PermPair: Android Malware Detection Using Permission Pairs

Anshul Arora, Sateesh K. Peddoju, *Senior Member, IEEE*, and Mauro Conti, *Senior Member, IEEE*

***Abstract*—The Android smartphones are highly prone to spreading the malware due to intrinsic feebleness that permits an application to access the internal resources when the user grants the permissions knowingly or unknowingly. Hence, the researchers have focused on identifying the conspicuous permissions that lead to malware detection. Most of these permissions, common to malware and normal applications present themselves in different patterns and contribute to attacks. Therefore, it is essential to find the significant combinations of the permissions that can be dangerous. Hence, this paper aims to identify the pairs of permissions that can be dangerous. To the best of our knowledge, none of the existing works have used the permission pairs to detect malware. In this paper, we proposed an innovative detection model, named *PermPair*, that constructs and compares the *graphs* for malware and normal samples by extracting the permission pairs from the *manifest* file of an application. The evaluation results indicate that the proposed scheme is successful in detecting malicious samples with an accuracy of 95.44% when compared to other similar approaches and favorite mobile anti-malware apps. Further, we also proposed an efficient edge elimination algorithm that removed 7% of the unnecessary edges from the malware graph and 41% from the normal graph. This lead to minimum space utility and also 28% decrease in the detection time.**

***Index Terms*—Android malware, android security, malware detection, permissions pair graph, smartphone security.**

I. Introduction

SMARTPHONES have gained popularity with the presence of feature-rich apps which provide services like social networking, online banking, online gaming, and location-based services, in addition to the conventional services like phone calls and messaging. A report [1] shows that there is tremendous growth in smartphone sales, with 82% Android smartphone users. Rising popularity has made them susceptible to malware attacks. The year 2013 recorded 1,45,000 new malware samples, with 97% of them targeted towards Android platform [2]. This indicates that Android is evidently the major target, with nearly 5000-6000 malware samples attacking them every 14 seconds, and the figure touched 3.5 million malicious samples in 2017 and is expected to rise to 25 million by 2019 [3]. This trend shows how malware attackers are relentlessly developing new malware samples targeting smartphones, especially Android. Apart from the traditional drive-by-downloads way of infecting the system, malware is also injected into smartphones through repackaging and update attacks. The threats posed by mobile malware include financial loss to users, information leakage, system damage, and mobile bots [4].

The increase in the number of Android malware attacks is mainly from three major sources: (a) App markets, an easy distribution gateway for malware developers; (b) Users, drive-by-downloads, and (c) Developers, weak code.

The design of the Android platform secures the system by restricting the access to local resources by the applications (apps) using the permission constraints. A user is prompted with the list of permissions during the installation of an application. This list is supposed to alert the users about the resources that the application accesses. Most of the users ignore them and grant the permissions liberally. They do not have adequate expertise to understand the significance of these permissions, and the harm caused by them if any [10]. This weakness of the users drew the attention of the attackers. Consequently, the researchers aim to analyze the permissions for detecting malicious behavior.

Several related works such as [5], [8], [14]–[16], [22], [23], [25] have used permissions to detect Android malware. They have analyzed the permissions in malicious apps, detected during the period, 2010-2012. These studies have examined the permissions of normal and malicious samples, and have reported that most of the top permissions in malware and normal apps are quite similar, and hence not distinguishable. For instance, the authors in [14] reported that the top five permissions in both the categories are exactly the same. Therefore, it is important to find the vital combinations of the permissions present in malware and normal samples. None of the previous works, except the one proposed in [22], have aimed at finding the permission patterns that can launch any malicious activity. They have also identified risky permission patterns in malware samples. However, they did not find the permission patterns that occur prominently in normal apps. Moreover, they did not propose any detection model.

Manuscript received August 9, 2018; revised December 11, 2018, April 3, 2019, May 17, 2019, July 14, 2019, and August 26, 2019; accepted October 11, 2019. Date of publication October 29, 2019; date of current version January 30, 2020. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Wojciech Mazurczyk. *(Corresponding author: Anshul Arora.)*

A. Arora is with the Discipline of Mathematics and Computing, Delhi Technological University Delhi, Delhi 110042, India (e-mail: anshul15arora@gmail.com).

S. K. Peddoju is with the Department of Computer Science and Engineering, IIT Roorkee, Roorkee 247667, India (e-mail: sateesh@ieee.org).

M. Conti is with the Department of Mathematics, University of Padua, 35122 Padova, Italy, and also with the Department of Electrical Engineering, University of Washington, Seattle WA 98195 USA (e-mail: conti@math.unipd.it).

Digital Object Identifier 10.1109/TIFS.2019.2950134

Motivation: We believe that the pairing of dangerous permissions together can be effective in detecting malicious apps. For instance, to leak device-specific information to the server, an application requires only two permissions: INTERNET and READ_PHONE_STATE. This permission pair, alone, is dangerous and can launch a malicious activity. However, to evade the detection, malware developers may supplement some additional permissions [11]. The presence of such dangerous permission pairs can help detect malicious behavior. Therefore, this work aims to analyze permissions in a group of two and proposes a new methodology to find such pairs that can distinguish normal and malicious samples. The following research questions emerge in the light of permission pair analysis:

1) How to represent the permission pairs extracted from the applications?
2) How to build a detection model using permission pairs?
3) Is there any change in the permission pairs of malware samples over a period of time?
4) What are the top dangerous permission pairs present in malicious samples but not in benign ones?
5) What are the top permission pairs present in normal apps and how are they different from malicious samples?

We are motivated to answer these questions with a vision to develop an Android malware detector based on permission pairs. We present *PermPair*: Permission Pair Based Android Malware Detection model, based upon permissions extracted from the manifest file of the applications. We use the graph data structure to represent these permission pairs. Our detection results are relatively better than the mobile anti-malware apps which we evaluate against the same dataset of malicious apps. The work proposed in this paper employs a mix of old and recent datasets for evaluation.

Contributions: The contributions of this research are highlighted below:

- Built the permission pair graphs for different malware datasets and analyzed the impact of the permission pairs on both, old and recent, malicious apps.
- Proposed a novel algorithm to merge graphs of different malware datasets to construct a single final malware graph of permission pairs (named as Malicious-Graph (G_M)). Similarly, a separate permission pair graph of normal apps known as Normal-Graph (G_N) was also established.
- Designed an algorithm to detect malicious apps by comparing both malicious (G_M) and normal (G_N) graphs.
- Compared the detection results with that of widely used mobile anti-malware apps and other similar defense mechanisms proposed in the literature. Concluded that the proposed approach is more effective in detecting malicious apps.
- Performed edge elimination to remove insignificant permission pairs from both the graphs, to reduce the size of the graphs and the detection time.

Organization: The rest of the paper is organized as follows: Section II provides the background knowledge of Android permissions, and related work proposed in the literature for Android malware detection. The discussion of the proposed *PermPair* model is in section III. A report of results and findings of the proposed work is presented in section IV. Finally, section V concludes with the scope of future work.

II. Background and Related Work

This section, initially, presents a brief description of Android permission system followed by a critical review of the studies that have been proposed for Android malware detection. In the end, a summary of the important takeouts from the review of the literature is presented.

A. Background

Every Android application[1] consists of *AndroidManifest.xml* file having permissions and other parameters required by the application. The user receives this list of permissions for additional resources at the time of installation. Once the user grants all the permissions, the app gets installed. The Java code of the application houses, possibly, the malicious component of the malware samples. If the manifest file has the required permissions, it invokes the API calls in the code. This is the primary reason why permissions have been the most used static feature in Android malware detection.

B. Related Work

Android malware detection is broadly categorized into three types: Static, Dynamic, and Hybrid Detection. This section reviews all these detection types published in the literature in the following subsections.

1) Static Detection: Static solutions aim to analyze the app's manifest file components, Java code or the sequence of API calls within the code. These related works can further be sub-divided into six categories: Permissions Analysis, Permissions Based Malware Detection, Permission Pattern Analysis of Android Malware, Manifest File-Based Detection, Permission Graph Analysis, and API Calls Based Detection.

a) Permissions analysis: Some of the earlier works like [8], [9] analyzed permissions to detect malicious behavior within the normal apps. Grace *et al.* [8] evaluated potential risks associated with in-app advertisement libraries by analyzing permissions and API calls. *Kirin* [9] model developed the security rules to identify the risky applications based upon permission combinations.

Holavanalli *et al.* [12] analyzed the cross-app, i.e., colluding apps, flow permissions to identify the interaction of apps with each other. Grace *et al.* [13] identified the permission leaks, i.e., several permissions that protect access to sensitive user data are unsafely exposed to other apps.

In all of these studies, the authors have analyzed permissions within the normal apps to look for any signs of dangerous behavior. They did not consider the malware samples in their analysis. However, we aim to find the dangerous permission pairs found in malware samples by analyzing different malware datasets.

[1] https://developer.android.com/guide/topics/manifest/manifest-intro

TABLE I

COMPARISON OF PROPOSED WORK WITH STATE-OF-THE-ART STATIC MOBILE MALWARE DETECTION MODELS

Tech.	Features	Perm. Pattrn Analy.	#	Source	Period	Unknown Detection
[14]	P	No	4301	VT	2010-2012	-
[15]	M	No	2808	VT	2010-2012	-
[16]	P	No	6909	G, D, C	2010-2011	-
[17]	P, APIs	No	15336	VS, C	2010-2016	-
[25]	M	No	238	C	2011-2012	-
[5]	M	No	5560	D	2010-2013	-
[30]	APIs	No	8407	G, C, VS	2010-2014	-
[31]	APIs	No	2200	G, MF	2010-2015	-
Prop.	**P**	**Pairs**	**6208**	**G, D, K, C, PZ**	**2010 - 2018**	**84.48%**

Prop. : Proposed Work; P: Permissions; M: Manifest File Components; VT: VirusTotal; G: Genome; D: Drebin; C: Contagio; VS: VirusShare; MF: Mcafee; PZ: Pwnzen Infotech

b) Permissions based malware detection: Sanz *et al.* [14] extracted top permissions in malicious and normal apps using machine learning classifiers. *MAMA* model [15] extracted not only the permissions but also other features of the manifest file. In both these works, the authors focused on extracting the widely used permissions by malware and normal apps.

Talha *et al.* [16] extracted permissions from the apps. For each permission, they calculated the score based upon the number of malware containing the permission, to the total number of malware. Tao *et al.* [17] analyzed permissions, APIs, and the correlation between them to detect malware.

Cen *et al.* [18] applied the probabilistic discriminative model on decompiled source code and the permissions for detecting malicious samples. In similar lines, Peng *et al.* [19] applied probabilistic generative models like Naïve Bayes for evaluating risks of the Android apps based upon the permissions requested. Few recent studies like [20], [21] applied permissions to detect Android malware.

However, none of these works discussed the significant presence of permission patterns in malware samples that can launch any malicious activity. In comparison to these studies, our work focuses on analyzing which permission patterns, in pairs, are significantly present in normal and malicious apps.

c) Permission patterns of android malware: Few studies like [22]–[24] have focused on the analysis of the permission patterns found in Android malware. In [22], permission patterns mining algorithm is applied to identify dangerous permission patterns. However, the model did not consider the normal apps and did not present any model for detection. Similarly, the model presented in [23] used the feature ranking methods to rank the risky permissions. It is identified that SVM gives high accuracy when the model uses a set of 40 permissions, and Random Forest gives better results with ten permissions. *DroidRanger* [24] model identified that a few permission combinations like ⟨SEND_SMS,RECEIVE_SMS⟩, and ⟨INTERNET,RECEIVE_SMS⟩ help in detecting malware. However, it does not discuss in detail which specific permission pairs are prominently present in normal or malicious apps.

Having realized that a large permission set is too complex to analyze, the proposed model selects permission pairs for analysis and detection. Unlike the results reported in [23] and [24], in our study, a thorough examination of which permission pairs exist in normal and malicious apps is made, and a detection model that was missing in [22] is proposed.

d) Manifest file based malware detection: Few studies like [5], [25]–[27] analyzed manifest file components in addition to the permissions. *DroidMat* [25] model analyzed permissions, intents, components, and API calls, and applied K-means clustering for detection. Arp *et al.* [5] used features like permissions, hardware components, API calls, and network addresses to detect malware. The work proposed in [26] used permissions and intents to detect malware. Kim *et al.* [27] formed a vector of the apps consisting of static features such as manifest file components, strings, and API calls.

The proposed work in this paper employs only significant permission pairs, instead of extracting all the components from the manifest file, to reduce the substantial computation overhead. It is also observed in our studies that many recent malware samples like Koodous dataset include only the permissions, not the other components from their manifest files.

e) Permission graph analysis: Solokova *et al.* [28] used graphs to represent the correlation between the permissions of normal apps. They grouped the apps of the same category and calculated metrics like node degree, weighted degree, and page rank score for each graph. However, they considered only the categorized normal apps and not the malicious samples. The proposed work in this paper considers both malware and normal apps to construct separate permission pair graphs.

Zhu *et al.* [29] built a system to evaluate the risks involved in an app based on the permissions. They created a bipartite graph, where one set of vertices constitute the apps and the other set constitute the permissions. There was no relation between the permissions, unlike in our proposed work.

f) API calls based detection: Fan *et al.* [30] formed API calls based frequent subgraphs to classify Android malware into their corresponding families. Zhang *et al.* [31] built dependency graphs of the API calls and applied similarity metrics to classify malware into their families. The *Stowaway* model in [32] determined the set of API calls used by an app and mapped them with the permissions. *Apposcopy* [33] model aimed to detect the apps that steal the user information, by analyzing control-flow and data-flow properties of the apps. Elish *et al.* [34] focused on user-trigger dependence and sensitive APIs to detect malware.

Often, if the APIs used are not related to any manifest file component, they act as noise in the detection process.

Table I compares the existing static works with the proposed work. Most of the works have used outdated malware samples for their experiments and have not analyzed dangerous permission patterns in malware apps. They have also not analyzed the detection rate of their proposed model on unknown samples.

2) Dynamic Detection: Dynamic solutions intend to analyze the run-time behavior of the applications. Related works in this field mainly fall into two aspects: OS-level Detection and Network-level detection.

a) OS-level detection: TaintDroid [35] model, based on dynamic taint analysis, tracked the flow of privacy-sensitive information through third-party apps. Many systems such as [36] are built on TaintDroid. *TaintART* [37] detected the privacy leakage from the apps on the Android Run Time (ART). Yang *et al.* [38] extended the TaintDroid model to detect the data leaks from the apps and also determine whether the leak is due to user intention or not. All these works focused

TABLE II
COMPARISON OF PROPOSED WORK WITH STATE-OF-THE-ART DYNAMIC MOBILE MALWARE DETECTION MODELS

Tech.	**Features**	**Features Overhead**	**#**	**Source**	**Period**	**Unknown Detec-tion**
[35]	DL	Yes	-	-	-	-
[36]	SE, APIs, KL	Yes	3	G	2010-2011	-
[37]	APIs, SC, KL	Yes	-	-	-	-
[43]	TCP	Yes	2267	G, D	2010-2014	-
[44]	HTTP	Yes	725	G	2010-2012	-
[46]	TCP	Yes	889	G	2010-2012	92%
Prop.	**P**	**No**	**6208**	**G, D, K, C, PZ**	**2010 - 2018**	**84.48%**

Prop. : Proposed Work; DL: Dynamic Logs; SE: System Events; KL: Kernel Logs; SC: System Calls; TCP: TCP Traffic; HTTP: HTTP Traffic; P: Permissions; G: Genome; D: Drebin; K: Koodous; C: Contagio; PZ: Pwnzen

TABLE III
COMPARISON OF PROPOSED WORK WITH STATE-OF-THE-ART HYBRID MOBILE MALWARE DETECTION MODELS

Tech.	**Features**	**Features Overhead**	**#**	**Source**	**Period**	**Unknown Detec-tion**
[47]	SC, APIs, AC	Yes	2784	G, C, VS	2010-2015	-
[48]	M, SC	Yes	3273	G, DA, C	2010-2016	-
[49]	API-S, API-D	Yes	1061	G, DB	2010-2015	-
[50]	P, J, DC	Yes	718	G	2010-2012	-
[51]	P, N	Yes	985	G	2010-2012	-
[52]	P, N	Yes	363	G	2010-2012	-
Prop.	**P**	**No**	**6208**	**G, D, K, C, PZ**	**2010-2018**	**84.48%**

Prop. : Proposed Work; SC: System Calls; AC: App Components; M: Manifest File; API-S: Static APIs ; API-D: Dynamic APIs; P: Permissions; J: Java Code; DC: Dalvik Code; N: Network Traffic; G: Genome; C: Contagio VS: VirusShare; DA: DroidAnalytics; DB: DroidBench; D: Drebin; K: Koodous; PZ: Pwnzen

on analyzing data-leaks from the apps, rather than detecting malicious apps.

Shabtai *et al.* [39] analyzed dynamic features such as the CPU usage, number of running processes, and the number of packets sent through Wi-Fi to detect malware. *CopperDroid* [40] model analyzed system calls of malware samples and described whether the malicious behavior is initiated from Java, JNI or native code execution. Afonso *et al.* [41] analyzed dynamic API calls and system calls to detect malicious apps.

All these solutions have relatively high computational overheads compared to static solutions. Moreover, stealthy malware samples try to evade such detection by gaining awareness about the simulation environment. Hence, we focus on static permissions based detection.

b) Network-level detection: Wang *et al.* [42] applied Natural Language Processing methods on the HTTP headers to detect malicious apps. Shabtai *et al.* [43] applied machine learning algorithms on traffic features to generate the patterns of normal traffic and used those patterns to detect malicious apps. Chen *et al.* [44] analyzed network traffic of malicious samples and found that the majority of those samples generated their malicious traffic within the first five minutes.

In one of our previous works [45], we deployed Android emulator to capture the network traffic of malicious and normal apps. Out of 16 network traffic features, 7 of them were found to distinguish normal and malicious traffic. In our other work [46], we used smartphones rather than emulators to capture the traffic. We did not observe any distinguishable features. Therefore, we minimized the number of features required to detect Android malware by ranking them.

All such network traffic based solutions detect only those samples that have network connectivity. For instance, any malware that only sends SMS in the background, will not generate any network traffic. The proposed approach in this paper, however, can be used to detect such type of malware.

Table II compares the existing dynamic models with the proposed work. All the dynamic solutions have feature extraction overheads, and none of them except one has evaluated their model on unknown samples. Moreover, the malware dataset used by other works is relatively older, whereas we have used recent malware samples for our experiments.

3) Hybrid Detection: Hybrid solutions aim to combine static and dynamic components to detect malware. Saracino *et al.* [47] analyzed various dynamic and static features such as system calls, API calls, user activity logs, and permissions. Sun *et al.* [48] proposed a hybrid model by generating static and dynamic graphs from manifest file components, and system calls respectively.

Xia *et al.* [49] performed static APIs analysis and dynamic bytecode analysis to detect the data leaks from the apps. *Riskranker* [50] analyzed dynamic features like run-time Dalvik code loading, and static features like permissions to detect malware. Arora *et al.* [51], [52] presented two different hybrid models by combining traffic features and permissions.

Table III compares the existing hybrid models with the proposed work. All these approaches also suffer from the drawbacks of the dynamic solutions, i.e., high computational overheads. Hence, the proposed work aims to analyze static permission pairs to detect Android malware. Additionally, it also analyzes and detects the recent unknown malware samples.

III. PROPOSED PERMPAIR DETECTION

Analyzing permissions in a group of more than two can be complex, as it can lead to a high number of permission patterns. If an application contains N number of permissions, and we want to analyze permission patterns in the group of R, then the total number of permission patterns will be ${}^{N}C_{R}$, i.e., $\frac{N!}{R!(N-R)!}$. Increasing the value of R from two will result in more number of permission patterns or will be complex to represent. Therefore, we focus on analyzing permission pairs.

In this section, we present our proposed novel *PermPair* model for detecting malicious Android apps.

A. System Design

In order to conduct extensive analysis on permission pairs, we considered malware samples from three different sources: Genome [4], Drebin [5], and Koodous [6]. Besides, we downloaded normal Android apps from the Google Play Store.

The proposed model consists of three phases as shown in Figure 1. The first phase, *Graph Construction phase*, extracts permission pairs from each application to form a graph. Four different permission pair graphs: Genome Graph (G_G), Drebin Graph (G_D), Koodous Graph (G_K) and Normal Graph (G_N) are built during this phase. Typically, every new dataset used in the analysis and detection needs the construction of the graph.

The next phase, *Graph Merge phase*, deals with the merging of all malicious graphs into a single malicious graph (G_M).

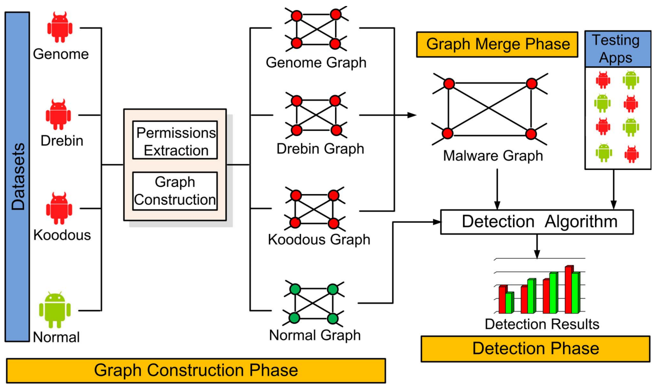


Fig. 1. System design of *PermPair* detection model.

Hence, the final model consists of two graphs: a normal graph (G_N) and a malicious graph (G_M). The subsequent phase proposes a detection algorithm to distinguish normal and malicious apps. Finally, the model eliminates the irrelevant edges, that do not affect the detection results, from both G_M and G_N, to optimize it.

1) Graph Construction Algorithm: The proposed model uses graphs to represent the relationship between various permissions in each application. The model applies the *Algorithm* 1 to construct a graph $G(V, E)$, by representing a unique permission at vertex V and a pair of permissions connected by a weighted edge E. Each edge weight increases with the occurrence of the same permission pair from different applications. An application with single or no permission does not contribute to any analysis or detection.

The algorithm extracts the permissions from each application and generates all the permission pairs. If an app has N number of unique permissions, then the algorithm creates N number of vertices, one for each permission. The number of possible permission pairs added to the graph among all is $\binom{N}{2}$. For instance, in an app, if there are two permissions P_1 and P_2, then, an edge, with an edge weight of one is created between them if it did not exist earlier. Otherwise, edge weight is incremented by one, as presented in Algorithm 1.

Once we process all the apps in a dataset, we divide each edge weight by the total number of apps in that dataset to normalize weights of permission pairs between different graphs. This phase answers our research question 1, describing how to represent the permission pairs extracted from the apps.

After generating separate Genome(G_G), Drebin(G_D) and Koodous(G_K) graphs, they are merged into a single graph (G_M) to represent a common malicious graph.

2) Graph Merge Algorithm: Graph Merge Algorithm, Algorithm 2, merges three malware graphs (G_G, G_D, G_K) to form a common malware graph (G_M) by combining their edges. There are two types of edges: common and disjoint.

Inserting all disjoint edges along with their edge weights to G_M is straightforward. The challenge lies in deciding the weights of the common edges. Consider a common edge e, with weights e_{w1}, e_{w2}, and e_{w3} in G_G, G_D, and G_K respectively. There are different possible ways the weight of edge e can be considered in the final graph G_M such

Algorithm 1 Graph Construction Algorithm

```
1:  Input: Set of Applications ($N$) in a dataset ($D$)
2:  Output: Permission Pair Graph $G(V,E)$
3:  Initially $V \leftarrow \emptyset$ and $E \leftarrow \emptyset$
4:  for $\forall\ N \in D$ do
5:      Extract its Manifest File $M$ using apktool
6:      Let P = ( $P_1$, $P_2$, $P_3$...$P_m$ ) be set of permissions in M
7:      if (|$P$| == 0 or |$P$| ==1) then
8:          $V \leftarrow V + \emptyset$ and $E \leftarrow E + \emptyset$      ▷ zero or single permission
9:      else
10:         For every permission $P_i \in M$, create a node $V_i$ if $V_i \notin V$
11:         Add an edge between every permission pair ($P_i$,$P_j$) derived from $M$
12:         if ($P_i$, $P_j$) $\notin E$ then
13:             $W_{(P_i, P_j)} = 1$
14:         else
15:             $W_{(P_i, P_j)}$ ++
16:         end if
17:     end if
18: end for
19: Divide each edge weight by $N$
20: Return Graph $G(V,E)$
```

as taking their average, minimum or maximum. However, we have adapted the multi-objective optimization approach for the reasons explained below.

The important merge criteria, to find the weights for common edges are (i) a high number of true positives, and (ii) a low number of false positives. This is similar to a multi-objective optimization where objectives are conflicting, i.e., no single solution can give optimum value for all the objectives. Weighted sum method is the most suited classical solution for solving such problems in which we assign weights to all the objectives, as shown in Equation 1.

$$\underset{x}{\text{minimize}}/\underset{x}{\text{maximize}}\ F(x) = [f_0(x), f_1(x), \ldots, f_k(x)],$$
$$\text{subject to } g_i(x) \le b_i, \quad i = 1, \ldots, m. \tag{1}$$

where f_0, $f_1, \ldots,$ f_k are k objective functions to be optimized, with m constraints. Weighted sum method applies scalar weights α_i for each objective such that the sum of all weights is one. Then, each objective is multiplied with its corresponding weight and all of them are added to form a single objective function which is to be optimized, as shown in Equation 2.

$$\underset{x}{\text{minimize}}\ Z, \quad \text{where } Z = \sum_{j=1}^{k} \alpha_j.f_j \ \text{ and } \sum_{j=1}^{k} \alpha_j = 1,$$
$$\text{subject to } g_i(x) \le b_i, \quad i = 1, \ldots, m. \tag{2}$$

Graph merging problem has two objective functions: one for maximizing true positives, say F_1, and other for minimizing false positives, say F_2. The weighted sum method requires weights for both the objectives. Consider an edge e having weights e_{w1} (from G_G), e_{w2} (from G_D), e_{w3} (from G_K), and

e_{wn} (from G_N) respectively. The weight assigned to function F_1, say α_1, is the ratio of the percentage of malware samples containing that edge e to the percentage of the total number of samples containing e, as mentioned in Equation 3.

$$\alpha_1 = \frac{\sum_i e_{\text{wi}}}{\sum_i e_{\text{wi}} + e_{\text{wn}}}. \quad (3)$$

Here $\sum_i e_{\text{wi}}$ is the sum of edge weights of any edge e in all the malicious graphs and e_{wn} represents its edge weight in the normal graph.

Similarly, the weight assigned to function F_2, say α_2, is the ratio of the percentage of normal samples containing that edge e to the percentage of the total number of samples containing the edge e.

Algorithm 2 describes the method to merge three malware graphs. We divide the complete edge set into disjoint and collectively exhaustive sets called $Common_{edge-set}$ and $Disjoint_{edge-set}$. We place the edges common in two or three graphs in $Common_{edge-set}$. We place the remaining edges in $Disjoint_{edge-set}$ and then add them forthwith to G_M.

Algorithm 2 Graph Merge Algorithm

```
1: Input: Three separate malware graphs: $G_G(V_g,E_g)$, $G_D(V_d,E_d)$ and $G_K(V_k,E_k)$
2: Output: Final Malware Graph $G_M(V_m,E_m)$
3: Distribute edge set of all three graphs in two subsets: $Common_{edge-set}$ and $Disjoint_{edge-set}$.
4: for each edge $e_i \in Disjoint_{edge-set}$ do
5:     $E_m \leftarrow (e_i, W(e_i))$
6: end for
7: for each edge $e_j \in Common_{edge-set}$ do
8:     $w_1 \leftarrow$ weight of $e_j \in G_G$
9:     $w_2 \leftarrow$ weight of $e_j \in G_D$
10:    $w_3 \leftarrow$ weight of $e_j \in G_K$
11:    $w_n \leftarrow$ weight of $e_j \in G_N$
12:    if $minimum\{w_1,w_2,w_3\} > w_n$ then
13:        $W(e_j) \leftarrow minimum\{w_1,w_2,w_3\}$
14:        $E_m \leftarrow (e_j, W(e_j))$
15:    else if $maximum\{w_1,w_2,w_3\} < w_n$ then
16:        $W(e_j) \leftarrow maximum\{w_1,w_2,w_3\}$
17:        $E_m \leftarrow (e_j, W(e_j))$
18:    else
19:        Let $m_j$ be weight of edge $e_j$ in final graph $G_M$
20:        Solve for $m_j$ the following optimization problem:
21:        Maximize $Z = \alpha_1(m_j - w_n)(w_1+w_2+w_3) + \alpha_2(w_n - m_j)(w_n)$;
               Subject to constraints: $\alpha_1+\alpha_2 = 1$ and $m_j <= max(w_1, w_2, w_3)$ and $m_j >= min(w_1, w_2, w_3)$
22:        $W(e_j) \leftarrow m_j$
23:        $E_m \leftarrow (e_j, W(e_j))$
24:    end if
25: end for
26: Return $G_M(V_m,E_m)$
```

We apply the weighted sum method to find the weight for the common edges. Let m_i be the weight of a common edge e_i in G_M. The Equation 4 denotes functions F_1 and F_2.

$$F_1 = (m_i - w_n)(w_1 + w_2 + w_3), \quad F_2 = (w_n - m_i)(w_n). \quad (4)$$

where w_1, w_2, w_3, and w_n denotes the weights for e_i in G_G, G_D, G_K, and G_N respectively. The probability of a sample to be classified as malware increases when the weight of any permission pair in G_M is higher than G_N. If m_i is greater than w_n, then the probability of $(w_1 + w_2 + w_3)$ percentage of samples to be classified as malware increases by $(m_i - w_n)$. If w_n is more than m_i, the probability of w_n percentage of samples to be classified as normal increases by $(w_n - m_i)$. Hence, we choose m_i such that both F_1 and F_2 are optimized. No single value of m_i can optimize both the objectives. Therefore, the weighted sum method is opted to solve, which reduces this to single-objective optimization as:

$$Maximize\ Z = \alpha_1(m_i - w_n)(w_1 + w_2 + w_3) + \alpha_2(w_n - m_i)(w_n), \quad (5)$$

subject to the three constraints:

$$\{\alpha_1 + \alpha_2 = 1, \quad m_i <= max(w_1, w_2, w_3),\ and\ m_i >= min(w_1, w_2, w_3)\}. \quad (6)$$

For every edge e_i in $Common_{edge-set}$, we calculate its weight m_i using equations 5 and 6. Then we add e_i and its weight m_i to the final malware graph G_M.

Let us consider an example where there is a common edge, say E_C, in all the three malware graphs and the normal graph. Suppose E_C has weights w_1, w_2, and w_3 in the three malware graphs and w_n in the normal graph respectively. We apply Algorithm 2 to find the weight of E_C in G_M. There can be three possibilities:

- If the edge weight of E_C in all the three malware graphs is higher than that in the normal graph, Algorithm 2 (Steps 12-14) gives *Minimum(*w_1*,* w_2*,* w_3*)* as the weight for the edge E_C in G_M.
- If the edge weight of E_C in all the three malware graphs is lower than that in the normal graph, Algorithm 2 (Steps 15-17) gives *Maximum(*w_1*,* w_2*,* w_3*)* as the weight for the edge E_C in G_M.
- Let w_1, w_2, w_3, and w_n be 0.6, 0.7, 0.9, and 0.8 respectively, i.e., edge weight of E_C in G_N lies between *Minimum(*w_1*,* w_2*,* w_3*)* and *Maximum(*w_1*,* w_2*,* w_3*)*. Algorithm 2 (Steps 18-23), formulates the following multi-objective optimization problem:

$$Maximize\ Z = \alpha_1(m_i - 0.8)(0.6 + 0.7 + 0.9) + \alpha_2(0.8 - m_i)(0.8), \quad (7)$$

where m_i is the required weight of edge E_C in G_M,

$$\alpha_1 = \frac{0.6 + 0.7 + 0.9}{0.6 + 0.7 + 0.9 + 0.8} = 0.73, \quad (8)$$

and

$$\alpha_2 = 1 - 0.73 = 0.27. \quad (9)$$

Hence, multi-objective optimization problem becomes:

$$Maximize\ Z = 0.73(m_i - 0.8)(2.2) + 0.27(0.8 - m_i)(0.8), \quad (10)$$

Solving the above equation, keeping the Equation 6 in mind, gives the value of m_i as 0.9. Therefore, the weight of the edge E_C in G_M turns out to be 0.9.

This graph merge procedure is not required for G_N because we get only one normal graph from the Algorithm 1.

Algorithm 3 Detection Algorithm

```
1: **Input**: Set of Applications ($A_1$, $A_2$,.....,$A_N$) to be Tested
2: **Output**: Outputs each application as Malware or Normal
3: for each $A_i$ do
4:     Extract its Manifest File $M$ using apktool
5:     Let P = ( P$_1$, P$_2$, P$_3$...P$_m$ ) be set of permissions in $M$
6:     if (|$P$| == 0 or |$P$| ==1) then
7:         Return                          ▷ zero or single permission
8:     else
9:         $Mal_{score}$ =0; $Norm_{score}$=0
10:        for Every permission pair ($P_i$, $P_j$) ∈ $M$ do
11:            $Mal_{score}$ + = $W_{(P_i, P_j)} \in G_M$
12:            $Norm_{score}$ + = $W_{(P_i, P_j)} \in G_N$
13:        end for
14:        if $Malicious_{score} > Normal_{score}$ then
15:            Return $A_i$ as malware
16:        else
17:            Return $A_i$ as normal
18:        end if
19:    end if
20: end for
```

3) Malicious App Detection Algorithm: Algorithm 3 describes the procedure that determines whether the app is malicious or not. For every testing app, the algorithm extracts its permission pairs using Algorithm 1. The procedure uses two scores $Normal_{score}$ and $Malicious_{score}$ for the testing apps. These scores can be calculated by searching every permission pair of the testing app in both G_M and G_N and adding the corresponding weights. If $Malicious_{score}$ is greater than $Normal_{score}$ then the procedure declares the app as malicious. This technique is based on the fact that *higher the weight of a permission pair in the malicious graph, higher is the chance of any app containing that permission pair to be malicious*. This phase answers our research question 2, describing the detection model using permission pairs.

4) Edge Elimination: The graphs, G_M and G_N, may contain several edges that do not contribute to the detection such as the edges that have similar weights in both G_M and G_N. We believe the removal of such edges may not alter the detection results. Consider an edge e_1 that has weight 0.1 in both the graphs. It is probable that the deletion of e_1 from both the graphs may not alter the accuracy because e_1 contributes equally to $Malicious_{score}$ and $Normal_{score}$. If on the removal of such edges, *True Positive Rate (TPR)* and *True Negative Rate (TNR)* do not decrease, then we can delete the aforesaid edges from the graphs. We call this procedure as *Edge Elimination*.

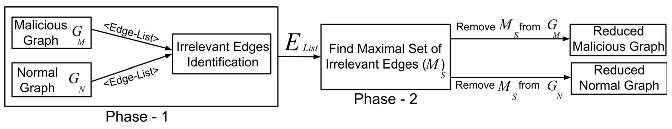


Fig. 2. Phases of edge elimination.

Figure 2 describes the Edge Elimination mechanism that generates Reduced Malicious and Normal graphs. The procedure consists of two phases: 1) Irrelevant Edges Identification, 2) Finding Maximal Set of Irrelevant Edges (M_S).

a) Irrelevant edges identification: For each unique edge existing in the union of G_M and G_N, we delete one edge, common or disjoint, at a time from both the graphs and apply Algorithm 3 to check the detection accuracy. Only if both *TPR* and *TNR* do not decrease, then the edge is considered to be inappropriate, and it is added to the set say $E_{reduced}$. Suppose an irrelevant edge has weight λ and μ in G_M and G_N respectively, then its weight difference from both the graphs comes out to be $| \lambda - \mu |$. We define the *Sum of Weight Differences (SWD)* of all these irrelevant edges as β. We insert all such weight differences for every irrelevant edge of $E_{reduced}$ in a separate list say E_{list}, sorted in increasing order. Now, we need to find the maximal possible set of edges, which is the subset of E_{list}, which we can remove from the graphs that do not lower the *TPR* or *TNR*.

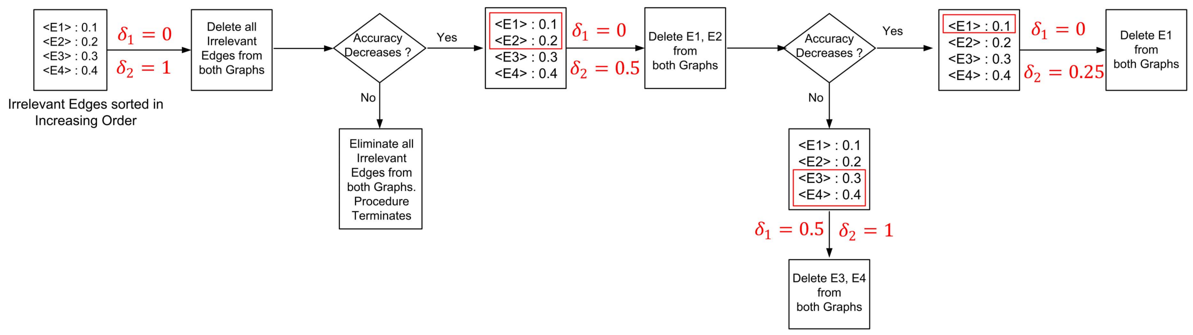
b) Find maximal set of irrelevant edges: We aim to find the maximal set of irrelevant edges that can be eliminated from the graphs. To begin with, we delete all the irrelevant edges, i.e., we delete the edges whose *SWD* is β and check the detection rate. If both *TPR* and *TNR* do not decrease, then we eliminate all the irrelevant edges and the algorithm ends. However, if it results in lowering of *TPR* or *TNR*, then we need to find the maximum possible set of edges that is a subset of E_{list}, which we can remove from the graphs.

We apply a technique similar to the Bisection method [53] to find the maximum possible subset. We set a lower limit, say $\delta 1$, initialized to zero, and an upper limit, say $\delta 2$, equal to β. Our objective is to find δmax, between $\delta 1$ and $\delta 2$, such that *TPR* and *TNR* remain same or higher on deleting all the edges whose weight differences summed up to δmax.

Initially, we set δmax equal to $\delta 2$. If, on deleting all the edges of set E_{list} from the graphs, either *TPR* or *TNR* decreases, we half the upper limit, i.e. $\delta 2 = \frac{\beta}{2}$ whereas the lower limit $\delta 1$ remains zero. The procedure terminates when both $\delta 1$ and $\delta 2$ converge to the same value, which we call as δmax. We delete the edges from the graphs whose *SWD* sums up to $\delta 2$ and test the detection results. Two cases may arise:

- If the detection rate does not decrease, then $\frac{\beta}{2}$ is the new lower limit and the previous value of $\delta 2$ becomes the new upper limit.
- If the detection rate decreases, then zero remains the lower limit and $\frac{\beta}{4}$ is the new upper limit.

The above steps repeat until $\delta 1 = \delta 2$ and we name that point as δmax. We finally remove those set of edges, whose weight difference sums up to the δmax, from G_M and G_N.



<E1> : 0.1
<E2> : 0.2
<E3> : 0.3
<E4> : 0.4
Irrelevant Edges sorted in Increasing Order
$\delta_1 = 0$
$\delta_2 = 1$
Delete all Irrelevant Edges from both Graphs
Accuracy Decreases ?
Yes
No
Eliminate all Irrelevant Edges from both Graphs. Procedure Terminates
$\delta_1 = 0$
$\delta_2 = 0.5$
Delete E1, E2 from both Graphs
$\delta_1 = 0$
$\delta_2 = 0.25$
Delete E1 from both Graphs
$\delta_1 = 0.5$
$\delta_2 = 1$
Delete E3, E4 from both Graphs

Fig. 3. Example of edge elimination from the graphs.

The following example, along with Figure 3, demonstrates the working of edge elimination. For simplicity, we consider four irrelevant edges, found at the end of phase 1. Let these edges between the permission pairs be E_1, E_2, E_3, and E_4. Let E_1 has a weight of 0.6 in G_M and 0.5 in G_N respectively. Hence, its weight Difference from both the graphs is: $| 0.6-0.5 |= 0.1$. Similarly, let us assume weight difference for the edges E_2, E_3, and E_4 is 0.2, 0.3 and 0.4 respectively. We arrange these irrelevant edges in increasing order of weight difference.

Deletion of these edges, individually, from G_M and G_N, does not decrease the detection rate. We assume that the deletion of all the four irrelevant edges from the graphs decreases the detection rate. Hence, phase 2 of the procedure is required to find the maximal set of irrelevant edges.

Sum of Weight Differences (SWD) from the E_{list} comes out to be $\beta = 0.1 + 0.2 + 0.3 + 0.4 = 1.0$. According to the procedure, the lower limit $\delta 1$ is set to zero, the upper limit $\delta 2 = \beta = 1.0$, and $\delta max = \delta 2 = 1.0$. Since it was assumed that deletion of all the four edges decreases the detection rate, in the next iteration, we decrease the upper limit $\delta 2$ from β to $\frac{\beta}{2}$, i.e., new $\delta 2$ is 0.5. Then we delete all the irrelevant edges whose SWD does not exceed $\delta 2$, i.e., 0.5. In this example, SWD for the edges E_1 and E_2 is $0.1 + 0.2 = 0.3$. Therefore, we delete only the edges E_1 and E_2 from both the graphs, and check for detection results. Further two cases may arise:

- If the detection rate decreases, then we further lower the upper limit from $\frac{\beta}{2}$ to $\frac{\beta}{4}$, i.e., new $\delta 2$ is 0.25. The lower limit remains as zero. Then we find the irrelevant edges whose SWD does not exceed 0.25. In this case only edge E_1 is selected for deletion from graphs.
- If the detection rate does not decrease, then $\delta 1$ is increased from zero to $\frac{\beta}{2}$, $\delta 1 = 0.5$ and $\delta 2$ is increased from $\frac{\beta}{2}$ to β, i.e., $\delta 2 = 1.0$. All the irrelevant edges whose SWD lie in the range from $\delta 1$ to $\delta 2$ are selected for deletion. We further test the detection rate by deleting the corresponding edges from both the graphs.

At every iteration, we check for the detection results after deleting some edges from the graphs, and we repeat the steps to decide for the values of $\delta 1$ and $\delta 2$. The procedure terminates when $\delta 1 = \delta 2$. Finally, we eliminate all the edges whose SWD does not exceed $\delta 2$ and return the reduced graphs.

TABLE IV
ANDROID MALWARE DATASET

Dataset Name	Detection Period	Number of Samples Used in		
		Graph Construction Phase	Detection Phase(Known Samples)	Detection Phase(Unknown Samples)
Genome [4]	2010-11	900	364	-
Drebin [5]	2010-12	1244	1500	-
Koodous [6]	2014-18	800	1400	775
Contagio [54]	2018	-	-	250
PwnZen [55]	2018	-	-	300
TOTAL	-	**2944**	**3264**	**1325**

TABLE V
VARIOUS ATTACK CATEGORIES OF ANDROID MALWARE

Spyware	Adware
Phishing Apps	Mobile Bots
System Damage	Information Leak
Financial Loss	Ransomware

IV. RESULTS AND DISCUSSION

This section reports the results of each phase of the *PermPair* model. We performed all the experiments on a desktop system with Ubuntu 12.04 OS and 8 GM RAM and used *apktool* to extract manifest files of the applications. We wrote *shell* scripts to analyze all the phases.

Datasets: In order to perform extensive analysis, we considered a total of 7533 malware apps from three different sources. We used 2944 samples for training during the Graph Construction phase and 3264 samples for testing the accuracy of the proposed model, as shown in Table IV. For the detection phase, we divided the malware samples into two types: *Known* and *Unknown* samples. Consider a typical malware family AnserverBot, say with a total of 180 samples. We used 100 of its samples in the training phase and remaining in the testing phase. Though the malware family is the same, the samples are different in the detection phase. Hence, we named them as *Known* samples. Table IV summarizes the number of *Known* samples from each dataset used for detection. Similarly, the *Unknown* samples are the ones whose family has not been considered in the graph construction phase. We used 1325 *Unknown* samples to test the detection accuracy of our proposed model. All such unknown samples were in the year 2018. We collected the diverse malware samples covering various attack categories as summarized in Table V.

TABLE VI

NUMBER OF NORMAL APPS USED (CATEGORY-WISE)

Category	Number of Apps used in	
	Training Phase	Detection Phase
Education	65	180
Entertainment & Books	94	200
Food & Drink	28	50
Health & Fitness	65	156
Lifestyle	52	96
Music & Video	124	372
Creativity	184	600
Shopping	65	120
Beauty	95	360
Photography	60	150
Social & Communication	88	250
News & Magazines	48	120
Medical	90	300
Business	65	156
Games	370	1390
TOTAL	1493	4500

TABLE VII

NOTATIONS USED FOR TOP PERMISSIONS

Permission	Notation
INTERNET	INT
WRITE_EXTERNAL_STORAGE	WES
ACCESS_NETWORK_STATE	ANS
WAKE_LOCK	WAKE
READ_PHONE_STATE	RPS
ACCESS_WIFI_STATE	AWS
RECEIVE_BOOT_COMPLETED	RBC
READ_SMS	READ
SEND_SMS	SEND
GET_ACCOUNTS	GA
GET_TASKS	GT
SYSTEM_ALERT_WINDOW	SAW

TABLE VIII

TOP TEN PERMISSION PAIRS ALONG WITH EDGE WEIGHTS IN MALWARE AND NORMAL GRAPHS

Genome (G_G)		Drebin (G_D)		Koodous (G_K)		Normal (G_N)	
Permission Pair	Edge Weight	Permission Pair	Edge Weight	Permission Pair	Edge Weight	Permission Pair	Edge Weight
INT:RPS	0.9574	INT:RPS	0.9153	ANS:INT	0.9378	ANS:INT	0.9538
ANS:INT	0.8314	INT:WES	0.6677	INT:WES	0.8261	INT:WES	0.7842
ANS:RPS	0.8062	RPS:WES	0.6530	INT:RPS	0.8210	ANS:WES	0.7670
INT:WES	0.6746	ANS:INT	0.6465	ANS:WES	0.7868	INT:WAKE	0.6457
RPS:WES	0.6702	ANS:RPS	0.6172	ANS:RPS	0.7753	ANS:WAKE	0.6381
AWS:INT	0.6454	INT:SEND	0.5472	RPS:WES	0.7449	WAKE:WES	0.5782
AWS:RPS	0.6412	RPS:SEND	0.5211	AWS:INT	0.6751	GA:INT	0.5320
INT:READ	0.6328	INT:RBC	0.5097	INT:RBC	0.6713	ANS:GA	0.5272
RPS:READ	0.6174	RPS:RBC	0.4934	ANS:AWS	0.6687	INT:RPS	0.4838
ANS:AWS	0.6134	ANS:WES	0.4934	RPS:RBC	0.6649	AWS:INT	0.4803

Additionally, we downloaded 1493 normal apps for training and 4500 for testing. We manually downloaded the normal apps on the same desktop machine via a freely available interface.[2] We selected 15 prominent categories from the Google Play Store having the highest number of apps, as summarized in Table VI. In each category, we selected the freely available apps having the rating of at least 4 in the Google Play Store. We have considered only those normal apps which pass the VirusTotal[3] test. The complete dataset of benign apps used in our experiments is made available at "www.iitr.ac.in/media/facspace/drpskfec/Dataset".

A. Phase-1: Permission Pair Graphs Analysis

First, we defined short notations for permissions as shown in Table VII for easy reference. We highlighted the top ten permission pairs found in G_G, G_D, G_K, and G_N datasets in decreasing order of weights in Table VIII. Permission pair of *INT : RPS* with a weight of 0.9574 in G_G (Table VIII) indicate that the permissions *INTERNET* and *READ_PHONE_STATE* occur together in 95.74% of the Genome samples. Similarly, one can infer the other pairs and their weights.

[2]https://www.apkpure.com

[3]https://www.virustotal.com/

TABLE IX

DIFFERENT MERGING SCHEMES TO GET G_M AND THEIR COMPARISON OF DETECTION RESULTS

Notation	Description	TPR(%)	FPR(%)	Accuracy(%)
Graph-1	*Minimum*$\{e_{w1}, e_{w2}, e_{w3}\}$	54.88	3.12	75.88
Graph-2	*Maximum*$\{e_{w1}, e_{w2}, e_{w3}\}$	99.31	41.35	78.98
Graph-3	*Average*$\{e_{w1}, e_{w2}, e_{w3}\}$	90.49	25.24	82.62
Graph-4	*Average*{*Graph-1, Graph-2*}	93.42	21.27	86.07
Graph-5	*Average*{*Graph-3, Graph-2*}	95.25	30.47	82.39
Graph-6	*Weighted Sum Approach*	95.13	4.25	**95.44**

TABLE X

TOP TEN PERMISSION PAIRS IN FINAL MALICIOUS GRAPH TAKEN FROM DIFFERENT COMBINATIONS

Minimum (Graph-1)		Maximum (Graph-2)		Average (Graph-3)		Weighted Sum	
Permission Pair	Edge Weight	Permission Pair	Edge Weight	Permission Pair	Edge Weight	Permission Pair	Edge Weight
INT:RPS	0.83	INT:RPS	0.95	INT:RPS	0.88	ANS:INT	0.92
INT:WES	0.67	ANS:INT	0.92	ANS:INT	0.79	INT:RPS	0.81
RPS:WES	0.64	INT:WES	0.83	ANS:RPS	0.74	INT:WES	0.80
ANS:INT	0.63	ANS:RPS	0.81	INT:WES	0.71	ANS:WES	0.78
ANS:RPS	0.61	ANS:WES	0.79	RPS:WES	0.69	ANS:AWS	0.68
INT:RBC	0.52	RPS:WES	0.75	ANS:WES	0.62	RPS:WES	0.66
RPS:RBC	0.48	AWS:INT	0.68	AWS:INT	0.59	AWS:INT	0.62
ANS:WES	0.47	INT:RBC	0.67	INT:RBC	0.58	ANS:RPS	0.61
AWS:INT	0.43	ANS:AWS	0.66	RPS:RBC	0.56	INT:WAKE	0.58
INT:SEND	0.42	RPS:RBC	0.66	ANS:AWS	0.55	ANS:WAKE	0.56

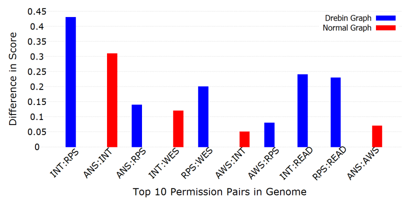
It can be observed that the top five permission pairs of all three malware graphs are quite similar. Pair {INT : RPS} has a high weight in all the malware graphs compared to that of G_N, indicating the leakage of private information of the device to a server. Permissions patterns consisting of ANS, INT, AWS, and WES are more prominent in recent malware compared to the older ones. This leads to the conclusion that the recent malware samples try to evade detection by retaining similar permission patterns as that of normal apps. It is also observed that some pairs containing permissions *WAKE_LOCK* and *GET_ACCOUNT* existed in the top ten pairs of normal apps but were missing in top pairs of malicious apps. Similarly, the pairs like {*ANS : RPS*}, {*RPS : WES*} appeared in the top ten pairs of malware samples but were not found in normal ones.

On closely analyzing the pairs, we found that *Koodous* samples have similarity not only with *Genome* and *Drebin*, but also with the normal apps. This answers our research question 3 that recent malware not only contains permission patterns of the yesteryears but also integrate permission patterns which match largely with the normal apps to evade detection.

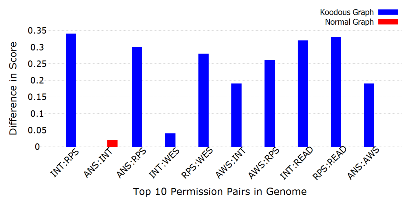
B. Phase-2: Graph Merge

For every common edge *e* with weights e_{w1}, e_{w2} and e_{w3} in three malware graphs, we considered different merging schemes as shown in Table IX. Note that Graph-4 and 5 are different from Graph-3. In Graph-3, the final weight of an edge is the average of weights from G_G, G_D, and G_K; whereas in Graph-4, the final weight of an edge is the average of its minimum and its maximum weight.

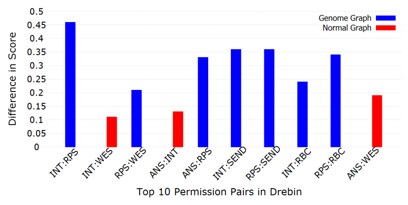
Table X presented the top ten pairs found from three merging possibilities: taking the *minimum* (Graph-1),



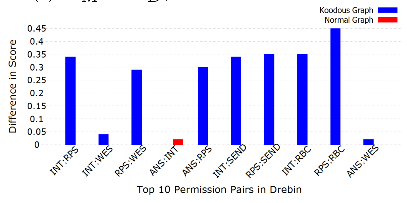
(a) $G_M = G_D$, Test Dataset = Genome



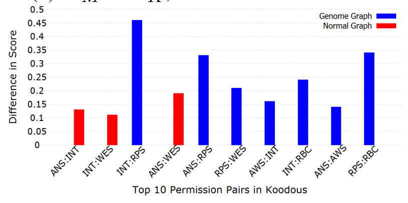
(b) $G_M = G_K$, Test Dataset = Genome



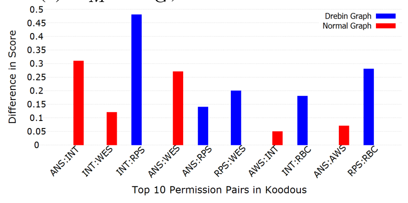
(c) $G_M = G_G$, Test Dataset = Drebin



(d) $G_M = G_K$, Test Dataset = Drebin



(e) $G_M = G_G$, Test Dataset = Koodous



(f) $G_M = G_D$, Test Dataset = Koodous

Fig. 4. Comparison of top permission pairs for the detection of Malware samples.

maximum (Graph-2) and *average* (Graph-3) of the edge weights. Similarly, Table X showed the top permission pairs found from the weighted sum approach (Graph-6). It is observed that there is some similarity between the top pairs of G_D and Graph-1, and G_K and Graph-2; because G_K and G_D have the highest and the lowest weights for most of the common edges.

Table IX summarized the *TPR* and False Positives Rate (*FPR*) for different malware graphs. G_N remained the same in all the experiments. A least *TPR* and *FPR* is observed in Graph-1 because of assignment of minimum weights to the edges in the G_M; the normal score is likely to beat the malware score. Graph-2, on the other hand, displayed the highest *TPR* and *FPR*. We noted that with an increase in the weights of the edges in G_M, both *TPR* and *FPR* increased.

To calculate the accuracy, we considered the average of *TPR* and *True Negative Rate (TNR)*. As can be seen from Table IX, the weighted sum method gave relatively better accuracy of 95.44%. Hence, our proposed graph merge algorithm is better than other possibilities to merge the malicious graphs.

TABLE XI
COMPARISON OF DETECTION RESULTS

Graph used as G_M for Testing	**Detection Accuracy (%)**				
	Genome Samples	**Drebin Samples**	**Koodous Samples**	**Normal Apps**	**Overall Accuracy**
Genome Graph (G_G)	90.65	87.86	58.92	83.88	80.32
Drebin Graph (G_D)	85.16	91.86	34.92	93.95	76.47
Koodous Graph (G_K)	99.17	99.13	94.71	31.08	81.02
Graph Merge of G_G and G_D	90.93	92.13	58.57	95.31	84.23
Graph Merge of G_G and G_K	98.35	97.80	88.35	61.55	86.50
Graph Merge of G_D and G_K	96.70	98.60	84.64	76.62	89.14
Graph Merge G_G, G_D and G_K	98.07	98.06	89.28	95.75	**95.44**

C. Phase-3: Detection

In this section, we analyzed the detection results obtained from the proposed *PermPair* approach for two cases: (i) **Individual Detection** in which we do not perform any graph merge, and every individual malicious graph is taken as G_M for detection, and (ii) **Graph Merge based Detection** in which we apply graph merging to get the G_M for detection.

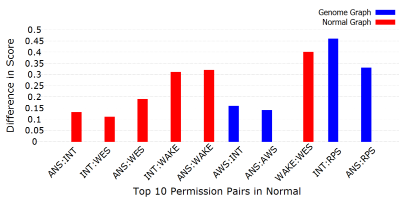
1) Individual Detection: As an outcome from Section III(1), three malware graphs G_G, G_D, and G_K were produced. These graphs individually were considered as G_M to observe the detection results. We tested all the datasets with all these malware graphs. Note that, we considered different samples for the graph construction and detection phases. Table XI summarized the detection results.

a) Detection of Genome and Drebin samples: Table XI indicates that G_K detected the malware samples with high accuracy because the permission pairs in G_K had a high similarity to G_G and G_D (Table VIII). Figure 4 showed the detection rate of Genome and Drebin samples. Figure 4(b) showed that among the top ten pairs in G_G, G_K scored more than G_N for nine pairs. Similarly, Figure 4(d) demonstrated that for top pairs in G_D, G_K scored more than G_N for nine pairs. Therefore G_K detected 99.17% and 99.13% of Genome and Drebin samples respectively. On the contrary, G_G and G_D detected the malware samples with relatively lower accuracy. This is because G_N scored higher for a few of the top permission pairs, as Figures 4(a) and 4(c) depicted it.

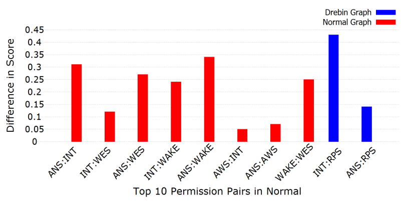
b) Detection of Koodous samples: Figures 4(e) and 4(f) showed that G_N scored higher than G_G and G_D for three and five permission pairs respectively out of the top ten pairs of G_K. As a result, G_G and G_D detected a low accuracy, 58.92%, and 34.92% respectively, of the Koodous samples.

c) Detection of normal samples: Figure 5(c) showed that many top permission pairs of normal apps scored high in G_K than G_N. Hence G_K generated a high false positives rate as compared to G_G and G_D. There were some of the permissions which existed only in one or two of the malicious datasets. Consider a case where some permission pairs existed only in G_G and G_D. Even on adding such pairs to G_K, the detection results did not alter. Similarly, some pairs existed only in G_K. Even on adding these pairs to G_G and G_D, the results did not improve because such pairs existed in very few samples as they were having a very low edge weight and therefore did not contribute to the detection results. Hence, the overall detection accuracy from individual malware graphs was relatively low.

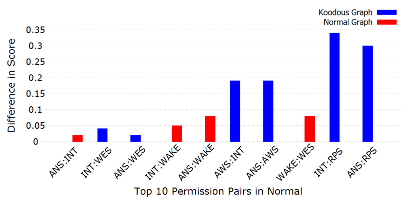
2) Graph Merge Based Detection: Refer to Tables IX and XI, as discussed in the previous subsection, the overall accuracy achieved by merging of malware graphs using



(a) $G_M = G_G$, Test Dataset = Normal



(b) $G_M = G_D$, Test Dataset = Normal



(c) $G_M = G_K$, Test Dataset = Normal

Fig. 5. Comparison of top permission pairs for the detection of normal samples.

TABLE XII

False Positives : Presence of Top Dangerous Permission Pairs in Normal Apps

Normal Apps	Type of App / Functionality	Top ten dangerous permission pairs sorted in decreasing order of edge weight									
		GT:RBC	ANS:GT	GT:WES	RPS:SND	INT:RPS	INT:SAW	INT:SND	GT:SAW	RPS:SAW	ANS:SAW
TextPlus	online SMS/calls	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fonetastic	blocks call/ SMS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Block sms call	blocks calls/SMS, deletes call history	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Android Assistant	deletes files and browser history	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Chaton	texting app	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Golocker	changes wallpaper	✓	✓	✓		✓	✓		✓	✓	✓
Omni Swipe	changes user profile,disables Wi-Fi	✓	✓	✓		✓	✓		✓	✓	✓
All in one Tool	reboots device, deletes files	✓	✓	✓		✓	✓		✓	✓	✓
Passcode Lock	disables HOME keys and sound	✓	✓	✓		✓	✓		✓	✓	✓
UR Launcher	change wallpaper and user profiles	✓	✓				✓		✓		✓
Surefox Browser	locks browser, disables websites	✓	✓	✓		✓	✓		✓	✓	✓
Shutapp	shut-downs apps	✓	✓	✓		✓	✓		✓	✓	✓
Vpn Proxy	encrypts network communication	✓	✓	✓		✓	✓		✓	✓	✓

weighted sum approach is 95.44%, which is better than the accuracy achieved by any of the individual malware graphs. We merged two malware graphs at a time, using the same Weighted Sum approach, and checked for detection accuracy, as summarized in Table XI. None of the two-graphs merge approaches yielded better accuracy than the three-graph merge approach. Hence, we focused our discussion on three-graphs merge approach. This subsection deals with the analysis of false positives and false negatives when we merge all three graphs to get G_M.

a) False positives analysis: Graph-6, from Table IX, when used as G_M, identified 4.25% False Positives (FPs). Table XII, having top ten *dangerous permission pairs*, presented the reason behind these results. A pair is said to be *dangerous* if it has more weight in G_M than in G_N. As big as 35% of the total *FPs* recorded high malware score. Table XII summarized many of such apps. We observed that apps like *Fonetastic*, *Block SMS and Call*, *Android Assistant*, etc. had all top ten *dangerous permission pairs*. Hence, they are detected as malicious with a high score. Besides, most of these normal apps indicated the signs of suspicious behavior. For instance, they can block calls/SMS, kill apps, delete files, and disable features like Wi-Fi. Though our model detected such apps as malware, we observed that the majority of the *FPs* belong to the *Social and Communication* category of the normal apps.

Interestingly, normal chat apps like *TextPlus* and *Chaton*, too had the dangerous permission pairs, but they did not pose any serious functionality. We tested other similar apps like *WhatsApp* and found that the proposed approach detected *WhatsApp* as malware with a low score difference of 0.04 in favor of G_M. On the contrary, *TextPlus* and *Chaton* scored a high difference of 21 and 10 respectively. *WhatsApp* also contained 6 of the top ten dangerous permission pairs, but it was the presence of permissions like *SYSTEM_ALERT_WINDOW*, *RECEIVE_MMS*, *BROADCAST_SMS*, *READ_CALL_LOG* and *WRITE_CALL_LOG* in *TextPlus* and *Chaton* which gave a high malicious score. The pairing of these permissions with *INTERNET* and *ACCESS_NETWORK_STATE* had a high weight in G_M than in G_N, hence giving a high malware score.

Remaining 65% of the FPs had a very low score difference, and they belonged to categories like online shopping apps, taxi booking apps, spy camera apps, and online games. Figure 6(a) showed the difference in the score of the testing set of normal apps. Most of the FPs had a score difference of as low as 0.05. Apps like *Fonetastic* and *Block SMS and call* had a high score difference of 41 in favor of G_M.

b) False negatives analysis: Table XIII and Figure 6(b) showed False Negatives (FNs) analysis. Analogous to *dangerous pairs*, we define the term *normal permission pairs*; i.e., pairs with the highest difference in favor of G_N. The proposed model detected samples of 19 malware families as FNs. Eight of them contained many of the top *normal pairs*. The remaining ones did not include any of the top *normal permission pair*, instead, they had a very less number of permissions (between two to five) as showed in Table XIII(b). Edge weight of these pairs was almost equivalent in both the graphs with very little difference in favor of G_N. 84% of the FNs had a score difference of less than or equal to one.

We conclude that our approach is effective in detecting Android malware with an accuracy of 95.44%, and the majority of the FPs (65%) and FNs (84%) had a very low

TABLE XIII

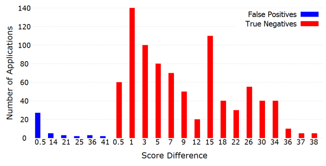
FALSE NEGATIVES ANALYSIS. (A) TOP NORMAL PERMISSION PAIRS IN MALICIOUS APPS. (B) LESS NUMBER OF PERMISSIONS IN MALICIOUS APPS

(a) Top normal permission pairs in malicious apps

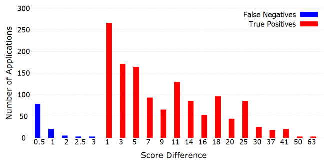
Malicious Apps	**Top ten normal permission pairs sorted in decreasing order of edge weight**									
	ANS:GA	GA:INT	GA:WAKE	GA:WES	WAKE:WES	AWS:WES	GA:VIB	RES:WAKE	AWS:GA	GA:RES
DroidKungFu3	✓	✓		✓		✓	✓		✓	
Opfake		✓	✓	✓	✓					
Faketimer		✓	✓				✓			
SMSReg	✓	✓	✓	✓	✓		✓			
Airpush	✓	✓	✓	✓	✓	✓	✓		✓	
Banker	✓	✓	✓	✓	✓		✓	✓		✓
Clicker	✓	✓	✓	✓	✓		✓	✓		✓
Dendroid	✓	✓	✓				✓			

(b) Less number of permissions in malicious apps

Malware	**Permissions**
Basebridge	INT, ANS, VIB
Hamob	INT, ANS
Fakeinstaller	INT, WAKE, WES
Hacktool	INT, AWS, VIB, ANS, WAKE
Mobidash	INT, AWS, VIB, ANS, WAKE
Slembunk	INT, AWS, VIB, ANS, WAKE
SLocker	INT, AWS, VIB, ANS, WAKE
Ransomware	INT, AWS, ANS, WES
Viking Horde	INT, RES, WES



(a) For normal apps



(b) For malware apps

Fig. 6. Difference in detection score.

score difference. Tables XII and XIII answered our research questions 4 and 5, highlighting the dangerous and normal permission pairs significantly present in malware and normal apps respectively.

c) Detection of unknown samples: We considered 775 recent samples from Koodous, 250 samples from Contagio, and 300 samples from Pwnzen Infotech [55] to test the detection rate of the proposed approach on the unknown samples. The results demonstrated that the proposed approach detected 714 of the Koodous, 205 of the Contagio, and 238 of the Pwnzen samples correctly with the detection rates of 92.12%, 82%, and 79.33% respectively. Hence, the overall detection rate on unknown samples was 84.48%. Many recent samples from Contagio and Pwnzen contained permissions similar to the normal apps, hence, gave a relatively lower detection rate of 82% and 79% respectively. This highlighted the fact that the proposed model should be continuously trained with the recent malware so that it can detect the unknown samples with high accuracy. While analyzing the false negatives for Koodous samples, we found similar permission patterns as in false negatives of known samples, i.e., containing less number of permissions such as *INT*, *ANS*, and *VIB*.

D. Phase-4: Edge Elimination

The number of edges in G_M and G_N is 7,658 and 10,661 respectively. We applied the Edge Elimination algorithm to eliminate the unnecessary edges. After the first phase, we found a total of 12,130 edges, i.e., deleting these edges from both the graphs, one at a time, did not decrease the detection accuracy. However, deleting all these edges together decreased the accuracy. Phase-2 of the algorithm aimed at finding the maximal subset of edges that can be deleted from the graphs and do not decrease the accuracy. We noticed that the accuracy of our model was 95.44%. Table XIV summarized the results of Phase-2. After a certain number of iterations, we got $\delta 1$ and $\delta 2$ nearly up to four decimal places. We observed that the algorithm could remove a maximum of 552 and 4382 edges from G_M and G_N respectively, thus, eliminating 7.2% edges in G_M and 41.1% edges in G_N. The detection time for testing 3264 malicious and 4500 normal apps was measured to be 2 minutes 45 seconds without edge elimination. This got reduced to 1 minute 58 seconds, i.e., a reduction of around 28% was possible. Even though this reduction is in seconds, this will have an impact if there are hundreds of thousands of files to be tested. Hence, the edge elimination algorithm reduces both, the number of edges in G_M & G_N, and the detection time, as shown in Table XV.

TABLE XIV

NUMBER OF EDGES REMOVED IN G_M AND G_N AFTER EVERY ITERATION OF PHASE 2 OF EDGE ELIMINATION

$\delta 1$	$\delta 2$	**Number of edges removed**		**TPR (%)**	**FPR (%)**	**Accuracy (%)**	**Computation Time**
		Malware Graph	**Normal Graph**				
0	97.1	6709	9665	90.46	2.9	93.78	0m 50 s
0	48.55	6164	9097	91.58	3.52	94.03	1m 8s
0	24.275	5145	8173	92.34	3.82	94.26	1m 17s
0	12.1375	3450	7044	93.26	3.98	94.64	1m 32s
0	6.06875	1909	5854	93.26	3.98	94.64	1m 48s
0	3.0343	684	4512	94.14	4.18	94.98	1m 59s
0	1.517	362	2347	95.13	4.25	95.44	2m 3s
1.517	3.0343	424	3500	95.13	4.25	95.44	1m 58s
2.275	3.0343	424	4046	95.13	4.25	95.44	1m 58s
2.6546	3.0343	464	4300	95.13	4.25	95.44	1m 58s
2.844	3.0343	595	4421	94.14	4.18	94.98	1m 58s
2.844	2.939	515	4352	95.13	4.25	95.44	1m 58s
2.891	2.939	569	4395	94.14	4.18	94.98	1m 58s
2.891	2.915	528	4366	95.13	4.25	95.44	1m 58s
2.903	2.915	563	4389	94.14	4.18	94.98	1m 58s
2.903	2.909	537	4371	95.13	4.25	95.44	1m 58s
2.906	2.909	545	4377	95.13	4.25	95.44	1m 58s
2.9075	2.909	552	4382	95.13	4.25	95.44	1m 58s
2.9082	2.909	563	4389	94.14	4.18	94.98	1m 58s
2.9082	2.9086	552	4382	95.13	4.25	95.44	1m 58s
2.9084	2.9086	563	4389	94.14	4.18	94.98	1m 58s
2.9084	2.9085	552	4382	95.13	4.25	95.44	1m 58s
2.90845	2.9085	563	4389	94.14	4.18	94.98	1m 58s
2.90845	2.908475	563	4389	94.14	4.18	94.98	1m 58s
2.90845	2.9084625	552	4382	95.13	4.25	95.44	1m 58s

E. Comparison With Anti-Malware Apps

To appraise the achieved results, we compared the malware detection rate of our model with 12 popular mobile anti-malware apps, taken from *VirusTotal*.[4] For comparison, we chose 275 random *unknown* malware from Koodous from a total of 775 *unknown* malicious samples to check manually for

[4]https://www.virustotal.com/

TABLE XV

Comparison of Different Parameters With and Without Graph Reduction Algorithm

	Without Graph Reduction	With Graph Reduction	Efficiency
Number of Edges in G_M	7658	7106	7.2%
Number of Edges in G_N	10661	6279	41.1%
Detection Time	2m 45s	1m 58s	28%
TPR (%)	95.13	95.13	Same
FPR (%)	4.25	4.25	Same

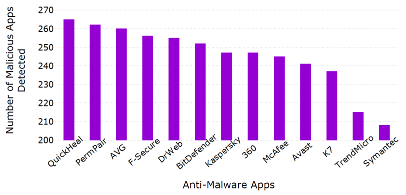


Fig. 7. Comparison of the proposed approach with different anti-malware apps.

TABLE XVI

Comparison of the Proposed Approach With Existing Permissions-Based Malware Detection Techniques. (a) TPR and FPR. (b) Precision and Recall

(a) TPR and FPR

Detection Techniques	**TPR (%)**	**FPR (%)**	**Permission Patterns**
Sanz et al. [14]	91	19	All permissions together
Sanz et al. [15]	94	5	All permissions together
Arp et al. [5]	94	1	All permissions together
Wang et al. [23]	94.6	0.6	Set of 40
Proposed Approach	**95.13**	**4.25**	Set of 2

(b) Precision and Recall

Detection Techniques	**Precision (%)**	**Recall (%)**
Zhua et al. [21]	88.16	-
Milosevic et al. [20]	95.8	95.7
Wu et al. [25]	96.7	87.3
Proposed Approach	**96.3**	**95.13**

each sample through *VirusTotal*. Figure 7 compared the results of *PermPair* model with widely known anti-malware apps. We noticed that except *QuickHeal*, all of the anti-malware apps had a lower malware detection rate than *PermPair*. *Quickheal* detected 265 samples, three more than *PermPair*. Those three samples had single permission defined in their manifest file, hence those three samples went undetected by our model. We conclude that (a) our approach is relatively better than eleven of the mobile anti-malware apps, and (b) most of the anti-malware apps are unable to detect newer malicious samples.

F. Comparison With Existing Approaches

In this subsection, we presented a comparative evaluation with other state-of-the-art permissions-based detection techniques, though they used different datasets for their experiments. To the best of our knowledge, no other work in the literature used the same datasets as ours. Some of the works reported their results in *TPR* and *FPR*, whereas others measured *Precision* and *Recall*. Table XVI compared the detection results of the proposed approach with other permissions-based solutions. The proposed approach gave nearly the same detection rate as that of other related works which used *Precision* and *Recall* for evaluation. Furthermore, the proposed approach outperformed all the works in terms of *TPR*. The better performance of our approach was due to the dangerous permission pairs analysis. Presence of such pairs contributed to detecting a high number of malicious samples.

Our model detected a few *FPs* with some malicious behavior. Few normal apps behaved maliciously such as blocking phone calls and deleting the files. Therefore, our model reported a little higher *FPR* than [5] and [23]. However, authors in [5] did not analyze the permission patterns of malicious samples. The authors in [23] reported dangerous permission patterns but in the set of 40. Representing and analyzing a large set of permissions can be complex. We analyzed permissions in a group of two and can be easily represented by graphs.

G. Limitations

The proposed approach has some gray areas which we intend to discuss in this subsection. The proposed model requires permission pairs to test the apps. Hence, the apps containing single or no permission cannot be analyzed. Some of the malware samples with few permissions evade the detection. Moreover, the proposed model gives relatively high *FPR* because many normal apps from *Social and Communication* category of the Google Play Store have been identified as malicious due to the presence of dangerous permission pairs. To evade detection, attackers may add widely used normal permissions in the malware samples, thus, generating more number of normal permission pairs. Besides, the proposed approach does not focus on detecting colluding apps [56], [57]. To overcome these limitations, in the future, we will analyze how efficiently other components such as intent filters, hardware components, and API call logs can be used for detection, in addition to the permissions. Since the proposed approach is static, malware with update capabilities (downloading of malicious components at update time) may evade the detection. To overcome this limitation, a dynamic detector can be deployed that could analyze the run time behavior of the apps. However, this will come at the cost of some computational overhead.

V. Conclusion and Future Work

In this paper, we proposed a novel approach for detecting malicious apps by using permission pairs extracted from the manifest files. We constructed the graphs to analyze permission pairs for both normal and malicious samples and assigned an edge weight to every pair depending upon the number of samples in which the pair is present. We subjected three malware datasets namely: Genome, Drebin, and Koodous to analysis. The datasets contained the newer samples detected in 2014-18 in addition to the older samples, detected in 2010-2014. Initially, we constructed three different graphs; one for each dataset and we observed certain deviations in permission pairs of newer samples compared to the older ones. We merged different malware graphs in a single graph using the weighted sum method. We further performed edge elimination to remove the unnecessary edges. Results showed that our proposed method is better than eleven of the popular mobile anti-malware apps. Our future work will focus on analyzing the other components of the manifest file like intent filters and hardware components to increase the detection accuracy.

ACKNOWLEDGMENT

We have taken the Genome dataset from the authors of [4], Drebin [5] dataset from Institute for System Security, Technische Universitat Braunschweig, Germany, and Koodous [6] dataset from their website that hosts a large number of malicious applications for analysis. We express deep gratitude to Genome, Drebin and Koodous projects for providing us the malware samples.

REFERENCES

[1] *Market Share Alert: Preliminary, Mobile Phones, Worldwide*, Gartner, Stamford, CT, USA, 2017.

[2] *97% of Mobile Malware is on Android. This is the Easy Way You Stay Safe*, Forbes Media, Jersey City, NJ, USA, 2014.

[3] *2018 Malware Forecast: The Onward March of Android Malware*, Security Report, 2017.

[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 23–26.

[6] *Koodous Malware Dataset*. Accessed: Nov. 25, 2019. [Online]. Available: https://www.koodous.com

[7] *Android Developers Guide*. Accessed: Nov. 25, 2019. [Online]. Available: https://developer.android.com/guide/index

[8] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM WiSec*, 2012, pp. 101–112.

[9] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM CCS*, 2009, pp. 235–245.

[10] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. 8th Symp. Usable Privacy Secur.*, 2012, Art. no. 3.

[11] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. 19th ACM CCS*, 2012, pp. 217–228.

[12] S. Holavanalli *et al.*, "Flow permissions for Android," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2013, pp. 652–657.

[13] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. NDSS*, 2012, p. 19.

[14] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission usage to detect malware in Android," in *Proc. Int. Joint Conf. CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*. Berlin, Germany: Springer, 2013.

[15] B. Sanz *et al.*, "MAMA: Manifest analysis for malware detection in Android," *Cybern. Syst.*, vol. 44, nos. 6–7, pp. 469–488, 2013.

[16] K. A. Talha, D. I. Alper, and C. Aydin, "APK Auditor: Permission-based Android malware detection system," *Digital Invest.*, vol. 13, pp. 1–14, Jun. 2015.

[17] G. Tao, Z. Zheng, Z. Guo, and M. R. Lyu, "MalPat: Mining patterns of malicious and benign Android apps via permission-related APIs," *IEEE Trans. Rel.*, vol. 67, no. 1, pp. 355–369, Mar. 2018.

[18] L. Cen, C. S. Gates, L. Si, and N. Li, "A probabilistic discriminative model for Android malware detection with decompiled source code," *IEEE Trans. Depend. Secure Comput.*, vol. 12, no. 4, pp. 400–412, Jul./Aug. 2015.

[19] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. ACM CCS*, 2012, pp. 241–252.

[20] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Comput. Elect. Eng.*, vol. 61, pp. 266–274, Jul. 2017.

[21] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of Android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, Jan. 2018.

[22] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious Android applications," *Future Gener. Comput. Syst.*, vol. 36, pp. 122–132, Jul. 2014.

[23] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zang, "Exploring permission-induced risk in Android applications for malicious application detection," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.

[24] 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*, 2012, pp. 50–52.

[25] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur. (Asia JCIS)*, Aug. 2012, pp. 62–69.

[26] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "Pindroid: A novel Android malware detection system using ensemble learning methods," *Comput. Secur.*, vol. 68, pp. 36–46, Jul. 2017.

[27] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for Android malware detection using various features," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 3, pp. 773–788, Mar. 2019.

[28] K. Sokolova, C. Perez, and M. Lemercier, "Android application classification and anomaly detection with graph-based permission patterns," *Decis. Support Syst.*, vol. 93, pp. 62–76, Jan. 2016.

[29] H. Zhu, H. Xiong, Y. Ge, and E. Chen, "Mobile app recommendations with security and privacy awareness," in *Proc. ACM KDD*, 2014, pp. 951–960.

[30] M. Fan *et al.*, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.

[31] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM CCS*, 2014, pp. 1105–1116.

[32] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. ACM CCS*, 2011, pp. 627–638.

[33] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Symp. Found. Softw. Eng.*, 2014, pp. 576–587.

[34] K. O. Elish, X. Shu, D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Comput. Secur.*, vol. 49, pp. 255–273, Mar. 2015.

[35] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014, Art. no. 5.

[36] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. ACM CODASPY*, 2013, pp. 209–220.

[37] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android runtime," in *Proc. ACM CCS*, 2016, pp. 331–342.

[38] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. ACM CCS*, 2013, pp. 1043–1054.

[39] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2011.

[40] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors," in *Proc. 6th Eur. Workshop Syst. Secur.*, 2013, pp. 1–6.

[41] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *J. Comput. Virology Hacking Techn.*, vol. 11, no. 1, pp. 9–17, 2015.

[42] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting Android malware leveraging text semantics of network flows," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1096–1109, May 2018.

[43] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Comput. Secur.*, vol. 43, no. 6, pp. 1–18, 2014.

[44] Z. Chen *et al.*, "A first look at Android malware traffic in first few minutes," in *Proc. IEEE Trustcom*, Aug. 2015, pp. 206–213.

[45] A. Arora, S. Garg, and S. K. Peddoju, "Malware detection using network traffic analysis in Android based mobile devices," in *Proc. 8th Int. Conf. Next Gener. Next Gener. Mobile Apps, Services Technol.*, Sep. 2014, pp. 66–71.

[46] A. Arora and S. K. Peddoju, "Minimizing network traffic features for Android mobile malware detection," in *Proc. 18th Int. Conf. Distrib. Comput. Netw.*, 2017, Art. no. 32.
[47] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.
[48] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A user-oriented behavior-based malware variants detection system for Android," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1103–1112, May 2017.
[49] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time Android application auditing," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 899–914.
[50] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 281–294.
[51] A. Arora and S. Peddoju, "NTPDroid: A hybrid Android malware detector using network traffic and system permissions," in *Proc. 17th IEEE TrustCom*, Aug. 2018, pp. 808–813.
[52] A. Arora, S. K. Peddoju, V. Chouhan, and A. Chaudhary, "Hybrid Android malware detection by combining supervised and unsupervised learning," in *Proc. 24th ACM MobiCom*, 2018, pp. 798–800.
[53] K. Atkinson, *An Introduction to Numerical Analysis*. Hoboken, NJ, USA: Wiley, 2008.
[54] *Contagio Mobile Malware Dump*. Accessed: Nov. 25, 2019. [Online]. Available: https://www.contagiominidump.blogspot.com
[55] S. Chen *et al.*, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Comput. Secur.*, vol. 73, pp. 326–344, Mar. 2018.
[56] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, p. 19.
[57] K. Elish, H. Cai, D. Barton, D. Yao, and B. Ryder, "Identifying mobile inter-app communication risks," *IEEE Trans. Mobile Comput.*, to be published.

Anshul Arora is currently pursuing the Ph.D. degree from the Department of Computer Science and Engineering, IIT Roorkee, India, under the guidance of Dr. S. K. Peddoju. He is also an Assistant Professor of discipline of mathematics and computing with the Delhi Technological University Delhi, India. His research interests include mobile security, mobile malware detection, and network traffic analysis.

Sateesh K. Peddoju (SM'18) has been with IIT Roorkee, India, since 2010. He has publications in reputed journals like IEEE POTENTIALS, MTAP, WPC, and IJIS and conferences, including TrustCom, MASS, ICDCN, and ISC. His research interests include cloud computing, ubiquitous computing, high-performance computing and security. He is currently a Senior Member of ACM. He was a recipient of University Rank and scholarship, and several Best Paper Awards and Best Teacher Award. He is also the Secretary of the IEEE Roorkee section, the Vice-Chair of IEEE Computer Society, India council, and a Founding Faculty Advisor of ACM Student Chapter-IIT Roorkee. He is also a Reviewer of top-rated journals like IEEE TCC, IEEE TSC, MTAP, COSE, COMNET, and JNCA. He is also the Founding General Co-Chair of SLICE-2018 and the General Chair of DSEA-2018. He is in several conferences like IEEE MASS, IEEE ATC, IEEE SmartComp, IEEE iNIS, and IoTSMS.

Mauro Conti (SM'14) received the Ph.D. degree from the Sapienza University of Rome, Italy, in 2009. After then, he was a Post-Doctoral Researcher with Vrije Universiteit Amsterdam, The Netherlands. In 2011, he joined the University of Padua as an Assistant Professor, where he became an Associate Professor in 2015, and a Full Professor in 2018. He was a Visiting Researcher with GMU in 2008 and 2016, UCLA in 2010, UCI from 2012 to 2014, and in 2017, TU Darmstadt in 2013, UF in 2015, and FIU from 2015 to 2016. He was awarded with a Marie Curie Fellowship in 2012 by the European Commission, and with a Fellowship by the German DAAD in 2013. He is currently a Full Professor with the University of Padua, Italy, and an Affiliate Professor with the University of Washington, Seattle, USA. His research is also funded by companies, including Cisco and Intel. His main research interests include security and privacy. In this area, he published more than 200 articles in topmost international peer-reviewed journals and conference. He is also an Area Editor-in-Chief of the IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, and an Associate Editor for several journals, including the IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and the IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. He was the Program Chair of TRUST 2015, ICISS 2016, WiSec 2017, and the General Chair of SecureComm 2012 and ACM SACMAT 2013.