IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 14, 2022, accepted February 25, 2022, date of publication March 2, 2022, date of current version March 8, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3156083

Deep-Layer Clustering to Identify Permission Usage Patterns of Android App Categories

ZAKEYA NAMRUD, SÈGLA KPODJEDO, AHMED BALI, AND CHAMSEDDINE TALHI
Department of Software Engineering and IT, École de Technologie Supérieure (ÉTS), Montreal, QC H3C 1K3, Canada
Corresponding author: Zakeya Namrud (zakeya.namrud.1@ens.etsmtl.ca)

ABSTRACT With the increasing usage of smartphones in banks, medical services and m-commerce, and the uploading of applications from unofficial sources, security has become a major concern for smartphone users. Malicious apps can steal passwords, leak details, and generally cause havoc with users' accounts. Current anti-virus programs rely on static signatures that need to be changed periodically and cannot identify zero-day malware. The Android permission system is the central security mechanism that regulates the execution of application tasks. Although recent advances in research have provided various approaches and detection methods for finding malware apps, the available literature lacks a full analysis of this subject. We fill this gap by: 1) Systematically and automatically building a large dataset of malware and benign apps, which we have made available to the community. Our dataset has around 16K apps and 118 features. 2) We offer a novel approach for automatically identifying permission usage patterns, which are groupings of permissions that developers frequently utilise together. The approach combines SOM and K-means clustering algorithms to classify permissions according to app usage categories. The results demonstrate that the proposed methodology is able to detect most of the consistent and coherent permission usage patterns across a wide variety of application categories. To assess our strategy, we add the identified patterns as features to our dataset and then apply an SVM classifier for malware detection. Our results indicate that the identified patterns improve the performance of the classifier.

INDEX TERMS Android permissions, android malware, empirical study, supervised and unsupervised machine learning, self organizing map (SOM).

I. INTRODUCTION

User statistics show that Android is the most widely used operating system (OS) on mobile devices and is expected to remain the most popular OS until 2023 [29]. Smartphones have been a key target for application developers who wish to exploit them for malicious purposes. Malicious tech is one of the biggest challenges with any software platform, and Android is no exception. Android apps can pose severe threats for Android users. According to Gartner, by the end of 2020, mobile applications were downloaded over 493 million times per day, generating more than $198 billion in revenue and making them popular computing tools for users worldwide. Such huge numbers are mostly driven by the Google Android mobile OS, which has an impressive smartphone market share of 82.8% [13]. This is mainly because it is open source and has a massive collection of applications in the official Android app store as well as in third-party Android app stores. However, their popularity comes at a cost: Android apps are also a vehicle for spreading vulnerabilities. A key security mechanism of Android is its permission system, which controls the privileges of applications. Under this system, apps must request access to particular permissions in order to perform certain functionalities. Moreover, the mechanism requires that app developers declare which sensitive resources will be used by their applications. Users have to agree with the requests when installing/running the applications. This constrains a given application to the resources it can request at run-time. Android has established a set of best practices designed to help developers properly define and operate permissions inside their source code. Unfortunately, there is no integrated security mechanism to guarantee that the apps only ask for the permissions they need. Moreover, developers do not always adhere to best practices guidelines [17], which makes the applications more sensitive to security issues.

The associate editor coordinating the review of this manuscript and approving it for publication was Shen Yin.

In this paper, we explore the use of 103 permissions for around 16K apps on the Android market. First, we investigate permission use for apps in different categories. Then we present a novel methodology for mining permission usage patterns, which we refer to as SOM+K-means. A permission use pattern is defined as a group of permissions utilised together in apps. Our strategy is based on a comparison of how permissions are used together and their correlation to apps across different categories. The patterns' permissions are dispersed over several use cohesion levels/layers. Each level indicates the frequency of co-usage of a set of permissions, while the distribution across various levels illustrates the degree of co-usage.

Our approach utilises a form of SOM+K-means, which is a commonly used clustering technique. SOM+K-means will identify probable permission usage patterns based on an investigation of its usage frequency and consistency across a number of apps within different categories. Utility permissions may be used by apps belonging to several categories. As a result, the logic behind distributing permissions in a pattern based on different levels of use cohesiveness is to distinguish between the most and least particular permissions. Additionally, our methodology is also designed to be used to find patterns associated with specific permissions that are of interest to a developer. SOM+K-means provides a pattern-recognition engine to aid developers in examining various permission usage patterns. So, we investigate the permission use for different categories of apps. Furthermore, we assess the scalability of SOM+K-means as well as the generalizability of the detected usage patterns to possible malware detection using Support Vector Machine (SVM). Our findings reveal that, across a wide range of apps in different categories, the detected usage patterns via SOM+K-means improve the malware detection model's effectiveness. The following is a brief summary of the paper's significant contributions:

1) Using an adapted combination of deep learning and the K-means clustering algorithm, we provide a novel strategy for mining deep-layer permission usage patterns.
2) We create and mine a big dataset of over 16K Android applications from the Google Play Store, investigating around 46 categories and studying their use of 103 permissions.
3) We assess our approach's efficacy by examining the coherence and generalizability of the identified patterns. The results reveal that our method was able to discover a greater number of usage patterns at various degrees of usage cohesiveness.

The remainder of this paper is structured as follow: We begin with a brief background in Section II. In Section III, we describe the data gathering procedure and the study's objectives. Section IV details our strategy. Section VI summarises the related work. Finally, Section VII concludes and outlines future work.

II. BACKGROUND

A. ANDROID SECURITY

Android's security is reflected in the application's deployment and execution. A digital certificate is required to sign Android apps for them to be installed on the mobile device. Android uses a separate process for each application to run. When an application is installed, it is given a unique and permanent user ID [25]. An application can't directly access another application's data over this boundary. Furthermore, Android provides a wide range of permissions for securing private data. To keep up with the demands of the development cycle, many developers employ permissions without considering their security implications, increasing the danger of security and privacy leaks.

B. PERMISSION SYSTEM

In a pessimistic scenario, all Android applications are considered to be implicitly buggy or malicious. The apps run in a process with a restricted user ID and are able to access their own files only by default. If a given application requires information or resources outside its sandbox, the permission must be explicitly requested. Permission may be granted automatically by the system, or the system may request the user to grant permission. Each Android application defines an XML-formatted file (Android Manifest.xml), which, along with other metadata such as minimal OS version requirements, contains the permission declarations to which it is requesting access [9]. The required permission attributes are used to declare permissions in the manifest, which is supplemented by a common namespace. For Google-defined permissions, this is usually Android.permission. Applications can demand self-declared permissions, while component permissions are identified by their tag names.

The Android manifestation includes entries automatically generated by the developer environment. However, some fields must be inserted manually, particularly those relating to permission declarations [5].

Android's permissions are classified into four levels of protection, as follows:

- Normal (lower-risk permission, which grants demanding applications access to isolated application level features).
- Dangerous (higher-risk permission, which grants a demanding application access to control the device or private user data).
- Signature (permission is granted only if the declaring application and requesting application have been done with the same certificate).
- SignatureOrSystem (A permission that the system only allows to apps that are in the Android system image or are signed with the same certificate as the app that declared the permission).

At runtime, Android apps enforce permissions, but at install time, the user must accept permissions. When a new application is installed by users in Android (regardless of how the application is obtained), the application prompts users

TABLE 1. Level of permission protection [8].

Level of protection	Description 10
Normal	A reduced risk that allows isolated application rights level features to be enabled while posing minimal danger to other applications, the system, or the user is available.
Dangerous	A higher-risk permission that A higher-risk permission that grants a requesting application access to sensitive user data or control over the device, both of which might have a detrimental impact on the user.
Signature	A permission that the system will only issue if the seeking application is signed with the same certificate as the one that declared the permission.
SignatureOrSystem	A permission that the system only allows to apps that are in the Android system image or are signed with the same certificate as the app that declared the permission.

to accept or deny the permissions requested. On Android 5.1 or earlier devices, application permissions are all required or all denied, which means that users have no choice. They can either accept all permissions or refuse the application altogether. In the latter case, they cannot use the application at all, because they did not agree with certain permissions.

Since version 6.0 of Android, however, users are able to grant permissions while running applications. This means that permission is no longer required to be granted during the initial installation of an application. Version 6.0 (update) has provided users with improved functionality and control over their applications. It gives them the possibility to revoke app permissions at any time and one by one via the application's setting interface. For instance, a user might choose to grant a particular mode of transport application access to the location of their device, while rejecting access to their contact list or SMS services. Tables 1 and 2 describe permission protection levels, including dangerous permissions.

C. CLUSTERING MODEL

1) SELF-ORGANIZING MAP

SOM [21] is an unsupervised learning network architecture in the area of machine learning. It is able to map high-dimensional data onto a two-dimensional space usually defined as a map. The map is given as the set of nodes within the input space field. This mapping indicates the similarity between the input patterns as the proximity to the map. It offers an understandable methodology to capture and classify the permissions of Android apps. Each SOM node is associated with a weight vector that has the same size as the input vector. The learning algorithm repeats over the input vectors and adjusts the weight vectors in accordance with what the algorithm pulls in. For each input vector, the equivalent weight vector is chosen and manipulated to be more like the original. Further, the neighbours of the best-matched weight vector are also modified using a learning algorithm. This helps ensure convergence over several iterations.

In 2000, Bengio *et al.* [6] proposed using SOM for data clustering in order to achieve better results and reduce computing time. In 2013, Abaei *et al.* [1] reported that SOM can be utilized instead of K-means for data clustering.

TABLE 2. Dangerous permissions and their related groups in Android 6.0 [8].

Permission Group	Permissions
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

TABLE 3. New or changed permission groups in Android 8 (marked with (*)) and 9 [8].

Permission Group	Permissions
CALL_LOG	READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS
PHONE	READ_PHONE_STATE READ_PHONE_NUMBERS(*), CALL_PHONE, ANSWER_PHONE_CALLS(*), ADD_VOICEMAIL, USE_SIP, ACCEPT_HANDOVER

In recent years, study and implementation in similar fields has shown that SOM and K-means can be merged to construct a better tool for data clustering [38].

2) CLUSTERING ANALYSIS BY K-MEANS METHOD

K-means is the simplest of the clustering algorithms. It employs squared error as its criterion [20]. K-means begins with a random initial partition and continues to reassign patterns to clusters on the basis of the similarities between the cluster centres and the pattern(s) until the convergence criteria are met. Patterns would not be reassigned from one cluster to another, as the squared error would then cease to decrease dramatically after a number of iterations.

3) SILHOUETTE INDEX

Silhouette index [31] is a highly useful indicator of cluster validity. It refers to methods for the interpretation and evaluation of consistency within clusters of data. The technique provides a sense of how well each object is categorised by displaying a clear picture of how successfully each element is classified. The silhouette value is used to determine how close an entity is to its own cluster in relation to other clusters (separation). The silhouette varies in accuracy from (-1) to $(+1)$, where a high value means that the object is well-suited to its own cluster and is poorly matched to neighbouring clusters.

III. STUDY OBJECTIVES AND DATA COLLECTION

Our motivation for this empirical study stems from: i) the absence of a built-in verification system to ensure that no unnecessary permissions are requested, which reduces the attack surface and makes the applications more exposed to security issues; and ii) the poor results of the Google Play Protect[1] system. Indeed, a recent evaluation of the best antivirus software for Androids, performed at the software testing laboratory AV-Test,[2] has reported that the Play Protect system detected 76.4% of threats in September 2020.[3]

Our main goals are the following: (1) To clarify the permission system use in different categories of Android applications, and (2) to investigate the potential risk for these applications to be harmful. In order to achieve our goals, we started by collecting our dataset and labeled the data with respect to the dangerousness of the required permission and the harmfulness risk of the application. In the following, we describe how we built the dataset used in our study.

A. DATA COLLECTION

For our data collection, we used the AndroZoo repository, which contained over 14,560,903 apps at the time we accessed it. The AndroZoo repository proposes data on the APKs it archived in a main CSV file containing important information for each application, including hash keys (such as sha256, sha1, md5), size information (for APKs and DEX), date of binary, package name, version code and market place, as well as information about how well the app fared on the VirusTotal website (number of antiviruses that flag the app as a malware, scan date).[4]

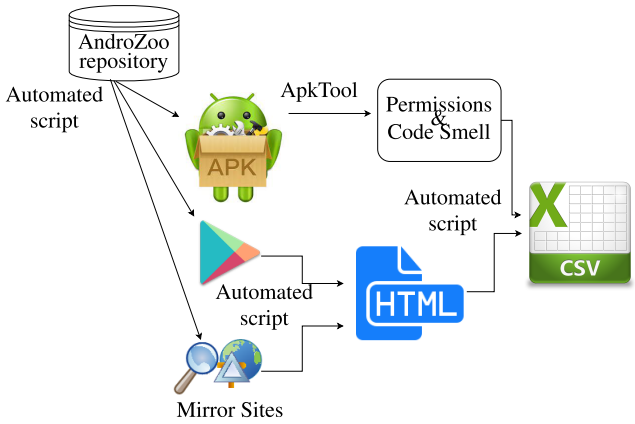


FIGURE 1. Overview of building the dataset.

In this section we explain the procedure that we followed to create our two datasets. Figure 1 illustrates the overview of collecting and building the data. The starting point was downloading the information file for the AndroZoo repository, targeting the apps from Google Play Store from 2019 and 2020. Then we randomly selected our 16K samples. Each app from AndroZoo has its info (i.e., sha256, sha1, md5, apk size, dex size, dex date, pkg name, vercodevt detection, vt scan date, markets). Next, we deployed the information from the AndroZoo for each app to download its APK file and HTML page. However, a significant number of those apps were removed from the Google Play Store for policy[5] reasons. This prompted us to search for it on mirror sites.

In finding the desired mirror site, however, we faced several issues, including language and difficulties downloading the html page automatically (no pattern used). To address these issues, we conducted extensive research and experiments to download the HTML pages automatically. By the end of this step, we had collected APK files and their HTML pages. The experimental dataset numbered 15,894 samples and 103 features (permissions). Clustering is an unsupervised process, so there is no need to know the class label of the samples. However, in order to check the efficiency and consistency of the clustering model, we need to know the class labels of the experimental cases, i.e., we must differentiate between "benign" and "malware" so that we can distinguish malware.

1) FEATURE EXTRACTION

We used the info related to the 15,894 samples to download their APK files from the AndroZoo repository. We then used these files as input to Apktool[6] (reverse engineer tool) and obtain the manifest file. Next, we modified AndroVul [28]. We employed the info related to those 15,894 to download their apk files from the AndroZoo repository, after which

[1] www.android.com/play-protect/
[2] www.av-test.org/en/about-the-institute/
[3] https://www.av-test.org/en/antivirus/mobile-devices/android/september-2020/google-play-protect-21.6-203809/
[4] More information here https://AndroZoo.uni.lu/lists
[5] https://play.google.com/about/developer-content-policy/
[6] https://ibotpeaches.github.io/Apktool/

Algorithm 1: Feature Extraction

Input : Android Applications, (Apk files & HTML pages & AndroZoo info) Dataset D
Output: A CSV file contains encoded feature vectors for each App in the dataset

for *(allf* $\in D$*)* **do**
  $APK_{file} \leftarrow Open(f)$
  $manifest_{File} \leftarrow ApkTool_{APK_{file}}$
  $Permissions_{list} \leftarrow Get\_Distinct\_Permissions(manifest_{File})$
  $Metadata_{list} \leftarrow Get\_Distinct\_Metadata(HTML_{File})$
  for *(each_permissions* $\in$ *Permissions)* **do**
    if $permissions \in Permissions_{list}$ **then**
      $Vector_{(Permissions)} \leftarrow 1$
    end if
    else
      $Vector_{(Permissions)} \leftarrow 0$
    end if
  end for
end for
$CSV_{(file)} \leftarrow Append(CSV_{(file)}, Concat(Vector_{(AndroZoo\_info)}, Vector_{(Metadata)}, Vector_{(Permission)}))$
return $(CSV_{(file)})$

the apk files were used as input to reverse-engineer and obtain the manifest file. To do so, we accessed Apktool, and then modified AndroVul [28] to extract all the relevant features, including 103 permissions in the manifest file. In the meantime, HTML pages were deployed to parse metadata and extract the desired features, such as category, rate, date of update, number of downloads, etc. This step resulted in a dataset that contains around 16K samples in a CSV file, including 118 features from different sources. Table 4 explains the dataset contents. Currently, there are 103 features (permissions) for the learning system. Algorithm 1 explains the procedure for extracting and mapping the features in our dataset. We labeled the permission list to distinguish between Dangerous, Normal, and Signature permissions according to the protection level reported in the Android documentation. We also used different tags to distinguish between permissions giving access to hardware and those giving access to user information in order to investigate the differences in terms of permission use between different categories of applications.

2) APPLICATIONS CATEGORIES

When a developer releases an application on Google Play Store, he/she is required to specify the category for the application's release. Currently, Google Play Store has around 46 categories. The distribution is shown in the dataset in Table 5. Applications are sorted within each category depending on a range of factors, such as ratings, reviews, downloads, country of origin, etc. We have done an exhaustive analysis and found that the number of malwares is not standardised across all categories. Certain categories such as education, entertainment, games, and tools are particularly vulnerable to malware, while others such as Word, comics, and events are slightly safer from security threats. In our research, we purposefully look for ways to better leverage this knowledge.

TABLE 4. The dataset contents.

Source	Feature's name	Feature's type
AndroZoo info	Package_ name	String
	SHA256	String
	SHA1	String
	MD5	String
	Apk _Size	Integer
	VT- Detection	Integer
	Date of scan	Date
	Dex_size	Integer
	Vercode	Integer
APK file	Permissions requested 103 features	Binary
	Category'name	String
HTML page	App's installs	Integer
	Updated_date	Date

IV. PROPOSED APPROACH

In this section, we introduce our approach and the methodology based on mining permission usage patterns of apps from different categories. Before delving into the algorithm, we present a brief background, an overview of our method, and a description of our experiments for investigating the identified permission usage patterns. Figure 2 shows the overview of the procedure of producing inferred pattern.

A. APPROACH OVERVIEW

Our technique begins with a collection of apps and a diverse range of permission schemes collected from their apk files. The output is a collection of permission usage patterns, each of which is a collection of apps arranged into distinct layers based on their frequency of co-use. We define a pattern of app co-usage as a collection of applications that are frequently used in conjunction with each other. A pattern is a collection of permissions that are dispersed over many usage cohesion layers. A cohesion layer reflects the frequency of co-use between apps. Indeed, similar permission usage patterns may exist across specific apps, and those apps are more typically classified as belonging to the same category. As a result, we are looking for an approach that can record co-usage relationships between permission usage patterns and app categories at various levels.

1) OUR APPROACH IS AS FOLLOWS

The input dataset is analysed to identify the various permissions that are unique to each app. Every application in the dataset is assigned a usage vector that contains information about used permissions. We aggregate the apps that are most commonly co-used by permissions using the K-means clustering algorithm based on the SOM deep learning cluster. Permissions that are not consistently used across apps in a category are segregated and treated as noisy data.

TABLE 5. The distribution of Benign & Malware App Categories in the dataset.

Category	*Benign*	*Malware*	*Category*	*Benign*	*Malware*
Action	243	32	Maps	101	11
Adventure	206	10	Medical	148	29
Arcade	266	32	Music	938	165
Art	158	64	News	370	68
Auto	105	34	Not found	193	97
Beauty	63	13	Parenting	12	3
Board	104	9	Personalization	651	62
Books	755	138	Photography	237	51
Business	671	96	Productivity	422	60
Card	80	9	Puzzle	423	27
Casino	48	8	Racing	95	12
Casual	327	42	Role playing	233	26
Comics	50	5	Shopping	273	51
Communication	292	44	Simulation	269	19
Dating	27	7	Social	177	48
Education	1183	246	Sports	331	55
Entertainment	839	201	Strategy	122	14
Events	39	7	Tools	994	345
Finance	482	62	Travel	266	38
Food	201	30	Trivia	53	12
Health	243	55	Video players	115	33
House	64	23	Weather	40	12
Libraries	26	8	Word	32	1
Lifestyle	411	101			

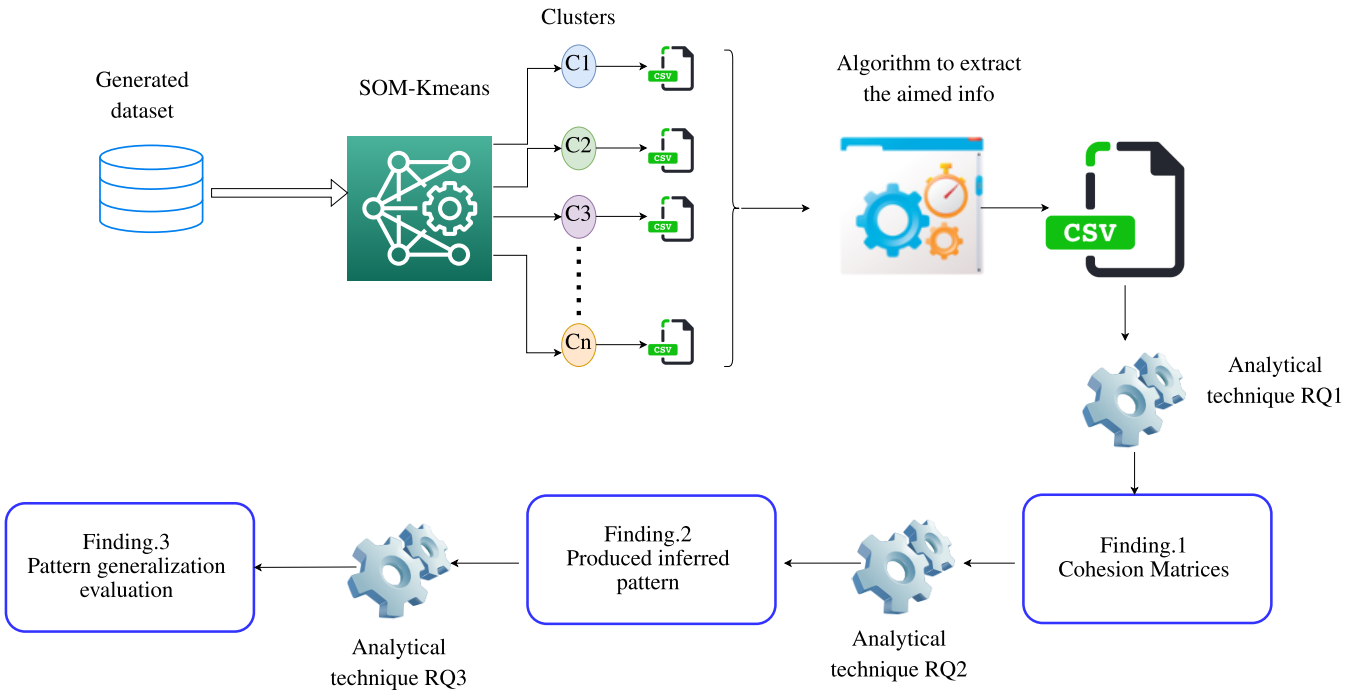


FIGURE 2. Overview of the procedure of producing inferred pattern.

B. DEEP-LAYER CLUSTERING

Our study aims to investigate the use of permissions, especially dangerous ones, in Android applications and their prediction potential for risk (malware). More specifically, we seek to understand and identify the weaknesses of the Android permission model. Although the various techniques of analysis and data mining are certainly applicable, we build a cluster model that combines the clustering of SOM and K-means centered on the silhouette index, which is a cluster validity measure. The model inherits SOM's advantage (unsupervised deep learning) and K-means clustering is applied to the SOM results, addressing one of the drawbacks (nodes with questionable clustering boundaries) of SOM. Furthermore, the findings do not always yield a simple clustering due to the number of initial nodes and the order of cases. The silhouette index is used by the model to assess the validity of various clustering outcomes. As previously mentioned, we suggested a two-stage strategy clustering approach to improve grouping accuracy.

SOM is a technique for mapping high-dimensional data to a low-dimensional space for easy understanding.

1) Weight values are initialised with random numbers.
2) Every neuron calculates the squared Euclidean distance between the vector being processed and its weight vector, which is a measure of the difference between the input pattern and the neuron's output.
3) The winning unit is the one that best approximates the input (the best matching unit). This formula is used for distance calculation, as follows:

$$Dist = \sqrt{\sum_{i=0}^{i=n}(Vi - Wi)^2} \tag{1}$$

where V is the current input vector and W is the node's weight vector. We take a set of inputs and measure the absolute difference between them and the neuron. Then we square the difference and sum the results. The winner will be the node that yields the smallest square root.

4) A topological neighborhood of excitable neurons appears around the winning node. The topological neighborhood model looks like this:

$$Tj, I(x) = \exp(-S^2 j, I(x)/2\sigma^2) \tag{2}$$

where Sj, I is the lateral distance between two neurons ($j\&I$), $I(x)$ is the winning neuron, and σ is the neighborhood size. The neighbourhood radius in an SOM must reduce over time and must be accomplished using an exponential formula. All excited neurons change their weight vectors values to align with the input patterns. The weight vectors of the winning unit are shifted closer to the input, and we change the weight vectors of the units in its neighbourhood, but to a smaller degree. The farther the unit is from the best matching unit, the less it is changed. The weight update formula used in this work is given below:

$$\Delta Wj, i = \eta(t) * Tj, I(x)(t) * (xi - wj, I) \tag{3}$$

where $\eta(t)$ is the learning rate, $Tj, I(x)(t)$ is the topological neighborhood, t is an epoch, i is neuron, j is another neuron, and $I(x)$ is best matching unit; Hence, this denotes the winning neuron.

The K-means algorithm is used in the second stage for cluster analysis by assigning the correct number of (K) clusters. The goal is to identify the distinct pattern in the data to find the smallest possible difference between the attributes in the same classes. We propose integrating the SOM and K-means approaches into the SOM+K-means architecture, as shown in Figure 3. K-means is very commonly used in machine learning. In our study, the K-means algorithm is used to obtain the best clustering results. The key idea is to identify K centroids, one for each cluster. The basic K-means algorithm randomly selects the centroid from the application list. After that, each item is placed according to its centroid

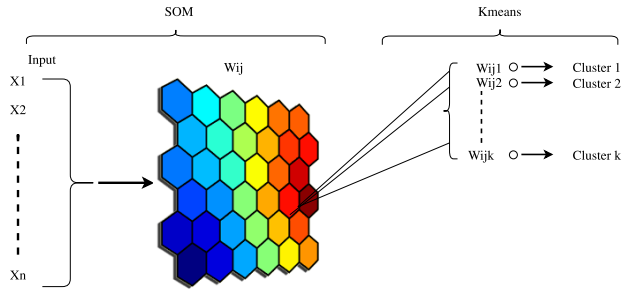


FIGURE 3. Architecture of the SOM-Kmeans model.

in a dataset. The K-means clustering partitions a dataset by reducing the total cost function of the squares.

$$J = \sum_{j=1}^{k}\sum_{i=1}^{x} \|Xi^{(j)} - Cj\|^2 \tag{4}$$

where $\|Xi^{(j)} - Ci\|^2$ is a chosen distance measure between an application $Xi^{(j)}$ and the cluster center, and Cj is a measure of the distance between applications and their cluster centroids [11]. We separate the applications into K clusters, so the application will be allocated to the one which is the smallest distance between K clusters. As a result, our SOM4+K-means builds the clusters based on improving overall average value of the silhouette index (the closer to 1, the better). Thus, we aim to increase the overall average silhouette. In order to help the SOM+K-means model succeed in its search, we tuned the K parameter in the K-means to gain a more qualitative interpretation of the acquired data. In so doing, we noted that (K = 250) led to an overall average silhouette of 99.4% and 250 clusters. Each resulting cluster was saved as an CSV file, including identified permission usage patterns, apps, and their info from the main dataset.

C. CLUSTERS ANALYSIS

This process generates clusters of permissions that are constantly used in conjunction with one another, as well as several noisy points that are omitted. We extract the use vectors of each generated cluster using logical disjunction in a single use vector. Each produced cluster's vector contains the name of the cluster, some statistical info, the permission usage pattern, and the number of apps per category. Algorithm 2 briefly explains the process of the produced results that were saved on one CSV file. This file will be exploited as a starting point to obtain the rest of the findings.

V. EMPIRICAL STUDY

We describe the findings from our study of the proposed methodology of SOM+K-means in this section. Our aim is to determine whether SOM+K-means can recognise usage patterns of applications that are 1) coherent enough to provide useful information for the relevant apps, and 2) generalizable for permission usage patterns. To do so,

Algorithm 2: Clusters Analysis

Input : $Cluster_i$, $i = 1, 2, \cdots, 250$, all Clusters.
Output: CSV_{file}.

for $app \in Category_{apps}$ **do**
  $cat_i = group\_category(app)$
  $Cluster_i =$ $get\_All\_info_{(Cluster_i)} \oplus pattern\_permissions_{Cluster_i}$
  $CSV_{(file)} \leftarrow$ $Append(CSV_{(file)}, Concat(Vector_{Cluster_{(i)}})$
  return $(CSV_{(file)})$
end for

we investigate the correlation between the resulting clusters and the permission usage patterns. We also investigate the permission patterns deployed to calculate the potential malware vector to train Support Vector Machine (SVM) and validate the enhancement of malicious application detection. For each experiment in this area, we present the study issues, the method used to address them, and the resulting findings.

A. ANALYSIS OF COHESION

As an initial experiment, we assessed the cohesion of the cluster's quality identified by SOM+K-means for various matrices, including the silhouette index metric, the Pattern Usage Cohesion (PUC) metric, and the Category Cohesion (CC) metric. We intend to answer the following research question:

RQ1. What is the quality of each resulting pattern and the correlation between its apps?

1) ANALYTICAL TECHNIQUE

Firstly, the similarity between an object and its own cluster has to be measured. Thus, we utilise a cohesiveness metric, namely the silhouette index metric. The silhouette value ranges between [-1,1], with a high value indicating that the object has a high affinity for its own cluster but a low affinity for neighbouring clusters. The silhouette index is calculated as follows:

For data point $i \in Ci$ (data point i in the cluster Ci), where $d(i, j)$ is the distance between data points i and j in the cluster Ci. We are able to interpret $a(i)$ as an indicator of how successfully i is assigned to its cluster (The better the assignment, the lower the value).

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j) \tag{5}$$

We then define the mean dissimilarity $b(i)$ of point i to some cluster C_k as the mean of the distance between i to all points in C_k (where $C_k \neq C_i$). For each data point $i \in C_i$.

$$b(i) = min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i, j) \tag{6}$$

We now define a silhouette (value) of one data point i

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}, \quad if\,|C_i| > 1 \tag{7}$$

Thus, the $s(i)$ over all the data in the entire dataset provides a measure for the data's clustering accuracy.

Next, we must determine whether the identified patterns are sufficiently coherent to reveal informative co-usage links between individual apps. As a result, we use a metric for cohesiveness called Pattern Usage Cohesion (PUC), to quantify the cohesion of the detected patterns. PUC was originally utilised for the cohesive utilisation that was inspired by Perepletchikov *et al.* [30]. It assesses the uniformity of co-use of an ensemble of entities, which in our context corresponds to a number of applications in the form of a used permission model. The range of PUC values is [0,1]. The greater the PUC number, the stronger the usage cohesion, i.e., a usage pattern shows optimal usage cohesion (PUC=1) if all permission patterns are always utilised together. If p is a pattern of permission usage, then its PUC is defined as follows:

$$PUC_p = \frac{\sum_{pp} ratio\_used\_apps(p, pp)}{|perm(p)|} \in [0, 1] \tag{8}$$

where pp denotes a permission that contains the pattern p, and the ratio_used_apps(p, pp) means the ratio of permissions that include the pattern. p and are used by each app. The perm(p) defines the set of all permissions that are used in the pattern p.

The last metric, Category Cohesion (CC), measures the ratio of apps that belong to the same category in each cluster. The CC confidence interval is [0,1]. The higher the CC number, the stronger the CC for each category $Cat(i)$ in the same cluster. Thus:

$$CC_{C_i} = max_{cat_i} \frac{|Apps(Cat_i, C_i)|}{|Apps(C_i)|} \tag{9}$$

where the ratio of used $Apps(Cat_i, C_i)$ denotes the number of app clusters (C_i) that belong to the same category, and $Apps(C_i)$ denotes the total apps in the cluster (C_i).

The analysis results of the three quality matrices are presented in the following subsection.

2) RESULTS FOR RQ1

The cohesion of the three quality matrices is calculated based on the results of overall average silhouette. Table 6 reports the silhouette cohesion matrix measuring the quality cohesion for each cluster. From the table, we observe that an average silhouette score is (98.1%) and standard deviation value is (0.1). These values are realistic, because our clustering is based on the silhouette index, which is already high. Further, these values reflect the co-usage relationships of the apps' patterns, making them more cohesive.

The PUC outcomes also provide evidence that SOM+K-means exhibits consistent cohesion with regard to the identified usage patterns. We found that at least 50% of the applications are used together with high PUC.

TABLE 6. SOM-Kmeans average cohesiveness and summary of inferred usage patterns.

	SIL	PUC	CC
Avg	0.981	0.6	0.4
StdDev	0.1	0.2	0.2
Nb Pattern	250		

A noteworthy number of the apps have 100% PUC. For example, an average PUC would be 60% and a standard deviation value (0.2). As well, the category cohesion matrix carried out the qualitative aspect of the obtained results. From it, we observe that an average 40% of the clusters contain apps that belong to the same categories. Indeed, it is worth mentioning that we observed a trade-off between usage cohesion of detected patterns and their distributed categories of apps.

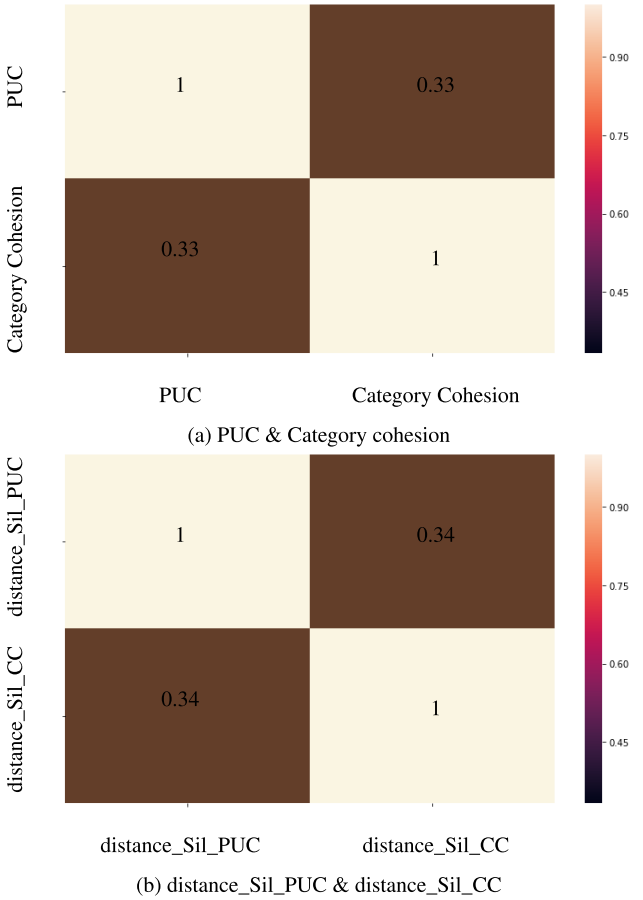
Next, to acquire a better understanding of the correlation of the findings between cohesion matrices with respect to the silhouette matrix, we calculated the distance between the silhouette matrix and each PUC and CC matrix. This resulted in two new matrices: (distance_Sil_PUC) and (distance_Sil_CC).

Figure 4 shows the correlation between the cohesion matrices on each axis. As can be seen, the correlation ranges from -1 to $+1$. Values closer to zero mean that the two cohesion matrices show no linear trend. The closer to 1 the correlation is, the stronger their correlation; in other words, as one increases, so does the other. Thus, the closer to 1, the stronger the relationship is. A correlation closer to -1 indicates similarity, However, rather than both rising, one variable will drop as the other increases. The diagonals are all 1 (light), since the squares relate each variable to themselves. Our motivation here is to study the correlation between the cohesion matrices in order to see the relation between them and possibly to discard some of them. Based on this motivation, we investigated the correlation between the PUC and CC matrices, and that between (distance_Sil_PUC) and (distance_Sil_CC). Figure 4a provides the correlation between the PUC and CC matrices, while Figure 4b shows the correlation between (distance_Sil_PUC) and (distance_Sil_CC). It worth noting that both correlations yield very close results. As well, Figure 5 shows the correlation between the clusters and cohesion matrices. We observe that the correlation result is not sufficiently close to be useless and not far enough away to be independent. Hence, it is important to consider all cohesion matrices. The presence of correlation implies the absence of a linear relationship that demonstrates the quality. From this, we can assume that cohesion matrices assess inferred patterns from various perspectives.

B. PRODUCED INFERRED PATTERN

The purpose of this study is to determine the reliability of the permission usage patterns detected using SOM+K-means. We seek to answer the following research question:

RQ2. How far does the concept of cohesive matrices go in obtaining representative permission usage patterns?



(a) PUC & Category cohesion

(b) distance_Sil_PUC & distance_Sil_CC

FIGURE 4. The correlation between the quality matrices.

1) ANALYTICAL TECHNIQUE

To address our second research question (RQ2), we study whether the patterns are representative of the permission usage by applying two selective thresholds to maintain a high level of usage cohesion.

First Selective Threshold: Based on PUC matrix, we calculate the median for the inferred patterns in our case ($Median = 0.42$). Our motivation for applying this threshold concept is as follows: We believe that the median for the PUC matrix is far enough away to include a valuable pattern. In other words, the median covers the patterns that have sufficient quality and generate a sufficient number of patterns. Thus, the median is considered the threshold, and the second cut follows this criterion. This step resulted in representative permission usage patterns, such that the representative permission usage pattern $\geq Median$.

Second Selective Threshold: To perform this step for each cluster, we only consider apps that belong to the same category and have the highest value. Thus, based on the app categories ($Apps_{Cat_i}$) and Category Cohesion (CC) matrix, we calculate the number of ($Apps_{Cat_i}$) in each cluster (C_i).. Then we calculate the average for the ($Apps_{Cat_i}$) matrix. Our motivation is to apply the average as the threshold. In so doing, we observe that the average is not particularly high when compared with total apps per cluster. This observation

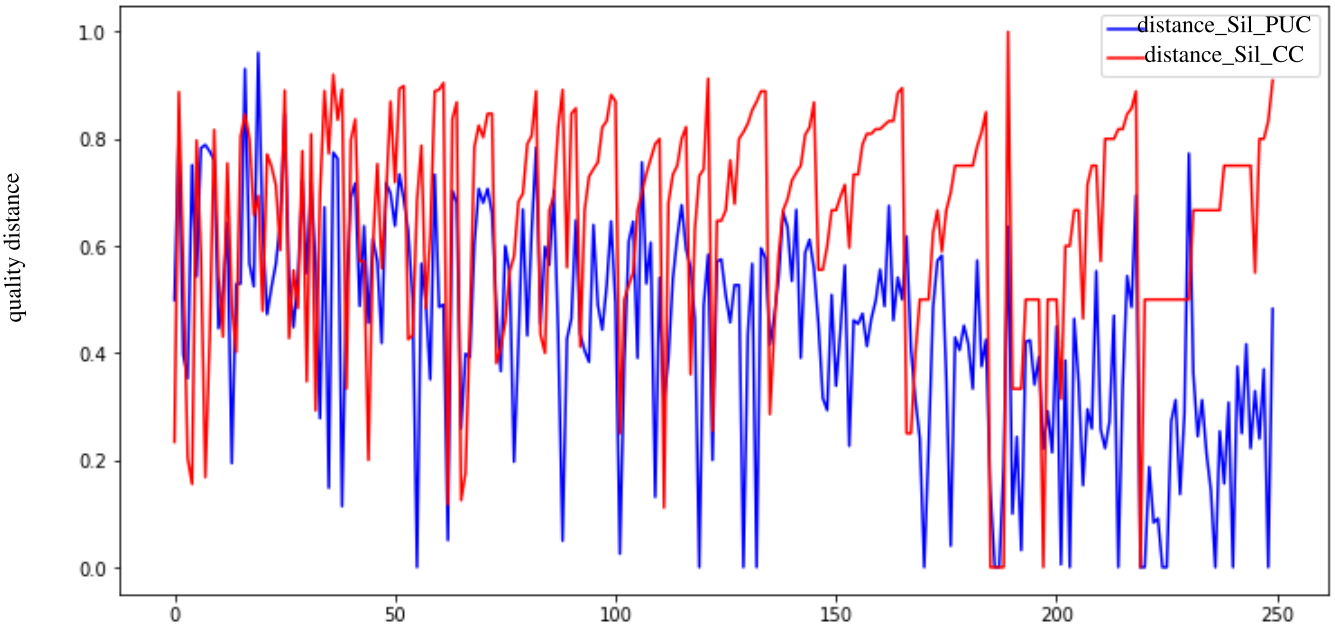


FIGURE 5. Overview of the quality matrices cohesion.

leads us to remove some clusters, even though this may badly impact our study.

As well, the average is not sufficiently small to be not representative enough; so, based on this motivation, the average was chosen as the first threshold. According to this criterion, the first cut was applied, leaving 58 clusters remaining. After this, each cluster was assigned to the more representative app category. In other words, we selected the category with the highest percentage of apps to represent the cluster's pattern, as follows. Category pattern $= Max(Apps_{Cat_i}) \in (C_i)$. In this step, we are logically motivated.

2) RESULTS FOR RQ2

The obtained results are as follows. The analysis study provides 30 representative permission usage patterns, including 12 different categories. Some of the categories have more than one pattern. This step resulted in a dataset of inferred patterns. Figure 6 shows the statistical distribution for the cohesion matrices and provides additional information about selective criteria.

C. PATTERN GENERALISATION EVALUATION

In this study, our objective is to evaluate whether the representative permission usage patterns identified with SOM+K-means can be generalizable in terms of being able to identify malicious and benign apps, which would then validate our work. Our goal is to address the following research question.

RQ3. To which extent are the discovered permission usage patterns consistent enough to increase the ability to distinguish between malware and benign apps?

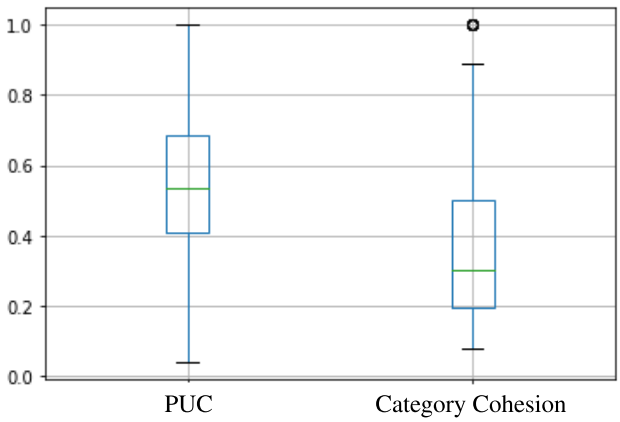


FIGURE 6. PUC & Category cohesion results of the identified permissions usage patterns.

1) ANALYTICAL TECHNIQUE

To answer RQ3, we look at whether the discovered patterns will be sufficiently consistent to aid in the differentiation of malicious and benign applications, and thus evaluate their generalizability. We address RQ3 through the following experiment: The inferred patterns are used as references to calculate the distance between each pattern's category P_{cat_i} in the inferred pattern dataset with patterns for the same category in the main dataset $P_{main_{cat_i}}$. We called this new set potential malware (PM). Hence, $PM_i = min_i|P_{cat_i} - P_{main_{cat_i}}|$. Our motivation here is to validate representative permission usage patterns and provide evidence of their quality.

Algorithm 3 explains the procedure to calculate potential malware (PM). As input 1, $Patterns_i$ is the inferred pattern category, and Apps refer to all apps in our dataset. After the

variables are initialized in Line 3, we filter the apps based on their categories. Then the app permissions were compared with inferred patterns, as shown in Line 9. We count the differences and store it in pm_i. If the cat_i has many inferred patterns, we select min pm_i., which means that the pattern has more similarities than the others. The chosen result is then stored in PM, as shown in Line 18. Next, Line 19 is reinitialized to the variables, after which we repeat all the procedures for all the apps. In the end, each app will be mapped with integer values in PM, as follows:

If the value in PM equals zero, the app's pattern is equal to one of the inferred patterns with respect to its category. Otherwise, we count the differences between the app's pattern and category's inferred patterns. The value with the smallest difference is then assigned to the category.

$$PM_{app} \rightarrow \begin{cases} 0, & ifIdenticalPattern \\ min, & otherwise \end{cases}$$

Algorithm 3: A Potential Malware(PM)

Input : $Patterns_i$, $i = 1, 2, \cdots, n$, where set of Patterns.
$Apps = [app, app_2, \ldots, app_m]$

1 p_i is a permission.
Output: PM list.
2 $PM = [\,]$
3 $count = 0, min = \infty, pm_i = 0$
4 **for** $app \in Apps$ **do**
5 $cat_i = get\_category(app)$ ▷getting all apps for same category
6 $pattern\_cat_i = get\_category(cat_i)$ ▷getting the pattern for each category
7 **for** $p \in pattern\_cat_i$ **do**
8 **for** $perm \in app.permissions$ **do**
9 **if** $perm \notin p.permissions$ **then**
10 $count = count + 1$
11 **end if**
12 **end for**
13 **if** $min < count$ **then**
14 $min = count$
15 $pm_i = min$
16 **end if**
17 **end for**
18 $PM.append(pm_i)$
19 $count = 0, min = \infty, pm_i = 0$ reinitialize the variables
20 **end for**

Study 1: We labelled our dataset based on vt_detection features as benign and malware by applying the Derbin [4] standard, which considers apps with 0& 1 flag as benign and apps with ≥ 2 as malware, as shown below.

$$App \rightarrow \begin{cases} Benign, & ifvt\_detection < 2 \\ Malware, & ifvt\_detection \geq 2 \end{cases}$$

Next, the machine learning classifier Support Vector Machine (SVM) was selected, as it has been successfully used in many research-related works. Therefore, in this study, the SVM method will be used to classify and distinguish between benign and malware apps. Also, we aim to validate our inferred patterns in this study. The SVM model is applied as follows:

1) SVM model were fed with the permissions as features.
2) The cross validation is applied 80% in the training phase and 20% in the testing phase.
3) The hyper parameters C& Gamma are tuned in the training phase to fit our data.
4) The model is tested using 20% cross-validation.

Study 2: In this study, we add MP to the dataset as a feature and deploy the same hyper parameters C & Gamma from **Study 1**. Thus, we apply the same model with respect to C& Gamma hyper parameters in order to observe the results under the same conditions.

To assess our model, we used three performance parameters: Accuracy, F1, and AUC. These parameters are frequently used in machine learning to evaluate performance models.

To assess our model, we used three performance parameters: Accuracy, F1, and AUC. These parameters are well-known in machine learning to evaluate the performance models.

1) This denotes the percentage of correctly classified apps: $(TP + TN)/(TP + TN + FP + FN)$ [7].
2) F1-Measure: This indicates a performance indicator that takes into account both the precision and recall of the obtained classification: $2 * (Recall * Precision)/(Recall + Precision)$ [7].
3) Area under ROC Curve (AUC): This is a measure of the predictive power of the classifier that basically informs us how much the model is capable of distinguishing between classes (benign apps vs malware).

2) RESULTS FOR RQ3

In Study 2, the training phase results were significantly higher than those in the testing phase, causing overfitting in the model. Thus, we solve the overfitting using the random oversampling technique. Random oversampling is the simplest strategy for balancing a dataset's imbalanced nature. It balances out the data by duplicating minority class samples. The overfitting was solved and the performance in the training phase was almost the same as that in the testing phase.

The obtained results are as follows.

Table 7 summarizes the results of the SVM classifier both without using the MP as a feature and including the MP as a feature. We observe that there are improvements in terms of distinguishing between malware and benign apps when we added the potential malware feature. Hence, adding the detected patterns is more informative and creates a notable change in the performance of the model. More specifically, the results from the experiment confirm the above-mentioned

TABLE 7. Comparison between two models (with and without *MP*).

Gamma = 1		C = 100	
Performance Matrix	Accuracy	F1	AUC
Without PM	93.5	92.9	92.5
With PM	94.1	93.2	94.1

findings. We believe that our approach can be achieved and will succeed at improving Android security for developers and users. The adaptation of our variant SOM+K-means method is one of the most important contributions of this work for mining permission usage patterns.

VI. RELATED WORK

A. RESEARCH RELATED TO DATASET GENERATION

Numerous repositories have been proposed over the years for the study of mobile apps. Recently, the AndroZoo[7] dataset was released, which includes over 13 million Android apps from Google Play, other stores, and app repositories. The aim of AndroZoo is to build robust app collections for software engineering research. F-Droid2 is a repository of free open-source Android apps that have been used in an impressive number of studies. Even more recently, Geiger *et al.* [14] made available a graph-based database with information (e.g., metadata and commit/code history) on 8,431 open-source Android apps located on GitHub and the Google Play Store. Also notable, although slightly older, is Krutz *et al.*'s study [22]], with a public dataset centered on the lifecycle of 1,179 Android apps from F-Droid. Arp *et al.* [4] established the well-known DREBIN dataset, which is comprised of 131,611 applications of benign and malicious software. Samples were obtained in the August 2010 to October 2012 time-frame. To find out whether an application is malicious or benign, each sample was sent to the VirusTotal service to examine the output of ten common antivirus scanners (AntiVir, AVG, BitDefender, ClamAV, ESET, FSecure, Kaspersky, McAfee, Panda, and Sophos). Any application that was scanned by at least two scanners was detected as malicious.

Li *et al.* [24] built a dataset of 1,497 apps pairs, where one application piggybacks another that may contain malicious payloads. Their work was based on AndroZoo. Using VirusTotal's results, they flagged the relevant malware apps. F. Wei *et al.* [34] prepared a dataset containing 24,650 samples dating from 2010 to 2016 that labeled Android malware. The samples were collected from several sources, including Google Play, VirusShare, and security companies of third-parties. VirusTotal was used to flag their apps. Zhou and Jiang [40] managed to collect around 1,200 malware samples in August 2010 and manually analyzed the malware samples. Wang H *et al.*built a dataset of 9,133 malware samples and set a threshold of 20 to indicate suspicious applications based on the number of VirusTotal[8] engines recorded.

[7]https://androzoo.uni.lu/

[8]https://www.virustotal.com/gui/

1) SIZE AND COVERAGE

Apart from AMD dataset [34], the great majority of datasets currently available are limited and obsolete. For example, MalGenome [40] and Drebin [4] are two of the most popular datasets. Their production was done five years ago, and only a limited number of samples are included. The literature also reports that the Drebin dataset has a replication issue [18]. The AMD dataset, which contains a large number of malware samples, was developed in 2016. It includes several samples that overlap with the MalGenome and Drebin projects, since it gathered samples from a broad range of sources, including previously collected malware datasets.

2) METHODS USED TO FLAG THE GROUND TRUTH

The rest of the three datasets heavily depend on VirusTotal for accuracy in labelling the ground truth. It is worth noting that various thresholds are utilized on VirusTotal to label malware samples. For example, Drebin was developed based on the findings of ten well-known engines on VirusTotal. At least two of the ten engines found one type of malicious activity in the original sample and flagged it as a malware. As a threshold, one engine was employed in the Piggybacking dataset, while AMD made use of 28 different engines (which, at that time, represented over 50% of the engines). Furthermore, despite the fact that VirusTotal is commonly used in academia and industry, it contains very little exclusivity.

3) APP METADATA

After looking into the issue, we assert that, to the best of our knowledge, no other studies have focused on metadata (e.g., app description, app ratings, etc.) relevant to malware in their samples. Furthermore, because previous works [15], [27] have suggested incorporating app metadata for malicious/anomaly detection, we believe it is critical to build a malware dataset containing all of the app metadata to enable malware detection evaluation.

B. PERMISSIONS BASED STUDY

The permission system has attracted considerable research interest. Several studies have been conducted recently to investigate how permissions are used in Android apps and whether or not they can help identify malware apps. In [10], Felt *et al.*conducted a survey of 100 paid apps and 856 free apps from the Android Market. They identified the most requested permissions and observed that both free and paid apps make requests for at least one dangerous permission. Additionally, they created a tool that is able to detect whether an app requests more permissions than necessary, noting that one-third of the examined applications were over-privileged. In [5], Barrera *et al.* conducted a survey of the 1,100 most popular applications downloaded in 2009. They discovered that only a small portion of the specified permissions are actively used by developers. In [35], Wei *et al.* investigated the evolution of permissions in the

Android ecosystem, finding that dangerous permissions often outnumber other permission types in all Android. Meanwhile, in [23], Krutz *et al.* also carried out a study on app permissions. They discovered that more experienced developers are more likely to make permission-based modifications, and that permissions are usually introduced earlier in an app's lifetime.

In [12], the authors selected 188,389 applications from the official Android market and studied the different requested permission combinations made by them. The authors identified more than 30 common patterns of permission requests and found that low-reputation applications often diverge from the permission request pattern observed in high-reputation applications.

Other research has focused on defining risk signal as a way to identify malware applications. In [32], Sarma *et al.* proposed a set of risk signals by analyzing the permission patterns in apps taken from the Android Market within a dataset of 121 malicious apps. in [41], Zhou *et al.* developed a system for detecting malicious applications in official and alternative Android markets.

In [33], the authors performed an empirical research of 574 open-source Android app GitHub repositories. They examined the incidence of four distinct sorts of permission-related concerns throughout the duration of the apps' lifetimes. Their findings indicate that permission-related difficulties are a common occurrence in Android applications. In [2], authors have conducted for the last five years' versions of the top Android apps to examine the Android platform's permissions mechanism. Additionally, the paper addresses Android's user-permissions model, which defines how applications manage sensitive data and resources. In [37], the authors introduced MPDroid, It is a new technique that combines static analysis and collaborative filtering to determine the minimum permissions required for an Android application based on its description and API usage. MPDroid begins by utilising collaborative filtering to determine the app's basic minimal permissions. Then, using static analysis, the final minimal permissions required by an app are determined. Finally, it assesses the danger of over privilege by analysing the app's excess privileges, i.e., the rights sought by the programme that are not essential. Experiments are run on 16,343 popular Google Play applications. In [36], the authors manually annotated 2,254 app descriptions from the Google Play Store to include 26 permissions classified into ten categories. They used two natural language processing approaches to enhance our annotated dataset in order to acquire additional permission semantics. In [3], the authors proposed a multi-criteria decision-making-based (MCDM) mobile malware detection system that evaluated Android mobile applications using a risk-based fuzzy analytical hierarchy process (AHP) method. The study focuses on static analysis, which employs permission-based features to evaluate the approach used by mobile malware detection systems. Risk analysis is used to raise the mobile user's awareness when accepting any permission request that carries a high risk level. 10,000 samples were collected from Drebin and AndroZoo for the assessment. The findings indicate a high rate of accuracy of 90.54%. In [19], the authors devised a method for identifying Android harmful applications called fine-grained dangerous permission (FDP), which collects characteristics that more accurately describe the difference between malicious and benign applications. Among these features, for the first time, a fine-grained feature for harmful permissions issued to components is offered. We examine 1700 benign and 1600 malicious apps and show that FDP has a 94.5% TP rate.

Our approach is similar to [2], [33], [36], [37] in terms of permission-related concerns, we dissimilar in terms of the dataset (including the size, features, and the number of permissions), using machine learning, and considering the categories'apps in their studies. In our present work, we expand on the existing research. We also investigate similar properties and propose new ones, which we define as application sustainability and malware risk.

C. CATEGORY BASED STUDY

Apps in Android app stores are classified into various categories, such as Health&Fitness, News&Magazine, Books&References, Music&Audio, etc. Each category has its own set of functionalities, which means that applications in the same category have similar functionalities. Permissions are one of these features. Several state-of-the-art studies make a link between the apps' requested permissions and the features that are standard in its category. Some researchers proposed using category-based machine learning classifiers to improve the efficiency of classification models in identifying malicious applications within a certain category.

In [16], as a feature, the authors used the category of applications named by Google Play. Their results reveal that by using machine learning technology to detect malicious malware, they used the applications' permissions at app-level. Further, they found that adding the application category feature improves detection efficiency and accuracy. In [26], the target consists of both static and dynamic analyses. The static analysis is focused on source code, user permissions and signatures, while the dynamic analysis is based on the behavior of applications in running time. A machine learning algorithm known as OKNN is then used to determine which category an application belongs to. The size of the dataset in that study is 3,600 apps. In [39] Yuan *et al.* presented an automated method for categorising Android apps. They conducted experiments with 13,005 applications composed of 18 categories with Naive Bayes. More specifically, in their approach, the malware application publisher can choose an application category at random in order to avoid detection by the application market. As a consequence, a method that can automatically categorize multiple types of apps can be useful for organizing the Android Market as well as identifying malicious applications. Studies show that the addition of an application category will greatly increase the efficiency and accuracy of the detection when using machine leaning

TABLE 8. Comparison between various state-of-art solutions.

Reference	Generated Dataset	Category-based	Identify permissions usage pattern	Machine Learning	Malware detection	Dataset size
[33]	✓	✗	✗	✗	✗	574
[2]	✓	✗	✗	✗	✗	20 apps with their versions (2016 to 2020)
[37]	✗	✗	✗	✗	✗	16,343
[36]	✗	10 categories	✗	✓	✗	2,254
[39]	✗	18 categories	✗	✓	✓	13,005
[3]	✗	✗	✗	✓	✓	10,000
[19]	✓	✗	✗	✓	✓	3100
Our work	✓	46 categories	✓	✓	✓	16,000

technology to detect malicious apps [16]. Thus, application category is important for Android malware detection. Several works involved category-based investigations, but for different purposes. The one most related to our work was conducted by Sarma *et al.* [32]. Thus, application category is important for Android malware detection.

Several works involved category-based investigations, but for different purposes. The one most related to our work was conducted by Sarma *et al.* [32]. Their approach is most similar to ours, in that it is also focused on permission use through categories. However, it has a different purpose, with Sarma *et al.* [32] focusing on the similarities between app permission usage and their categories to distinguish between malware and benign entities. We, on the other hand, are more concerned with the overall app permission usage and in finding requested permission patterns among different categories. Moreover, our work takes into account a different level of granularity than previous works whose approaches infer malware app usage permissions at the category level.

Nonetheless, to improve Android security, Sarma *et al.* [32] investigated the feasibility of using the permissions that an app requires, the category of the app, and the permissions that other apps of the same category require. They created their 158,062-app dataset in February 2011. The malware dataset consists of 121 apps obtained from the Contagio Malware Dump. Some related work used category as a feature [16] in their training model to improve performance, whereas in our case, we are more interested in exploring possible use permissions patterns across whole categories of applications. Previous approaches assumed that the necessary permissions were selected by the developer in advance and that he/she chooses an application category at random in order to avoid detection by the application market [39]. Without using this assumption, our study will meaningfully supplement other research. Indeed, our approach may be used as a preliminary step to infer sets of permissions that are consistently used together, such that existing approaches could be used to learn how to improve the ability to distinguish between benign and malware within the patterns' permissions and category apps. Our novel findings focus on producing usage patterns of permissions for various categories and on providing in-depth analysis of pattern cohesion and the impact of patterns on malware detection. Table 8 shows the comparison between various state-of-art solutions that study the Android permissions system in different purposes.

VII. CONCLUSION

With the exponential growth in the number of smartphones being used in services such as banks, hospitals, and m-commerce, smartphone security has become a major concern. The use of unofficial sources to upload applications is likewise concerning. Malicious apps can be used to steal passwords, leak information, and build windows into phones. Existing anti-virus software relies on static signatures that must be modified on a regular basis and are incapable of detecting zero-day malware. The Android permission scheme is the core Android security framework that governs application task execution. Despite recent advancements in research that have provided a variety of approaches and detection methods for locating malware applications, the available literature lacks a comprehensive examination of the topic. We addressed this deficiency in this work by investigating all the larger issues, resulting in two main achievements. 1) We created a huge dataset of malware and benign apps in a systematic and automated manner and made it accessible to the community. 2) We conducted a preliminary analytical analysis of various forms of Android permissions and their potential associations with malicious intents, as well as users' impressions of the nature of the applications that use them.

Our research examined 118 separate features, 103 of which are permissions, on approximately 16K apps. Further, we proposed tentative findings on the ties between the use of Android permissions tagged as unsafe by the permission scheme. Additionally, we introduced a model that combines a self-organizing map (SOM) and K-means clustering. Based on a clustering validity test, we built the resultant SOM+K-means using permissions as features. Our overall achieved purpose was to describe pictures or patterns of how applications in a particular category behave by optimizing our model.

REFERENCES

[1] G. Abaei, Z. Rezaei, and A. Selamat, "Fault prediction by utilizing self-organizing map and threshold," in *Proc. IEEE Int. Conf. Control Syst., Comput. Eng.*, Nov. 2013, pp. 465–470.

[2] I. M. Almomani and A. A. Khayer, "A comprehensive analysis of the Android permissions system," *IEEE Access*, vol. 8, pp. 216671–216688, 2020.
[3] J. M. Arif, M. F. Ab Razak, S. R. T. Mat, S. Awang, N. S. N. Ismail, and A. Firdaus, "Android mobile malware detection using fuzzy AHP," *J. Inf. Secur. Appl.*, vol. 61, Sep. 2021, Art. no. 102929.
[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.
[5] D. Barrera, H. G. Ü. Ä. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 73–84.
[6] Y. Bengio, J. Buhmann, Y. Abu-Mostafa, M. Embrechts, and J. Zurada, "Special issue on neural networks for data mining and knowledge discovery," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 545–822, Oct. 2000.
[7] D. Chicco and G. Jurman, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, pp. 1–13, 2020.
[8] (Mar. 2021). *Permissions on Android*. [Online]. Available: https://developer.android.com/
[9] (Mar. 2021). *Replication Package*. [Online]. Available: http://www-etud.iro.umontreal.ca/~saiedmoh/MobileSoftRP/index.html
[10] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proc. 2nd USENIX Conf. Web Appl. Develop.*, 2011, p. 7.
[11] R. Ferdous, "An efficient K-means algorithm integrated with Jaccard distance measure for document clustering," in *Proc. 1st Asian Himalayas Int. Conf. Internet*, 2009, pp. 1–6.
[12] M. Frank, B. Dong, A. Porter Felt, and D. Song, "Mining permission request patterns from Android and Facebook applications," in *Proc. IEEE 12th Int. Conf. Data Mining*, Dec. 2012, pp. 870–875.
[13] (Mar. 2021). *Mobile Users Will Provide Personalized Data Streams*. [Online]. Available: http://www.gartner.com/newsroom/id/2654115
[14] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world Android apps," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 30–33.
[15] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 1025–1035.
[16] V. Grampurohit, V. Kumar, S. Rawat, and S. Rawat, "Category based malware detection for android," in *Proc. Int. Symp. Secur. Comput. Commun.* Springer, 2014, pp. 239–249.
[17] (Mar. 2021). *Smartphone OS Market Share*. [Online]. Available: http://www.idc.com/promo/smartphone-market-share/os
[18] P. Irolla and A. Dey, "The duplication issue within the drebin dataset," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 3, pp. 245–249, Aug. 2018.
[19] X. Jiang, B. Mao, J. Guan, and X. Huang, "Android malware detection using fine-grained features," *Sci. Program.*, vol. 2020, pp. 1–13, Jan. 2020.
[20] K. Kirchner, B. Delibašić, and M. Vukićević, "Designing clustering process with reusable components," *Info M*, vol. 9, no. 34, pp. 23–29, 2010.
[21] T. Kohonen, *Self-Organizing Maps* (Information Sciences), vol. 30. Springer. 2001.
[22] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, "A dataset of open-source Android applications," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 522–525.
[23] D. E. Krutz, N. Munaiah, A. Peruma, and M. Wiem Mkaouer, "Who added that permission to my app? An analysis of developer permission changes in open source Android apps," in *Proc. IEEE/ACM 4th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 2017, pp. 165–169.
[24] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. L. Traonm, and D. Lo, "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.
[25] X.-T. Li, S. Ren, W. Cheng, L.-S. Xiang, and X.-Y. Liu, "SmartPhone: Security and privacy protection," in *Proc. Joint Int. Conf. Pervasive Comput. Networked World*. Springer, 2013, pp. 289–302.
[26] J. Lin, X. Zhao, and H. Li, "Target: Category-based Android malware detection revisited," in *Proc. Australas. Comput. Sci. Week Multiconf.*, 2017, pp. 1–9.
[27] S. Ma, S. Wang, D. Lo, R. H. Deng, and C. Sun, "Active semi-supervised approach for checking app behavior against its description," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 179–184.
[28] Z. Namrud, S. Kpodjedo, and C. Talhi, "AndroVul: A repository for Android security vulnerabilities," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2019, pp. 64–71.
[29] S. O'Dea. (Feb. 2021). *Smartphone Unit Shipments Worldwide by Operating System From 2016 to 2023*. [Online]. Available: https://www.statista.com/statistics/
[30] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Proc. 7th Int. Conf. Quality Softw. (QSIC)*, 2007, pp. 328–335.
[31] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, no. 1, pp. 53–65, 1987.
[32] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proc. 17th ACM Symp. Access Control Models Technol.*, 2012, pp. 13–22.
[33] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission issues in open-source Android apps: An exploratory study," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2019, pp. 238–249.
[34] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Springer, 2017, pp. 252–276.
[35] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the Android ecosystem," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 31–40.
[36] Z. Wu, X. Chen, M. U. Khan, and S. U.-J. Lee, "Enhancing fidelity of description in Android apps with category-based common permissions," *IEEE Access*, vol. 9, pp. 105493–105505, 2021.
[37] J. Xiao, S. Chen, Q. He, Z. Feng, and X. Xue, "An Android application risk evaluation framework based on minimum permission set identification," *J. Syst. Softw.*, vol. 163, May 2020, Art. no. 110533.
[38] S. Yu, M. Yang, L. Wei, J.-S. Hu, H.-W. Tseng, and T.-H. Meen, "Combination of self-organizing map and k-means methods of clustering for online games marketing," *Sensors Mater.*, vol. 32, no. 8, pp. 2697–2707, 2020.
[39] C. Yuan, S. Wei, Y. Wang, Y. You, and S. G. ZiLiang, "Android applications categorization using Bayesian classification," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery (CyberC)*, 2016, pp. 173–176.
[40] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.
[41] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS*, Feb. 2012, vol. 25, no. 4, pp. 50–52.