Zohaib *et al. (IJECI)* 2023

International Journal for
Electronic Crime Investigation

(IJECI)
ISSN: 2522-3429 (Print)
ISSN: 2616-6003 (Online)

DOI: https://doi.org/10.54692/ijeci.2023.0704163

Research Article

Vol. 7 issue 4 Oct-Dec 2023

A Comparative Analysis of Malware Detection Methods Traditional vs. Machine Learning

Zohaib Ahmad[1], Muhammad Salman Pathan[2], and Ahsan Wajahat[3]

[1]College of Electronics and Information Engineering, Beijing University of Technology, Beijing, China

[2] School of Computer Science, National University of Ireland, Maynooth

[3]Faculty of Information Technology, Beijing University of Technology, Beijing, China

Corresponding author: ahmedzohaib03@gmail.com

Received: September 20, 2023; **Accepted:** November 08,2023; **Published:** December 22, 2023

ABSTRACT

Mobile devices have been the target of malicious software since their beginnings. Two known types of malware threats can intrude into mobile, independently injected applications and fraudulent applications that are developed to breach the security of mobile devices. Mostly these fraudulent applications access data using API calls and permission requests. API calls and permission requests are important for smooth conversation between mobile devices and database servers. This research proposes an efficient classification model that concatenates API calls and permission requests to detect malicious applications. We have used a dataset that contained more than 15 thousand Android devices' malware. We have divided data into three groups to differentiate between the malicious permission requests and malicious API calls with normal permission requests and normal API calls. To increase the probability of recognizing Android malware applications, we develop three distinct grouping strategies for selecting the most valuable API calls that are obscure, critical, and obstreperous and are chosen because Android apps extensively use several application programming interfaces (APIs). According to the results, malware applications require authorizations to access confidential information very frequently than normal Android applications do. Also, malicious Android applications raise a diverse set of API calls to access sensitive data, evidenced by malware applications making a distinct set of API calls. Our proposed method attains an F-score of 94.04%, which suggests that it is efficient at discovering mobile malware applications. Our model can be of significant assistance in conducting mobile application analysis and forensic investigations into malware.

Keywords: Permission, Genetic Algorithm, Mobile, Hybrid analysis, Dynamic analysis, Deep learning, intent, API calls

1. INTRODUCTION

There are many ways to access the internet today, and the most common method is via mobile devices. The Internet's explosive growth, combined with recent increases in automation via intelligent applications, creates a favorable environment for attackers using

malicious software. Malware threats have increased due to the global adoption of cloud computing and Internet of Things (IoT) [1]. Such malicious actions have the potential to compromise the integrity, confidentiality, or availability of mobile systems [2]. The people who make mobile malware have taken PC malware and added new features to it to make new threats to mobile platforms. Putting in place forensic identification security controls will make it less likely that digital systems will be broken into [3]. Mobile malware are virulent software that explicitly developed and considered to attack mobile devices, e.g., Smartphones and other devices [4]. Mobile malware refers to any form of harmful code or software that compromises the safety and performance of a mobile device without the knowledge or permission of the device's owner. Ransomware, Trojan horses, worms, spyware, rootkits, and botnets are all examples of different types of malware. Mobile malware is becoming more sophisticated and dangerous because it collects user data, sends premium text messages, makes calls, etc. Mobile platforms are now the primary target for cybercriminals that create malicious software, resulting in a 1,800% spike in mobile malware in 2016. Check Point did an international survey of 850 businesses and found that all of them had been attacked by mobile malware. According to Kaspersky, the number of users who have been infected with Android malware has more than tripled to 1.7 million worldwide in 2019. As can be seen in Fig. 1, the number of mobile malicious installation packages that Kaspersky found in 2021 was 3,464,756, which decreased of 2,218,938 from the previous year. The total number of mobile malware installation packages has decreased to levels roughly equivalent to those seen in 2019. The number of attacks continued to go down steadily throughout the reporting period, and in the second half of 2021 they reached their lowest monthly average in the previous two years can be seen in Fig. 2.

In the last quarter of 2017, McAfee Labs discovered 16M mobile malware [5], and Juniper reported a 400\% increase in Android malware. Over 1.05 million Android malware apps have been detected by Sophos Labs since 2010 [6]. Smartphone malware is always busy updating new features, e.g. always looking for new ways to shift into new distribution, methods and avoid detection techniques, such as obfuscation technique stealth methods, and repacking methods [7]. An old Study [8] shows that most of the Android malware used a repackaged technique they merged codes in other legitimate known applications to avoid security checkpoints.

Google Play Store is used by malware developer to download popular Android applications. Then decompile the applications, insert malicious code into the apps, and then reupload the applications with the malignant content to third-party markets for user adoption [9]. Existing mobile anti-malware applications were found to be unable to detect malware apps that have been obfuscated or repackaged.

Using ten malicious apps from six different families, researchers were able to test the effectiveness of mobile anti-malware scanners using a variety of obfuscation tactics [10]. It was then tested versus 10 reputable anti-malware scanners using these new obfuscated binaries. As per the results, there is not a single antimalware that was able to detect any malicious applications. Because there are so

many mobile apps out there, it's vital to assess and check what's available in marketplaces swiftly and intelligently [11]. An automated system that can identify and remove harmful programs from both official and unofficial markets must be established to prevent them from being downloaded. As part of [12] malicious content was introduced in Android apps resources to assess if ten anti-malware scanners can detect it. While the remainder of the anti-malware scanners were unable to discover any dangerous information in their results, just one anti-malware scanner detected two hiding tactics [13].

2. RELATED WORKS

In the last ten years, substantial advancement has been made in the study and discovery of malware in mobile devices. This section in total investigates challenges associated with the detection of mobile malware and investigates expressively related methods that have been suggested by a body of previous research. The issues surrounding mobile malware are discussed in this section identification and explores the literature's substantially related approaches. Three techniques used by security companies and researchers for extracting features from mobile applications are static, dynamic and hybrid

2.1. Static analysis

Static analysis is the name given to the study of computer programs that do not include the actual running of the program's code. Static analysis techniques and processes include those that employ analytic approaches to examine computer programs. According to the most recent findings of the research that has been conducted, it was found that there exists a variety of attributes that are utilized for the purpose of static analysis of applications in order to identify malicious applications. The primary purpose of permissions is to store all the information on the permissions necessary to execute an app in the system/mobile. Therefore, developers can investigate the behavior or intent of an application based on the permissions requested. The detecting mechanism makes use of the regularity of approvals utilized by malevolent and benign programs. It may be used alone or in conjunction with one or more additional features. Any application's .apk file's manifest file is where permissions are extracted [14]. It has contrasted the usage of regular and malicious applications using both unique requested permissions (URP) and unique utilized permissions (UUP). API calls are the second-most utilized functionality. The application's class.dex file may be used to extract these API calls. The application's API calls may be examined during the investigation process. Skeptical, and potentially harmful API calls are recognized, and apps are categorized based on this information. Droidlogger [15] employed API call blocks, which is a collection of APIs used for a certain purpose and operation, and outperformed that single API call analysis. Permissions and API are often used in combination with one another [16]. The core components of the program, like the manifest file and class.dex file, are covered by both of these. Metadata information, string searches, call graphs, hardware elements, and other aspects are made use of also, nonetheless less often than permissions and APIs.

2.2. Dynamic analysis

The term "dynamic analysis" makes mention of a set of methods and procedures that are used when analytical techniques are applied to the

research of any software in which program implementation is involved with monitoring and parallel outcomes. In dynamic analysis, the two primary approaches are in-box analysis and out-of-box analysis. To utilize such strategies, an isolated atmosphere is required in order to execute the program and see the data in real-time [17]. Vibrant analysis is more difficult than static examination, and it requires a variety of tools and expertise to observe and draw conclusions. In [18] employs system call patterns for any inquiry process. A distinctive strategy that researchers have conducted is a filtering mechanism that calls for abstraction and improved results; moreover, it substitutes mechanism calls with aliases, and created a method to detect malicious program activity based on how often system calls occur. The entire dissimilarity between weighted system calls (ADWSC) and ranked system calls utilizing a large population test are the two methods used to assess system calls (RSLPT) [19]. The analysis of numerous parameters, including average packet size, total count of packets transmitted and acquired, duration in-between packets, ratio of incoming and outgoing packets, etc., is done in order to identify malicious activity in network traffic [20]. In [21], the authors examined the HTTP flow request by making use of natural language processing for string analysis while treating the HTPP flow as a document. Other attributes Hardware resources including CPU, memory, battery, and other hardware resources are also employed as features for behavior analysis, however, they are less effective than system call and network traffic. In general, authorizations as well as APIs are utilized for static analysis, however, sometimes, they are also examined during execution. In article [22] and [23], the researchers performed a taint analysis, which is a sort of behavioral analysis that makes use of the source and sink paradigms for data flow. Finally, it has been seen that most researchers prefer network traffic and system calls as features in their dynamic analysis.

2.3. Hybrid analysis

Static and dynamic analysis are combined in the ensemble analysis. Using AAPT (Android asset packaging tool), researchers [24] used static analysis in order to acquire permission requests from the manifest file. In addition to that, they made advantage of dynamic analysis when tracing system calls with the Strace tool. One hundred and eighty-eight benign applications and one hundred and eleven malicious ones were gathered by the researchers. Static and dynamic analysis aspects were merged by the authors. For the aim of evaluating their technology, they used four different ML algorithms, with the best detection accuracy being 70.31 percent. Static analysis using the APK tool yielded 135 permissions. The top 87 permissions were retrieved by utilizing IG as a search engine. The system calls were dynamically recorded by combining an Android emulator with the Strace program. System calls were examined to see if a certain call was invoked more frequently in malignant code than in benign code. They used six different ML algorithms to see how well their approach worked. Static analysis accuracy was 0.972, and dynamic analysis accuracy was 0.884, thanks to the random forest they used. The researcher concludes that a permission-based static feature is substantially more informative than a system-based dynamic feature [25]. As a result, in this work, we decide to employ static analysis in conjunction with permissions-based features to study API calls.

3. MACHINE LEARNING

Some researchers have tried applying deep learning and ML based on API call relationships to find behavioral patterns in benign and malicious applications to develop a detection system. These efforts failed. It was reported that the authors of the paper [26] had obtained an accuracy of 96 percent on Drebin (beneficial 5.09K) dataset and AMD (benign 20.05K and malware 20.08K). UniPDroid, developed by [27], combines static analysis as well as ML methods to classify malicious software families. Throughout all, they found 15,884 harmful programs in their research. They gathered 560 features through static analysis. Meta-transformer and extra-trees classifiers were used by the authors to narrow the list of candidates. They tested their technique on 78 different malware families, using the XGBoost classifier, and got an average classification accuracy of 92%. MalDozer, a program developed by [28], studied the efficiency of API call raw sequences and deep learning algorithms in detecting malicious software. A total of 33,000 malicious programs from Drebin, MalDozer, and Malgenome as well as 38000 benign apps from Google Play Store were analyzed for API method call sequences using Dexdump by the researchers. MalDozer uses two-word embedding algorithms, GloVe and word2vec, to normalize the feature vector. The detection accuracy was between 96% and 99.6%, with a false positive rate between 0.06% and 2%. Permission requests were not considered when writing the article. We looked at permissions and the frequency with which Android platform APIs, such as classes, packages, methods, and constructors were utilized in our research. The authors conducted an extensive research on Android malware recognition using a deep learning approach. The authors used a mobile security framework to extract permissions, intent filters, incorrect certificates, and API calls from the asset folder that contained APK files (MobSF). After that, all five features were transformed into vector space. They used a neural network to test their technique on both benign and malicious software to see how well it worked. They used 80\% for training and 20% for testing, and their detection accuracy was 96.81 percent.

4. DATASETS

We use two different kinds of datasets for the evaluation experiments a benign dataset and a malicious dataset. A benign dataset contains an application that is well-intentioned, and harmless while a malware dataset contains an application that is malicious and harmful, as indicated in Fig. 5. We do the assessment tests under both data sets to see which one performs better. We use reference datasets like Contagio, VirusShare, MalShare, AndroZoo, and VirusTotal for the malware dataset. There are 5,560 malware programs in this dataset, and at least ten anti-malware products have scanned and identified them such as VirusTotal. We were unable to locate a standard benign dataset, so we decided to develop our own and run it through VirusTotal to ensure that it was complete and accurate. AndroZoo was used to gather useful apps from the PlayStore. There are 9,476 benign applications in this dataset. Since the application's data gathered in June 2021 were tested by utilizing VirusTotal, the app categories reflect this. In case all anti-malware vendors in the database found a program to be safe, we consider it to be safe as well. To remedy the issue of class imbalance, we utilized SMOTE (synthetic minority over-sampling technique).

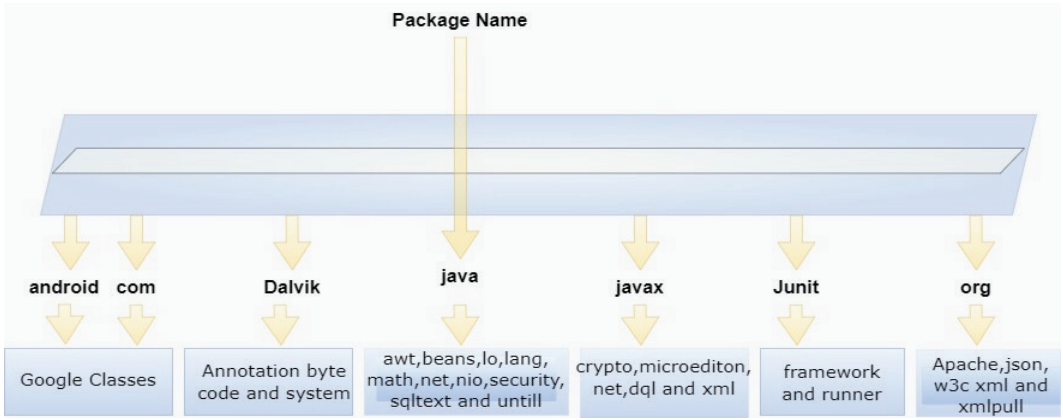

Fig 1: Architecture for ranking the API calls

Table 1: Dataset Description

Mobile Apps	Number
Malware	5,560
Benign	9,470
Total	15,036

5. METHODOLOGY

It is possible to extract and correlate the behavior of permission requests and API calls displayed by malicious programs in order to improve the preciseness of mobile malware detection. To increase possibility of discovering malicious apps, we make use of permissions analysis and analysis of the frequency of API calls. Applying the framework that we have suggested, it is possible for us to determine which API calls made by malicious Android software are the most important by using a scoring and categorizing method. Finding repackaging apps by comparing their names, hash values, or entry in a blacklist database is a fruitless endeavor. Instead, we propose a system that compares the regularity of the API calls and permissions across two programs in order to detect comparable repackaged applications. We created a three-stage analysis model for our suggested mobile malware analysis method for research purposes. Pre-processing stage, Extraction stage, and Grouping stage.

5.1. Pre-processing stage

The Java programming language is used to write Android apps, which are then converted into Java bytecode and then converted to Dalvik executable bytecode using the Dalvik virtual machine. Many files with the .class extension are created when the Java code is built. The Java source code, when compiled, results in the creation of many files along with the extension .class. The dx tool is used to combine all of the separate class files into one.dex file. An Android app's binary data is stored in the APK file. It's critical to decompile the Android app first before doing any more investigation. Android apps can be disassembled or decompiled using a variety of reverse engineering tools, including dex2jar, Apktool, Android Multitool, and the JADX. During this stage, we make use of Androguard, a tool for static analysis reverse engineering that is open source.

5.2. Extractions stage

Android SDK (software development kit) offers programmers a combination of API calls (comprising of a fundamental collection of package constructors, classes, fields, and

methods) that they can use to communicate with the operating system, software, or hardware as shown in Fig. 4. The SDK offers a wide range of APIs for developers to choose from when building an app. Malware writers can use these API calls to exploit mobile devices illegally. The same API call, for instance, may be requested by a benign or malicious program to access and receive particular data from an OS. There are several libraries included with the Android SDK as depicted in Fig. 4, including Android, Junit, and Org. The "android.jar" file in the Android SDK contains a reference to these libraries. API call features and Permission features are assigned to each Android app individually. We used the following method to extract permission requests and API calls from APK files.

A script developed in Python programming language which automatically runs and decompiles the complete dataset as follows:

1. Androguard can be used to generate all the different packages called from within an APK.
2. You can get the API call details and package level information from the entire package if it contains important methods and classes (like Java, and Android). More crucially, a few delicate API calls are shielded by Android's permission system, making them critical.
3. To extract the apps' requested permissions we used the methodology described by the authors [29], and we defined the set of all requested API calls, and all Android Permissions in the following way.

$$D_i = \{D_1, D_2, \dots, D_n\} \quad (1)$$

4. Each application should be represented in a form of a binary vector of API calls,

Where $App_i = \begin{cases} 1 \\ 0 \end{cases}$

If API is utilized in the application and if the corresponding application does not use API.

5. The association map is defined as follows D_i to map API calls to permissions P_i

$$A = \{(P, D | P \epsilon P, D \epsilon D\} \quad (2)$$

Where P is controlling the D.

6. For each permission, calculate the number of API calls and the numerical count for each API request, as follows:

$$MP = \{MP_1, MP_2, \dots, MP_n\}, \quad (3)$$

Where $MP_i = \begin{cases} 1, if \exists D_i \\ 0, if \nexists D_i \end{cases}$

$$C_i = \sum D_i | (P_i, D_i)$$

6. GROUPING STAGE

We used a grouping method to better highlight the complexities of utilization of the API calls in malignant applications in order to give comprehensive coverage of the detection performances..

7. OBSCUR

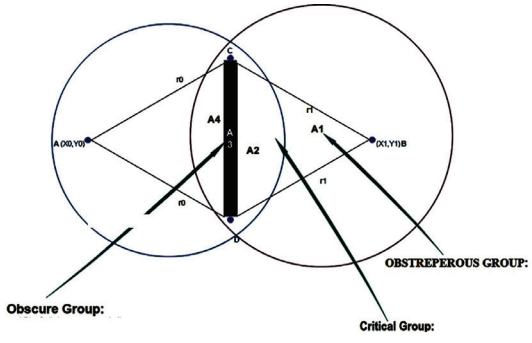

Fig 2: Architecture for ranking the API calls

This allowed us to present a high-cost coverage of the detection performances. We could create API call groups that were quite effective in spotting fraudulent apps. We designated as irrelevant API calls those calls that were in benign apps but were deemed insignificant. We concentrated on the APIs utilized by the malicious applications the most. Some of the aspects we looked at were prevalent in malicious applications. By using complementing techniques that avoid fingerprinting malware, this system aimed to classify API calls into three distinct levels and then classify them according to the level of hazard they posed. For the investigation, we used a full set of 15,036 API calls, each of which might be used by malicious programmers to carry out a variety of tasks throughout the system. Our research identified three distinct types of API requests founding malicious apps: Obscure (A3), Critical (A2), and Obstreperous (A1). We figured out which API methods were used more frequently in malware as compared to benign applications and then took the intersection of those two numbers. The ' critical API calls' category is what we term it because malicious apps frequently use these APIs rather than benign apps.

8. OBSCURE GROUP (A3)

We found intersections between API calls used by both malicious and benign applications and discovered that the total count of API calls in both benign and malicious apps was about equal. When looking at the API calls made by both malware and benign applications, the frequency with which each API request is made is also considered. For the sake of clarity, we took into consideration the frequency with which each API call is utilized by the benign applications. When it came to potentially harmful calls, we followed the same line of thinking. Thus let represent a collection of API calls which are used by malicious applications and their frequency and represent a collection of API calls which are used by benign applications and their corresponding frequency. Thus let $M=\{M1,M2,\ldots,Mj\}$ represent a collection of API calls which are used by malicious applications and their frequency and $C=\{C1,C2,\ldots,Ci\}$ represent a collection of API calls which are used by benign applications and their corresponding frequency. Thus, we isolated the obscure combination by extraction of API calls that appeared approximately equally in benign and malicious applications. In Area 3 (see Fig. 5), the calls are obscure (A3). We took a set theory method and made use of the intersection process in order to differentiate between benign and malware combinations. This process detects API requests which are frequent and similar to one another. However, a threshold value was added as a result of a shift in the frequency of API calls; because of this threshold constant, the classification of API calls as either malicious or benign and therefore belonging to obscure or critical groups is unstable. Out of a total of 15,036 calls, only 1,687 were placed in the obscure group.

9. CRITICAL GROUP (A2)

We were able to obtain the intersection of the API calls that were present in malicious applications more frequently than they were in benign applications by using our data. Since harmful apps often use these API calls rather than benign apps, we label this group "the critical API calls group. Thus let $M=\{M1,M2,\ldots Mj\}$ represent a collection of API calls which are used by malicious applications and their frequency and

let *C={C1,C2,…Ci}* represent a collection of API calls which are used by benign applications and their corresponding frequency. As illustrated in Figure 7, we can extract Area 2 (A2) by extracting the API calls which were used regularly in malicious files rather than in benign files since we know the exact time each API call appeared in both benign and harmful apps. The set-theoretic intersection process that takes place between the benign and malicious groups was implemented in the 'critical' group as we did in the first group. This was done in order to assess the level of danger posed by the 'risky' group. However, the characteristic that distinguished this group from others was that the frequency of these API calls for malicious applications was significantly higher than the frequency of API calls for clean apps. Because of this, a threshold value must be used when using the intersection operation to detect comparable API requests. API calls from benign apps are less likely to be considered active, specified, and notable even when the threshold is set at a high value. In other words, malicious API calls are increasingly common. C and D are the intersection points of the two circles (benign and malicious circles). Let A be center point of the benign circle (x_0, y_0) of radius r_0 and B be the center point of the malware circle (x_1, y_1) of radius r_1. There are three subareas at the intersection part *{A2,A3,…,A4}* that need to be calculated (see Figure 7), *{A2,A3,…,A4}* make up the right, left, and center sides of the intersection, respectively. The area A3 contains confusing API calls, while A2 contains critical API calls.

Step 1: We may compute the intersection's total area by using Area *{A2,A3,…,A4}*

Step 2: *{A2,A3,…,A4}* In order to determine the three individual subareas

$$A_2 = A_{pie}(CBD) - A_{CBD}$$
$$A_3 = A_{pie}(CD) - A_{CD}$$
$$A_4 = A_{pie}(DAC) - A_{DAC}$$

Step 3: Because the intersection is connected to the two circles, the area's angle where the circles meet (area that looks like a pie) may be stated using the following relation

$$A_{pie} * 2\pi = a * A_{circle}$$
$$A_{pie} = a * \frac{A_{circle}}{2\pi}$$
$$A_{pie} = a * \frac{\pi r^2}{2}$$
$$A_{pie} = 0.5 * a * r^2$$
$$A_{pie} = (DAC) = .5 * \overline{DAC} * r_0^2$$
$$A_{pie} = (CBD) = 0.5 * \overline{CBD} * r_1^2$$
$$A_{pie} = (CD) = 0.5 * \overline{CD}$$

Step 4: Use the cosine rule to get the angles.

$$r_0^2 = r_1^2 + AB^2 - 2 * r_1 * AB * \cos(CBA)$$

Step 5: We can compute distance AB using the coordinates of point A and point B:

$$AB = sqrt(x_1 - x_0)^2 - sqrt(y_1 - y_0)^2$$
$$\cos(BAC) = r_0^2 + AB^2 + \frac{r_1^2}{2} * r_0 * AB$$
$$\overline{BAC} = acos(r_0^2 + AB^2 + \frac{r_1^2}{2} * r_0 * AB)$$
$$\overline{ABD} = acos(r_1^2 + AB^2 + \frac{r_0^2}{2} * r_1 * AB)$$

Step 6: To get the triangles, we can compute the following: Knowing the distances and angles between two triangles

$$A_{DAC} = 0.5 * r_0^2 \sin(\overline{DAC})$$
$$A_{CBD} = 0.5 * r_0^2 \sin(\overline{CBD})$$

Step 7: We have reached at the final step where we can calculate the total area. $Area = A_1 + A_2 + A_3)$

$$A = A_{pie}(DAC) - A_{DAC} + A_{pie}(CBD) - A_{CBD} + A_{pie}(CB) - A_{CB}$$
$$A = 0.5 * (\overline{DAC}) * r_0^2 - 0.5 * r_0^2 * sin(\overline{DAC}) + 0.5 * (\overline{CBD}) * r_1^2 - 0.5 * r_0^2 * sin(\overline{CBD}) + 0.5 * (\overline{CD})$$

Following that, extraction of the linked API calls with malicious applications is allowed, as

illustrated in Fig. 7. The building of the intersection of value combinations and frequency displayed details regarding such characteristics which were present within several malware applications. When compared to the count of presence within benign applications, value combination of API calls used among the malignant applications has a higher total count. To demonstrate this idea, let's say that a particular API call was required 10 times by the concerned malware combination, but the benign set only asked it twice. As a result, given that the sample API call was found to occur more frequently in the malicious dataset, we have reason to believe that it is connected to a malicious dataset. In addition, there were only 737 potentially critical calls out of 15,036 total calls. While comparing it to the benign dataset, we see that the malicious dataset makes a significantly higher number of API calls in order to communicate with the system. For instance, the collected malware apps use the APIs for telephone controller, short message service manager, storage, system service, logs, databases, and device details often more than benign application does. This is because malware applications are designed to exploit vulnerabilities in mobile operating systems. The differential ranking of the API calls is mentioned in Table 2.

Table 3 presents a subgroup of multiple API calls that were in this set and which are utilized much regularly among malicious applications than they are in benign applications. This was necessary because of the limited amount of space available. The characteristics that malicious application employ requires critical API calls to get access to the system, according to our investigation. Examples include "*detDeviceId*" and "*getSubscriberId"* methods for stealing sensitive data such as (IMEI) and Identity (IMSI) numbers and sending them through the network using *setWifiEnabled* or *execHtpRequest*. Malware programs can be affected from techniques linked to sending messages and receiving messages (such as "*sendTextMessage," "getDefault,",* and ". *SetMessage*") according to the findings. The malware dataset, it turns out, affects obfuscation and other static analysis elimination strategies (e.g., *Cipher.getInstance*).For this reason, we hypothesized that classes like "*Getdeviceid*" and "*TelephonyManager*" could have needed additional rights to keep them safe from potentially malicious apps like "*SmsManage*" and "*SmsMessage".*Some API requests, such as "*Getdeviceid*,","*Getsubscriberid*," and "*Setwifienabled*, were already blocked by Google permissions.

10. OBSTREPEROUS GROUP (A1)

We only included APIs that were found in malicious apps and were absent from benign ones. Illustrated in Fig. 7. $C=\{C1,C2,\ldots,C_i\}$ Let C be a product of the collection of API requests which happen most frequently among benign applications is C_i while $M=\{M1,M2,\ldots,M_j\}$ is defined as API calls that appear frequently in malware programs. We compute the following to find the Obstreperous calls:

$$R = {}^{M}/_{C} \qquad (4)$$

In relation to the equation shown above, the characteristics of obstreperous calls are more obviously geared toward harmful applications. Concerning Fig. 7, in contrast to the other two categories, no explicit criterion was found for the frequency of API calls because this set is unquestionably more skewed from malignancy. Because no single particular frequency term must be met, and because there is a greater

potential for malicious API calls in most of these conditions, a previously implemented threshold is rendered meaningless, and its function is rendered moot within the context of this scenario. API generation technology such as the one described here was employed to create API requests that were then incorporated into the malicious dataset in their entirety. The outcomes of the experiments demonstrate several API calls. (i,e.

(*Lorg/w3c/dom/DOMException.getMessage*,) (*Java/lang/Thread;.setContextClassLoader,)* (*Android/content/Context;.deleteFile,)* *(Android/database/sqlite/SQLiteDatabase;query, Java/net/URL;.openConnection,)* (*sAndroid/telephony/TelephonyManager;getLine1Number)* are specifically discovered in malicious applications, not in benign applications. Among the 15,036 API calls, only 4 are found to be Obstreperous calls.

TABLE 2

DIFFERENTIAL RANKING OF THE API CALLS

API Calls Name	Meaning
Android/Telephony/ Telephonymanager;. Getnetworkoperator	To gain access to sensitive data
Java/Util/Gregoria ncalendar;. Set	To gain access to sensitive data (Current Time)
Java/Io/Bytearrayo utputstream; Reset	To gain access to se nsitive data
Java/Lang/Stringb uffer;.Insert.	For obfuscation purposes
Cipher. Getinstance ()	For obfuscation purposes
Sendtextmessage () Smsmanager () Setmessage ()	in order to send and receive SMS messages
Setwifienabled () Exechttprequest ()	For communicating over the network
Getdeviceid () Getsubscriberid ()	To gain access to sensitive data (phone's unique device ID)
RuntimeException ()	For the execution of external commands

Get Last Known Location: it communicates the device's location information to a remote site and returns the device's last known location from the specified provider. This technique is employed by the Geinimi family.

Get Line 1 Number: It sends sensitive information, such as a phone number, to a remote server as a string; we've seen this on nearly all Android devices. This is how the Fokonge family do things.

Set Context Class Loader: It can be used to dynamically load harmful software since it loads exterior classes or sources from certain repositories. Malware applications could use the Class Loader class to replace the corresponding software with malicious software to get around current countermeasures. It is quite likely that the malicious code is concealed either underneath the next route (/assets) or within the safe digital (SD) card. This strategy appears to be used by most members of Android. Steek family. Malicious software only requested permission for full Internet access once throughout the installation. It might appear as a smaller threat to possible sufferers if this malicious software just asks for permission during installation. Installing the malware on a smartphone triggers it to open and show information about any fraudulent apps that have been installed. The set Context Class Loader technique is called a similar amount of times by malicious applications that belong to the Steek family.

DOM Exception: It is possible to use it when certain events take place. In our research, we discovered that malicious apps used *Lorg/w3c/dom/DOMException)*

(Lorg/w3c/dom/DOMException;<init>.(SJava/lang/String;)V) .This method appears to be used by much of Android Steek group.

Open Connection: This method is belonging to Android. Generisk group. The group establishes linking to predetermined distant server, loads it, and then runs the code that it contains.

11. STATISTICAL ANALYSIS

Figure.3 summarizes all the attributes from both the “critical calls" as well as the “Obstreperous calls" groups that we considered in our experiment.

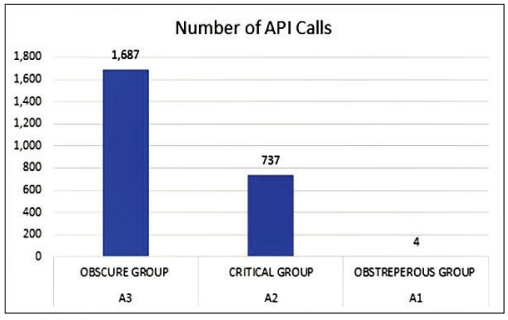

Fig 3: Distribution of Features across the Groups

In order to choose the most relevant characteristics from those that were available, we computed the IG for each one of them. Classifications of API calls can be made with reasonable ease using the primary categories. There is a variety of APIs available for each type of group. The following criteria will be used to assign a value to each feature: ‘very important’, ‘important’, ‘normal’, or ‘unimportant’.

Figure.4 demonstrates that the IG has been implemented in each feature. The score indicates the importance, in the opinion of the IG, of each of the best 12 characteristics found within the risky set. It approves that in mobile malware detection, the features chosen are very much important. In order to determine the significance of each feature to the data set that has been provided, the procedure calculates the splitting conditions regarding decision trees. Each permission's IG is determined by the formula below.

gain (c,r_i)=entropy (c)-entropy (c|r_i) C refers to the class value (i.e., malignant or benign) and is the attribute. The entropy (*c*) is the information entropy. The ideal collection of features depends on the classifier and is fewer than the total number of available features. We begin by utilizing all of the features in combination with ML techniques, and after that, we pick the attributes for evaluation from either the critical calls or Obstreperous calls.

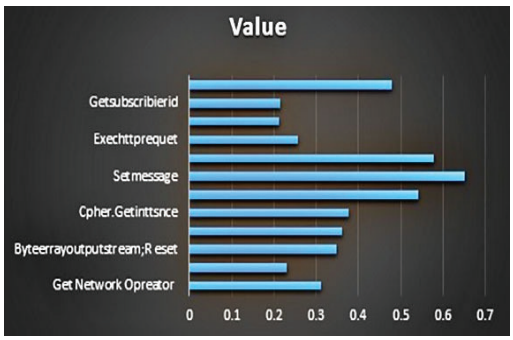

Fig 4: Information gain for the top features in the critical group

12. LEARNING-BASED DETECTION

12.1. Data normalization

Before using ML algorithms, it is critical to normalize the data. Now that we have compiled the essential characteristics (critical and Obstreperous attributes set), weights need to be assigned for those attributes and express them using a vector space.

Normalization of the term frequency (TF) was used in order to minimize situations in which the classifier has varying weights when making decisions. The represents the extracted dictionary, where the dictionary was drawn from both groups of data (critical and Obstreperous groups). A weighted vector space, $(w_{1n}, \ldots w_{wn}{}^{""}, w_{in} \in \{0, 1\}$ shows the presence or absence of a precise attribute in an app in the form of a (TF) representation. This word denotes the regularity with which the functionality can be accessed within the application. The (TF) can be scaled to values by the division of the frequency of an appearance by the amount of features within the application (0, 1). The following is a formula for calculating the $tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$ (TF) the normalization of the dataset provides for a matrix-like view of the vector representation where rows indicate application vectors and columns represent features. Performing this action enables the application of a variety of ML algorithms, and it also enables us to identify areas of similarity and difference by employing similarity-measuring algorithms.

13. EVALUATION METRICS

In order to determine the effectiveness of classification models, we have selected the following: accuracy, precision, recall, and the F-measure standard metrics. Estimates for these measures are derived from values of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

- **TP:** The count of correctly discovered malware applications is represented.
- **TN:** The count of correctly discovered benign applications is represented.
- **FP**: Count of benign apps mistakenly categorized as malware applications.
- **FN:** Count of malware an application mistakenly categorized as benign applications.

Accuracy It calculates an estimate of the proportion of successfully detected connection records relative to the total test dataset. When there is a higher level of accuracy, the ML model is considered to be superior. The accuracy is a useful measurement for the test dataset since it consists of classes that are evenly distributed, and it is explained in following manner:

$$Accuracy = {(TP + TN)}/{(TP + TN + FP + FN)}$$

Precision You can figure out the percentage of correctly identified data by doing the following calculation.

$$\boldsymbol{Precision = {TP}/{(TP + FP)}}$$

Recall You can figure out the percentage of correctly identified malicious data by doing the following calculation.

$$Recall = {TP}/{(TP + FN)}$$

F-Measure The following formula can be used to calculate the precision and recall combined

$$F\ Measure = 2 * \frac{Precision * \text{Recall}}{Precision + \text{Recall}}$$

14. RESULTS AND ANALYSIS

The primary objective is to investigate whether the considered attributes from the critical set and Obstreperous calls can be utilized to construct complicated classifiers that can forecast the classes of mobile malware, or the risk factors associated with it. After ignoring the features of the dataset that were deemed to be insignificant, all 741 different features were

collected from either the critical or Obstreperous categories. 6 ML algorithms J48, random forest (RF), k-nearest neighbors (k-NN), random tree (RT), naive Bayes (NB), Support vector machine (SVM) were used in 10- fold cross-validation for each group (critical group and Obstreperous group). The empirical findings imply that the proposed method is effective at recognizing mobile malware, as evidenced by the fact that it attained an F-measure of 94.04\%, as displayed in Fig. 8. In the process of conducting mobile application analysis and forensic investigations into malware, our model can substantial assistance.
The k-NN and random tree algorithms are the fastest when it comes to training and testing a classifier; both require 200 milliseconds. J48 is the most time-consuming, requiring 920 milliseconds. SVM takes 980 milliseconds in order to complete the training and testing, and for a random forest, an average of 0.73 seconds is required. Overall, the system is predictable and dependable in real-time applications, with a speed that is suitable for all five classifiers.

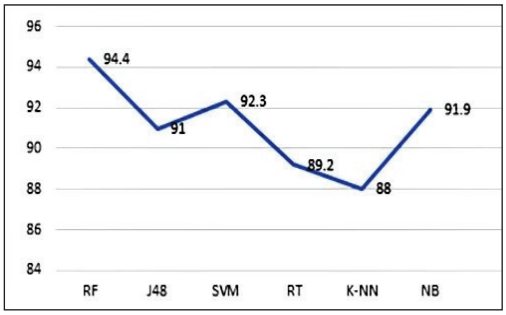

Fig 5: F Measure Score

15. CONCLUSION

The development of a secure mobile computing environment, the protection of sensitive data, and the detection of malicious software all need the identification of the most prevalent features demanded by malware applications. The behavior of an Android app is reflected in permissions and important API calls that the program makes. To identify malicious programs, we present a classification approach that includes the consideration of authorizations and API requests. This was provoked by the growing count of applications and the absence of efficient malware recognition technologies. There are three stages to our analysis: preprocessing, extraction, and grouping stage. With so many APIs used by Android apps, we devised a grouping method in order to target only the top important ones to increase the chances of finding Android malware.

Obscure set (common API calls in both malicious and normal applications).

Critical set (common API calls in a malicious application which are less like those in normal applications).

Obstreperous set (API calls that are present in the malicious applications and absent in normal applications).In order to find the top discriminating set of attributes for malware detection, a frequency examination is run on the important groups. As a result of the findings, it's clear that malicious Android apps make a distinct set of API calls and request permissions more frequently than normal Android applications to access user data than benign apps. For instance, the API requests for the SMS manager, storage, telephone manager, system service, device information, logs, and database are substantially more prevalent in malware applications. According to our suggested method's empirical results, which used an actual malware dataset of 15,036 Android applications, it is successful at recognizing mobile malware and can greatly

contribute in malware forensic investigation and mobile app analysis. Using IG and API frequency calculations, a useful subset of features is narrowed down, and the TF is then utilized to reduce the dimensionality of the narrowed-down set. The J48, k-NN, RT, RF, and NB algorithms are among the ML approaches we use in our research. The findings of the experiments show that our model is capable of reaching an F-measure of 94.03%.

REFERENCES

[1] R. Husnain, A. Nauman, A. Muhammad, I. Biju and R. Hamid, "AndroMalPack: enhancing the ML-based malware classification by detection and removal of repacked apps for Android systems," Scientific Reports, vol. 12, no. 1, pp. 19-34, 2022.

[2] F. Faezeh, H. M. Sayad J. Alireza and A. Mamoun, "Artificial intelligence for detection, estimation, and compensation of malicious attacks in nonlinear cyber-physical systems and industrial IoT," IEEE transactions on industrial informatics, vol. 16, no. 4, pp. 2716-2725, 2019.

[3] A. M. Taleby, L. Qianmu, R. Mahdi and R. A. Raza, "A survey on smartphones security: software vulnerabilities, malware, and attacks," arXiv preprint arXiv: 2001.09406, 2020.

[4] M. Anjali, Permissions Ranking With Statistical Techniques for Android Malware Detection, “Doctoral dissertation, 2022.

[5] M. Sreenath and S. Anuradha, “The political economy of digital automation: measuring its impact on productivity, economic growth, and consumption, “Routledge, 2020.

[6] Z. Jason, "Machine learning with feature selection using principal component analysis for malware detection: a case study," arXiv preprint arXiv: 1902.03639, 2019.

[7] A. Saba, S. M. Ali, A. Khan and A. Mansoor, "Android malware detection \& protection: a survey," International Journal of Advanced Computer Science and Applications, vol. 7, no. 2, 2016.

[8] P. Faruki, B. Ammar, V. Laxmi, Ganmoor, Vijay and Gaur, Manoj Singh and Conti, Mauro and Rajarajan, Muttukrishnan, "Android security: a survey of issues, malware penetration, and defenses," IEEE communications surveys \& tutorials, vol. 17, no. 2, pp. 998-1022, 2014.

[9] Z. Yajin and J. Xuxian, "Dissecting android malware: Characterization and evolution," in 2012 IEEE symposium on security and privacy, IEEE, pp. 95-109, 2012.

[10] R. Vaibhav, C. Yan and J. Xuxian, "Droidchameleon: evaluating android anti-malware against transformation attacks," in Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, pp. 329-334, 2013.

[11] I. Mülhem, B. Issa, and M. B. Jasser., "A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning," IEEE Access, vol. 10, pp. 117334-117352, 2022.

[12] B. Shikha, and S. Muttoo, "Evading android anti-malware by hiding malicious application inside images," International Journal of System Assurance Engineering and Management, vol. 9, pp. 482-493, 2018.

[13] I. Rejwana, S. M. Islam, S. Sajal, H. M. Jamal and M. Md Abdul, "Android malware classification using optimum feature selection and ensemble machine learning," Internet of Things and Cyber-Physical Systems, vol. 3, pp. 100-111, 2023.

[14] W. Chao, X. Qingzhen, L. Xiuli and L. Shouqiang, "Research on data mining of permissions mode for Android malware detection," Cluster Computing, vol. 22, pp. 13337-13350, 2019.

[15] D. Shuaifu, W. Tao and Z. Wei, "Droid-Logger: Reveal suspicious behavior of Android applications via instrumentation," in 2012 7th international conference on computing and convergence technology (ICCCT), IEEE, pp. 550-555, 2012.

[16] S. A. Kumar, C. D. Jaidhar, and K. MA Ajay, "Experimental analysis of Android malware detection based on combinations of permissions and API-calls," Journal of Computer Virology and Hacking Techniques, vol. 15, pp. 209-218, 2019.

[17] Tao, Guanhong, Zibin Zheng, Ziying Guo, and Michael R. Lyu, "MalPat: Mining patterns of malicious and benign Android apps via permission-related APIs," IEEE Transactions on Reliability, vol. 67, no. 1, pp. 355-369, Dec. 2017.

[18] A. Abdelfattah, R. Jean-Marc and T. Chamseddine, "Enhancing malware detection for Android systems using a system call filtering and abstraction process," Security and Communication networks, vol. 8, no. 7, pp. 1179-1192, 2015.

[19] P. Vinod, Z. Akka and C. Mauro, "A machine learning based approach to detect malicious android apps using discriminant system calls," Future Generation Computer Systems, vol. 94, pp. 333-350, 2019.

[20] Z. Aqil, H. I. Rahmi, S. Wahidah Md and A. Zubaile, "Android malware detection based on network traffic using decision tree algorithm," in Recent Advances on Soft Computing and Data Mining: Proceedings of the Third International Conference on Soft Computing and Data Mining (SCDM 2018), Johor, Malaysia, Springer, pp. 485-494, 2018.

[21] W. Shanshan, Y. Qiben, C. Zhenxiang, Y. Bo, Z. Chuan and C. Mauro, "Detecting android malware leveraging text semantics of network flows," IEEE Transactions on Information Forensics and Security, vol. 13, no. 5, pp. 1096-1109, 2017.

[22] W. Ping, C. W. Jie, C. Kuo-Ming and L. Chi-Chun, "Using taint analysis for threat risk of cloud applications," in 2014 IEEE 11th International Conference on e-Business Engineering, IEEE, pp. 185-190, 2014.

[23] B. James, A. Mohd and D. Gerry, "Detection of mobile malware: an artificial immunity approach," in 2016 IEEE Security and Privacy Workshops (SPW), IEEE, pp. 74-80, 2016.

[24] K. Pallavi and J. Amit, "Malware detection techniques in android," International Journal of Computer Applications, vol. 122, no. 17, 2015.

[25] W. Ahsan, I. Azhar, L. Jahanzaib, N. Ahsan and B. Anas, "A novel approach of unprivileged keylogger detection," in 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), IEEE, pp. 1-6, 2019.

[26] Z. Hanqing, L. Senlin, Z. Yifei and P. Limin, "An efficient Android malware detection system based on method-level behavioral semantic analysis," IEEE Access, vol. 7, pp. 69246-69256, 2019.

[27] F. Hossein, M. Veelasha, C. Mauro and B. Lejla, "Efficient classification of android malware in the wild using robust static features," Protecting mobile networks and devices: challenges and solutions, vol. 1, pp. 181-209, 2016.

[28] K. ElMouatez Billah, D. Mourad, D. Abdelouahid and M. Djedjiga, "Mal-Dozer: Automatic framework for android malware detection using deep learning," Digital Investigation, vol. 24, pp. S48-S59, 2018.

[29] Q. Mengyu, S. Andrew and L. Qingzhong, "Merging permission and api features for android malware detection," in 2016 5th IIAI international congress on advanced applied informatics (IIAI-AAI), IEEE, pp. 566-571, 2016.