electronics

MDPI

Article

Innovative Approach to Android Malware Detection: Prioritizing Critical Features Using Rough Set Theory

Rahul Gupta [1,*], Kapil Sharma [1,†] and Ramesh Kumar Garg [2,†]

[1] Department of Information Technology, Delhi Technological University, New Delhi 110042, India; kapil@ieee.org
[2] Department of Mechanical Engineering, Deenbandhu Chhotu Ram University of Science and Technology, Murthal 131039, India
* Correspondence: rahulgupta100689@gmail.com
† These authors contributed equally to this work.

Abstract: The widespread integration of smartphones into modern society has profoundly impacted various aspects of our lives, revolutionizing communication, work, entertainment, and access to information. Among the diverse range of smartphones available, those operating on the Android platform dominate the market as the most widely adopted type. With a commanding 70% share in the global mobile operating systems market, the Android OS has played a pivotal role in the surge of malware attacks targeting the Android ecosystem in recent years. This underscores the pressing need for innovative methods to detect Android malware. In this context, our study pioneers the application of rough set theory in Android malware detection. Adopting rough set theory offers distinct advantages, including its ability to effectively select attributes and handle qualitative and quantitative features. We utilize permissions, API calls, system commands, and opcodes in conjunction with rough set theory concepts to facilitate the identification of Android malware. By leveraging a Discernibility Matrix, we assign ranks to these diverse features and subsequently calculate their reducts–streamlined subsets of attributes that enhance overall detection effectiveness while minimizing complexity. Our approach encompasses deploying various Machine Learning (ML) algorithms, such as Support Vector Machines (SVM), K-Nearest Neighbor, Random Forest, and Logistic Regression, for malware detection. The results of our experiments demonstrate an impressive overall accuracy of 97%, surpassing numerous state-of-the-art detection techniques proposed in existing literature.

Keywords: android; malware; ranking; reduct; rough sets; prediction

check for updates

Citation: Gupta, R.; Sharma, K.; Garg, R.K. Innovative Approach to Android Malware Detection: Prioritizing Critical Features Using Rough Set Theory. *Electronics* **2024**, *13*, 482. https://doi.org/10.3390/electronics13030482

Academic Editors: Zbigniew Kotulski and Juan M. Corchado

Received: 5 October 2023
Revised: 11 January 2024
Accepted: 11 January 2024
Published: 23 January 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Smartphones, in reality, are the personal desktop computers of today's world. With smartphones, we can do almost anything we intend to do with personal desktop computers. Smartphones have become integral to modern society, impacting various aspects of our lives. Their versatility and functionality have revolutionized how we communicate, work, entertain ourselves, and access information. In addition to being a communication device to place calls and send SMSs, smartphones are used for internet browsing, social media, emails, photography and videography, navigation, online shopping, banking, health and fitness, education and learning, personal organization, and intelligent home control. The current mobile market share is 20% higher than desktops, meaning smartphones are leading the way ahead of desktops in terms of usage [1].

Among all the types of smartphones available in the market, smartphones with the Android operating system are the most popular. The reason for the popularity is that Android is an open-source platform many manufacturers adopt. As per the report of Statcounter [2], the global market share of mobile operating systems is entirely dominated by the Android operating system with a 70% share. The high market share of the Android

operating system is one of the primary reasons for many malware attacks on the Android platform over the past few years. As per the report [3], several mobile trojan subscribers were found on Google's official app marketplace in 2022. As per the blog post by renowned antivirus firm McAfee [4], 60 Android apps with 100 million downloads were found to be spreading a new malware strain to unsuspecting users. According to the news article on TechRadar [5], a new ransomware, "Daam", capable of hiding from antivirus software, was detected. The report on the development of new Android malware shared by Statistica [6] shows that 5.6 million Android malware samples were seen in 2020. Moreover, millions of Android malware have been detected yearly from 2016 to 2020. Hence, there is a dire need to develop Android malware detection mechanisms to see malicious applications.

Malware analysis is a technique by which we can analyze the functionality and source of malware. Malware analysis can be broadly classified into three types [7]: static analysis, dynamic analysis, and hybrid analysis. These different analysis techniques can form a detection model to classify an Android application as malicious or benign. Three types of detection models could be possible: static, dynamic, and hybrid. In static detection, features are extracted using static analysis, which is the art of malware analysis that extracts features from the application without installing or executing the application. In dynamic detection, features are extracted using dynamic analysis, performed by executing or running the application and capturing the features at run time. The hybrid detection model is based upon hybrid analysis, which extracts both static and dynamic features from the application.

In static detection methods, the standard static techniques employed for extracting static features are manifest file-based detection, API calls-based detection, and Java code-based detection methods. In the manifest file-based detection technique, features are extracted from the manifest file of the Android application. Li et al. [8] used permissions from the manifest file and achieved an accuracy of 90%. Arora et al. [9] used permissions pairs from the manifest file and achieved an overall accuracy of 95.44%. IPDroid [10] used both permissions and intents from the manifest file and achieved an accuracy of 94.73% with a Random Forest classifier.

API calls-based feature detection is based on extracting APIs to be called by Android applications. Droidmat [11] used API calls along with manifest file components to detect malicious applications with an accuracy of 97.87%. Elish et al. [12] extracted sensitive API calls invocation by the user to build a detection model. Zhang et al. [13] created association rules between API calls and achieved an accuracy of 96%.

The Java code-based detection methods use Dex files that contain Java code in Android applications for extracting features. Zhu et al. [14] converted the vital parts of Dalvik byte code into RGB images and trained the Convolution neural network to devise the malware detection system with an accuracy of 96.9%. Fang et al. [15] also converted Dex files to RGB images to do malware familial classification with a precision of 96%. The work [16] is based on eliminating code confusion and achieves an overall accuracy of 92.67%.

Contributions

In the current work, we have used permissions, API calls, system commands, and op-codes with rough set theory for Android malware detection. To the best of our knowledge, we are the first to apply rough set theory to the static features mentioned above. The rough set theory has several advantages, such as attribute selection and its ability to work with qualitative and quantitative attributes. We used a Discernibility Matrix to rank and further calculate the reduct of the above features. Ranking of features is done to highlight essential features. Reduct, a reduced feature set, is estimated to improve the overall detection rate with the most minor features. We applied several Machine Learning (ML) algorithms such as Support Vector Machines (SVM), K-Nearest Neighbor, Random Forest, and Logistic Regression for malware detection. Our results demonstrate an overall accuracy of 97%, better than many state-of-the-art detection techniques proposed in the literature. The main contributions of this paper are summarized below.

- Firstly, we performed data pre-processing, in which we eliminated co-related features and features not dependent on the class variable.
- We calculated the ranking score with the help of the discernibility concept of rough set theory to rank the features according to their importance.
- We applied an algorithm for rough set reduct computation to reduce the number of features in each category using the ranking score and discernibility concept of rough set theory.
- We further applied machine learning algorithms to evaluate the detection accuracy with the reduct calculated in the previous step.
- We compared the results of our proposed model with other state-of-the-art detection techniques, and our results highlight that the proposed model outperforms similar state-of-the-art models.

The rest of the paper is organized as follows. We discuss related work in Section 2. The detailed methodology is discussed in Section 3. Section 4 is dedicated for results and discussions, and we conclude the paper with future directions in Section 5.

2. Related Work

This section describes the work on static malware detection methods based on manifest file-based detection, API calls-based detection, and Java code-based detection. Therefore, this section is divided into three subsections: manifest file-based detection, API calls-based detection, and Java code-based detection.

2.1. Manifest File-Based Detection

In this sub-section, we review the works that have analyzed features from manifest files for Android malware detection. Grace et al. [17] established the relationship between embedded ad libraries and host apps as a significant risk factor for Android-based smart devices. Enck et al. [18] developed a lightweight certification model based on security configuration within an application to identify risky applications. A malware detection system, SIGPID, was proposed by Li et al. [8], in which three-level pruning is applied to filter out a minimal permission set that can be used for malicious app identification. Talha et al. [19] developed a client–server-based tool known as APK-auditor to detect malicious applications that maintain the Android profile database based on permission analysis.

The authors in [20] developed context category ontology based on permissions to detect the potential risk of information leakage with the help of a malicious activity. A prototype ASE was developed by Song et al. [21], which applied four levels of filtering based on static analysis to detect an application as benign or malicious. DroidChain [22] is another malware detection approach that applies static analysis and a behavior chain model to detect four types of malicious behaviors, i.e., privacy leakage, SMS financial charges, malware installation, and privilege escalation. ProDroid [23] is another behavior-based malware detection model that uses the biological sequence technique and Markov chain model to match the classes and API of decompiled applications to the stored malicious behavior models. Moonsamy et al. [24] used both requested and used permissions to mine contrasting permission sets, which were further used to detect an application as benign or malware.

Idrees et al. [25] used intent filters and permissions to identify an application as benign and malicious. Wang et al.'s [26] work is based on rankings of requested permissions based on risk, and the selection of risky permission subset is made to train machine learning models. DroidRanger was developed by the authors in [27] as a malicious application detection tool using permission behavior and heuristic filtering, which also successfully discovered zero-day malware. Qiu et al. [28] did a unique work of annotating the capabilities of detected malware regarding security and privacy concerns. APIs, intents, and permissions were used in [29] to perform similarity associations with malware samples and detect malicious applications using hamming distance.

Bai et al. [30] tried to develop a fast malware detection model using multiple features such as permissions and opcode sequences. Drebin [31] is a lightweight malware detection technique that can be deployed on the smartphone. The developed method can detect malicious applications on the smartphone within ten seconds of download. Varsha et al.'s [32] work is based on selecting prominent features from various sets of static features. Mahindru et al. [33] used ten different feature selection techniques to find the best possible combination of features to detect malicious applications effectively. Firdaus et al. [34] is a genetic search-based technique, in which a genetic algorithm minimizes the number of features.

Khariwal et al. [10] proposed a novel method to find the best permissions and intents combined to detect malicious applications. PermPair [9] is another malicious application detection technique that creates permission pairs from each application and further constructed malicious and normal permissions pair graphs used for the detection mechanism. The work in [35] uniquely compared the dynamics between requested permissions and intent filters. In manilyzer [36], stress was given on using different manifest components along with requested permissions. Sanz et al. [37] developed a malware detection model based on used permissions. Li et al. [38] developed a malware detection model using multiple features both from the manifest file as well as from the source file, whereas Sato et al. [39] used multiple features from the manifest file only.

2.2. API Calls-Based Detection

Several researchers have utilized static API calls to identify Android malware. The Droidmat model [11] employed a combination of manifest file features and API calls and applied K-means and KNN algorithms for malware detection. Another study [12] examined user-triggered dependencies and sensitive APIs in malicious apps, while Zhang et al. [40] constructed dependency graphs of API calls to categorize malicious apps into Android malware families using similarity metrics. The authors of [41] introduced a model called Apposcopy, which examined control-flow and dataflow properties derived from API calls to detect malware. Wang et al. [42] focused on analyzing string features like permissions and intents, as well as structural features such as API calls and function call graphs, on detecting malicious behaviour in Android apps. Similarly, the work described in [13] involved the analysis of API calls and their call graphs for malware detection.

2.3. Java Code-Based Detection

Zhu et al. in [14] developed an image-based malware detection method that extracts important parts of Dalvik code and converts it to RGB images. Fang et al. [15] also used RGB images generated from Dex files but, apart from classifying an application as benign or malicious, did malware familial classification. The work [16] eliminated code confusion and calculated scores for every code word based on their importance, which deep learning models then used to detect malicious applications. CDGDroid [43] is another technique to detect Android malware based on control flow graphs and data flow graphs that are constructed from the code of the application with the help of program analysis techniques and later on used as features for the CNN model. Xiao et al. [44] developed a method that captures the system call sequence from the code of the application, and the captured system call sequence is used to train LSTM to detect malicious applications. To form the malware detection model, MSNDroid [45] incorporated native-layer code features and combined them with permissions and Java layer components.

To the best of our knowledge, no other work has used rough set theory to form reducts of multiple features in the area of Android malware detection.

3. The Proposed Methodology

This section describes the overall approach to classifying Android applications as malware or benign. The process is divided into four phases, as depicted in Figure 1. The first phase of the approach is the pre-processing phase, the second phase is feature ranking,

the third phase is the Rough Set Reduct Computation Phase, and the fourth phase is the detection phase.

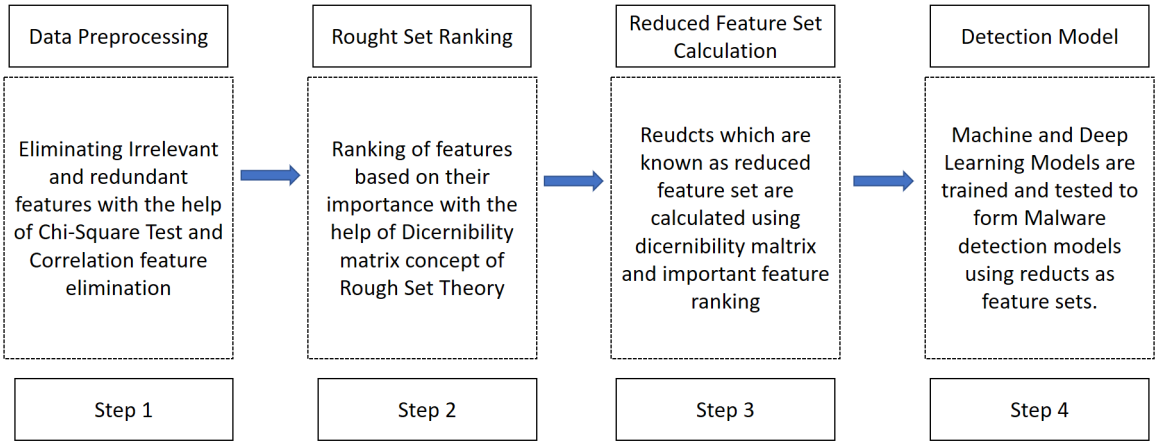


Figure 1. Proposed Methodology.

3.1. Data Pre-Processing Phase

These data pre-processing phase is more focused on the primary feature selection phase. The whole process of this phase is depicted in Figure 2.

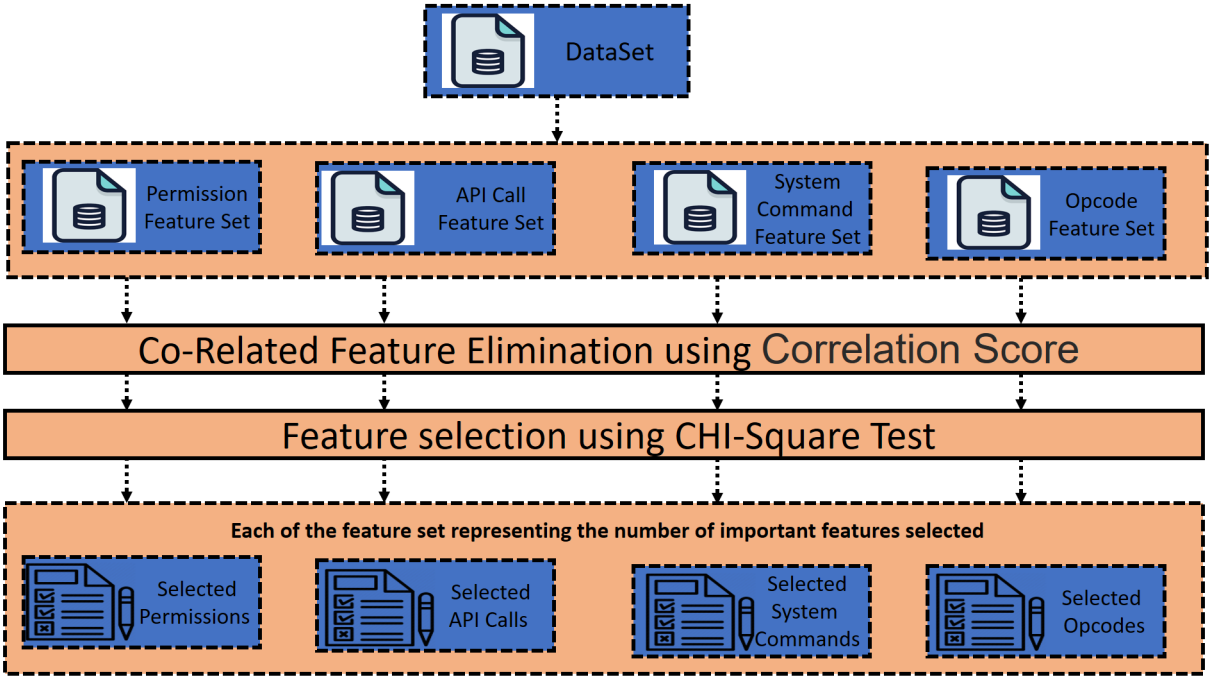


Figure 2. Data Pre-Processing.

The figure referred to here summarizes the whole process of phase 1. The proposed technique first considers the OmniDroid Dataset [46] and Androzoo Dataset [47]. The OmniDroid dataset is the data set in which various features are extracted from an extensive collection of 22,000 APKs. The dataset consists of an equal number of benign and malicious applications, i.e., 11,000 each. An additional 8000 applications are taken from the Androzoo Data set, spreading from 2015 to 2023, making it more diverse. These 8000 apps consist of an equal number of benign and malicious applications, i.e., 4000 each. The features from these 30,000 apps were extracted with the AndroPytool [48]. The AndroPyTool extracts

features from the Android application supplied as input to the tool. Specifically, the AndroPyTool extracts three types of features: pre-static features, static features, and dynamic features. This paper focuses on static features, i.e., permissions, API calls, system commands, and opcodes. The following is the description of each static feature considered in this paper.

1. Permissions: Every Android application requests and requires a particular set of permissions for its functioning. The apps need these permissions to access some data or specific resources. These permissions are listed in the Android Manifest file. The OmniDroid dataset consists of 5501 unique permissions.
2. API Calls: The Application Programming Interface (API) is a set of code snippets that the underlying systems use for communicating. API calls are the calls to such code snippets with some functionality that must be invoked to perform specific tasks. The dataset in consideration consists of 2128 API Calls.
3. System commands: Android applications must access the kernel to perform specific tasks and services. So, the services that need to be accessed by the app are done by calling the OS routines. The calls to such kinds of OS routines are known as system commands. The OmniDroid dataset consists of 103 system commands.
4. Opcodes: The Dalvik Bytecode generated by compiling the Android apps consists of instructions that need to be executed in terms of opcodes. The data set consists of 224 opcodes.

First, each of the features mentioned above are considered individually to be preprocessed. The feature correlation score is calculated for each of the separate features, i.e., permissions, API calls, system commands, and opcodes. This feature correlation score is the basis for eliminating such features that are highly correlated with the existing features. The elimination of closely interconnected attributes occurs due to their strong correlation, resulting in a high degree of linear dependence. As a result, they exert nearly identical influences on the dependent variable. Thus, in cases where two attributes exhibit significant correlation, it is feasible to omit one of them. A correlation score of 90 per cent or more is used to select one feature out of two and eliminate the other. After this round, we received 4428 permissions out of 5501 permissions, 1589 API calls out of 2128 API calls, 93 System commands out of 103 System calls, and 159 Opcodes out of 224 Opcodes.

Further, the Chi-Square test is executed to select a subset of features for each feature set. The Chi-square test is a statistical test used to determine whether there is a significant association between two categorical variables. The chi-square test works based on the following equation:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{1}$$

The formula for the Chi-square test involves several key terms and calculations, as shown in Equation (1), where χ^2 is the Chi-square test statistic, n is the number of categories in the contingency table, O_i is the observed frequency of category i, and E_i is the expected frequency of category i under the null hypothesis. The sum is taken over all categories in the contingency table.

Initially, the null hypothesis assumes that the two variables, i.e., the feature variable and the class variable, do not have any association. Then, the Chi-square test is applied to those variables using Equation (1) to calculate the Chi-square test statistic value. The Chi-square distribution table maps the estimated chi-square test statistic value to the corresponding p-value. The p-value less than 0.05 means that the null hypothesis assumed is false, and the alternate hypothesis that the class variable and the feature variable are associated with each other is true. All those features in each feature set that fall in the rejection region whose p-value is less than 0.05 are selected as a new feature subset for each feature set. Choosing a p-value of less than 0.05 means the features falling in this rejection region have 95% confidence of dependency on the class variable. Hence, in the end, we got

a subset of each feature space with the minimized feature set from each feature space, i.e., permissions, API calls, system commands, and opcodes' feature space.

For the permission feature set, we achieved 206 permissions selected as a subset of permission out of 4428 permissions. For the API calls feature set, we achieved 1264 API calls selected as a subset of API calls out of 1589 API calls. For system command feature space, out of 94 system commands obtained in the previous step, we achieved a minimized feature space of 52. Similarly, for the opcodes-based feature set, which consisted of 204 opcodes from the previous step, we achieved 158 opcodes. The entire process is also summarized in the Algorithm 1.

Algorithm 1 Data Pre-Processing

Input: A feature set of 22,000 APK regarding four types of features, i.e., Permissions (f_p), API Calls (f_a), System commands (f_s), and Opcodes (f_o).
Output: For each of the feature set f_p, f_a, f_s, and f_o, a minimal feature space of important features is obtained as $min_f_p, min_f_a, min_f_s$, and min_f_o, respectively.
for each of features set f_i in feature spaces f_p, f_a, f_s, f_o **do**
    set $|f_i| = N$
    for each of the feature x_i in feature set f_i **do**
        set $Truth_val(x_i) = True$
    end for
    for each of the feature x_i in feature set f_i **do**
        if $Truth_val(x_i) == True$ **then**
            for each of the feature x_j in feature set f_i where $j : i + 1 \rightarrow N$ **do**
                if $(correl[x_i, x_j] > 0.9)$ **then**
                    $Select(x_i)$
                    $Reject(x_j)$ and set $Truth_val(x_j) = False$
                end if
            end for
        end if
    end for
end for
for each of the features set f_i in the new feature space f_p, f_a, f_s, and f_o obtained **do**
    for each of the feature x_i in feature set f_i **do**
        Apply Chi-Square Test (x_i)
        if $p_value[x_i] < 0.05$ **then**
            $Select(x_i)$
        else
            $Reject(x_i)$
        end if
    end for
end for

3.2. Feature Ranking Phase

In this phase, the ranking of minimal feature sets obtained in the previous step is performed to rank the features of each type according to their importance. Feature ranking is performed through the Discernibility Matrix concept of rough set theory. The whole process flow of this phase is depicted in Figure 3.

Rough set theory is a mathematical approach to data analysis and data mining. This mathematical tool is powerful in dealing with improper, imprecise, inconsistent, incomplete information, and knowledge [49,50]. The rough set theory has several advantages, such as attribute selection and its ability to work with qualitative and quantitative attributes. The critical concepts of the rough set theory used in this paper are explained below.

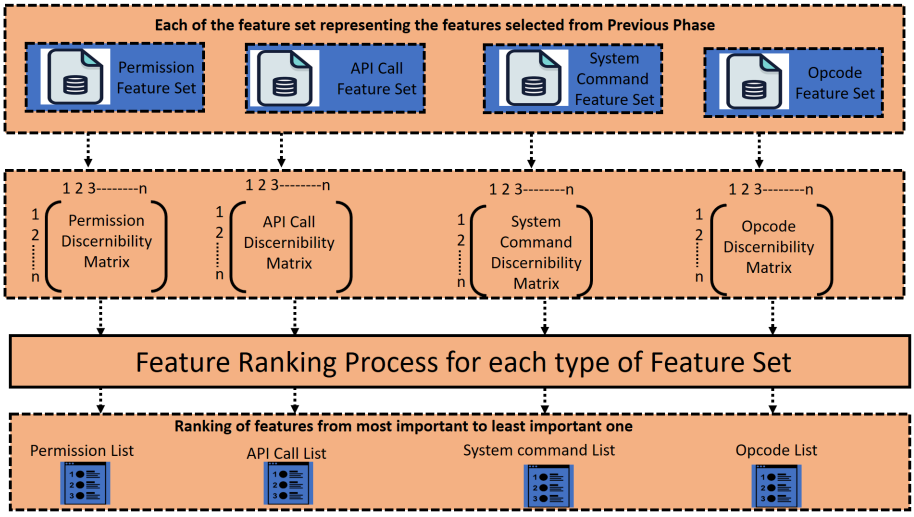


Figure 3. Feature Ranking Phase.

3.2.1. Information System

It is defined as an ordered pair, in which the first element of this ordered pair is called the universe. In our case, the universe is the set of both malicious and benign applications that are considered. We represent the ordered pair as $D = (A, F \cup \{l\})$, where D is the data set under consideration, and A is the non-empty finite set called the universe of Android application, consisting of both the malicious and benign types. F is the non-empty set of features in the data set. In our case, these features are in terms of permissions, API calls, system commands, and opcodes. Here, l is the special attribute known as the label attribute, which stores the type of label corresponding to each application in set A. This label attribute stores whether a particular application in set A is malicious or benign. Table 1 shows an instance of the permission information system for five applications assumed as A_1, A_2, A_3, A_4, and A_5. Here, feature attributes are the Content Provider Access, Settings App widget Provider, and JPUSH Message. These are permissions, with corresponding values such as 0 or 1 for each application A_i. The value 0 signifies that particular permission is not present in application A_i, whereas the value 1 signifies otherwise. The label column has the value BW, signifying that the particular A_i is Benignware, whereas the label value MW signifies that application A_i is malware, i.e., a malicious one.

Table 1. Instance of permission information system.

Application	P_1 = CONTENT PROVIDER ACCESS	P_2 = Settings App Widget Provider	P_3 = JPUSH MESSAGE	Label
A_1	0	1	0	BW
A_2	1	1	0	MW
A_3	0	1	1	BW
A_4	0	0	1	BW
A_5	1	1	1	MW

Similarly, the information systems for API calls, system commands, and opcodes are shown in Tables 2–4, respectively. From these information systems, a Discernibility Matrix is formed. The concept of Discernibility is explained in the following section.

Table 2. Instance of API calls information system.

Application	AP_1 = APICALL-android.view.SubMenu	AP_2 = APICALL-android.net.RouteInfo	AP_3 = APICALL-android.app.Activity	Label
A_1	0	0	0	BW
A_2	1	1	0	MW
A_3	1	1	1	MW
A_4	1	0	1	BW
A_5	1	1	1	MW

Table 3. Instance of system command information system.

Application	S_1 = SYSTEMCOMMAND-svc	S_2 = SYSTEMCOMMAND-stagefright	S_3 = SYSTEMCOMMAND-nandread	Label
A_1	0	1	0	MW
A_2	1	1	0	MW
A_3	0	1	1	BW
A_4	0	0	1	BW
A_5	1	1	1	MW

Table 4. Instance of opcode information system.

Application	O_1 = OPCODE-rem-float/2addr	O_2 = OPCODE-div-int/lit8	O_3 = JOPCODE-sparse-switch	Label
A_1	1	1	0	BW
A_2	0	1	0	MW
A_3	1	0	1	MW
A_4	0	0	1	BW
A_5	0	1	1	BW

3.2.2. Discernibility Matrix

This matrix is created from the information system. The Discernibility Matrix is a symmetric $|A|X|A|$ matrix corresponding to each information system. Each entry C_{ij} is defined as $\{f \in F | f(A_i) \neq f(A_j)\}$ if $l(A_i) \neq l(A_j)$, Φ otherwise. Tables 5–8 show the instances of Discernibility Matrices corresponding to information system shown in Tables 1–4, respectively.

Table 5. Instance of permission discernibility.

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	P_1				
A_3		P_1, P_3			
A_4		P_1, P_2, P_3			
A_5	P_1, P_3		P_1	P_1, P_2	

Table 6. Instance of API calls discernibility.

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	AP_1, AP_2				
A_3	AP_1, AP_2, AP_3				
A_4		AP_2, AP_3	AP_2		
A_5	AP_1, AP_2, AP_3			AP_2	

Table 7. Instance of system call discernibility.

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2					
A_3	S_3	S_1, S_3			
A_4	S_2, S_3	S_1, S_2, S_3			
A_5			S_1	S_1, S_2	

Table 8. Instance of opcode discernibility.

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	O_1				
A_3	O_2, O_3				
A_4		O_2, O_3	O_1		
A_5		O_3	O_1, O_2		

For each of the selected minimal permission set, API call feature set, system command feature set, and opcode feature set, we create the Permission Discernibility Matrix, API Call Discernibility Matrix, System Call Discernibility Matrix, and Opcode Discernibility Matrix, respectively. Algorithm 2 depicts the whole process. The algorithm creates a Discernibility Matrix for each minimal feature set obtained in the previous step and further calls the rough set ranking algorithm described in the next section.

Algorithm 2 Feature Ranking

1: **Input:** Minimal feature space, i.e., $min_f_p, min_f_a, min_f_s$, and min_f_o obtained as output of Algorithm 1.
2: **Output:** For each of the minimal feature space, ranked minimal feature list L_p, L_a, L_s, and L_o, respectively, sorted in decreasing order as per the importance of the features.
3: **for each** of the minimized feature space min_f_i in $min_f_p, min_f_a, min_f_s$, and min_f_o **do**
4: create Discernibilty Matrix for min_f_i.
5: **end for**
6: Let D_p, D_a, D_s, and D_o be Discernibility Matrix corresponding to min_f_i in $min_f_p, min_f_a, min_f_s$, and min_f_o, respectively.
7: **for each** of Discernibility Matrix D_i in D_p, D_a, D_s, and D_o **do**
8: call Algorithm 3 for each D_i in order to perform rough set ranking of each of the features in D_i
9: Let L_p, L_a, L_s, and L_o be be the sorted list of important features for each of min_f_i in $min_f_p, min_f_a, min_f_s$, and min_f_o, respectively
10: **end for**

3.2.3. Rough Set-Based Feature Ranking

After creating each of these Discernibility Matrices, a rough set-based feature ranking methodology, summarized in Algorithm 3, is applied on each matrix to rank each of the Permission, API Call, System Command, and Opcode features separately. The algorithm takes the Discernibility Matrix as an input and initializes the weight of each feature in the corresponding minimal feature set to zero. Then, the Discernibility Matrix is traversed, and each entry in the Discernibility Matrix, which consists of one or more features, receives the updated weight of the features as per Equation (2).

$$w(x_k) = w(x_k) + |min_f_i|/|C_{ij}| \tag{2}$$

In the above equation, C_{ij} is the entry in the Discernibility Matrix corresponding to applications A_i and A_j, and the entry C_{ij} may contain one or more features. Hence, $|C_{ij}|$

represents the count of features in the entry. min_f_i is the minimal feature set corresponding to the Discernibility Matrix, and $|min_f_i|$ is the count of features in the minimal feature set. $w(x_k)$ is the weight of k^{th} feature in the entry C_{ij}, which may contain $x_1, x_2, x_3, \ldots$ and x_n as the features in the entry.

Algorithm 3 Rough Set Ranking

Input: Discernibility Matrix D_i with dimensions nXn.
Output: Ranked feature list L_i w.r.t D_i.
for each x_i in the minimal feature space min_f_i corresponding to D_i **do**
    Let $w(x_i)$ denote the weight of the feature x_i and let $w(x_i) = 0$.
end for
for each $i : 1 \rightarrow N$ **do**
    for each $j : 1 \rightarrow i$ **do**
        Let C_{ij} be the entry in Discernibility Matrix D_i containing features $x_1, x_2, x_3, \ldots and\ x_n$.
        for each x_k in $x_1, x_2, x_3, \ldots and\ x_n$ corresponding to entry C_{ij} **do**
            update weight of every feature $x_k \in C_{ij}$ as
            $w(x_k) = w(x_k) + |min_f_i|/|C_{ij}|$
        end for
    end for
end for
Sort every x_i in the minimal feature space min_f_i w.r.t $w(x_i)$ in decreasing order of $w(x_i)$

This ranking is obtained by arranging each of these features in descending order in terms of their importance.

The Rough Set-based feature ranking embodies the following idea [49].

(1) The more times an attribute appears in the discernibility, the more important is the attribute.
(2) The shorter the entry is, the more important the attributes is in the entry.

3.3. Rough Set Reduct Computation Phase

This phase focuses on reducing the feature space so that, with as few features as possible, i.e., a reduced feature space, the classification algorithms could be applied to detect an application as benign or malicious. The reduced feature space obtained using the underlying principles of rough set theory is called reduct in rough set theory. The Discernibility Matrix and rough set feature ranking obtained in the previous phase are used to attain reducts for each of the permission, API call, system command, and opcode feature spaces. Hence, after this phase, for each feature space, i.e., permission, API call, system command, and opcode, we get a reduced feature space, which we call a reduct in rough set theory. Figure 4 depicts the current phase under discussion. Algorithm 4 describes the whole process in pseudo-code form. The algorithm first calculates the net weight $w(net_{ij})$ of each of the entry C_{ij} in the Discernibility Matrix D_i, containing features $x_1, x_2, x_3, \ldots$ and x_n by summing their individual weights $w(x_1), w(x_2), w(x_3), \ldots\ldots$ and $w(x_n)$, respectively. Then, all the entries C_{ij} in the Discernibility Matrix D_i are copied in the list LD_i, and then LD_i is sorted as per the net weight calculated in the previous step. Initially, Red_i is assumed to be an empty set. For each entry C_z of the sorted list LD_i containing features $x_1, x_2, x_3, \ldots\ldots$ and x_n, we check weather the Red_i contains any common feature in C_z. If no common feature exists, we select the feature x_i with maximal $w(x_i)$ in C_z; otherwise, we skip the entry C_z. The set Red_i is the reduct computed for min_f_i.

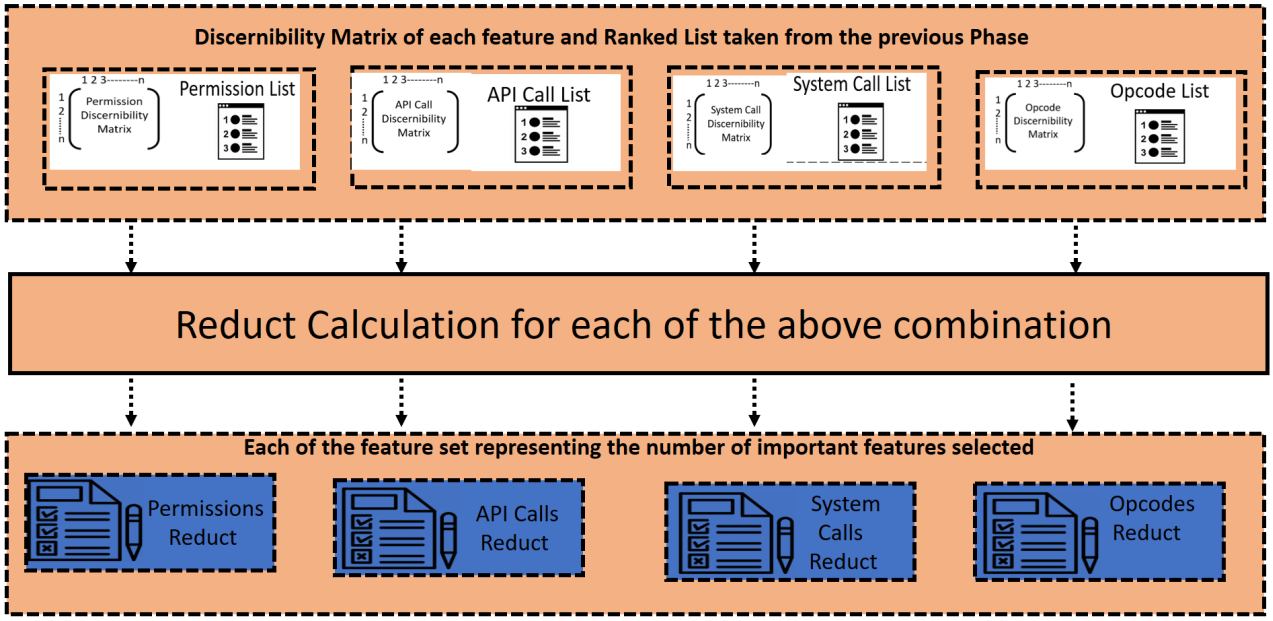


Figure 4. Rough Set Reduct Computation Phase.

Algorithm 4 Rough Set Reduct Computation

Input: Discernibility Matrix D_i with dimensions nXn and weight $w(x_i)$ of every feature x_i in the minimal feature space min_f_i in $min_f_p, min_f_a, min_f_s$, and min_f_o corresponding to D_i in D_p, D_a, D_s and D_o .
Output: Red_i as the reduct of minimal feature space min_f_i corresponding to D_i
Let LD_p, LD_a, LD_s, and LD_o be the empty list corresponding to permission, API call, system command and opcode feature space.
for each D_i in D_p, D_a, D_s, and D_o **do**
    Let $Red_i = \Phi$ denote the empty reduct set corresponding to minimal feature space min_f_i.
    for each $i : 1 \rightarrow N$ **do**
        for each $j : 1 \rightarrow i$ **do**
            Let C_{ij} be the entry in Discernibility Matrix D_i containing features $x_1, x_2, x_3, \ldots$ and x_n .
            Let $w(net_{ij})$ be the cumulative weight of entry C_{ij} having features as $x_1, x_2, x_3, \ldots$ and x_n .
            $w(net_{ij}) = w(x_1) + w(x_2) + w(x_3) \ldots\ldots\ldots + w(x_n)$
            $LD_i = append(C_{ij})$
        end for
    end for
    $Sort(LD_i)$ based on $w(net)$ calculated previously.
    for each $z : 1 \rightarrow |LD_i|$ **do**
        Let C_z be the entry in List LD_i containing features $x_1, x_2, x_3, \ldots$ and x_n .
        if $C_z \cap Red_i = \Phi$ **then**
            Select attribute x_i with maximal $w(x_i)$ in C_z
            $Red_i = Red_i \cup x_i$
        end if
    end for
end for

3.4. Detection Phase

For building our Android Malware detection system, we experimented with four machine learning algorithms, i.e., the Support Vector Machine (SVM), Random Forest, Logistic Regression, and K-nearest neighbour algorithms, to train and test the dataset. We also performed training and testing on two deep learning models, i.e., Artificial neural network (ANN) and Convolution Neural Network (CNN). Figure 5 portrays the overall purpose of the current phase, i.e., with the help of the machine learning models mentioned above and the different reducts, i.e., permission reduct, API call reduct, system call reduct, and opcode reduct calculated in the previous phase; the machine learning models are trained to build the system capable of detecting an Android application as benign or malware.

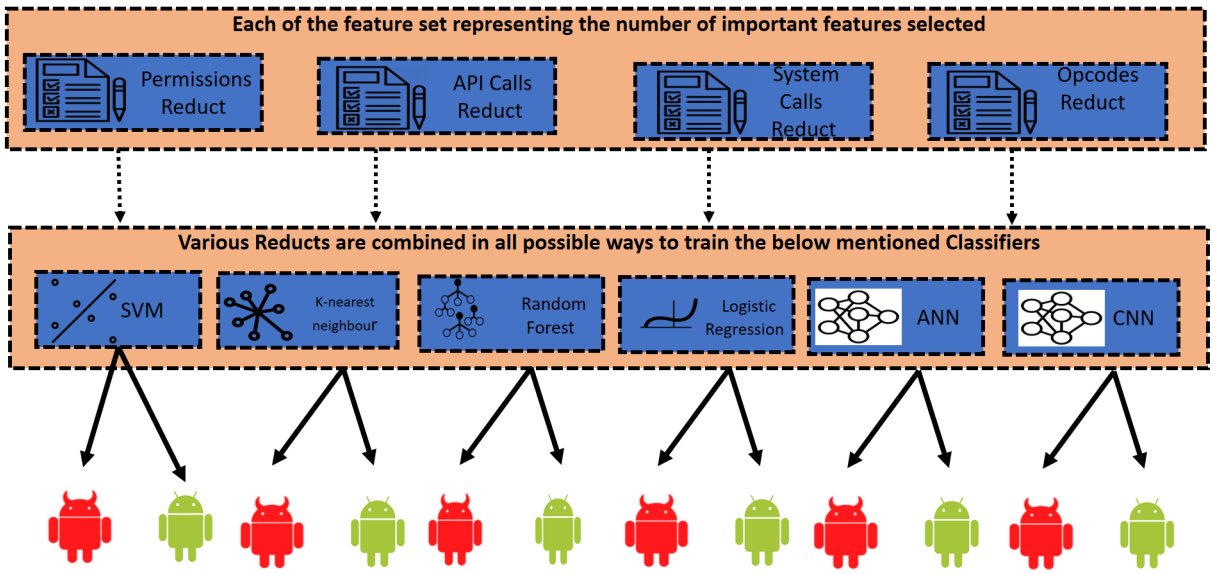


Figure 5. Detection Phase.

4. Results and Discussion

In the current section, we present the results of the evaluation carried out on the proposed malware detection approach. The code and details about data set are available at https://github.com/rahulgupta100689/Rough-Set.git, accessed on 10 January 2024.

4.1. Results of Ranking Phase

Table 9 shows the top 10 important permissions for malware detection. Similarly, the Tables 10–12 represent the top 10 important opcodes, API calls, and system calls, respectively.

Table 9. Top ten important permissions.

Rank	Permission Name	Score
1	*READ_PHONE_STATE*	1.56E9
2	*ACCESS_WIFI_STATE*	1.33E9
3	*WRITE_EXTERNAL_STORAGE*	1.26E9
4	*WAKE_LOCK*	1.2E9
5	*ACCESS_COARSE_LOCATION*	1.03E9
6	*ACCESS_NETWORK_STATE*	1.02E9
7	*ACCESS_FINE_LOCATION*	1.01E9
8	*GET_TASKS*	9.52E8
9	*RECEIVE_BOOT_COMPLETED*	8.82E8
10	*GET_ACCOUNTS*	8.4E8

Table 10. Top ten important opcodes.

Rank	Opcode	Score
1	$OPCODE - xor - int$	3.09E8
2	$OPCODE - rem - float$	3.07E8
3	$OPCODE - rem - float/2addr$	2.98E8
4	$OPCODE - float - to - long$	2.82E8
5	$OPCODE - and - long$	2.78E8
6	$OPCODE - aget - short$	2.7E8
7	$OPCODE - iget - byte$	2.67E8
8	$OPCODE - aput - short$	2.66E8
9	$OPCODE - iget - short$	2.65E8
10	$OPCODE - rem - double/2addr$	2.65E8

Table 11. Top ten important API calls.

Rank	API Call	Score
1	$APICALL - android.app.ActionBar$	2.2E8
2	$APICALL - android.widget.PopupWindow$	2.2E8
3	$APICALL - android.widget.BaseAdapter$	2.17E8
4	$APICALL - android.view.ScaleGestureDetector$	2.15E8
5	$APICALL - android.widget.CheckBox$	2.12E8
6	$APICALL - android.widget.AbsListView$	2.1E8
7	$APICALL - android.widget.ListPopupWindow$	2.1E8
8	$APICALL - android.content.res.XmlResourceParser$	2.1E8
9	$APICALL - android.graphics.Path$	2.09E8
10	$APICALL - android.webkit.MimeTypeMap$	2.09E8

Table 12. Top ten important system commands.

Rank	System Command	Score
1	$SYSTEMCOMMAND - top$	4.27E8
2	$SYSTEMCOMMAND - id$	3.75E8
3	$SYSTEMCOMMAND - start$	3.7E8
4	$SYSTEMCOMMAND - service$	3.58E8
5	$SYSTEMCOMMAND - gzip$	3.54E8
6	$SYSTEMCOMMAND - date$	3.44E8
7	$SYSTEMCOMMAND - log$	3.17E8
8	$SYSTEMCOMMAND - stop$	3.1E8
9	$SYSTEMCOMMAND - mv$	3.06E8
10	$SYSTEMCOMMAND - input$	3.02E8

4.2. Detection Results with Individual Features

Table 13 shows the results of four sets of permissions, i.e., all permissions in the data set, reduced permissions obtained after applying correlation feature elimination as permission correlation, permission chi as a set of permissions obtained after applying chi-square test on permission correlation, and finally permission reduct as the reduced permission feature set obtained after applying rough set reduct on permission chi. All four permission sets are used to train all six classifiers, i.e., Support Vector Machines (SVM), K-nearest neighbours, Random Forest, Logistic regression, ANN, and CNN. The results indicate that, for each type of classifier, as the feature set is changed from all permissions to permission correlation then to permission chi and finally to permission reduct, the

accuracy increases. This means that reduct feature sets are the best for building malware detection systems.

Table 13. Detection results based on permission.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All Permissions	78	78	79	78
	Permissions Correlation	79	78	79	78
	Permissions Chi	79	79	79	79
	Permissions Reduct	80	79	79	80
K-Nearest Neighbor	All Permissions	77	77	79	78
	Permission Correlation	78	78	79	78
	Permissions Chi	80	81	78	80
	Permissions Reduct	82	81	82	79
Random Forest	All Permissions	82	84	80	82
	Permissions Correlation	82	84	80	82
	Permissions Chi	83	82	80	81
	Permissions Reduct	83	84	80	81
ANN	All Permissions	76	76	76	77
	Permission Correlation	78	78	79	78
	Permissions Chi	79	79	78	79
	Permissions Reduct	80	80	80	79
CNN	All Permissions	77	77	79	78
	Permission Correlation	78	78	79	78
	Permissions Chi	79	80	79	79
	Permissions Reduct	81	80	81	79
Logistic Regression	All Permissions	79	79	77	78
	Permissions Correlation	79	79	77	78
	Permissions Chi	79	80	78	79
	Permissions Reduct	80	80	77	79

Likewise, Tables 14–16 summarize the results for the three types of features, i.e., opcode feature, API call feature, and system command feature, respectively. The same phenomenon is observed in these three tables as was observed in Table 13, i.e., for each type of classifier, as the feature set is changed from all feature to feature correlation then to feature chi and finally to feature reduct, the accuracy increases, and training and testing time gets reduced drastically.

Table 14. Detection results based on opcode.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All Opcodes	80	88	82	82
	Opcodes Correlation	80	75	87	82
	Opcodes Chi	81	75	87	81
	Opcodes Reduct	81	79	89	81
K-Nearest Neighbor	All Opcodes	84.50	86	83	84
	Opcodes Correlation	85	86	83	84
	Opcodes Chi	85	85	83	84
	Opcodes Reduct	85	85	83	84
Random Forest	All Opcodes	86	88	85	87
	Opcodes Correlation	86.80	88	86	87
	Opcode Chi	87	88	86	87
	Opcode Reduct	87	88	86	87
ANN	All Opcodes	78	77	86	81
	Opcodes Correlation	79	77	85	82
	Opcodes Chi	80	73	82	80
	Opcodes Reduct	79	73	83	78
CNN	All Opcodes	79	75	83	82
	Opcodes Correlation	80	74	85	80
	Opcodes Chi	80	74	84	78
	Opcodes Reduct	79	74	84	78
Logistic Regression	All Opcodes	79	76	86	81
	Opcodes Correlation	80	76	86	81
	Opcodes Chi	80	75	84	79
	Opcodes Reduct	80	74	85	79

Table 13 shows that the Random Forest classifier with the permission reduct feature attains a maximum accuracy of 83% and emerges as the best performing model. Similarly, for Tables 14–16, the best models are Random Forest with opcode reduct having an accuracy of 87%, Random Forest with API calls reduct having an accuracy of 90%, and Random Forest with system command reduct having an accuracy of 83%. Among all the models mentioned in the above four tables, Random Forest with API calls reduct emerges as the best model.

Table 15. Detection results based on API calls.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All API calls	84	83	88	85
	API calls Correlation	85	83	88	85
	API calls Chi	85	83	88	86
	API calls Reduct	86	85	89	83
K-Nearest Neighbor	All API calls	85	86	85	85
	API Calls Correlation	85.70	87	85	86
	API Calls Chi	86	87	85	86
	API Calls Reduct	86	87	84	86
Random Forest	All API Calls	88	90	88	90
	API Calls Correlation	89	88	90	88
	API Calls Chi	89	90	88	89
	API Calls Reduct	90	90	88	89
ANN	All API Calls	83	82	86	83
	API Calls Correlation	83	82	86	83
	API Calls Chi	83	82	86	83
	API Calls Reduct	84	81	86	83
CNN	All API Calls	84	82	87	84
	API Calls Correlation	84	82	87	84
	API Calls Chi	84	82	87	84
	API Calls Reduct	85	82	87	84
Logistic Regression	All API Calls	85	83	88	85
	API Calls Correlation	85	83	88	85
	API Calls Chi	85	83	88	85
	API Calls Reduct	86	83	88	85

Table 16. Detection results based on system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All Sys cmd	62	59	86	70
	Sys cmd Correlation	62.50	59	86	70
	Sys cmd Chi	62.80	59	86	70
	Sys cmd Reduct	63	59	86	70
K-Nearest Neighbor	All Sys cmd	79	77	82	80
	Sys cmd Correlation	79	77	82	80
	Sys cmd Chi	79	77	83	80
	Sys cmd Reduct	79	77	83	80

Table 16. *Cont.*

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Random Forest	All Sys cmd	82	81	85	83
	Sys cmd Correlation	82	81	86	83
	Sys cmd Chi	83	82	86	83
	Sys cmd Reduct	83	82	86	83
ANN	All Sys cmd	63	59	80	68
	Sys cmd Correlation	63	60	80	68
	Sys cmd Chi	64	60	81	70
	Sys cmd Reduct	64	60	81	71
CNN	All Sys cmd	64	60	81	70
	Sys Cmd Correlation	65	60	81	70
	Sys cmd Chi	65	62	82	70
	Sys cmd Reduct	65	61	82	71
Logistic Regression	All Sys cmd	65	61	82	70
	Sys cmd Correlation	65.13	61	82	70
	Sys cmd Chi	65.98	62	83	71
	Sys cmd Reduct	66	62	83	71

4.3. Detection Results with Combinations of Two Features

Table 17 shows that Random Forest emerges as the best algorithm in terms of accuracy, precision, recall, and F1-score for permissions and opcodes reduct as the feature set.

Table 17. Detection results based on permissions and opcodes.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + Opcode Reduct	85	86	83	84
K-Nearest Neighbor	Permission + Opcode Reduct	87	88	86	86
Random Forest	Permission + Opcode Reduct	90	92	88	90
ANN	Permission + Opcode Reduct	82	83	82	82
CNN	Permission + Opcode Reduct	83	84	82	83
Logistic Regression	Permission + Opcode Reduct	84	85	83	84

Table 18 shows results of permission and API calls reduct as feature set used by SVM, K-nearest neighbour, Random Forest, and Logistic regression. Random Forest emerges as the best, with an accuracy of 92%.

Table 18. Detection results based on permissions and API calls.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + API calls Reduct	87	87	85	87
K-Nearest Neighbor	Permission + API calls Reduct	88	88	87	88
Random Forest	Permission + API calls Reduct	92	92	90	91
ANN	Permission + API calls Reduct	85	84	84	84
CNN	Permission + API calls Reduct	86	85	85	85
Logistic Regression	Permission + API calls Reduct	87	86	86	86

Table 19 shows that Random Forest with an accuracy of 88% is proved to be the best detection model among all the other three detection models using permissions and system command reduct as the feature set.

Table 19. Detection results based on permissions and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + Sys Cmd Reduct	83	84	81	83
K-Nearest Neighbor	Permission + Sys Cmd Reduct	84	84	84	84
Random Forest	Permission + Sys Cmd Reduct	88	91	85	88
ANN	Permissions+ Sys Cmd Reduct	80	82	79	81
CNN	Permissions + Sys Cmd Reduct	81	83	80	82
Logistic Regression	Permission + Sys Cmd Reduct	82	84	81	83

Table 20 shows that the detection model with classifier as Random Forest and feature set as a combination of opcodes reduct and API calls reduct outperforms the other three detection models with an accuracy of 93%.

Table 20. Detection results based on opcodes and API calls.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Opcode + API call Reduct	88	88	86	88
K-Nearest Neighbor	Opcode + API call Reduct	90	89	88	89
Random Forest	Opcode + API call Reduct	93	93	92	93
ANN	Opcode + API call Reduct	86	85	85	86
CNN	Opcode + API call Reduct	87	86	86	87
Logistic Regression	Opcode + API call Reduct	88	87	87	88

Table 21 shows the detection results of the models that used opcode reduct and system command reduct as the feature set. The Random Forest classifier performed best with an accuracy of 90%.

Table 21. Detection results based on opcodes and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Opcode + Sys Cmd Reduct	84	85	82	84
K-Nearest Neighbor	Opcode + Sys Cmd Reduct	85	87	85	86
Random Forest	Opcode + Sys Cmd Reduct	90	90	88	89
ANN	Opcode + Sys Cmd Reduct	81	83	84	83
CNN	Opcode + Sys Cmd Reduct	82	84	85	84
Logistic Regression	Opcode + Sys Cmd Reduct	83	85	86	85

Table 22 shows that the detection model formed with the help of Random Forest as the classifier and API calls reduct and system command reduct as the feature set attains an accuracy of 91%, which is best among all the models in the table.

Table 22. Detection results based on API calls and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	API Call + Sys Cmd Reduct	85	86	84	86
K-Nearest Neighbor	API Call + Sys Cmd Reduct	86	88	86	87
Random Forest	API Call + Sys Cmd Reduct	91	91	88	90
ANN	API Call + Sys Cmd Reduct	82	84	86	83
CNN	API Call + Sys Cmd Reduct	83	85	87	84
Logistic Regression	API Call + Sys Cmd Reduct	84	86	88	85

4.4. Detection Results with Combinations of Three Features

Table 23 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 95%.

Table 23. Detection results based on permissions, API calls, and opcodes.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	permissions + API Call + opcode + Reduct	90	90	88	89
K-Nearest Neighbor	permissions + API Call + opcode + Reduct	92	91	90	91
Random Forest	permissions + API Call + opcode + Reduct	95	94	93	95
ANN	permissions + API Call + opcode + Reduct	88	89	89	88
CNN	permissions + API Call + opcode + Reduct	90	91	90	91
Logistic Regression	permissions + API Call + opcode + Reduct	90	89	89	90

Table 24 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 93%.

Table 24. Detection results based on permissions, API calls, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permissions + API Call + Sys Cmd + Reduct	86	87	85	87
K-Nearest Neighbor	Permissions + API Call + Sys Cmd + Reduct	87	89	87	88
Random Forest	Permissions + API Call + Sys Cmd + Reduct	93	92	90	92
ANN	Permissions + API Call + Sys Cmd + Reduct	84	85	84	84
CNN	Permissions + API Call + Sys Cmd + Reduct	85	87	87	85
Logistic Regression	Permissions + API Call + Sys Cmd + Reduct	85	87	88	86

Table 25 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 93%.

Table 25. Detection results based on permissions, opcodes, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + Opcode + Sys Cmd + Reduct	86	87	83	85
K-Nearest Neighbor	Permission + Opcode + Sys Cmd + Reduct	87	87	86	87
Random Forest	Permission + Opcode + Sys Cmd + Reduct	93	92	89	91
ANN	Permission + Opcode + Sys Cmd + Reduct	84	86	86	86
CNN	Permission + Opcode + Sys Cmd + Reduct	84	86	86	86
Logistic Regression	Permission + Opcode + Sys Cmd + Reduct	84	86	86	86

Table 26 shows combining the opcodes, API calls, and system command reducts and applying all four classifiers the detection model with the Random Forest as the classifier is best among all, with an accuracy of 94%.

Table 26. Detection results based on opcodes, API calls, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Opcode + API Call + Sys Cmd Reduct	90	91	88	90
K-Nearest Neighbor	Opcode + API Call + Sys Cmd Reduct	92	90	90	91
Random Forest	Opcode + API Call + Sys Cmd Reduct	94	94	93	94
ANN	Opcode + API Call + Sys Cmd Reduct	87	87	86	87
CNN	Opcode + API Call + Sys Cmd Reduct	88	88	87	88
Logistic Regression	Opcode + API Call + Sys Cmd Reduct	89	88	88	89

4.5. Detection Results with Combinations of all Four Features

Table 27 shows that all the feature set reducts, i.e., permissions, opcodes, API calls, and system commands reducts, used together with Random Forest emerge as the best classifier, with an accuracy of 97%.

Table 27. Detection results based on permissions, opcodes, API calls, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permissions + Opcode + API Call + Sys Cmd Reduct	92	91	90	91
K-Nearest Neighbor	Permissions + Opcode + API Call + Sys Cmd Reduct	93	93	92	92
Random Forest	Permissions + Opcode + API Call + Sys Cmd Reduct	97	95	95	95
ANN	Permissions + Opcode + API Call + Sys Cmd Reduct	89	88	87	89
CNN	Permissions + Opcode + API Call + Sys Cmd Reduct	90	89	88	89
Logistic Regression	Permissions + Opcode + API Call + Sys Cmd Reduct	91	90	89	91

4.6. Discussion and Findings

In this subsection, we summarize the reasoning behind the detection results achieved from the proposed model. The proposed model gives us the highest accuracy of 97% with all four features. We observe that the detection accuracy significantly improves when we combine all four types of features for detection. The reason behind this observation is that different categories of features capture different behavioral aspect of malicious application. Hence, combining different types of features gives us wholesome characteristics of a malicious application under consideration. Therefore, increasing the number of features in combinations helps to improve the overall accuracy.

Secondly, we observed that API calls give us better results as compared to other features individually. Malicious software developers might try to avoid being detected by

obscuring their code. Nevertheless, concealing the need for specific API calls is challenging. Identifying these calls can aid in the identification of malware that is intentionally concealed. Hence, results with API calls are better when compared to other features.

Thirdly, Random Forest gives us the better results when compared to other classifiers. The reason lies in the fact that Random Forest is a technique that employs a collection of decision trees to formulate predictions. This method of ensembling is beneficial for mitigating the variance and overfitting risks inherent in individual decision trees. Through the consolidation of predictions from numerous trees, there is a consistent enhancement in overall performance.

Fourthly, we can combine two types of feature sets out of permissions, opcodes, API calls, and system commands in six ways. We experimented on all these six combinations and observed that opcodes and API calls combinations give the highest accuracy among all the six types of combinations. The reason seems to be obvious from the fact that opcodes constitute low-level instructions executed by a processor. In the realm of Android malware detection, scrutinizing the sequence of opcodes within an app's code can unveil discernible patterns suggestive of malicious behavior. Simultaneously, examining the app's API calls offers insights into its conduct. Specific API calls may serve as markers for malicious activities, and analyzing their patterns can assist in identifying malware. Consequently, the amalgamation of opcodes and API examines a more detailed perspective on an app's behavior at the code level. This approach proves advantageous for pinpointing nuanced and sophisticated malware that may employ obfuscation techniques. In contrast, permissions offer a higher-level overview based on declared capabilities, potentially offering less specificity.

Similarly, we combined all these four features in the group of three. The total number of combinations possible are four. Experimenting with all these four combination establishes that combining permissions, opcodes, and API calls is best among them. The reason seem to be that permissions offer insights into an app's officially declared capabilities. Integrating permissions with opcodes and API calls enhances our comprehension of an app's intended functionality and the possibility of misuse. Specific combinations of permissions can act as early indicators of potential issues, even before delving into the details of code execution. While opcodes and API calls provide a granular view of an app's behavior at the code level, incorporating permissions into the analysis contributes to a broader context, fostering a more comprehensive understanding of an app's intentions and the associated risks.

4.7. Comparison with Other Related Work

We conducted a thorough comparison between the detection outcomes achieved by our suggested approach and those of other studies found in the existing literature regarding the detection of Android malware. We implemented several other state-of-the-art techniques on our data sets and to facilitate this comparison; we present a concise summary of the findings in Table 28, which encompasses the results obtained by various works that have utilized certain or all components of the manifest file for detection purposes. By examining these results, it becomes evident that our proposed methodology surpasses all of the aforementioned related works in terms of detection accuracy, signifying its superior performance in comparison to existing approaches.

Table 28. Comparison of proposed model with related works.

Detection Technique	**Feature Set Used**	**Detection Accuracy**	**No. of Applications**	**Feature Ranking Method**	**Feature Selection Method** (%)
SIGPID [8]	Permissions	92	5494 malicious & 310,926 benign apps	Negative rate & support	Sequential Forward Selection (SFS) & Principal Component Analysis (PCA).
PermPair [9]	Permissions	94.60	5993 benign & 7533 malicious applications	Ranked Permission-pairs	Not Used
Alazab et al. [51]	Permissions and API Calls	94.22	14,172 Benign & 13,719 Malicious Application	No ranking done	Not used
Urooj et al. [52]	Permissions, API Calls and other manifest components	96.40	50,000 Benign & 50,000 Malicious Applications	No ranking done	Not used
Zhu et al. [53]	Permissions, API Calls and hardware components	96	1061 Benign & 2126 Malicious Applications	No ranking done	Not used
Firdaus et al. [34]	Java Code and Permissions	94.20	5555 Malicious & 550 Begin	No Ranking Done	Genetic Search
DREBIN [31]	Manifest File Components & Dis ambled Java code	95	123,452 benign & 5560 malicious applications	no ranking done	Not used
Proposed Approach	Permissions, Opcode, API Calls, and System Calls	97	15,000 benign & 15,000 malicious	Rough Set based Ranking	Rough Set Reduct

4.8. Limitations

The work performed in this research paper is based on static analysis. Static Android malware analysis has shortcomings, such as not capturing the run time behavior of applications like data leakage and network communications. Due to obfuscation techniques employed by malware writers, static analysis may not be able to capture the true intention of the code. With these limitations in the picture, static analysis may miss the malicious behaviour of Android applications, which may show its actual hostile conduct at run time.

Additionally, the current proposed model is an off-device model, and hence it can not be installed on smartphones for real-time detection.

5. Conclusions and Future Scope

In this work, we have proposed a novel Android malware detection model based on rough set theory. We have considered the combination of four static features, namely, permissions, opcodes, API calls, and system commands. Initially, a data pre-processing stage was executed, during which correlated features, as well as features that exhibited no dependence on the class variable, were eliminated. A ranking score was computed to establish the significance of each feature by employing the concept of Discernibility Matrices from rough set theory. Subsequently, an algorithm for computing rough set reducts was employed to reduce the number of features within each category based on the ranking score and the Discernibility Matrix concept of rough set theory. Following this reduction step, machine learning algorithms were applied to assess the detection accuracy using the calculated reducts. To gauge the effectiveness of the proposed model, a comparison was drawn against other advanced detection techniques. The results of this comparison underscored the superior performance of the proposed model over other state-of-the-art models in the field.

In our future work, we will try to work on the limitations of static analysis, i.e, we will be using dynamic analysis techniques along with static analysis techniques to build hybrid malware detection methods. Currently, the proposed model is an off-device model, and hence it can not be installed on smartphones for real-time detection. In future, we will propose a client–server-based model to integrate in smartphones.

Author Contributions: R.G. contributed to the conception of design of algorithms. R.G. and K.S. contributed to the implementation of the algorithms. R.G., K.S. and R.K.G. contributed in the acquisition, analysis, and interpretation of data. Each author approved the version of the research paper to be published. Each author verified the integrity of the results and all other details related to the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare that they have no conflicts of interest.

References

1. Petrov, C. 51 Mobile vs. Desktop Usage Statistics for 2023; Technical Report. Available online: https://techjury.net/blog/mobile-vs-desktop-usage/ (accessed on 27 July 2023).
2. Statcounter: Mobile Operating System Market Share Worldwide. 2023. Statcounter. Available online: https://gs.statcounter.com/os-market-share/mobile/worldwide (accessed on 27 July 2023).
3. SHISHKOVA, T. The Mobile Malware Threat Landscape in 2022. Technical Report. SECURELIST by Kaspersky (February 2023). Available online: https://securelist.com/mobile-threat-report-2022/108844/ (accessed on 27 July 2023).
4. McAfee. Goldoson: Privacy-Invasive and Clicker Android Adware Found in Popular Apps in South Korea. 2023. McAfee. Available online: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/goldoson-privacy-invasive-and-clicker-android-adware-found-in-popular-apps-in-south-korea/?AID=11552066&PID=9129747&SID=tomsguide-in-8524612056782596000 (accessed on 27 July 2023).
5. Fadilpašić, S. This Dangerous New Malware Also Has Ransomware Capabilities. Techradar, 2023. Available online: https://www.techradar.com/news/this-dangerous-new-malware-also-has-ransomware-capabilities (accessed on 27 July 2023).

6. Petrosyan, A. Development of Android Malware Worldwide 2016–2020. Statista, 2020. Available online: https://www.statista.com/statistics/680705/global-android-malware-volume/ (accessed on 27 July 2023).
7. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The evolution of android malware and android analysis techniques. *ACM Comput. Surv. (CSUR)* **2017**, *49*, 1–41. [CrossRef]
8. Li, J.; Sun, L.; Yan, Q.; Li, Z.; Srisa-An, W.; Ye, H. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Inf.* **2018**, *14*, 3216–3225. [CrossRef]
9. Arora, A.; Peddoju, S.K.; Conti, M. Permpair: Android malware detection using permission pairs. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 1968–1982. [CrossRef]
10. Khariwal, K.; Singh, J.; Arora, A. Ipdroid: Android malware detection using intents and permissions. In Proceedings of the 2020 Fourth World Conference on Smart Trends in Systems, Security, and Sustainability (WorldS4), London, UK, 27–28 July 2020; pp. 197–202.
11. Wu, D.-J.; Mao, C.-H.; Wei, T.-E.; Lee, H.-M.; Wu, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In Proceedings of the 2012 Seventh Asia Joint Conference on Information Security, Tokyo, Japan, 9–10 August 2012; pp. 62–69.
12. Elish, K.O.; Shu, X.; Yao, D.D.; Ryder, B.G.; Jiang, X. Profiling user-trigger dependence for android malware detection. *Comput. Secur.* **2015**, *49*, 255–273. [CrossRef]
13. Zhang, H.; Luo, S.; Zhang, Y.; Pan, L. An efficient android malware detection system based on method-level behavioral semantic analysis. *IEEE Access* **2019**, *7*, 69246–69256. [CrossRef]
14. Zhu, H.; Wei, H.; Wang, L.; Xu, Z.; Sheng, V.S. An effective end-to-end android malware detection method. *Expert Syst. Appl.* **2023**, *218*, 119593. [CrossRef]
15. Fang, Y.; Gao, Y.; Jing, F.; Zhang, L. Android malware familial classification based on dex file section features. *IEEE Access* **2020**, *8*, 10614–10627. [CrossRef]
16. Yen, Y.-S.; Sun, H.-M. An android mutation malware detection based on deep learning using visualization of importance from codes. *Microelectron. Reliab.* **2019**, *93*, 109–114. [CrossRef]
17. Grace, M.C.; Zhou, W.; Jiang, X.; Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, Tucson, AZ, USA, 16–18 April 2012; pp. 101–112.
18. Enck, W.; Ongtang, M.; McDaniel, P. On lightweight mobile phone application certification. In Proceedings of the 16th ACM Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 235–245.
19. Talha, K.A.; Alper, D.I.; Aydin, C. Apk auditor: Permission-based android malware detection system. *Digit. Investig.* **2015**, *13*, 1–14. [CrossRef]
20. Choi, J.; Sung, W.; Choi, C.; Kim, P. Personal information leakage detection method using the inference-based access control model on the android platform. *Pervasive Mob. Comput.* **2015**, *24*, 138–149. [CrossRef]
21. Song, J.; Han, C.; Wang, K.; Zhao, J.; Ranjan, R.; Wang, L. An integrated static detection and analysis framework for android. *Pervasive Mob. Comput.* **2016**, *32*, 15–25. [CrossRef]
22. Wang, Z.; Li, C.; Yuan, Z.; Guan, Y.; Xue, Y. Droidchain: A novel android malware detection method based on behavior chains. *Pervasive Mob. Comput.* **2016**, *32*, 3–14. [CrossRef]
23. Sasidharan, S.K.; Thomas, C. Prodroid—An android malware detection framework based on profile hidden markov model. *Pervasive Mob. Comput.* **2021**, *72*, 101336. [CrossRef]
24. Moonsamy, V.; Rong, J.; Liu, S. Mining permission patterns for contrasting clean and malicious android applications. *Future Gener. Comput. Syst.* **2014**, *36*, 122–132. [CrossRef]
25. Idrees, F.; Rajarajan, M. Investigating the android intents and permissions for malware detection. In Proceedings of the 2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Larnaca, Cyprus, 8–10 October 2014; pp. 354–358.
26. Wang, W.; Wang, X.; Feng, D.; Liu, J.; Han, Z.; Zhang, X. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Trans. Inf. Forensics Secur.* **2014**, *9*, 1869–1882. [CrossRef]
27. 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, 50–52.
28. Qiu, J.; Zhang, J.; Luo, W.; Pan, L.; Nepal, S.; Wang, Y.; Xiang, Y. A3cm: Automatic capability annotation for android malware. *IEEE Access* **2019**, *7*, 147156–147168. [CrossRef]
29. Taheri, R.; Ghahramani, M.; Javidan, R.; Shojafar, M.; Pooranian, Z.; Conti, M. Similarity-based android malware detection using hamming distance of static binary features. *Future Gener. Comput. Syst.* **2020**, *105*, 230–247. [CrossRef]
30. Bai, H.; Xie, N.; Di, X.; Ye, Q. Famd: A fast multifeature android malware detection framework, design, and implementation. *IEEE Access* **2020**, *8*, 194729–194740. [CrossRef]
31. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. *NDSS* **2014**, 14, 23–26.
32. Varsha, M.; Vinod, P.; Dhanya, K. Identification of malicious android app using manifest and opcode features. *J. Comput. Virol. Hacking Tech.* **2017**, *13*, 125–138. [CrossRef]

33. Mahindru, A.; Sangal, A. Fsdroid:-a feature selection technique to detect malware from android using machine learning techniques: Fsdroid. *Multimed. Tools Appl.* **2021**, *80*, 13271–13323. [CrossRef] [PubMed]
34. Firdaus, A.; Anuar, N.B.; Karim, A.; Razak, M.F.A. Discovering optimal features using static analysis and a genetic search based method for android malware detection. *Front. Inf. Technol. Electron. Eng.* **2018**, *19*, 712–736. [CrossRef]
35. Kumaran, M.; Li, W. Lightweight malware detection based on machine learning algorithms and the android manifest file. In Proceedings of the 2016 IEEE MIT Undergraduate Research Technology Conference (URTC), Cambridge, MA, USA, 4–6 November 2016; pp. 1–3.
36. Feldman, S.; Stadther, D.; Wang, B. Manilyzer: Automated android malware detection through manifest analysis. In Proceedings of the 2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems, Philadelphia, PA, USA, 28–30 October 2014; pp. 767–772.
37. Sanz, B.; Santos, I.; Laorden, C.; Ugarte-Pedrero, X.; Nieves, J.; Bringas, P.G.; Álvarez Marañón, G. Mama: Manifest analysis for malware detection in android. *Cybern. Syst.* **2013**, *44*, 469–488. [CrossRef]
38. Li, C.; Mills, K.; Niu, D.; Zhu, R.; Zhang, H.; Kinawi, H. Android malware detection based on factorization machine. *IEEE Access* **2019**, *7*, 184008–184019. [CrossRef]
39. Sato, R.; Chiba, D.; Goto, S. Detecting android malware by analyzing manifest files. *Proc. Asia-Pac. Adv. Netw.* **2013**, *36*, 17. [CrossRef]
40. Zhang, M.; Duan, Y.; Yin, H.; Zhao, Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 1105–1116.
41. Feng, Y.; Anand, S.; Dillig, I.; Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 576–587.
42. Wang, W.; Gao, Z.; Zhao, M.; Li, Y.; Liu, J.; Zhang, X. Droidensemble: Detecting android malicious applications with ensemble of string and structural static features. *IEEE Access* **2018**, *6*, 31798–31807. [CrossRef]
43. Xu, Z.; Ren, K.; Qin, S.; Craciun, F. Cdgdroid: Android malware detection based on deep learning using cfg and dfg. In Proceedings of the Formal Methods and Software Engineering: 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, 12–16 November 2018; Proceedings 20; pp. 177–193.
44. Xiao, X.; Zhang, S.; Mercaldo, F.; Hu, G.; Sangaiah, A.K. Android malware detection based on system call sequences and lstm. *Multimed. Tools Appl.* **2019**, *78*, 3979–3999. [CrossRef]
45. Qin, X.; Zeng, F.; Zhang, Y. Msndroid: The android malware detector based on multi-class features and deep belief network. In Proceedings of the ACM Turing Celebration Conference, Chengdu, China, 17–19 May 2019; pp. 1–5.
46. Martín, A.; Lara-Cabrera, R.; Camacho, D. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Inf. Fusion* **2019**, *52*, 128–142. [CrossRef]
47. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, Austin, TX, USA, 14–15 May 2016; pp. 468–471. [CrossRef]
48. Martín, A.; Lara-Cabrera, R.; Camacho, D. A new tool for static and dynamic Android malware analysis. In Data Science and Knowledge Engineering for Sensing Decision Support. In Proceedings of the 13th International FLINS Conference (FLINS 2018), River Edge, NJ, USA, 21–24 August 2018; World Scientific: Singapore, 2018; pp. 509–516.
49. Pawlak, Z. Rough Set Theory and its Applications to Data Analysis. *Cybern. Syst.* **1998**, *29*, 661–688. [CrossRef]
50. Zhang, Q.; Xie, Q.; Wang, G. A survey on rough set theory and its applications. *CAAi Trans. Intell. Technol.* **2016**, *1*, 323–333. [CrossRef]
51. Alazab, M.; Alazab, M.; Shalaginov, A.; Mesleh, A.; Awajan, A. Intelligent mobile malware detection using permission requests and API calls. *Future Gener. Comput. Syst.* **2020**, *107*, 509–521. [CrossRef]
52. Urooj, B.; Shah, M.A.; Maple, C.; Abbasi, M.k.; Riasat, S. Malware detection: A framework for reverse engineered android applications through machine learning algorithms. *IEEE Access* **2022**, *10*, 89031–89050. [CrossRef]
53. Zhu, H.J.; Gu, W.; Wang, L.M.; Xu, Z.C.; Sheng, V.S. Android malware detection based on multi-head squeeze-and-excitation residual network. *Expert Syst. Appl.* **2023**, *212*, 118705–118713. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.