Familial Clustering For Weakly-labeled Android Malware Using Hybrid Representation Learning

Yanxin Zhang, Yulei Sui*, Shirui Pan, Zheng Zheng, Baodi Ning, Ivor Tsang and Wanlei Zhou

***Abstract*—Labeling malware or malware clustering is important for identifying new security threats, triaging and building reference datasets. The state-of-the-art Android malware clustering approaches rely heavily on the raw labels from commercial AntiVirus (AV) vendors, which causes misclustering for a substantial number of *weakly-labeled malware* due to the inconsistent, incomplete and overly generic labels reported by these closed-source AV engines, whose capabilities vary greatly and whose internal mechanisms are opaque (i.e., intermediate detection results are unavailable for clustering). The raw labels are thus often used as the only important source of information for clustering.**

To address the limitations of the existing approaches, this paper presents ANDRE, a new ANDroid Hybrid REpresentation Learning approach to clustering weakly-labeled Android malware by preserving heterogeneous information from multiple sources (including the results of static code analysis, the meta-information of an app, and the raw-labels of the AV vendors) to jointly learn a hybrid representation for accurate clustering. The learned representation is then fed into our outlier-aware clustering to partition the weakly-labeled malware into known and unknown families. The malware whose malicious behaviours are close to those of the existing families on the network, are further classified using a three-layer Deep Neural Network (DNN). The unknown malware are clustered using a standard density-based clustering algorithm. We have evaluated our approach using 5,416 ground-truth malware from Drebin and 9,000 malware from VIRUSSHARE (uploaded between Mar. 2017 and Feb. 2018), consisting of 3324 weakly-labeled malware. The evaluation shows that ANDRE effectively clusters weakly-labeled malware which cannot be clustered by the state-of-the-art approaches, while achieving comparable accuracy with those approaches for clustering ground-truth samples.

I. INTRODUCTION

ANDROID devices have represented around 80% of all smartphones since 2017 and are forecast to maintain their leadership with over 85% market share by 2020 [1]. The increasing popularity of Android devices has witnessed an unprecedented growth in emerging Android apps, which has also become the prime targets of attackers. It has been reported that the total number of Android malware had reached 856.52 million by the end of 2018. There were more than 137.5 million newly discovered malicious apps in 2018 (i.e., 350,000 new malware per day) [2]. Android malware is a major source of cyberattacks and is a serious threat to smartphone users.

Labeling a malicious app as an unknown or a variant of an existing family is important for identifying new threats, determining the severity of the threat, creating signatures for malware detection, malware triaging, and building reference datasets. Labeling malware is a non-trivial task. The raw labels reported by AntiVirus (AV) vendors are well-known to be inconsistent (e.g., two vendors may report two aliased family names for the same type of malware. `solimba` and `firseria` are aliases) without a standard naming convention (e.g., the conventions CARO [3] and CME [4] are not often used by AV vendors). Although manual inspection for malware labeling by an expert can provide an accurate solution, it is extremely costly in practice due to the huge number of apps being released every day.

*Corresponding author.

TABLE I
MALWARE WITH EMPTY LABELS. THE TABLE GIVES THE NUMBER OF MALICIOUS APPS WHICH DO NOT HAVE A FAMILY NAME AFTER CLUSTERING BY EUPHONY AND AVCLASS DUE TO OVERLY GENERIC RAW LABELS BEING REPORTED BY AV VENDORS. THE 9000 MALICIOUS APPS ARE FROM VIRUSSHARE UPLOADED BETWEEN 03/2017 AND 02/2018.

Clusters	AVCLASS		EUPHONY	
#App	# Empty-labeled	# percentage	#Empty-labeled	# percentage
9000	3066	34.06%	**1534**	17.04%

TABLE II
MALWARE WITH CONTROVERSIAL LABELS. THE TABLE GIVES THE NUMBER OF MALICIOUS APPS WHOSE TOP TWO FREQUENT FAMILIES REPORTED BY A CLOSE NUMBER OF VENDORS BASED ON PLURALITY VOTING USING EUPHONY. OF THE 7466 APPS WHICH HAVE FAMILY NAMES (EXCLUDING THE 1534 EMPTY LABELED APPS IN TABLE I), PLURALITY VOTING IS NOT CONFIDENT IN 1790 APPS SINCE THE TWO MOST FREQUENT NAMES ARE REPORTED BY AN EQUAL NUMBER OF VENDORS FOR EACH APP.

#Gap = # vendor reporting the most frequent name - # vendor reporting the second most frequent name.

#Gap	0	1	2	3	4	≥ 5	# Apps	# Vendor
# App No.	**1790**	1389	825	594	438	2430	7466	67

Existing Efforts. To address these issues, the recently proposed clustering approaches, such as AVCLASS [5] and EUPHONY [6], perform an automatic malware labeling based on the outputs from a collection of AntiVirus vendors in VIRUSTOTAL [7]. Due to the inconsistency among the raw labels reported by a wide variety of AV vendors, the existing approaches normally perform a pre-processing to extract the family names from the raw labels based on vendor-specific or self-defined heuristic rules [5, 6], including generic token removal and alias token reduction. Plurality voting [8] is then applied to disambiguate the inconsistent family names.

Limitations and Our Observations. These *raw-label-based* approaches rely heavily on the original labels from the AV vendors. A substantial number of *weakly-labeled malware* are unable to be clustered because their raw labels are often incomplete, inconsistent and overly generic, as reported by incompatible commercial vendors with different capabilities

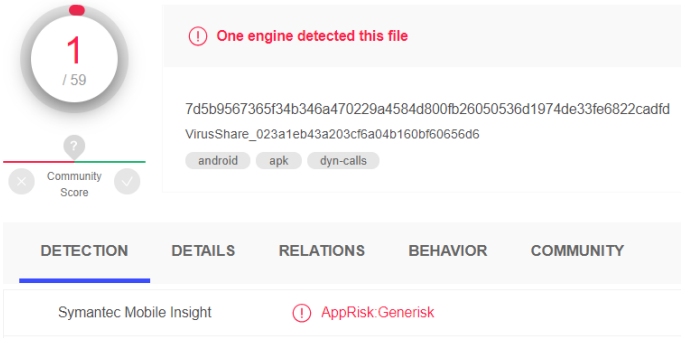

Fig. 1. An example of malware with empty label. Only one AV vendor (`Symantec Mobile Insight`) reports it as malware. The raw label `AppRisk.Generisk` given by this vendor is very generic. According to the rules of AVCLASS and EUPHONY, the raw label `AppRisk.Generisk` is parsed as two generic tokens `AppRisk` and `Generisk` with no actual family information, thus returning an empty label after their generic token removal phases.

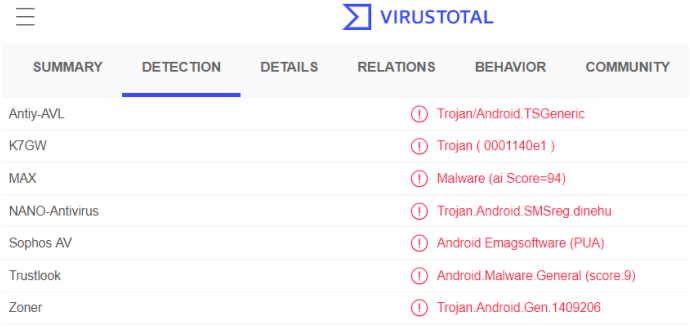

Fig. 2. An example of malware with controversial labels. There are seven AV engines report this app as malware. According to EUPHONY, five of them are generic labels (it becomes empty labels after generic token removal by EUPHONY). The remaining two AV engines label this malware as two different families, i.e., `dinehu` by `NANO-Antivirus` and `Emagsoftware` by `Sophos AV`. Thus, the family name of this app becomes controversial if the malware labeling is purely based on the raw labels produced by AV vendors.

in the presence of rapidly evolving malware.

We define two types of weakly-labeled malware based on the results from the state-of-the-art approaches [5, 6] with their statistics given in Tables I and II: (1) *Malware with empty label*. The tokens from the raw labels being recognized as generic tokens (e.g., `apprisk`) or unable to be parsed by the rules of AVCLASS and EUPHONY are discarded, resulting in weakly-labeled apps without a family name. Figure 1 illustrates an concrete example of malware with empty label. In addition, for example, as listed in Table I, AVCLASS and EUPHONY are unable to produce a family name for 34.06% and 17.04% of the total 9000 apps which were recently uploaded to VIRUSTOTAL between Mar. 2017 and Feb. 2018, resulting in **1534** weakly-labeled malware. (2) *Malware with controversial labels*. The plurality voting strategy is also hard to disambiguate inconsistency between two family names reported by a close number of vendors among the 7466 apps which have family names extracted from the raw labels by EUPHONY. In figure 2, an example of malware with controversial labels is given. Table II shows that the top two most frequent names are reported by an equal number of vendors for **1790** apps (24%), causing the plurality voting to be less confident for these weakly-labeled malware compared to their strongly-labeled counterpart with an uncontroversial family name reported by the majority (e.g., 80%) of all available vendors in VIRUSTOTAL [7].

Insights and challenges. A strongly-labeled malware with a clear and unambiguous family name is easy to be clustered. However, relying on vendors' raw labels as the only source of information for labeling weakly-labeled malware is inherently partial and shallow. Apart from the reports from AV vendors, an Android app itself is a comprehensive package containing source code and meta-info (e.g., configuration and resource files), which are crucial for extracting the behaviors of an app. This information cannot be easily obtained and is not often considered by the existing raw-label-based clustering approaches. This is because inferring the malware family by analyzing an app, in fact, is to develop another AV engine, competing with dozens of available AV vendors, which is hard and likely to cause inconsistent results due to lack of ground-truth family names and the unknown mechanisms of the commercial AV vendors, whose capabilities and mechanisms vary greatly with no intermediate detection results reflected in their reports. Thus, the raw labels generated from these vendors are the only easy-to-use and important data source for the existing clustering approaches.

Our solution. Deep representation learning (DRL) [9] is a new and promising branch of machine learning. DRL learns the representation of the target data via deep architectures in a layer-wise manner, through which the higher abstraction level of the features is embedded in a lower unified representation, making it easy for later classification and clustering tasks. Recently, Hybrid Representation Learning approaches [10–13] have significantly enhanced the existing DRL techniques in terms of both efficiency and accuracy for training and predicting large-scale networks by extracting heterogeneous information in a variety of formats (e.g., texts [14] or graphs [15]) and from multiple data sources.

Inspired by the recent advances in DRL (Deep Representation Learning), this paper proposes ANDRE, a new hybrid representation learning approach to clustering weakly-labeled malware by utilizing multiple sources of data (including raw labels from AV vendors, meta-info of apps and results from source code analysis). ANDRE jointly learns a hybrid representation that allows heterogeneous information to be integrated into one neural network pipeline that distills the discriminative features for accurate clustering. ANDRE is based on a new Android malware network, in which every node represents an app whose label contains its meta-info and the raw reports from AV vendors, and every edge denotes the similarity between two apps inferred by our static analysis which exploits a pairwise analysis of code similarity. Weakly-labeled malware in the network, together with existing strongly-labeled malware, are fed into our hybrid representation learning model to embed all the nodes on the network into a continuous and low-dimensional space that preserves comprehensive heterogeneous information.

The learned representation is then used for our outlier-aware clustering to partition the weakly-labeled malware into

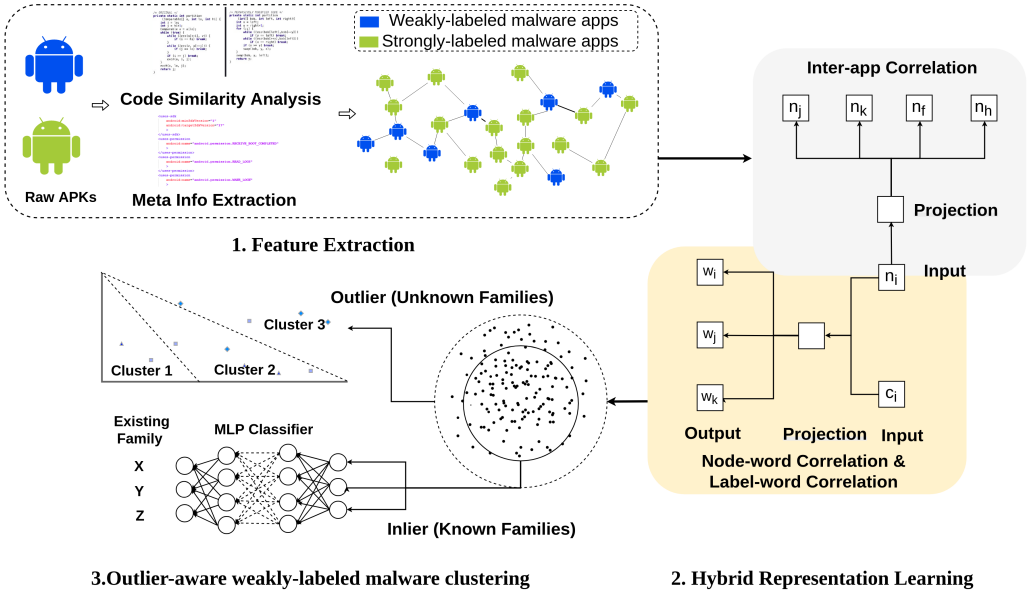

Fig. 3. The overview of ANDRE. n_i represents a single malicious app, and c_i denotes a known family name if n_i is a strongly-labeled. c_i is empty if n_i is weakly-labeled for prediction purposes. w_j denotes a word in the meta-info associated with a node.

known and unknown families. The malware whose malicious behaviours are close to the existing families on the network, are further classified using a three-layer Deep Neural Network (DNN). The unknown malware are clustered using a standard density-based clustering algorithm.

The key contributions of our paper are as follows:

- We present a new hybrid representation learning approach to cluster weakly-labeled Android malware.
- We propose a new representation by successfully preserving heterogeneous information, including raw labels from AV vendors, meta-info and results from source code analysis. This provides a compact and low-dimensional representation for effective Android malware clustering.
- We have conducted a comprehensive evaluation using 5,416 ground-truth samples from the Drebin dataset and 9,000 malware from VIRUSSHARE, uploaded between Mar. 2017 and Feb. 2018, consisting of 3324 weakly-labeled malware. The results show that our approach has comparable accuracy to the state-of-the-art approaches in clustering ground-truths, and can effectively cluster weakly-label malware which are unable to be clustered by AVCLASS and EUPHONY.

The rest of the paper is structured as follows. Section II defines the malware clustering problem and introduces the overall framework of ANDRE. Section III details our approach including feature extraction, network construction, representation learning and outlier-aware clustering.

II. PROBLEM DEFINITION AND FRAMEWORK OVERVIEW

A. Problem Definition

An Android malware network is represented as $G = (N, E, W, C)$, where the node set N = $\{n_1, n_2, \ldots, n_{|N|}\}$ denotes a set of Android malware (including both strongly- and weakly-labeled apps), and an edge $e_{i,j} = (n_i, n_j) \in E$ between two nodes encodes the code similarity between two apps. Each node n_i is associated with content information d_i consisting of a sequence of word tokens $w_j \in d_i$ extracted from the app's meta-info and the raw labels produced by AV vendors. We use $W = \{w_1, \ldots, w_{|W|}\}$ to denote the words of all nodes in this network.

Every node n_i has a label $l_i \in C = L \cup U$, where U denotes a set of labels with no family name for prediction purposes, and L are known family names of the strongly-labeled malware (either from ground-truths in Drebin or downloaded from VIRUSSHARE with each app's unique family name reported by over 80% of all available vendors). The numbers of ground-truth and weakly-labeled apps are configurable in the network. If the label set $L = \emptyset$, i.e., $C = U$, the representation learning becomes purely unsupervised, and our proposed solution is still valid but imprecise.

Our hybrid representation learning problem is formulated as maximizing an objective function $\mathcal{J}$ (Equation 1), which aims to jointly learn a k-dimensional vector $\mathbb{V}_{n_i} \in \mathbb{R}^k$ (k is a smaller number) for each node n_i in the network, such that nodes, which are neighbors based on the network topology or have similar meta-info or family names are close to one another in the latent embedding space.

$$\mathcal{J} = \alpha_1 \sum_{i=1}^{|N|}\sum_{j=1}^{|N|} \mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j}) + \alpha_2 \sum_{i=1}^{|N|}\sum_{p=1}^{|W|} \mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(\mathbb{V}_{n_i}, \mathbb{V}_{w_p}) + \alpha_3 \sum_{q=1}^{|L|}\sum_{p=1}^{|W|} \mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(\mathbb{V}_{w_p}, \mathbb{V}_{c_q}) \quad (1)$$

where $\mathbb{V}_{n_i}$, $\mathbb{V}_{w_p}$ and $\mathbb{V}_{c_q}$ denote the learned representation vectors of the three entities, i.e., node n_i, word w_p and family c_q in the representation space $\mathbb{R}^k$. $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}$, $\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}$ and $\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}$ are affinity functions to capture the correlation between two entities in $\mathbb{R}^k$. α_1, α_2 and α_3 are the weights that balance network structure, meta-info, and family information ($\alpha_1 + \alpha_2$

$+\alpha_3 = 1$).

B. Framework Overview

Figure 3 gives the overview of our framework, which consists of the following three major components.

1. Feature Extraction. There are two steps for extracting features. (1) Code similarity analysis. ANDRE implements a pairwise comparison scheme to dissect the similarities between apps. Two apps are provided as inputs and our method yields a similarity profile for each pair. The similarity profile summarizes similarity facts relating to similarity scores at file level, which means calculating similarity score for two source code files. The method builds an invertd-index to quickly calculate the similarity score of a pair of Android source code files. In addition, there is also a filtering heuristics to reduce the size of the index, which reduces the number of pairs needed to evaluate the similarity scores. A large portion of an Android app contains standard Android and safe third party libraries [16], which can be seen as noise in our representation learning. Following [16], we maintain a whitelist of common libraries to remove those library-related code segments from the application code of a malicious app in our code similarity analysis. Whitelist is a standard way for reducing noise in code similarity analysis. In addition, whitelist also provides a flexible and customized way for adding any new APIs which are safe.

(2) Analyzing the Android manifest files. We extract meta-info of each malware app from its manifest files, including package names, API versions, launcher activities, permissions, etc, which are associated with the corresponding node in the network. In addition, the raw labels from the existing AV vendors are also attached to each node in our network. Finally, all the ground-truths (strongly-labeled) malware are labeled with their unique family names.

2. Hybrid Representation Learning. An Android network G is constructed from the code similarity analysis, where each edge encodes the similarity between node n_i and n_j. The meta-info and raw labels associated with node n_i are represented by d_i. Our network G contains both weakly- and strongly-labeled (as the ground-truths) malware nodes with configurable portions of both kinds. The Hybrid (or multi-modal) Representation Learning (HRL) module [11, 17, 18] jointly embeds nodes N, words W from the meta-info, and family names L in a low-dimensional space, so that the affinity of heterogeneous information can be captured. The malware clustering can be accurately performed based on the comprehensive representation space via Equation 1, through which the following three relationships are preserved.

(1) *Inter-app correlation* $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j})$, which captures the relationship between two apps via a neural network based on the randomly generated sequences (random walks) from the network structure [19], where each sequence $s = n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_n$ can be seen as a phrase in natural language model. Given a node n_j in each random walk sequence s within a sliding window b, the neural network maximizes the log-likelihood of observing a set of neighboring nodes $\{n_{j-b}, n_{j-b+1}, \cdots, n_{j+b-1}, n_{j+b}\}$, so that the representation vectors $\mathbb{V}_{n_i}$ and $\mathbb{V}_{n_j}$ are close to each other if $e_{i,j} \in E$, i.e., $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j})$ is maximized.

(2) *Node-word correlation*, which maximizes the co-occurrence of a node and a word in the meta-info, i.e., maximizing the affinity value of $\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(\mathbb{V}_{n_i}, \mathbb{V}_{w_p})$ in the embedding space, modeling the fact that if a word w_p from the meta-info appears in node n_i, then the two vectors $\mathbb{V}_{n_i}$ and $\mathbb{V}_{w_p}$ are near in the representation space.

(3) *Label-word correlation*, which maximizes the co-occurrence of a family name and a word in the meta-info, i.e., maximizing $\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(\mathbb{V}_{w_p}, \mathbb{V}_{c_q})$, such that the label c_q and word w_p are forced to be closed to each other in the resulting representation space.

Lastly, the above three relations are jointly learned in a unified mode in an iterative manner. We further employ a hierarchical softmax modeling [20] to reduce the time complexity of our model, enabling ANDRE to scale to large malware datasets.

3. Outlier-aware weakly-labeled malware clustering. Given our learned hybrid representation space $\mathbb{R}^k$, this step performs prediction of a weakly-labeled app in the network G. When predicting its family name, this malware may fall into an existing family in L from the strongly-label malware in G or it may be an unknown type of malware with different behaviors which are different from any known families in L since the unknown malware types do exist and the ground-truth labels from strongly-labeled malware in the network may be limited. To address this issue, ANDRE performs an outlier-aware clustering that first employs outlier detection to partition all the weakly-labeled malware apps into (1) inlier apps whose behaviors are close to those of known families, and (2) anomalous apps that unlikely belong to any existing family in the network. Malware apps in the outlier set are clustered using a density-based algorithm, while apps in the inlier set are fed into a designed neural network to train the multi-classifier for classifying these malware into known families. The neural network consists of three layers, where the first two 1024-node layers comprise a dropout followed by a dense layer with a parametric rectified linear unit (ReLU) activation function in the first two layers and the sigmoid function in the third layer, which is also used for prediction.

III. Our Approach

This section introduces the detailed approaches of ANDRE, including Android malware feature extraction, hybrid representation learning and outlier-aware clustering for weakly-labeled malware.

A. Android Malware Feature Extraction

The aim of our feature extraction is to build the network via code similarity analysis between two nodes (apps) and associate each node with its meta-info extracted from its corresponding app.

Our source code analysis and meta-data extraction are complementary. Performing code similarity analysis on possible malicious code segments from two programs is helpful for grouping malware apps with similar runtime behaviors. The

meta-info of an Android app, including permissions, Android services, activities, providers, receivers, intent filters, security settings and referenced libraries are important building blocks for understanding about the special characteristics of an app. Combining code similarity and meta-info for constructing the network provides more comprehensive understanding of app relationships.

1) Code Similarity Analysis: Directly applying the existing code clone analysis, such as SOURCERCC [21] and DECKARD [22] on Android apps to obtain their similarity information is ineffective, because an Android app commonly contains a large portion of safe library code, either by invoking Android SDK APIs or third party libraries. The malicious code segments which reflect the malware type may be hidden in the application code. The safe (library) code segments may become noisy, resulting in inaccurate identification of code similarity between the two malware families. In addition, some malware apps are developed by repackaging an existing benign app [23]. By injecting two different types of malicious code into one benign app, the two repackaged apps can be of different malware families. Therefore, the benign code segments from Android SDK and third-library are the major noises, impeding analyzing code similarity to differentiate the two malware families.

Our code similarity analysis consists of four steps. First, all the apks are decompiled using `dex2jar` by converting their original DEX files to Java files for our source code analysis. Next, we perform a common library removal to reduce as many noise as possible, and our token-based code similarity analysis is performed to group apps based on file-level similarity. We then perform fine-grained code-block-level similarity analysis by considering existing available malicious payloads [16, 24] to refine the similarity scores between two apps. Lastly, an edge in the network is connected between two apps if their similarity score is above a pre-defined threshold.

Common library removal. Previous study [25] shows that over 60% of Android application code (in terms of low-level bytecode instructions after compilation) is contributed by library code. To address the noises introduced by a large portion of library code when conducting code similarity analysis, we follow [16, 26–28] to perform a common library removal by maintaining comprehensive whitelists of common libraries [29] to exclude library-related code segments prior to our similarity analysis.

Code Similarity. Our code similarity analysis applies a bag-of-words-based code clone approach [21], which has been shown to be the most efficient strategy and has a good precision for scaling to large code bases. Tokenization is first performed to remove comments, white space, and terminal. A token is extracted from each source file into Java keywords, literals, and identifiers. A string literal is split on whitespace and operators are not included. Tokens in known malicious payload code segments [16, 24] are given higher weights for code similarity analysis. The source code of an Android app n is represented as a set of code blocks (basic blocks of the control-flow graph of a program) $Source(n) = \{B_1, ..., B_{num}\}$ with each block B_i denoting a bag-of-tokens $B_i = \{T_1..., T_k\}$. One token may appear multiple times in a block, therefore each token is qualified with its occurrence frequency inside a block, $T_j = (token, frequency)$ to differentiate between the frequency of words in the bag-of-words model. Formally, given two apk files A_x and A_y, a similarity function f, and a threshold θ, the aim is to find all the code block pairs A_x.B and A_y.B s.t $f(A_x.B, A_y.B) \geq [\theta \cdot max(|A_x.B|, |A_y.B|)]$.

There are several choices of similarity function for measuring the similarity between two code pieces, we use the Jaccard index, or Jaccard similarity, which is defined as the size of intersection divided by the size of the union of two sets. The similarity of two code blocks B_x and B_y is defined as follows:

$$J(B_x, B_y) = \frac{|B_x \cap B_y|}{|B_x \cup B_y|} = \frac{|B_x \cap B_y|}{|B_x| + |B_y| - |B_x \cap B_y|} \quad (2)$$

2) Meta-Info Extraction: Our meta-info is mainly extracted from `AndroidManifest.xml`, which is the key file at the root of an Android project. It provides all the essential information of an app to the building tools, Android OS, and app stores. Once an app is launched, `AndroidManifest.xml` is the first file to be consulted by the Android system, providing the first-hand information to understand the characteristics and security settings of the app.

The detailed content of a manifest file is illustrated in Table III, including (1) `permissions`, which are responsible for protecting the application from accessing any protected parts, (2) `instrumentation` classes, which provide profiling and other dynamic monitoring information, (3) `application-level Android APIs` that an app is going to use, (4) four types of Android components: `activity`, `service`, `content provider`, and `broadcast receiver`. The names of an app's components may help identify the known malware components. (5) `hardware component`, which is helpful to identify malicious behaviors reflected by access requests to specific device components, e.g., touchscreen, camera, or sensors. (6) `intent and intent filter`, which can be used to trigger malicious activities, thus it is also necessary to be collected, and (7) `package name, version, referenced libraries`.

*B. Android Network Representation Learning (*ANDRE*)*

This section presents our malware representation learning which jointly exploits the network structure, the meta-info, and vendors' labels to embed heterogeneous information into a latent feature space for each node in the network.

The process of ANDRE consists of two steps:

- Network node (app) sequence generation with random walks. This step randomly generates a set of walks on the Android malware network, where each sequence starts with a node n_i and randomly jumps to other nodes at each iteration. The random walk sequences capture the topological relationships between nodes.
- Neural network architectures. This step encodes each node (app) into a vector representation by preserving multi-source information from (1) an android sequence which captures the inter-app correlation, (2) the meta-info which represents the node-word correlation, and

TABLE III
META-DATA INFORMATION EXTRACTED FROM AN ANDROID APP

Type	Description
Permission	Permissions constitute an important security feature in Android apps. A user has to grant them to install applications or access particularly sensitive data.
Instrumentation	Declares the code used to test this package or other package directive components. A manifest can contain zero or more of this element.
Application	Contains the root node of the application-level component declaration in the package and contains global and default properties in the application, such as tags, icons, themes, necessary permissions, and more.
Activity	Activity is the primary Android component used to interact with users.
Service	A service is a component that can run any time in the background.
Content provider	A content provider is a component that is used to manage persistent data and publish it to other applications.
Broadcast receiver	The receiver enables the application to obtain data changes or operations that occur even if it is not running.
Intent/intent-filter	The IntentFilter is formed by declaring the Intent values supported by the specified set of components.
Action	The intent action supported by the component.
Category	Intent Category supported by the component.
Type	Intent data MIME type supported by the component.
Schema	Intent data URI scheme supported by the component.
Authority	Intent data URI authority supported by the component.
Path	Intent data URI path supported by the component.

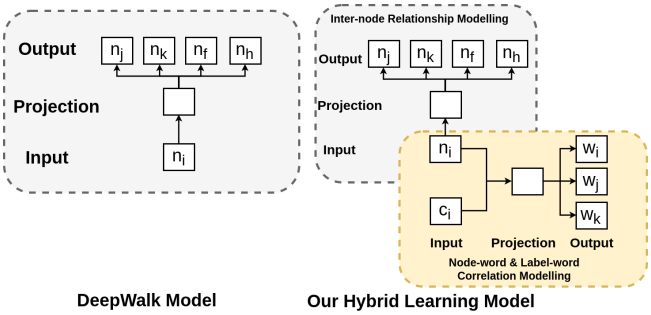

Fig. 4. The DEEPWALK (Skip-Gram) method vs our proposed hybrid learning method. The DEEPWALK approach learns the network representation based only on the network structure. Our hybrid method couples two neural networks to learn the representation from three parties (i.e., node structure, meta-info, and label information from strongly-labeled malware) to capture the inter-node, node-word, and label-word relationships. The input, projection, and output indicate the input layer, hidden layer, and output layer of a neural network model.

(3) the label information which records the label-word correlation. Lastly, a hybrid representation model is built by jointly learning the above three correlations.

1) Model Architecture: The proposed joint neural network model is illustrated on the right of Figure 4. It has key properties:

Inter-node Correlation. Assuming that neighbor nodes are highly correlated and statistically dependent on each other, the upper layer of ANDRE exploits the network structure from a set of generated random walk sequences.

Node-word correlations. The lower layer of ANDRE captures the relations between nodes $n_i \in N$ and the meta-info, which is considered as a document with a set of words $w_j \in W$. If a word is used to describe a node, the node and the word are correlated.

Label-word Correlation. We use the labels $c_i \in C$ from strongly-labeled malware (a ground-truth) and its corresponding node $n_i \in N$ as input and learn the label vector and word vector simultaneously, so that the label-word correlation can be well captured to improve the learning model.

Joint Training Model. We integrate these two layers by the node n_i in the model. The three kinds of correlations are jointly learned in a unified way, so that they can benefit each other and ultimately converge to a steady stage.

2) Model Details: ***Inter-node Correlation.*** To capture the inter-node correlation, we assume that the nodes with linkages (edges) in a network are statistically dependent on each other. This idea is inspired by DEEPWALK [19], which extends the SKIP-GRAM [30] algorithm.

The SKIP-GRAM model [30] is a language model that learns the word embedding by exploiting word orders in a sequence and assuming that words close to one another are statistically more related. Due to its simplicity and efficiency, The SKIP-GRAM model is widely used in many NLP tasks. Given a word w_m, Skip-Gram maximizes the log-likelihood of w_m's surrounding words within a certain window b.

$$\mathcal{J}_1 = \sum_{m=1}^{T} \log \mathbb{P}(w_{m-b} : w_{m+b} | w_m) \tag{3}$$

where b is the window size and $w_{m-b} : w_{m+b}$ is a word sequence in a window, which excludes the word w_m itself. The probability $\mathbb{P}(w_{m-b} : w_{m+b} | w_m)$ is defined as:

$$\prod_{-b \le j \le b, j \ne 0} \mathbb{P}(w_{m+j} | w_m) \tag{4}$$

Equation (4) makes the assumption that the words in a window $w_{m-b} : w_{m+b}$ are independent of each other. $\mathbb{P}(w_{m+j} | w_m)$ is defined as:

$$\mathbb{P}(w_{m+j} | w_m) = \frac{exp(\mathbb{V}_{w_m}^{\top} \mathbb{V}'_{w_{m+j}})}{\sum_{w=1}^{W} exp(\mathbb{V}_{w_m}^{\top} \mathbb{V}'_{w})} \tag{5}$$

where $\mathbb{V}_w$ and $\mathbb{V}'_w$ are the *input vector* and *output vector* of w. Once the training is finished, the *input vector* $\mathbb{V}_w$ is used as the final representation of word w.

DEEPWALK [19] extends SKIP-GRAM and employs the neural language model to learn the embedding for a network. Specifically, DEEPWALK generates a set of random walk sequences S based on the network structure. Each random walk $s = n_{i-b} \rightarrow ... \ n_i \ ... \rightarrow n_{i+b}$ is regarded as a sentence with a window of size b in the natural language model, and each node n_i is considered as a word. Then DEEPWALK employs the SKIP-GRAM algorithm [30] on the node sequences to obtain a latent representation for each node. This is achieved by solving an objective function which maximizes the log-likelihood of the observed nodes on a target node n_i for all random sequences $s \in S$.

$$\begin{aligned}\mathcal{J}_2 &= \sum_{i=1}^{|N|}\sum_{s\in S}\log \mathbb{P}(n_{i-b}:n_{i+b}|n_i) \\ &= \sum_{i=1}^{N}\sum_{s\in S}\sum_{-b\leq j\leq b, j\neq 0}\log \mathbb{P}(n_{i+j}|n_i)\end{aligned} \tag{6}$$

The DEEPWALK neural network is illustrated on the left of Figure 4. It exploits the network structure for learning representations without considering other information (such as meta-info and label-info), which is exploited by our model.

Node-word Correlation. To enable joint learning together with inter-node correlations, ANDRE captures the node-word correlations as depicted in the right lower panel in Figure 4, which applies a DOC2VEC style model built on top of SKIP-GRAM and CBOW (Continuous Bag-Of-Words) [30], with the aim of learning the distributed representation for a document. In our setting, for a node n_i, we learn the input node vector $\mathbb{V}_{n_i}$ and output word vector $\mathbb{V}'_{w_j}$ based on meta-info $d_i = [w_0, w_1, \ldots, w_{|d_i|}]$ associated with n_i using a sliding window of size b for repeatedly picking a sequence of words centering w_j within d_i.

Label-word Correlation We enable semi-supervising by leveraging uncontroversial family names from strongly-labeled malware, benefiting from the knowledge of existing AV vendors. By applying DOC2VEC, we use the ground-truth label information together with the meta-info as inputs and simultaneously learn the input label vector $\mathbb{V}_{c_i}$ of node n_i and output word vector $\mathbb{V}'_{w_j}$ based on the meta-info associated with n_i, modeling the correlation between the nodes' labels and the nodes' meta-info.

Since the first correlation is captured via DEEPWALK and the last two correlations are modeled via DOC2VEC, it can intuitively be seen that our hybrid learning couples these two modelings using two panels, as illustrated in Figure 4. The node n_i shared by both panels indicates that n_i is influenced by the two models to produce a hybrid representation which preserves the heterogeneous information and the relations between the three parties (e.g., node sequences from random walks, word sequences from meta-info, and label information).

Joint Learning Model. Given a network G consisting of nodes N = $\{n_1, n_2, \ldots, n_{|N|}\}$, our hybrid learning model in Equation 7 implements the pre-defined objective function in Equation 1 by considering the three aforementioned correlations. The aim is to jointly learn the following three affinity functions with random walks S generated for node n_i, and a sliding window for a sequence of nodes or words.

$$\begin{aligned}\mathcal{J} = (1-\alpha)\sum_{i=1}^{|N|}\sum_{s\in S}\sum_{-b\leq j\leq b, j\neq 0}\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(n_{i+j}, n_i) \\ + \alpha\sum_{i=1}^{|N|}\sum_{-b\leq j\leq b}\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(w_j, n_i) + \alpha\sum_{i=1}^{|L|}\sum_{-b\leq j\leq b}\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(w_j, c_i)\end{aligned} \tag{7}$$

where α is the weight that balances the network structure, meta-info, and label information. b is the window size of a node or a word sequence, and w_j indicates the j-th word in a contextual window. Given Equation 7, the first term computes the affinity function $\mathcal{A}(n_{i+j}, n_i)$, and the log-likelihood probability of observing surrounding nodes given node n_i, using the log-likelihood softmax functions as Equation 8:

$$\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(n_{i+j}, n_i) = \log \mathbb{P}(n_{i+j}|n_i) = \log \frac{exp(\mathbb{V}_{n_i}^\top \mathbb{V}'_{n_{i+j}})}{\sum_{x=1}^{|N|} exp(\mathbb{V}_{n_i}^\top \mathbb{V}'_{n_x})} \tag{8}$$

where $\mathbb{V}_{n_x}$ and $\mathbb{V}'_{n_x}$ are the input and output vector representations of node n_x. $|N|$ is the number of the nodes in the network. The probability of observing contextual words $w_{i-b} : w_{i+b}$ given current node n_i is:

$$\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(w_j, n_i) = \log \mathbb{P}(w_j|n_i) = \log \frac{exp(\mathbb{V}_{n_i}^\top \mathbb{V}'_{w_j})}{\sum_{x=1}^{|W|} exp(\mathbb{V}_{n_i}^\top \mathbb{V}'_{w_x})} \tag{9}$$

where $\mathbb{V}'_{w_j}$ is the output representation of word w_j, and $|W|$ is the number of distinct words on the whole network. Similarly, the probability of observing the words given a class label c_i is then defined as:

$$\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(w_j, c_i) = \log \mathbb{P}(w_j|c_i) = \log \frac{exp(\mathbb{V}_{c_i}^\top \mathbb{V}'_{w_j})}{\sum_{x=1}^{|W|} exp(\mathbb{V}_{c_i}^\top \mathbb{V}'_{w_x})} \tag{10}$$

Equations 9 and 10 reflect the correlation between nodes and meta-info, and the correlation between meta-info and label information, so that ANDRE jointly learns the output representation vector $\mathbb{V}'_{w_j}$ of word w_j, which will propagate back to influence the input representation of $n_i \in N$ in the network as also illustrated in Figure 4. As a result, the node representation (i.e., the input vectors of nodes) is enhanced by both the meta-info and the label information.

3) Training Hybrid Learning Model: We train our model Equation 7 using stochastic gradient ascent, which is a standard solution for training. However, computing the gradients in Equations 8, 9 and 10 is expensive, as the computation is proportional to the number of nodes and words in the network G. To address this issue, we resort to hierarchical softmax [11, 20], which reduces the time complexity to $O(|W|log(W) + Nlog(N))$.

The hierarchical model in our algorithm uses two binary trees, one with distinct nodes as leaves, and another with distinct words and labels as leaves. The trees are built using HUFFMAN algorithm, so that each vertex in a tree has a binary

code, in which more frequent nodes (or words) have shorter codes. There is a unique path from the root to each leaf in a tree. The interval vertices of the trees are represented as real-valued vectors with the same dimension as the leaves. Instead of enumerating all nodes in Equation 7 in each gradient step, we only need to evaluate the path from the root to the corresponding leaf in the Huffman tree. Suppose the path to the leaf node n_i is a sequence of vertices $(l_0, l_1, ..., l_h)$ reaching n_i, where l_h is n_i and l_0 is the root node in the Huffman tree. The probability is computed as follows:

$$\mathbb{P}(n_{i+j}|n_i) = \prod_{t=1}^{h} \mathbb{P}(l_t|n_i) \tag{11}$$

$$\mathbb{P}(l_t|n_i) = \sigma(\mathbb{V}_{n_i}^{\top} \mathbb{V}_{l_t}^{'}) \tag{12}$$

where P($l_t|n_i$) is a binary classifier with $\sigma(.)$ as the sigmoid function, and $\mathbb{V}_{l_t}^{'}$ is the representation of node l_t, which is the parent of n_i in the Huffman tree. Thus, the time complexity is reduced to $O(NlogN)$. Likewise, we can use hierarchical softmax technique [14] to compute the words and labels in Equations 9 and 10.

C. Outlier-aware Android Malware Clustering

Outlier detection (a.k.a. anomaly detection [31]) detects rare samples that do not meet the expected patterns or behaviors of the majority of samples in the dataset. The technique is essentially a density-based outlier detection algorithm that constructs a graph of the data using nearest neighbors, instead of calculating local densities. A malware sample whose malicious behaviours are closed to those of known families in the network G will be classified into an existing family in L, otherwise, it will be identified as an unknown type of malware. This is due to the fact that the unknown malware types do exist and the number of ground-truth labels from strongly-labeled malware in the network may always be limited.

Finally, the outlier set of the collected malware apps is clustered to produce unknown family clusters, while the inlier set is fed into the designed Multi-layer Perceptron (MLP) classifier (Supervised Neural Network) to train the multi-classification model, thereby classifying these malware into known families.

IV. Evaluation

The objective of our evaluation is to show that ANDRE is effective in clustering weakly-labeled malware that cannot be clustered by AVCLASS and EUPHONY, while achieving comparable accuracy with the state-of-the-art tools evaluated using the ground-truth samples in the Drebin dataset.

A. Experimental Setup and Implementation

To evaluate the effectiveness of ANDRE, 5,416 ground-truth malware samples from the Drebin dataset [32] and 9,000 recent malware from VIRUSSHARE [33] (uploaded between March 2017 and October 2018) were used. The Android apks were first decompiled to Java files using `dex2jar` for our source code analysis. Note that there are 5560 malware samples in the Drebin dataset, but only 5416 samples were used since the remaining 144 samples could not be correctly decompiled by `dex2jar`. Out of the 9,000 recently collected malware from VIRUSSHARE, 4314 are smaller than 5MB, 1969 are between 5MB and 10MB, and 2717 are greater than 10MB.

To build the edge relations between nodes (apps) of the network, we adopted the bag-of-words model [21] to perform code similarity analysis for all pairs of malware apps after excluding their library-related code, following [16] by using a self-maintained whitelist containing common third-party libraries [29] and the available malicious payload [16, 24]. An edge is connected between two nodes if their similarity score is above 0.8 (with the maximum score 1 using Jaccard similarity in Equation 2 after normalization).

For each app in the network, we use the ANDROID ASSET PACKAGING TOOL [34] to extract meta-info of an app. The raw labels reported by Anti-Virus vendors were obtained by uploading an app or its hash value to VIRUSTOTAL [7], an online malware scanning service which integrates a set of existing commercial Antivirus vendors, such as `Comodo` and `Kaspersky`. Given VIRUSTOTAL's reports, we can identify weakly-labeled malware, including malware with no label (1534) and malware with controversial family names (1790) by using EUPHONY. The results are shown in Table I and Table II.

We implemented DEEPWALK, DOC2VEC and our hybrid learning models to embed the constructed network using Python. Our outlier detection approach was implemented based on TOPOLOGICAL ANOMALY DETECTION (TAD) [35]. To classify the inlier known families, we applied and compared different classification algorithms, including the neural network classifier MULTI-LAYER PERCEPTRON (MLP) [36] and traditional classification algorithms SVM [37], KNN [38], DSTREE [39] and RDFOREST [40] whose implementations are available from Scikit-learn library [41].

B. Evaluation Methodology

We evaluate the effectiveness of our approach from the following four aspects: (1) comparing ANDRE with the state-of-the-art tools using Drebin's ground-truth samples (a typical dataset widely used by existing clustering tools) to show the effectiveness and applicability of our representation learning approach for clustering all types of malware (not limited to weakly-labeled malware) with good precision, (2) comparing ANDRE against different baseline settings (i.e., different representation learning models including DEEPWALK and DOC2VEC) to show whether our hybrid learning model can obtain a promising level of accuracy by preserving heterogeneous information, (3) comparing different classification models given the learned representation, and (4) validating our outlier-aware malware clustering results with comprehensive case studies to show ANDRE's effectiveness in clustering weakly-labeled malware.

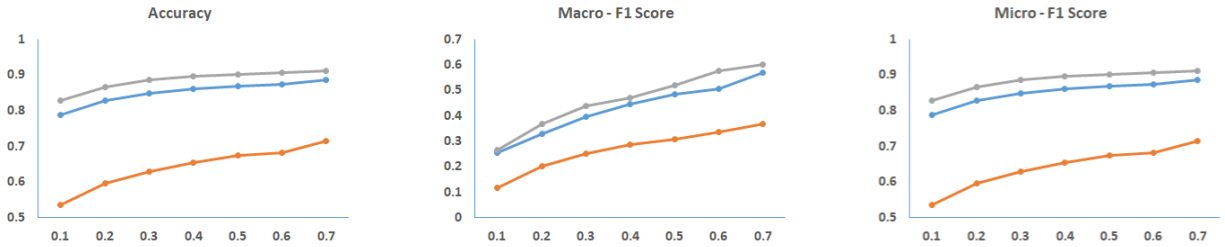

Fig. 5. Performance comparison with respect to different learning models. ● represents the DEEPWALK method, ● represents the DOC2VEC method, ● represents our ANDRE method. In the three subgraphs, the horizontal axis represents the ratio of the training set, which is 10% to 70%. The vertical axes in the three subgraphs represent the fractions of accuracy, macro and micro score respectively.

TABLE IV
COMPARISON WITH AVCLASS [5] AND EUPHONY [6]. THE ACCURACY, F1 AND RECALL DATA ARE DIRECTLY FROM THEIR PAPERS

Method	Accuracy	F1 Score	Recall
AVCLASS	95.2%	93.9%	92.5%
EUPHONY	95.0%	95.5%	96.1%
ANDRE	93.1%	93.3%	93.1%

TABLE V
COMPARISON WITH DIFFERENT CLASSIFIERS

Classifiers	Accuracy	F1 Score	Speed(Seconds)
KNN	0.8677	0.4468	225
SVM	0.8295	0.2580	256
Random Forest	0.4911	0.0491	224
Decision Tree	0.4031	0.0357	230
MLP Classifier	0.9126	0.6079	356

C. Comparing with the state-of-the-art tools and different baseline settings

Comparing with non-learning approaches. We compare ANDRE with the two state-of-the-art tools AVCLASS and EUPHONY by using ground-truth samples from Drebin to show ANDRE's applicability and effectiveness in clustering malware which are not weakly-labeled. Our hybrid representation learning approach randomly selects 70% of malware as the training set and 30% of samples in the dataset for testing. Table IV shows that ANDRE achieves 93.1% for accuracy, 93.3% for F1 score and 93.1% for recall, which are comparable or slightly less than the results of the two tools (reported in their papers [5] and [6]).

Note that we do not claim that our learning-based approach is superior to the non-machine-learning methods in clustering malware apps that are not weakly-labeled. This is because plurality voting does work well for strongly-labeled samples with uncontroversial raw labels from AV vendors after label preprocessing, such as generic token removal and alias detection [5]. Rather, our aim is to show that our approach successfully preserves necessary information for effective clustering and can achieve comparable accuracy with the existing tools for ground-truth samples. Due to the nature of the learning-based algorithm, its performance depends on several factors, such as ratio selection of the training and testing sets, clustering algorithm, network structure (from the code similarity analysis) and feature attentions (importance of different meta-information). The following subsections conduct comprehensive evaluations and discussions of these factors.

Comparing with different representation learning models. To demonstrate that our hybrid network representation learning is able to achieve better performance, we compare ANDRE with the mainstream learning methods, i.e., DOC2VEC [14] (a paragraph vectors algorithm for embedding texts and phrases in a distributed vector using neural networks) and DEEPWALK [19] (which applies language modeling to capture the topological relationships between nodes in a social network).

Figure 5 shows our comparison results of the three approaches. By preserving the network structure (node-node correlation) and the co-occurrence relations between apps and their corresponding meta-info (node-word and label-word correlations), our approach performs better than the individual learning methods, i.e., DEEPWALK and DOC2VEC. In general, the accuracy values and F1 scores of the three approaches increase when the sizes of the training sets increase. The trend becomes stable and gradually reaches a convergence when the training set occupies around 70% of the total samples. ANDRE achieves up to 93% in accuracy, while DEEPWALK and DOC2VEC can only achieve 89.5% and 70.1% respectively.

Comparisons using different multi-class classifiers. Given the learned representation, we compare the classification performance when applying MULTI-LAYER PERCEPTRON (MLP), a deep neural network classifier and conventional classification algorithms, including SVM (separating hyperplane in the feature space to maximize the interval between positive and negative samples), KNN (distance measurement between different eigenvalues), RANDOM FOREST (integrating multiple trees through Ensemble Learning) and DECISION TREE (mapping relationships between object attributes and object values through a tree structure).

To fairly compare the results of different classifiers, we keep their underlying network embeddings unchanged (i.e., by associating all meta-info for each node, using the same network structure and same representation space with the dimension K=400). As shown in Table V, the accuracy of MLP classifier exceeds that of all other methods, demonstrating the recent advances in deep learning that enable the precise capture of input-output data relations correlations, i.e., the input features are connected to the neurons of the hidden layer, and the neurons of the hidden layer are then linked with the

TABLE VI
PERFORMANCE COMPARISON REGARDING DIFFERENT K, I.E., THE NUMBER OF DIMENSIONS IN THE REPRESENTATION SPACE.

Number of K	Accuracy	Macro-F1 Score	Micro-F1 Score
5	0.7230	0.2360	0.7230
10	0.8505	0.4661	0.8505
30	0.9022	0.5851	0.9022
50	0.9083	0.6115	0.9083
100	0.9151	0.6027	0.9151
200	0.9138	0.6189	0.9138
300	0.9169	0.5927	0.9169
400	0.9175	0.6282	0.9157

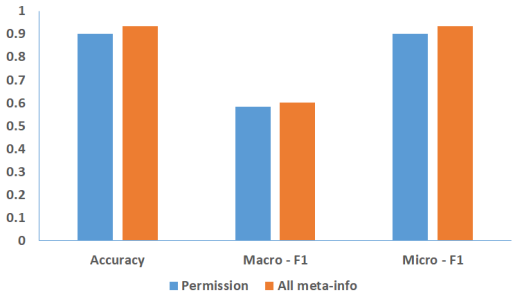

Fig. 6. Performance comparison (permissions vs all meta-info)

neurons of the input layer. The multi-layer perceptron layer is fully associated (the full connection means that any single neuron in the upper layer is connected to all neurons in the next layer).

Comparisons when selecting K-dimensional representation space. Table VI shows the results when the number of representation dimensions K are varied from 5 to 400 in our hybrid representation learning. There are noticeable increases for accuracy, macro-F1 score and micro-F1 score when K is increased from 5 to 50. The subsequent differences are negligible especially when K is greater than 100.

All meta-info vs. permission information. In this experiment, we choose the representation dimension K = 400 for our MLP classifier. This experiment aims to show that of all the meta-info components, permission information makes the major contribution to the accuracy of our clustering results. All other components also contribute to the improvement in accuracy. As illustrated in Figure IV-C, the mean accuracy is 90% when only permission strings were used for our network embedding. The accuracy increases to 93.4% when all elements of meta-info are used. The improvement shows that the complete meta-info includes not only permission information, but also many other types of useful information, such as activity, intent, service, etc., which represents much richer text information for capturing the correlations of apps whose behaviors are similar.

D. Result analysis for weakly-labeled malware

When clustering the weakly-labeled malware from VIRUSSHARE, we apply our outlier-aware clustering on top of the hybrid representation learned from the constructed malware network, which consists of 5676 strongly-labeled and 3324 weakly-labeled samples.

Our outlier detection identifies that 3242 malware apps are close to existing malware families, falling into the inlier zone, and 82 apps have behaviors that are unlikely to correspond to those of the known families, thereby falling into the outlier zone.

TABLE VII
RESULTS OF OUTLIERS AND INLIERS OF WEAKLY-LABELED MALWARE INCLUDING MALWARE WITH EMPTY LABELS AND CONTROVERSIAL LABELS (I.E., THE TOP TWO MOST FREQUENT FAMILY NAMES REPORTED BY AN EQUAL NUMBER OF VENDORS USING EUPHONY).

#Weakly-labeled malware	#Malware in outlier		#Malware in inlier	
	Empty-labels	Controversial-labels	Empty-labels	Controversial-labels
# 3324	35	47	1499	1743

Table VII gives the inliers and outliers of the 3324 weakly-labeled malware. The number of outlier samples is relatively small, comprising only 2.5%. A major portion of the weakly-labeled malware are detected as inliers. For the 1534 empty-labeled malware (Table I), 35 and 1449 are outliers and inliers respectively. Of the 1790 malware (Table II) who have controversial family names (i.e., the top two most frequent family names reported by an equal number of vendors using EUPHONY), 47 and 1743 are outliers and inliers respectively.

Figure 7 shows the outlier results with 58 outlier (unknown) families clustered by anomaly detection. 47 out of 58 outliers contain only 1 malware sample, and the remaining 11 include more than 1 candidate. The largest outlier cluster contains 9 samples.

For inliers, there are 60 families out of the 176 known families. The distribution is uneven. The malware apps in the top 10 families occupy the majority (88.26%) of all the weakly-labeled malware. Regarding malware in the inlier, we observe that our method is not limited by the number of malware samples in the training set. For example, the `plankton` family ranks first in terms of the prediction result in the Drebin dataset, whereas it only ranks as the third largest family. Similarly, `Vdloader` ranks fifth according to our experiment result, while it only ranks 167th in the Drebin dataset with only 16 samples inside.

Due to the lack of ground-truths for weakly-labeled malware, manual checking is the best and only way to validate the results [24]. Following [16], we have also conducted manual inspections (costing two people for four weeks) of the malware inside the same cluster to check the similarity of their malicious behaviors (by running and looking into the source code and meta-info of the apps) to validate ANDRE's clustering results. All the malware in outliers were checked since all the outlier clusters have very small sizes. For inlier families, we randomly checked 3 apps for each family if the family contains more than 3 apps, otherwise, all the apps within the family were checked. This process includes running the app (in a virtual Android environment) and checking the meta-info, vendors' raw labels and malicious code (decompiled by `dex2jar`) to understand their familial behaviors. We will select and illustrate in detail the representative apps within 3 different clusters in the inlier set in the following subsection.

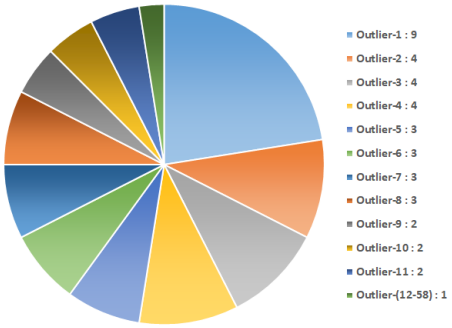

Fig. 7. Weakly-labeled malware detected as outliers (unknown families in the network). There are 58 outlier clusters; their corresponding malware numbers listed on the right-hand side.

E. Case studies for weakly labeled malware in inlier

This section conducts in-depth case studies to demonstrate representative inlier weakly-labeled malware clustered by ANDRE. Since every malware has a unique hash value (MD5), we will refer to their hash values in the following studies.

Inlier case Study 1 - YzchBgserv family. Malware "327a0387a3114ab4e4d18a81ad9fca8b", which is a malware with an empty label being processed by EUPHONY, is clustered into the YzchBgserv family by ANDRE based on our hybrid representation. The app has similar malicious behaviors to other apps in YzchBgserv, which is essentially a type of ransomware. The app runs as a background thread which is automatically triggered when the app starts. It first fetches a target service provider number (component info) from a remote server and then sends an SMS message (permission info) to the number, incurring a charge on the user's phone bill. The SMS messages (code info) sent will always start with a string "YZHC".

Inlier case study 2 – DroidKungFu family. Malware "00307253cd9ad3d1120df85beb473801" which is unable to be clustered by EUPHONY, is clustered into the DroidKungFu family by ANDRE. We manually checked the app whereas observing that such malware infects a self-defined service and receiver (code-info), which can be automatically launched without user interaction. Once the service gets started, it will collect three kinds of user information on the infected device, including the IMEI number, phone model, and the Android OS version (meta-info). These behaviors are typical symptoms of being infected by the Droidkungfu family.

Inlier case Study 3 – Fakeupdates family. Malware "005054fdf485fdbbada461bd7824cfb2", whose top two families are reported by an equal number of vendors ("fakeupdates":1, "igexin":1). After manual validation, we established that the `plankton.device.android.service` package is its source of malice. By looking at several other peers within the same cluster, we find that they all share this package (code-info) with an identical code segment to start the malicious service (i.e., AndroidMDKService. initMDK ()).

F. Case study for weakly-labeled malware in outlier

We also performed a manual inspection of outliers whose malicious behaviours are different from the known families in the network. As per our discussions in Section III-C, the results of our outlier detection often rely on the availability of the existing known families in the network, which may not cover all emerging malware families. The following case studies report the outlier clusters, inside which an app does not conduct behaviors that are similar to those of the existing ground-truths (e.g., Drebin).

Outlier-1 case study. This cluster, containing 9 malware, is the largest of all outlier clusters. Our manual inspection reveals that the apps in this cluster share the same risky permission combinations (9 suspicious permissions in total, as shown in Table VIII) to trigger similar malicious behaviors, which induce users to click an inbound link address to install a trojan package that modifies the app's configuration files. For example, the malware candidate "ee93c3eefb81151eca7346db74d613c" has the top two family names "autoins" and "letang" both reported by 2 vendors using EUPHONY. These two family names are new and do not appear in our ground-truths.

Outlier-2 case study. This cluster contains 4 malware sharing similar malicious behaviors as triada [42], a recent malware family, which does not belong to any of the existing families in the network. The malware in this cluster first collects sensitive information including Android OS versions and the list of installed applications. Then it sends this information to the command and control (C&C) server [43].

Outlier-3 case study. This cluster also contains 4 malware in total. The malice of the malware in this cluster starts from their in-app advertisements. In particular, when a user clicks a pre-defined advertisement link, the malicious program will display as many ads as possible to the user. These malware also have a risky combination of permissions using CHANGE_WIFI_STATE and CHANGE_NETWORK_STATE whose intention is to switch the Wifi and network status respectively after deriving their current status.

V. RELATED WORK

Android malware detection. A number of solutions to Android malware detection [26, 32, 44–46] exist that gives a binary decision to identify whether or not an app is malicious. DREBIN [32] proposes a malware detection approach through a two-class SVM by performing a lightweight static analysis

TABLE VIII
SUSPICIOUS PERMISSIONS USED BY APPS IN OUTLIER-1

Permission Strings
ACCESS_COARSE_LOCATION
ACCESS_FINE_LOCATION
ACCESS_FINE_LOCATION
INTERNET
READ_PHONE_STATE
SYSTEM_ALERT_WINDOW
WRITE_EXTERNAL_STORAGE
ACCESS_DOWNLOAD_MANAGER
DOWNLOAD_WITHOUT_NOTIFICATION
WRITE_SETTINGS

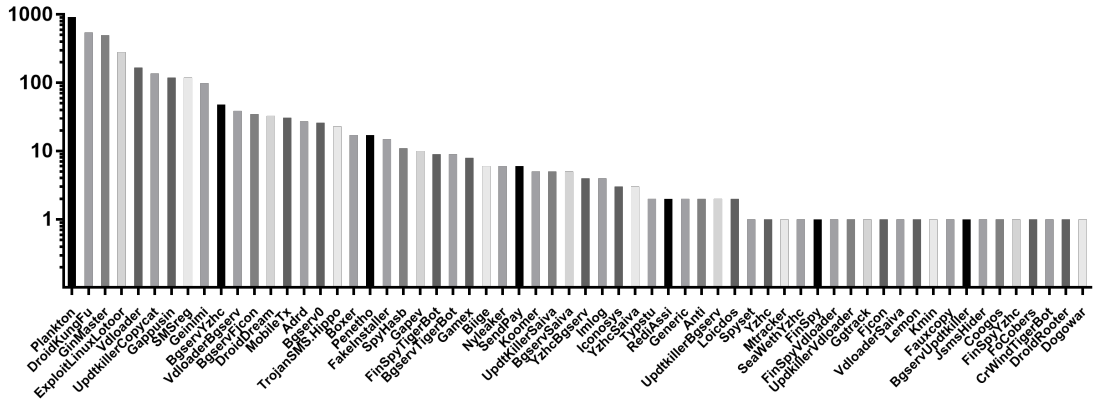

Fig. 8. Weakly-labeled malware detected as inliers (known families in the network). The horizontal axis in the figure shows the family names of all malware in the inlier clusters. The vertical axis indicates the log value of the number of apps in the corresponding families.

to extract API calls and manifest files as the input features. MaMaDroid [47] leverages sequences of abstracted method calls to create a probabilistic representation of program behaviors in the form of Markov chains. DroidAPIMiner [44] mines API features for malware detection in Android using a lightweight KNN-based binary classification. HINDroid [48] proposes an Android malware detection approach using heterogeneous information network. Kim et al. [49] present a deep-learning-based approach to malware detection by utilizing the features extracted from an Android app. Unlike malware detection, malware clustering or labeling (such as ANDRE) performs fine-grained familial identification, which in turn can be used to build references datasets for supervised detection and classification [5].

Android malware classification and clustering. DroidMiner [50] proposes a two-level behavioral graph model and extracts sensitive paths to represent malicious behavioral patterns for malware classification. DroidSIFT [51] classifies Android malware through dependency graphs. The approach excludes common third-party libraries to improve precision by using a set of benign subgraphs to remove the common subgraphs of the whole dependency graph. DroidCat [52] proposes a dynamic malware detection and classification approach that complements existing static program analyses by supporting dynamic features such as reflections and callbacks. SMART [53] presents a malware classification approach based on semantic clones using deterministic symbolic automation.

Malware clustering mainly works in an unsupervised or semi-supervised scenario to cluster unlabeled samples. Most existing works on Android malware clustering rely on the raw labels reported by AntiVirus vendors [32]. However, the lack of common naming standards results in inconsistencies among the reports from different vendors [54]. To address this problem, AVCLASS [5] processes the raw label from vendors by performing alias detection and generic token removal based on vendor-specific rules to produce a single label per sample. EUPHONY further enhances its accuracy by using self-defined extraction rules to distinguish fields of a raw label. Consensus between different vendors is then reached through plurality voting. Li et al. [16] present a malicious payload mining method for clustering malware by considering the source code information of an app. However, the approach relies on a pre-labeling phase similar to AVCLASS and weakly-labeled samples are not included in their clustering process. This paper addresses the limitations of previous approaches to clustering weakly-labeled malware using a new representation learning approach that preserves code similarity, meta-info and the raw labels of vendors.

Representation Learning. Representation learning takes samples and their corresponding raw features as input and produces a unified and low-dimensional representation, which is particularly useful for later machine-learning tasks, such as classification and clustering. Many existing methods perform representation learning based on a single source of information, such as natural languages (word2vec [30] and doc2vec [14]) and graph structures (DeepWalk [19], Node2vec [55], and LINE [56]).

HRL model has recently emerged as a promising branch of representation learning, demonstrating its power in embedding heterogeneous information into a unified low dimensional space. The resulting representation is particularly useful for many complicated machine-learning tasks. This paper is the first work that leverages the recent advances in HRL to perform comprehensive Android malware clustering by preserving multi-source heterogeneous information by jointly learning three correlations: node sequences, node content, and node labels. This comprehensive representation learning approach is shown as a promising solution for clustering Android malware.

VI. DISCUSSIONS AND LIMITATIONS

First, like all learning-based approach, our approach requires samples for training our model for precise clustering of weakly-labeled malware. The more samples we have, the more precise and robust our clustering model will be.

Second, this approach relies on a whitelist [29] to exclude common Android application third-party libraries, which may potentially lead to false positives or negatives, if the whitelist is incomplete for emerging APIs. Nevertheless, whitelist is still a standard way for reducing noise in code similarity analysis. In addition, whitelist also provides a flexible way for adding any new APIs which are safe.

Third, regarding the calculation of the similarity distance between two Android applications, our method uses Jaccard for measuring the similarity distances, which achieves good results for code analysis. However, other methods, such as Cosine [57], can also be used as an alternative approach to similarity analysis.

Fourth, ANDRE aims to develop a representation learning framework for clustering weakly-labeled malware. When comparing the similarity between Android apps, our method does not focus on obfuscation or deobfuscation, which may lead to imprecise results if malicious apps are obfuscated. Investigating obfuscated app is an orthogonal but an interesting future topic.

Finally, when calculating the similarity between two apps, we made a pairwise comparison for all Android malware. Applying parallel computing or enable similarity comparison between apps using multiple threads can reduce the code analysis overhead and save comparison time.

VII. CONCLUSION

This paper proposes ANDRE, a new approach to Android malware clustering that utilizes heterogeneous information including code similarity, the raw labels of AV vendors and meta-data information to jointly learn an effective representation that embeds all malware in the network into a low-dimensional and compact hybrid feature space for effectively clustering weakly-labeled malware. The experimental results show that ANDRE achieves comparable accuracy to the state-of-the-art approaches for clustering ground-truth samples and that ANDRE can effectively cluster weakly-labeled malware which cannot be clustered by those approaches.

VIII. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful comments. This research is supported by Australian Research Grant DE170101081, DP180102828, LP150100671, DP180100106 and the National Natural Science Foundation of China under Grant 61772055.

REFERENCES

[1] T. I. Murphy, "Android-statistics & facts," https://www.statista.com/topics/876/android, 2018.
[2] "Avtest," https://www.av-test.org/en/statistics/malware/, 2018.
[3] "Caro naming convention," http://www.caro.org/articles/naming.html.
[4] J. Kuo and D. Beck, "The common malware enumeration initiative," *Virus Bulletin*, pp. 14–15, 2005.
[5] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *RAID*. Springer, 2016, pp. 230–253.
[6] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *MSR*, 2017, pp. 425–435.
[7] "Virustotal," https://www.virustotal.com/home/upload, accessed 2018.
[8] "Pluralityvoting," https://en.wikipedia.org/wiki/Plurality_voting.
[9] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *TPAMI*, vol. 35, no. 8, pp. 1798–1828, 2013.
[10] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, 2015, pp. 2111–2117.
[11] S. Pan, J. Wu, X. Zhu, C. Zhang, and Y. Wang, "Tri-party deep network representation," in *AAAI*, 2016, pp. 1895–1901.
[12] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017, pp. 1024–1034.
[13] W. Yu, C. Zheng, W. Cheng, C. C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang, "Learning deep network representations with adversarially regularized autoencoders," in *SIGKDD*, 2018, pp. 2663–2671.
[14] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, 2014, pp. 1188–1196.
[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint:1901.00596*, 2019.
[16] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *RAID*, 2017, pp. 192–214.
[17] C. Shi, B. Hu, W. X. Zhao, and S. Y. Philip, "Heterogeneous information network embedding for recommendation," *TKDE*, vol. 31, no. 2, pp. 357–370, 2019.
[18] Y.-H. H. Tsai, P. P. Liang, A. Zadeh, L.-P. Morency, and R. Salakhutdinov, "Learning factorized multimodal representations," in *ICLR*, 2019.
[19] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *SIGKDD*, 2014, pp. 701–710.
[20] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model." in *AISTATS*, vol. 5. Citeseer, 2005, pp. 246–252.
[21] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *ICSE*, 2016, pp. 1157–1168.
[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.
[23] L. Li, T. F. Bissyande, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *TSE*, 2019.
[24] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 252–276.
[25] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: a scalable and accurate two-phase approach to android app clone detection," in *ISSTA*, 2015, pp. 71–82.
[26] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play

scale." in *USENIX*, vol. 15, 2015.
[27] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014, pp. 175–186.
[28] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *SANER*, vol. 1, 2016, pp. 403–414.
[29] ——, "An investigation into the use of common libraries in android apps," in *SANER*, vol. 1, 2016, pp. 403–414.
[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.
[31] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.
[32] 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 *Ndss*, vol. 14, 2014, pp. 23–26.
[33] "Virusshare," https://www.virusshare.com.
[34] "Android asset packaging tool," https://developer.android.com/studio/command-line/aapt2.
[35] "Topological anomaly detection," https://unsupervisedlearning.wordpress.com/2014/08/04/topological-anomaly-detection/.
[36] "Mlpclassifier," https://scikit-learn.org/stable/modules/neural_networks_supervised.html .
[37] "Svm," https://scikit-learn.org/stable/modules/svm.html.
[38] "Knn," https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.
[39] "Decisiontree," https://scikit-learn.org/stable/modules/tree.html.
[40] "Randomforest," https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.
[41] "Sklearn," https://scikitlearn.org.
[42] "triada," https://www.kaspersky.com/blog/triada-trojan/11481/.
[43] "Command and control server (c&c)," https://www.trendmicro.com/vinfo/us/security/definition/commandandcontrol-server.
[44] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.
[45] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *FSE*. ACM, 2014, pp. 576–587.
[46] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *ICSE*, 2015, pp. 303–313.
[47] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *NDSS*, 2017.
[48] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *SIGKDD*. ACM, 2017, pp. 1507–1515.
[49] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *TIFS*, vol. 14, no. 3, pp. 773–788, 2019.
[50] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *European symposium on research in computer security*. Springer, 2014, pp. 163–182.
[51] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *SIGSAC*. ACM, 2014, pp. 1105–1116.
[52] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *TIFS*, 2018.
[53] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 306–317.
[54] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware," in *DIMVA*, 2016, pp. 142–162.
[55] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *SIGKDD*, 2016, pp. 855–864.
[56] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW*, 2015, pp. 1067–1077.
[57] "Cosine similarity," https://en.wikipedia.org/wiki/Cosine_similarity, accessed 2018.