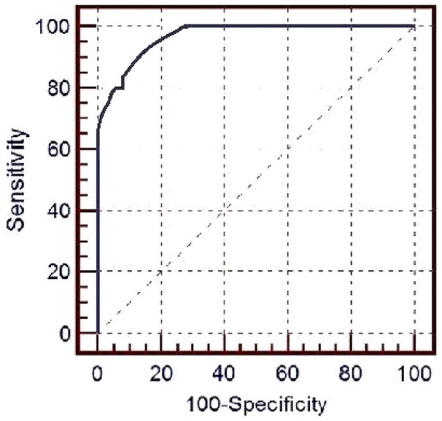

Fig. 9. ROC for high ranked individual classifier MLP.

6.2. Results of the parallel classifiers

The second part of the research is conducted using the composite model where the results (obtained from Section 6.1) are executed in a parallel manner. Four parameters are considered to estimate the efficiency of the cumulative approach. Probability of detection of malware from individual classifiers is combined to create parallel classifiers. The two classes obtained are malicious (Mal) and benign (Ben).

6.2.1. Average probabilities (AvgProb)

It is an average of the probabilities of each class. If the average of the probabilities from malicious *classMal* $[(P1+P2+P3+P4)/4]$ is greater than the average of the probabilities from benign *classBen* $[(P1+P2+P3+P4)/4]$ then the APK is categorized as malicious, otherwise benign.

6.2.2. Product of probabilities (ProdProb)

It is a product of the probabilities of each class. If the product of the probabilities from malicious *classMal*($P1 .P2 .P3 .P4$) is greater than the product of the probabilities from benign *classBen*($P1 .P2 .P3 .P4$), then the APK is categorized as malicious, otherwise benign.

6.2.3. Maximum probability (MaxProb)

It is the maximum of the probabilities for each class. If the maximum of the probabilities from malicious *classMax*[*Mal*($P1$, $P2$, $P3$, $P4$)] is greater than the maximum of the probabilities from benign *classMax*[*Ben*($P1$, $P2$, $P3$, $P4$)], then the APK is categorized as malicious, otherwise benign.

6.2.4. Majority vote (MVote)

It is the decision that is taken when an individual classifier has obtained the highest majority of the vote. If the majority votes from the classifier *MVote*($C1$, $C2$, $C3$, $C4$) = *Mal*, then the APK is categorized as malicious, otherwise benign.

Table 8 presents the result of the four different parameters used in the proposed methodology. After comparing Table 7 and Table 8, it is seen that TPR has improved in MaxProb and ProdProb.

Table 8
Performance metrics of parallel classifiers (in%).

Ensemble technique	TPR	TNR	FPR	FNR	Acc	Err
MaxProb	98.79	97.75	2.25	1.21	98.27	1.73
ProdProb	96.02	95.81	4.19	3.98	95.91	4.09
AvgProb	95.82	93.63	6.37	4.18	94.72	5.28
MVote	92.06	93.13	6.87	7.94	92.59	7.41

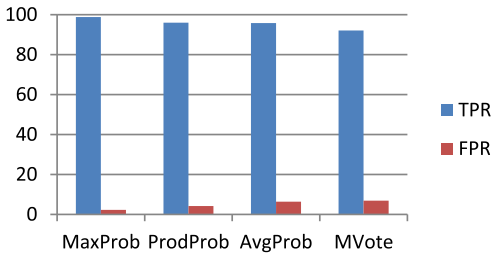

Fig. 10. Comparison of different parallel classifiers.

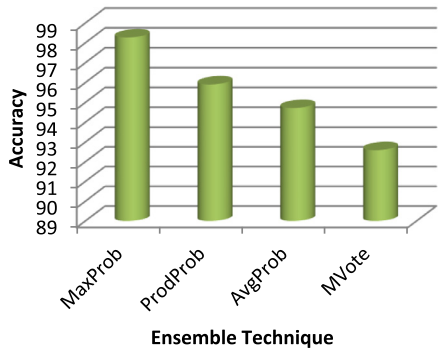

Fig. 11. Accuracy values for different algorithm combination.

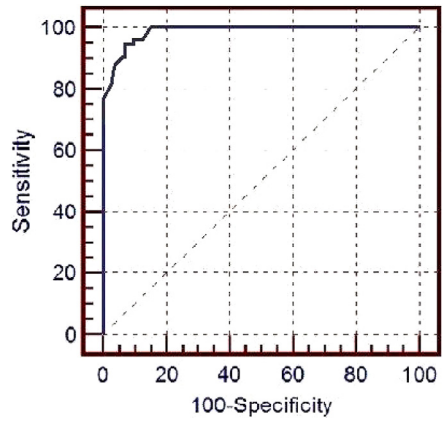

Fig. 12. ROC for high ranked parallel classifier.

The results in Table 8 validate the need for implementing the parallel ML classifiers for detecting malware in Android. Parallel ML classifiers (MaxProb and ProdProb) not only improve the accuracy of detection of malware but also help overcome the gaps and limitations of the existing techniques (described in Section 1). As per obtained results, the best parallel classifier is MaxProd and the best individual classifier is MLP. The TPR of MaxProd stands at 98.79%, which is 2.68% higher than the TPR obtained from MLP. In addition, MaxProd shows an improvement in accuracy by 2.38% and reduction in error rate by 2.38% over MLP. Fig. 10 compares TPR and FPR for all parallel classifiers. Accuracy for different parallel classifiers is shown in Fig. 11. ROC curve for MaxProd (0.9827) is shown in Fig. 12.

7. Conclusion and future scope

Android mobile platforms are susceptible to malware threats and, although previously researchers have handled such malware detection and categorization, the accuracy of their methods is not beyond improvement. Hence, the paper demonstrates the use of parallel classifiers to efficiently detect and classify malware (as malicious or benign) in Android applications. A number of static features (e.g., Permissions, API Calls, Version, Receiver Broadcast, Services and Libraries used), as well as dynamic features (e.g., battery temperature, battery charging percentage, network traffic, SMS sent and received, CPU and memory usage, etc.), were tested and correlated. The methodology proposed in the paper uses individual classifiers, namely, MLP, SVM, PART, and RIDOR and ensembles them to create a solution that is more accurate and efficient. This method not only detects the malware in the Android OS with an accuracy of 98.27% but also overcomes the gaps and limitations of the previous approaches. This methodology leverages the capabilities and strong-points inherent to each individual classifier and averages out their individual limitations. Hence, the overall method is more robust and has a lower error rate.

As part of the future work, we aim to create more parallel classifiers by taking new combinations of individual classifiers. Our goal is also to detect and predict vulnerabilities and malware using deep-learning models on a large real-world dataset. There is a strong hypothesis that the existent malware can be clustered into various families and useful patterns can be drawn from them using advanced ML models. In addition, we also aim to perform an in-detail impact analysis of Android vulnerabilities on Confidentiality, Integrity and Availability triad at the architectural level.

Conflicts of interest

The authors have no conflicts of interest to disclose.

Acknowledgements

We thankfully acknowledge all the help and support of Dr. Arun Sharma in guiding us throughout the project. We would also like to acknowledge AMD community and Androzoo for providing us with the dataset of 24,650 malware and 60,000 of benign applications respectively.

References

[1] Al-Turjman F. 5G-enabled devices and smart-spaces in social-IoT: an overview. Future Generation Comput Syst 2019;92:732–44. https://doi.org/10.1016/j.future.2017.11.035.
[2] Smith G. Mobile Malware: gGrowth stuns, storms and threatens, anewdomain.net. [Online]. Available: http://anewdomain.net/mobile-malware-growth-stuns-storms-and-threatens. [Accessed: Feb. 12, 2018].
[3] Statista Number of smartphone users in India from 2015 to 2022. The Statistics Portal 2017. [Online]. Available: https://www.statista.com Accessed:Mar. 20, 2018.
[4] Maier D, Müller T, Protsenko M. Divide-and-conquer: why android malware cannot be stopped. 9th international conference on availability, reliability and security. IEEE; 2014.
[5] González D. In Android the Malware has a free way: 350 malicious applications increase every hour, voltaico.lavozdegalicia.es. [Online]. Available: https://voltaico.lavozdegalicia.es/2017/05/android-malware-via-libre-350-aplicaciones-maliciosas-aumentan-hora/. [Accessed: Mar. 18, 2018].
[6] Song H, Fink GA, Jeschke S. Security and privacy in cyber-physical systems: foundations, principles, and applications. John Wiley & Sons; 2017.
[7] Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. AndroSimilar: robust statistical feature signature for Android malware detection. In: Proceedings of the 6th international conference on security of information and networks. ACM; 2013. p. 152–9. doi: 10.1145/2523514.2523539.
[8] Zheng M, Sun M, Lui J. Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. arXiv:1302.7212, 2013. doi:10.1109/TrustCom.2013.25.
[9] Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In Proceedings of the 18th ACM conference on computer and communications security, ACM. pp. 627–638.
[10] Sato R, Chiba D, Goto S. Detecting Android malware by analyzing manifest files. Proc Asia-Pacific Adv Netw 2013;36(23–31):17.
[11] Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Bringas PG, Álvarez G. Puma: permission usage to detect malware in android. In: International joint conference CISIS'12-ICEUTE 12-SOCO 12 special sessions. Berlin, Heidelberg: Springer; 2013. p. 289–98.
[12] Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices. ACM; 2011. p. 15–26. doi: 10.1145/2046614.2046619.
[13] Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. "Andromaly": a behavioral malware detection framework for android devices. J Intell Inf Syst 2012;38(1):161–90.
[14] Zhao M, Ge F, Zhang T, Yuan Z. AntiMalDroid: an efficient SVM-based malware detection framework for android. In: International conference on information computing and applications. Berlin, Heidelberg: Springer; 2011. p. 158–66.
[15] Enck W, Gilbert P, Seungyeop H, Tendulkar V, Byung-Gon C, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 2014;32(2):5.
[16] Yan L-K, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android Malware analysis. In: USENIX security symposium; 2012. p. 569–84.
[17] Blasing T, Batyuk L, Schmidt A-D, Camtepe SA, Albayrak S. An android application sandbox system for suspicious software detection. In: 2010 5th international conference on malicious and unwanted software (MALWARE 2010). IEEE; 2010. p. 55–62. doi: 10.1109/MALWARE.2010.5665792.
[18] Bae C, Shin S. A collaborative approach on host and network level android malware detection. Secur Commun Netw 2016;9(18):5639–50. https://doi.org/10.1002/sec.1723.
[19] Spreitzenbarth M, Freiling F, Echtler F, Schreck T, Hoffmann J. Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th annual ACM symposium on applied computing. ACM; 2013. p. 1808–15. doi: 10.1145/2480362.2480701.
[20] Barsiya TK, Gyanchandani M, Wadhwani R. Android Malware analysis: a survey paper. Int J Control Autom Commun Syst (IJCACS) 2016;1(1):35–42.
[21] Arshad S, Shah MA, Wahid A, Mehmood A, Song H, Yu H. SAMADroid: a novel 3-level hybrid malware detection model for android operating system. IEEE Access 2018;6:4321–39.
[22] Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. NDSS 2012;25(4):50–2.

[23] Wu W-C, Hung S-H. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In: Proceedings of the 2014 conference on research *in Adaptive and Convergent Systems*. ACM; 2014. p. 247–52. doi: 10.1145/2663761.2664223.
[24] Baskaran B, Ralescu A. A study of android malware detection techniques and machine learning. MAICS 2016:15–23.
[25] Mehmood A, Mukherjee M, Ahmed SH, Song H, Malik KM. NBC-MAIDS: naïve Bayesian classification technique in multi-agent system-enriched IDS for securing IoT against DDoS attacks. J Supercomput 2018:1–15. https://doi.org/10.1007/s11227-018-2413-7.
[26] Rehman Z-U, Khan SN, Muhammad K, Lee JW, ZhihanLv SW, Shah PA, Awan K, Mehmood I. Machine learning-assisted signature and heuristic-based detection of malwares in Android devices. Comput Electric Eng 2018;69:828–41.
[27] Yerima SY, Sezer S, Muttik I. Android malware detection using parallel machine learning classifiers. In: 2014 eighth international conference on next generation mobile apps, services and technologies. IEEE; 2014. p. 37–42. doi: 10.1109/NGMAST.2014.23.
[28] Garg S, Baliyan N. Data on vulnerability detection in Android. Data in Brief 2019;22:1081–7.
[29] Milosevic N, Dehghantanha A, Choo K-KR. Machine learning aided android malware classification. Comput Electric Eng 2017;61:266–74.
[30] Ham YJ, Moon D, Lee H-W, Lim JD, Kim JN. Android mobile application systems call event pattern analysis for determination of malicious attack. Int J Secur Appl 2014;8(1):231–46. doi: 10.14257/ijsia.2014.8.1.22.

Shivi Garg received her M.Tech in Information Security from Delhi Technological University, India, in 2014. She is an Assistant Professor in J.C. Bose University of Science & Technology YMCA, since September 2018. She is also a Ph.D. scholar at Indira Gandhi Delhi Technical University for Women, since August 2016. Her research interests include Cyber Security, Data Mining and Machine learning.

Niyati Baliyan is an Assistant Professor at Information Technology Department, Indira Gandhi Delhi Technical University for Women, Delhi. She received her Ph.D. from Indian Institute of Technology Roorkee. She also attained Post Graduate Certificate with Honors in Information Technology from Sheffield Hallam University, UK, as an exchange student on scholarship. Her research interests are Knowledge Engineering and Machine Learning.