Apk2vec : semi-supervised multi-view representation learning for profiling Android applications

Narayanan, Annamalai; Soh, Charlie; Chen, Lihui; Liu, Yang; Wang, Lipo

2018

Narayanan, A., Soh, C., Chen, L., Liu, Y., & Wang, L. (2018). Apk2vec : semi-supervised multi-view representation learning for profiling Android applications. Proceedings of 2018 IEEE International Conference on Data Mining (ICDM), 357-366. doi:10.1109/ICDM.2018.00051

https://hdl.handle.net/10356/142658

https://doi.org/10.1109/ICDM.2018.00051

apk2vec: Semi-supervised multi-view representation learning for profiling Android applications

Annamalai Narayanan*, Charlie Soh*, Lihui Chen, Yang Liu and Lipo Wang
[annamala002,csoh004]@e.ntu.edu.sg,[elhchen,yangliu,elpwang]@ntu.edu.sg
Nanyang Technological University, Singapore

ABSTRACT

Building behavior profiles of Android applications (apps) with holistic, rich and multi-view information (e.g., incorporating several semantic views of an app such as API sequences, system calls, etc.) would help catering downstream analytics tasks such as app categorization, recommendation and malware analysis significantly better. Towards this goal, we design a semi-supervised Representation Learning (RL) framework named apk2vec to automatically generate a compact representation (*aka* profile/embedding) for a given app. More specifically, apk2vec has the three following unique characteristics which make it an excellent choice for large-scale app profiling: (1) it encompasses information from multiple semantic views such as API sequences, permissions, etc., (2) being a semi-supervised embedding technique, it can make use of labels associated with apps (e.g., malware family or app category labels) to build high quality app profiles, and (3) it combines RL and feature hashing which allows it to efficiently build profiles of apps that stream over time (i.e., online learning).

The resulting semi-supervised multi-view hash embeddings of apps could then be used for a wide variety of downstream tasks such as the ones mentioned above. Our extensive evaluations with more than 42,000 apps demonstrate that apk2vec's app profiles could significantly outperform state-of-the-art techniques in four app analytics tasks namely, malware detection, familial clustering, app clone detection and app recommendation.

KEYWORDS

Representation Learning, Graph Embedding, Skipgram, Malware Detection, App Recommendation

1 INTRODUCTION

The low threshold for entering the official and third-party Android app markets has attracted a large number of developers, resulting in an exponential growth of apps. At the point of this writing, Google play [1] is hosting more than 3.6 million apps [2], with thousands of new apps being added daily. Due to the wide range of functionalities provided by the apps (e.g., online shopping, banking, etc.), they have become an indispensable part of people's daily life. However, this astronomical volume of functionality rich apps, has raised several challenging issues. A few significant ones are as follows: (i) app markets are facing difficulties in organizing large volumes of diverse apps to allow convenient and systematic browsing by the users, (ii) due to the rapid growth rate in app volumes, it is becoming increasingly tough for markets to recommend up-to-date and meaningful apps that matches users' search queries, and (iii) with a significant number of plagiarists and malware authors hidden among app developers, these markets have been plagued with app clones and malicious apps.

One could observe that a systematic and deep understanding of apps' behaviors is essential to solve the aforementioned issues. Building high-quality behavior profiles of apps could help in determining the semantic similarity among the apps, which is pivotal to addressing these issues. Recent research [23, 24, 29, 32–38] reveals that compared to primitive representations of programs (e.g., counts of system-calls, Application Programming Interfaces (APIs) used etc.) graph representations (e.g., Control Flow Graphs (CFGs), call graphs, etc.) are ideally suited for app profiling, as the latter retain program semantics well, even when the apps are obfuscated. Reinforcing this fact, many recent works achieved excellent results using graph representations along with Machine Learning (ML) techniques on a plethora of program analytics tasks such as malware detection [23, 24, 32, 33, 38], familial classification [37], clone detection [29, 42], library detection [39] etc. In effect, these works cast their respective program analytics task as a graph analytics task and apply existing graph mining techniques [33] to solve them. Typically, these ML algorithms work on vectorial representations (*aka* embeddings) of graphs. Hence, arguably, one of the most important factors that determines the efficacy of these downstream analytics tasks is the quality of such embeddings.

Besides the choice of graph representations, another pivotal factor that influences the aforementioned tasks are the features that could be extracted from them. In the case of app analytics, the most prominent features in recent literature include API/system-call sequences observed [23], permissions [27] and information source/sinks used [31], etc. Evidently, each of these feature sets provides a different semantic *perspective* (interchangeably referred as *view*) of the apps' behavior with different inherent strengths and limitations. As revealed by existing works [24, 27], capturing multiple semantic views with different modalities would help to improve the accuracy of downstream tasks significantly. Furthermore, any form of labeling information (e.g., malware family label, app category label, etc.) could be of significant help in building semantically richer and more accurate app profiles.

Towards catering the above-mentioned applications, in this work, we propose a Representation Learning (RL) technique to build data-driven, compact and versatile behavior profiles of apps. Based on the above observations, the following challenges have to be addressed to obtain such a profile:

(C1) Handcrafted features. Graph representations of programs such as CFGs are highly expressive data structures. Consequently,

* indicates equal contributions.

ICDM, 2018, Singapore

...$15.00
https://doi.org/

representing them as vectors without losing much of their expressiveness is non-trivial. A typical solution is to use graph kernels [19, 20, 40, 41] which leverage on graph substructures (e.g., shortest paths, graphlets etc.) to build graph embeddings. However, these substructures are *handcrafted*. Therefore, when used on large datasets, these features lead to building high dimensional, sparse and non-smooth graph embeddings which do not generalize well and thus yield suboptimal accuracies [10, 11].

(C2) Fully supervised or unsupervised RL. Addressing the limitations of graph kernels, several data-driven supervised (e.g., CNNs [16], RNNs [17]) and unsupervised (e.g.,skipgrams [9, 25], autoencoders [18]) graph embedding approaches have been proposed. Both types of approaches exhibit good generalization and offer excellent accuracies. However, the supervised embedding methods suffer from the following limitations: (i) they require typically large volumes of labeled graphs to learn meaningful embeddings, which is undesirable and often impractical for large-scale app analytic tasks, and (ii) the embeddings thus learnt are specific to one particular analytics task and may not be transferable to others. On the other hand, unsupervised embedding methods do not exhibit these limitations. However, in many cases, a portion of the dataset may have labels or some samples may have more than one label (e.g., an app may have several labels such as category to which it belongs, whether or not it is malicious, etc.). Unsupervised embedding approaches are incapable of leveraging such labels which contain valuable semantic information.

(C3) Scalability. Though RL based methods provide graph embeddings which generalize well, when used on large datasets, they exhibit poor scalability both in terms of memory and time requirements. This is because they have extremely large number of parameters to train (especially in models such as RNNs and skipgrams).

(C4) Integrating information from multiple views. All the above mentioned approaches are typically designed to capture only one semantic view of the program through their embeddings. This severely limits their potentials to cater to a wide range of downstream tasks. Effectively integrating information from multiple views is challenging, but of paramount importance in building comprehensive embeddings capable of catering to a variety of tasks.

Our approach. Driven by these motivations, we develop a static analysis based semi-supervised multi-view RL framework named apk2vec to build high-quality data-driven profiles of Android apps. apk2vec has two major phases: (1) *a static analysis phase* in which a given *apk* file is disassembled and three different dependency graphs (DGs), each representing a distinct semantic view are extracted and, (2) *an embedding phase* in which a neural network is used to combine the information from these three DGs and label information (if available) to learn one succinct embedding for the *apk*. To this end, apk2vec combined and extends several state-of-the-art RL ideas such as multimodal (*aka* multi-view) RL, semi-supervised neural embedding and feature hashing.

apk2vec addresses the above-mentioned challenges in the following ways:

- **Data-driven embedding:** Unlike graph kernels, apk2vec uses a skipgram neural network [4, 25] that automatically learns features from large corpus of graph data to produce high quality dense embeddings. This in effect addresses challenge C1.
- **Semi-supervised task-agnostic embedding:** apk2vec's neural network facilitates using class labels of *apk*s (incl. multiple labels per sample) if they are available to build better app profiles. However, these embeddings are still task-agnostics and hence can be used for a variety of downstream tasks. This helps addressing challenge C2.
- **Hash embedding:** Recently, Svenstrup et al [5] proposed a scalable feature hashing based word embedding model which required much lesser number of trainable parameters than conventional RL models. Besides this improvement in efficiency, hash embedding model also facilitates learning embeddings when instances stream over time. Inspired by this idea, in apk2vec, we develop an efficient hash embedding model for graph/subgraph embedding, thus addressing challenge C3.
- **Multi-view embedding:** apk2vec's neural network facilitates multimodal RL through a novel learning strategy (see §4.3). This helps to integrate three different DGs that emerge from a given *apk* file and produce one common embedding. Thus apk2vec facilitates combining information from different views in a systematic and non-linear manner, thereby addressing challenge C4.

Experiments. To evaluate our approach, we perform a series of experiments on various app analytics tasks (incl. supervised, semi-supervised and unsupervised learning tasks), using a dataset of more than 42,000 real-world Android apps. The results show that our semi-supervised multimodal embeddings can achieve significant improvements in terms of accuracies over unsupervised/unimodal RL approaches and graph kernel methods while maintaining comparable efficiency. The improvements in prediction accuracies range from 1.74% to 5.93% (see §5 for details).

In summary, we make the following contributions:

- We propose apk2vec, a static analysis based data-driven semi-supervised multi-view graph embedding framework, to build task-agnostic profiles for Android apps (§4). To the best of our knowledge, this is the first app profiling framework that has three aforementioned unique characteristics.
- We propose a novel variant of the skipgram model by introducing a view-specific negative sampling technique which facilitates integrating information from different views in a non-linear manner to obtain multi-view embeddings (§4.7).
- We extend the feature hashing based word embedding model to learn multi-view graph/subgraph embeddings. Hash embeddings improve apk2vec's overall efficiency and support online RL (§4.6).
- We make an efficient implementation of apk2vec and the profiles of all the apps used in this work publicly available at [22].

2 PROBLEM STATEMENT

Given a set of *apk*s $\mathbb{A} = \{a_1, a_2, ...\}$, a set of corresponding labels $\mathbb{L} = \{l_1, l_2, ...\}$ (some of which may be empty i.e., $\forall l_i \in \mathbb{L}, |l_i| \geq 0$) for each app in $\mathbb{A}$ and a positive integer δ (i.e., embedding size), we intend to learn δ-dimensional distributed representations for every *apk* $a_i \in \mathbb{A}$. The matrix representations of all *apk*s is denoted as $\Phi^{\mathbb{A}} \in \mathbb{R}^{|\mathbb{A}| \times \delta}$.

More specifically, $a_i \in \mathbb{A}$ can be represented as a three-tuple: $a_i = (G_i^v)$ where $v \in \{A, P, S\}$ and G_i^A, G_i^P, G_i^S denote its API Dependency Graph (ADG), Permission Dependency Graph (PDG), and information Source & sink Dependency Graph (SDG), respectively (refer to §4.2 for details on constructing these DGs). Furthermore, a DG can be represented as $G_i^v = (N_i^v, E_i^v, \lambda^v)$, where N_i^v is the set of nodes and $E_i^v \subseteq N_i^v \times N_i^v$ is the set of edges in G_i^v. A labeling function $\lambda^v : N_i^v \rightarrow L^v$ assigns a label to every node in N_i^v from alphabet set L^v.

Given $G^v = (N^v, E^v, \lambda^v)$ and $sg^v = (N_{sg}^v, E_{sg}^v, \lambda_{sg}^v)$. sg^v is a subgraph of G iff there exists an injective mapping $\mu : N_{sg}^v \rightarrow N^v$ such that $(n_1, n_2) \in E_{sg}^v$ iff $(\mu(n_1), \mu(n_2)) \in E^v$. In this work, by subgraph, we strictly refer to a specific class of subgraphs, namely, rooted subgraphs. In a given graph G^v, a rooted subgraph of degree d around node $n \in N^v$ encompasses all the nodes (and corresponding edges) that are reachable in d hops from n.

3 BACKGROUND & RELATED WORK

Our goal is to build compact multi-view behavior profiles of *apk* files in a scalable manner. To this end, we develop a novel *apk* embedding framework by combining several RL ideas such as word, document and graph embedding models and feature hashing. Hence, in this section, the related background from these areas are reviewed.

3.1 Skipgram word and document embedding model

The popular word embedding model `word2vec` [4] produces word embeddings that capture meaningful syntactic and semantic regularities. To learn word embeddings, `word2vec` uses a simple feed-forward neural network architecture called *skipgram*. It exploits the notion of context such that, given a sequence of words $\{w_1, w_2, ..., w_t, ..., w_T\}$, the target word w_t whose representation has to be learnt and the length of the context window c, the objective of skipgram model is to maximize the following log-likelihood:

$$\sum_{t=1}^{|\mathcal{T}|} \log \Pr(w_{t-c}, \ldots, w_{t+c} | w_t) \approx \sum_{t=1}^{|\mathcal{T}|} \log \prod_{-c \leq j \leq c, j \neq 0} \Pr(w_{t+j} | w_t) \quad (1)$$

where $w_{t-c}, ..., w_{t+c}$ are the context words and $\mathcal{T}$ is the vocabulary of all the words. Here, the context and target words are assumed to be independent. Furthermore, the term $\Pr(w_{t+j}|w_t)$ is defined as: $\frac{e^{(\vec{w}_t \cdot \vec{w}'_{t+j})}}{\sum_{w \in \mathcal{T}} e^{(\vec{w}_t \cdot \vec{w})}}$ where $\vec{w}$ and $\vec{w}'$ are the input and output embeddings of word w, respectively. In the face of very large $\mathcal{T}$, the posterior probability in eq.(1) could be learnt in an efficient manner using the so-called *negative sampling* technique.

Negative Sampling. In each iteration, instead of considering all words in $\mathcal{T}$ a small subset of words that do not appear in the target word's context are selected at random to update their embeddings. Training this way ensures the following: *if a word w_t appears in the context of another word w_c, then the embedding of w_t is closer to that of w_c compared to any other randomly chosen word from $\mathcal{T}$.* Once skipgram training converges, semantically similar words are mapped to closer positions in the embedding space revealing that the learnt embeddings preserve semantics.

Le and Mikolov's `doc2vec`[6] extends the skipgram model in a straight forward manner to learn representations of arbitrary length word sequences such as sentences, paragraphs and whole documents. Given a set of documents $\mathbb{D} = \{d_1, d_2, ...\}$ and a set of words $c(d_i) = \{w_1, w_2, ...\}$ sampled from document $d_i \in \mathbb{D}$, `doc2vec` skipgram learns a δ dimensional embeddings of the document d_i and each word $w_j \in c(d_i)$. The model works by considering a word $w_j \in c(d_i)$ to be occurring in the context of document d_i and tries to maximize the following log likelihood: $\sum_{j=1}^{|c(d_i)|} \log \Pr(w_j|d_i)$ where the probability $\Pr(w_j|d_i)$ is defined as: $\frac{e^{(\vec{d} \cdot \vec{w}_j)}}{\sum_{w \in \mathcal{T}} e^{(\vec{d} \cdot \vec{w})}}$ Here, $\mathcal{T}$ is the vocabulary of all the words across all documents in $\mathbb{D}$. Understandably, `doc2vec` skipgram could be trained efficiently using negative sampling.

Model parameters. From the explanations above, it is evident that the total number of trainable parameters of skipgram word and document embedding skipgram models would be $2|\mathcal{T}|\delta$ and $\delta(|\mathbb{D}| + |\mathcal{T}|)$, respectively.

3.2 Hash embedding model

Though skipgram emerged as a hugely successful embedding model, it poses scalability issues when the vocabulary $\mathcal{T}$ is very large. Also, its architecture does not support learning embeddings when new words (*aka new tokens*) stream over time. To address these issues, Svenstrup *et al.*, [5] proposed a feature hashing based word embedding model. This model involves the following steps:
(1) A token to id mapping function, $\mathcal{F} : \mathcal{T} \rightarrow \{1, ..., K\}$ and k hash functions of the form $\mathcal{H}_i : \{1, ..., K\} \rightarrow \{1, ..., B\}, i \in [1, k]$ are defined (B is the number of hash buckets and $B << K$).
(2) The following arrays are initialized: $\Phi^{B \times \delta}$: a pool of B component vectors which are intended to be shared by all words in $\mathcal{T}$, and $p^{K \times k}$: contains the importance of each component vector for each word.
(3) Given a word $w \in \mathcal{T}$, hash functions $\mathcal{H}_1, ...\mathcal{H}_k$ are used to choose k component vectors from the shared pool Φ.
(4) The component vectors from step (3) are combined as a weighted sum to obtain the hash embedding of the word w: $\vec{w} = \sum_{w, i=1}^{i=k} p_w^i \mathcal{H}_i(w)$.
(5) With hash embeddings of target and context words thus obtained, skipgram model could be used to train for eq. (1). However, unlike regular skipgram which considers Φ alone as a set of trainable parameters, one could train p as well.

Thus Svenstrup et al's framework reduces the number of trainable parameters from $2K\delta$ to $2(B\delta + Kk)$, which helps reducing the pretraining time and memory requirements. The effect of collisions from K to B could be minimized by having more than one hash function and this helps in maintaining accuracies on-par with `word2vec`. Besides, when new words arrive over time, a function like MD5 or SHA1 could be used in place of $\mathcal{F}$ to hash them to a fixed set of integers in range $[1, K]$. This helps learning word embeddings in an online fashion.

3.3 Graph embedding models

Analogously, `graph2vec`[25] considers simple node labeled graphs such as CFGs as documents and the rooted subgraphs around every node in them as words that compose the document. The intuition

is that different subgraphs compose graphs in a similar way that different words compose documents. In this way, graph2vec could be perceived as an RL variant of Weisfeiler-Lehman kernel (WLK) which counts the number of common rooted subgraphs across a pair of graphs to estimate their similarity. As such, graph2vec is capable of learning embeddings of arbitrary sized graphs.

Given a dataset of graphs $\mathbb{G} = \{G_1, G_2, ...\}$, graph2vec extends the skipgram model explained in §3.1 to learn embeddings of each graph. Let $G_i \in \mathbb{G}$ be denoted as (N_i, E_i, λ) and the set of all rooted subgraphs around every node $n \in N_i$ (up to a certain degree D) be denoted as $c(G_i)$. graph2vec aims to learn a δ dimensional embeddings of the graph G_i and each subgraph sg_j sampled from $c(G_i)$ i.e., $\vec{G}_i, \vec{sg}_j \in \mathbb{R}^\delta$, respectively by maximizing the following log likelihood: $\sum_{j=1}^{|c(G_i)|} \log \Pr(sg_j|G_i)$, where the probability $\Pr(sg_j|G_i)$ is defined as: $\frac{e^{(\vec{G} \cdot \vec{sg}_j)}}{\sum_{w \in \mathcal{T}} e^{(\vec{G} \cdot \vec{sg})}}$. Here, $\mathcal{T}$ is the vocabulary of all the subgraphs across all graphs in $\mathbb{G}$. The number of trainable parameters of this model will be $\delta(|\mathbb{G}| + |\mathcal{T}|)$.

Similar to graph2vec, many recent approaches such as sub2vec [9], GE-FSG [45] and Anonymous Walk Embeddings (AWE) [46] have adopted skipgram architecture to learn unsupervised graph embeddings. The fundamental difference among them is the type of graph substructure that they consider as a graph's context. For instance, sub2vec considers nodes, GE-FSG considers frequent subgraphs (FSGs) and AWE considers walks that exist in graphs as their contexts, respectively.

3.4 Semi-supervised embedding model

Recently, Pan *et al.* [14] extended the skipgram model to learn embedding of nodes in Heterogeneous Information Networks (HINs) in a semi-supervised fashion. Specifically, their extension facilitates skipgram to use class labels of (a subset of) samples while building embeddings. For instance, when l_i, the class label of a document d_i is available, the doc2vec model could maximize the following log likelihood:

$$\beta \sum_{j=1}^{|c(d_i)|} \log \Pr(w_j|d_i) + (1-\beta) \sum_{j=1}^{|c(d_i)|} \log \Pr(w_j|l_i) \qquad (2)$$

to include the supervision signal available from l_i along with the contents of the document made available through $c(d_i)$. Here, β is the weight that balances the importance of the two components in document embedding. Pan *et al.* empirically prove that embeddings with this form of semi-supervision significantly improves the accuracy of downstream tasks.

In summary, all above-mentioned embedding models possess some strengths for RL in various areas mainly for natural language texts. Differing from them, we propose a new and efficient data-driven graph embedding model/framework for app behaviour profiling. The new framework has three unique characteristics which brings in crucial strengths for holistic app behavior profiling, namely: (i) multi-view RL, (ii) semi-supervised RL, and (iii) feature hashing based RL. We illustrate (both in principle and through experiments) that this new framework possess all these strengths and caters a multitude of downstream tasks.

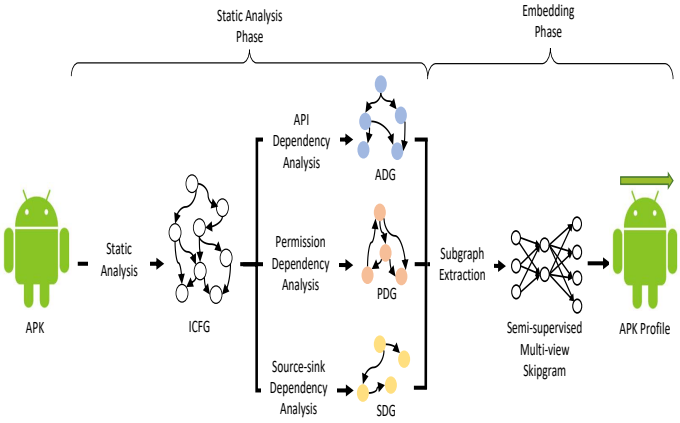

Figure 1: APK2VEC: Framework overview

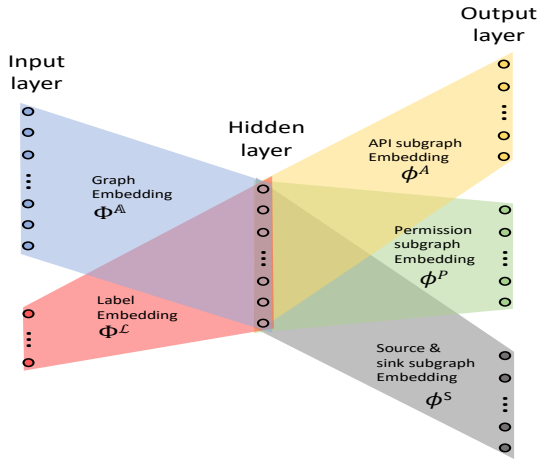

Figure 2: Semi-supervised Multi-view Skpigram

4 METHODOLOGY

In this section, we explain the apk2vec app profiling framework. We first present an overview of the framework which encompasses two phases, subsequently, we discuss the details of each component in the following subsections.

4.1 Framework Overview

As depicted in Figure 1, the workflow of apk2vec can be divided into two major phases, namely, the *static analysis phase* and the *embedding phase*. The static analysis phase encompasses of only program analysis procedures which intend to transform *apk* files into DGs. The subsequent embedding phase encompasses RL techniques that transform these DGs into *apk* profiles.

Static analysis phase. This phase begins with disassembling the *apk*s in the given dataset and constructing their interprocedural CFG (ICFG). Further static analysis is performed to abstract each ICFG into three different DGs, namely, ADG, PDG and SGD. Each of them represent a unique semantic view of the app's behaviors with distinct modalities. Detailed procedure of constructing these DGs is presented in §4.2.

Embedding phase. After obtaining the DGs for all the *apk*s in the dataset, rooted subgraphs around every node in the DGs are extracted to facilitate the learning *apk* embeddings. Once the rooted subgraphs are extracted, we train the semi-supervised multi-view skpigram neural network with them. The detailed procedure is explained in §4.3.

4.2 Static Analysis Phase

ICFG construction. Given an *apk* file, the first step is to perform static control-flow analysis and construct its ICFG. We chose ICFG over other program representation graphs (e.g., DFGs, call graphs) due to its fine-grained representation of control flow sequence, which allows us to capture finer semantic details of the *apk* which is necessary to build a comprehensive *apk* profile. Formally, $ICFG = (N, E)$ for an *apk* a is a directed graph where each node $bb \in N$ denotes a basic block[1] of a method m in a, and each edge $e(bb_1, bb_2) \in E$ denotes either an intra-procedural control-flow or a calling relationship from bb_1 to bb_2 and $E \subseteq N \times N$.

Abstraction into multiple views. Having constructed the ICFG, to obtain richer semantics from the *apk*, we abstract it with three Android platform specific analysis, namely API sequences, Android permissions, and information sources & sinks to construct the three DGs, respectively. The abstraction process is described below.

To obtain the ADG from a given ICFG, we remove all nodes that do not access security sensitive Android APIs. This will leave us with a subset of sensitive nodes from the perspective of API usages, say $N^{\mathcal{A}} \subseteq N$. Subsequently, we connect a pair of nodes $n_1, n_2 \in N^{\mathcal{A}}$ iff there exist a path between them in ICFG. This yields the ADG, $G^{\mathcal{A}}$ which could be formally represented as a three tuple $G^{\mathcal{A}} = (N^{\mathcal{A}}, E^{\mathcal{A}}, \lambda^{\mathcal{A}})$, where $\lambda^{\mathcal{A}} : N^{\mathcal{A}} \rightarrow L^{\mathcal{A}}$ is a labeling function that assigns a security sensitive API as a node label to every node in $N^{\mathcal{A}}$ from a set of alphabets $L^{\mathcal{A}}$. We refer to existing work [23] for the list of security sensitive APIs. Similarly, we use works such as PScout [30] and SUSI [31] which maps APIs to Android permissions and information source/sinks to obtain set of node labels $L^{\mathcal{P}}$ and $L^{\mathcal{S}}$, respectively. Subsequently, adopting the process mentioned above using them we abstract the ICFG into PDG and SDG using $L^{\mathcal{P}}$ and $L^{\mathcal{S}}$, respectively.

4.3 Embedding Phase

In the embedding phase, our goal is to take the DGs and class labels that correspond to a set of *apk*s and train the skipgram model to obtain the behavior profile for each *apk*. To this end, we develop a novel variant of the skipgram model which facilitates incorporating the three following learning paradigms: semi-supervised RL, multi-view RL and feature hashing.

Network architecture Figure 2 depicts the architecture of the neural network used in apk2vec's RL process. The network consists of two shared input layers and three shared output layers (one for each view). The goal of the first input layer ($\Phi^{\mathbb{A}}$) is to perform multi-view RL. More precisely, given an *apk* id a_i in the first layer, the network intends to predict the API, permission and source-sink subgraphs that appear in a_i's context, in each of the output layers. Similarly, the goal of the second input layer ($\Phi^{\mathcal{L}}$) is to perform semi-supervised RL. More specifically, given a_i's class label as input in the second layer, the network intends to predict subgraphs of all three views that occur in a_i's context in each of the output layers. Thus, the network forces API, permission and source-sink subgraphs that frequently co-occur with same class labels to have similar embeddings. For instance, given a malware family label †, subgraphs that characterize †'s behaviors would end up having similar embeddings. This in turn would influence *apk*s that belong to † to have similar embeddings.

[1] A basic block is a sequence of instructions in a method with only one entry point and one exit point which represents the largest piece of the program that is always executed altogether.

Algorithm 1: APK2VEC ($\mathbb{A}, \mathbb{L}, D, \delta, \mathcal{E}, B^{\upsilon}, k, \alpha$)

Input: $\mathbb{A} = \{a_1, a_2, \ldots\}$: set of *apk*s such that $a_i = \{G_i^{\upsilon}\}$ for $\upsilon \in \{\mathcal{A}, \mathcal{P}, \mathcal{S}\}$
$\mathbb{L} = \{l_1, l_2, \ldots\}$: Set of labels for each *apk* in $\mathbb{A}$. Note that there may be zero or more labels for an *apk*. Hence $\forall l_i \in \mathbb{L}$, $|l_i| \geq 0$. Let the total number of unique labels across $l_i \in \mathbb{L}$ be denoted as $\mathcal{L}$.
D : Maximum degree of rooted subgraphs to be considered for learning embeddings. This will produce a vocabulary of subgraphs in each view, $\mathcal{T}^{\upsilon} = \{sg_1^{\upsilon}, sg_2^{\upsilon}, \ldots\}$ from all the graphs G_i^{υ}. Let $|\mathcal{T}^{\upsilon}|$ be denoted as K^{υ}.
δ : Number of dimensions (embedding size)
$\mathcal{E}$: Number of epochs
B^{υ}: Number of hash buckets for view υ
k: Number of hash functions (maintained same across all views)
α : Learning rate

Output: Matrix of vector representations of *apk*s $\Phi^{\mathbb{A}} \in \mathbb{R}^{|\mathbb{A}| \times \delta}$

1 Initialization: Sample $\Phi^{\mathbb{A}}$ from $\mathbb{R}^{|\mathbb{A}| \times \delta}$, $\Phi^{\mathcal{L}}$ from $\mathbb{R}^{|\mathcal{L}| \times \delta}$, ϕ^{υ} from $\mathbb{R}^{B^{\upsilon} \times k}$, and p^{υ} from $\mathbb{R}^{K^{\upsilon} \times k}$
for $e = \{1, 2, \ldots, \mathcal{E}\}$ **do**
2 **for** $a_i \in$ SHUFFLE($\mathbb{A}$) **do**
3 **for** $G_i^{\upsilon} \in a_i$ **do**
4 $sg_c :=$ GETSUBGRAPHS(G_i^{υ}, D)
5 $J(\Phi^{\mathbb{A}}, \phi^{\upsilon}, p^{\upsilon}) :=$
$$-\log \prod_{sg \in sg_c} \frac{e^{(\Phi^{\mathbb{A}}(a_i) \cdot \text{HASHEMB}(sg, \phi^{\upsilon}, p^{\upsilon}, \upsilon))}}{\sum_{sg' \in \mathcal{T}^{\upsilon}} e^{(\Phi^{\mathbb{A}}(a_i) \cdot \text{HASHEMB}(sg', \phi^{\upsilon}, p^{\upsilon}, \upsilon))}}$$
6 $\Phi^{\mathbb{A}} := \Phi^{\mathbb{A}} - \alpha \frac{\partial J}{\partial \Phi^{\mathbb{A}}}$; $\phi^{\upsilon} := \phi^{\upsilon} - \alpha \frac{\partial J}{\partial \phi^{\upsilon}}$; $p^{\upsilon} := p^{\upsilon} - \alpha \frac{\partial J}{\partial p^{\upsilon}}$
7 **for** $l \in l_i$ **do**
8 $J(\Phi^{\mathcal{L}}, \phi^{\upsilon}, p^{\upsilon}) :=$
$$-\log \prod_{sg \in sg_c} \frac{e^{(\Phi^{\mathcal{L}}(l) \cdot \text{HASHEMB}(sg, \phi^{\upsilon}, p^{\upsilon}, \upsilon))}}{\sum_{sg' \in \mathcal{T}^{\upsilon}} e^{(\Phi^{\mathcal{L}}(l) \cdot \text{HASHEMB}(sg', \phi^{\upsilon}, p^{\upsilon}, \upsilon))}}$$
9 $\Phi^{\mathcal{L}} := \Phi^{\mathcal{L}} - \alpha \frac{\partial J}{\partial \Phi^{\mathcal{L}}}$; $\phi^{\upsilon} := \phi^{\upsilon} - \alpha \frac{\partial J}{\partial \phi^{\upsilon}}$; $p^{\upsilon} := p^{\upsilon} - \alpha \frac{\partial J}{\partial p^{\upsilon}}$
10 **return** $\Phi^{\mathbb{A}}$

Hash embeddings. Considering the real-world scenario where Android platform evolves (i.e., APIs/permissions being added or removed) and apps stream rapidly over time, it is obvious that the vocabulary of subgraphs (across all DGs) would grow as well. Regular skipgram models could not handle such a vocabulary and as mentioned in §3.2, hash embeddings could be used effectively to address this. Note that in our framework, the vocabulary of tokens is only present in the output layer. Hence, in apk2vec, hash embeddings are used only in the three output layers (ϕ^A, ϕ^P and ϕ^S) and the two input layers ($\Phi^{\mathbb{A}}$ and $\Phi^{\mathcal{L}}$) uses regular embeddings.

The process through with our skipgram model is trained is explained through Algorithm 1.

4.4 Algorithm: apk2vec

The algorithm takes the set of *apk*s along with their corresponding DGs ($\mathbb{A}$), set of their labels ($\mathbb{L}$), maximum degree of rooted

subgraphs to be considered (D), embedding size (δ), number of epochs ($\mathcal{E}$), number of hash buckets per view (B^v), number of hash functions (k) and learning rate (α) as inputs and outputs the *apk* embeddings ($\Phi^{\mathbb{A}}$). The major steps of the algorithm are as follows:

(1) We begin by randomly initializing the parameters of the model i.e., $\Phi^{\mathbb{A}}$: *apk* embeddings, $\Phi^{\mathcal{L}}$: label embeddings, ϕ^v: embeddings of each hash bucket for each of the three views, and p^v: importance parameters for each of the views (line 1). It is noted that except the *apk* embeddings, all other parameters are discarded when training culminates.
(2) For each epoch, we consider each *apk* a_i as the target whose embedding has to be updated. To this end, each of its DG G_i^v is taken and all the rooted subgraphs upto degree D around every node are extracted from the same (line 4). The subgraph extraction process is explained in detail in §4.5.
(3) The set of all such subgraphs sg_c, is perceived as the context of a_i. Once sg_c is obtained, we get their hash embeddings and compute the negative log likelihood of them being similar to the target *apk* a_i's embedding (line 5). The hash embedding computation process is explained in detail in §4.6.
(4) With the loss value thus computed, the parameters that influence the loss are updated (line 6). We propose a novel view-specific negative sampling strategy to train the skipgram and the same is explained in §4.7.
(5) Subsequently, for each of a_i's class labels i.e., $l \in l_i$, we compute the negative log likelihood of their similarity to the context subgraphs in sg_c and update the parameters that influence the same (lines 7-9). This step amounts to semi-supervised RL as l_i could be empty for some *apk*s.
(6) The above mentioned process is repeated for $\mathcal{E}$ epochs and the *apk* embeddings (along with other parameters) are refined.

Finally, when training culminates, *apk* embeddings in $\Phi^{\mathbb{A}}$ are returned (line 10).

4.5 Extracting context subgraphs

For a given *apk* a_i, extracting rooted subgraphs around each node in each G_i^v and considering them as its context is a fundamental task in our approach. To extract these subgraphs, we follow the well-known Weisfeiler-Lehman relabeling process [19] which lays the basis for `WLK` [11, 19]. The subgraph extraction process is presented formally in Algorithm 2. The algorithm takes the graph G from which the subgraphs have to be extracted and maximum degree to be considered around root node D as inputs and returns the set of all rooted subgraphs in G, S. It begins by initializing S to an empty set (line 2). Then, we intend to extract rooted subgraph of degree d around each node n in the graph. When $d = 0$, no subgraph needs to be extracted and hence the label of node n is returned (line 6). For cases where $d > 0$, we get all the (breadth-first) neighbours of n in $\mathcal{N}_n$ (line 8). Then for each neighbouring node, n', we get its degree $d-1$ subgraph and save the same in a multiset $M_n^{(d)}$ (line 9). Subsequently, we get the degree $d-1$ subgraph around the root node n and concatenate the same with sorted list $M_n^{(d)}$ to obtain the subgraph of degree d around node n, which is denoted as $sg_n^{(d)}$ (line 10). $sg_n^{(d)}$ is then added to the set of all subgraphs (line 11). When all the nodes are processed, rooted subgraphs of degrees $[0, D]$ are collected in S which is returned finally (line 12).

Algorithm 2: GetSubgraphs (G, D)

```
1  begin
2      S := {} //Initialize with an empty set
3      for n ∈ N do
4          for d ∈ {0, 1, ..., D} do
5              if d = 0 then
6                  sg_n^(d) := λ(n) //node label
7              else
8                  N_n := {n' | (n, n') ∈ E}//neighboring nodes
9                  M_n^(d) := {{GetWLSubgraph(n', G, d − 1) | n' ∈ N_n}}
                     //multiset of rooted subgraphs around neighboring nodes
10                 sg_n^(d) := GetWLSubgraph(n, G, d − 1) ⊕ sort(M_n^(d))
11             S := S ∪ sg_n^(d)
12     return S //set of all rooted subgraphs in G
```

Algorithm 3: HashEmb (sg, ϕ, p, v)

```
1  begin
2      sg_id := F^v(sg) //Token to id mapping function
3      components := (φ(H_1(sg_id)), ..., φ(H_k(sg_id)))^T //shape of components: k × δ
4      weights := (p_1^v(sg_id), p_2^v(sg_id), ..., p_k^v(sg_id))^T //shape of weights: k × 1
5      s⃗g := weights^T · components //1 × k · k × δ
6      return s⃗g
```

4.6 Obtaining hash embeddings

Once context subgraphs are extracted, we proceed with obtaining their hash embeddings and training the same along the target *apk*'s embedding. Given a subgraph, the process of extracting its hash embedding involves four steps which are formally presented in Algorithm 3. Following is the explanation of this algorithm:

(1) Given a subgraph sg, we begin by mapping to an integer sg_{id}, using a function $\mathcal{F}^v$ (line 2). When $\mathcal{T}^v$, the vocabulary of all the subgraphs in view v could be obtained ahead of training, a regular dictionary *aka* token-to-id function which maps each subgraph to a unique number in the range $[1, K^v]$ (where $K^v = |\mathcal{T}^v|$) could be used as $\mathcal{F}^v$. In the online learning setting, such a dictionary could not be obtained. Hence, analogous to feature hashing [26], one could use a regular hash function such as MD5 or SHA1 to hash the subgraph to an integer in the predetermined range $[1, K^v]$ (here, an arbitrarily large value of K^v is chosen to avoid collisions).
(2) sg_{id} is then hashed using each of the k hash functions. Each function $\mathcal{H}_i, i \in [1, k]$ maps it to one of the B^v available hash buckets which in turn maps to one of the B^v component embeddings in ϕ^v. Thus we obtain k component embeddings for the given subgraph and save them in *components* (line 3). In other words, *components* contains k δ-dimensional embeddings.
(3) Similarly, using sg_{id}, we then lookup the importance parameter for each hash function, $p_i^v(sg_{id}), i \in [1, k]$ and save them in *weights* (line 4). In other words, *weights* contains k importance values.
(4) Finally, the hash embedding of the subgraph is obtained by multiplying k δ-dimensional component vectors with k corresponding importance values (line 5).

Once the hash embeddings of the context subgraphs are obtained using the above mentioned process, one could train them along with the target *apk*'s embedding using a learning algorithm such as Stochastic Gradient Decent (SGD).

4.7 View-specific negative sampling

Similar to other skipgram based embedding models such as `graph2vec` [25], we could efficiently minimize the negative log likelihood in lines 5 and 8 of Algorithm 1. That is, given an *apk* a and a subgraph sg^v which is contained in view v, the regular negative sampling intends to maximize the similarity between their embeddings. Besides, it chooses η subgraphs as negative samples i.e., that do not occur in the context of a and minimizes the similarity of a and these negative samples. This could be formally presented as follows,

$$\Pr(sg^v|a) = \sigma(\vec{a}^T \cdot \vec{sg^v}) \prod_{j=1}^{\eta} \mathbb{E}_{sg_j \sim \Pr_n(\mathcal{T})} \sigma(-\vec{a}^T \cdot \vec{sg}_j) \quad (3)$$

where, $\mathcal{T} = \bigcup_v \mathcal{T}^v$ is union of vocabularies across all views and $\mathbb{E}$ is expectation of choosing a subgraph sg_j from the smoothed distribution of subgraphs $\Pr_n$ across all the three views.

In simpler terms, eq. (3) moves $\vec{a}$ closer to $\vec{sg^v}$ as it occurs in a's context and also moves $\vec{a}$ farther away from $\vec{sg}_j$ (which may not belong to view v) as it does not occur in a's context.

However, in our multi-view embedding scenario, the distribution of subgraphs is not similar across all views. For instance, in our experiments reported in §5, the API view produces millions of subgraphs, where as the permission and source-sink view produce only thousands. Hence, eq. (3) which ignores the view-specific probability of subgraph occurrences is not suitable in this scenario. Therefore, we propose a novel view-specific negative sampling strategy as described by the equation below:

$$\Pr(sg^v|a) = \sigma(\vec{a}^T \cdot \vec{sg^v}) \prod_{j=1}^{\eta} \mathbb{E}_{sg_j^v \sim \Pr_n(\mathcal{T}^v)} \sigma(-\vec{a}^T \cdot \vec{sg_j^v}) \quad (4)$$

In simpler terms, eq. (4) moves $\vec{a}$ closer to $\vec{sg^v}$ as it occurs in a's context and also moves $\vec{a}$ farther away from $\vec{sg_j^v}$ (which also belongs to view v) as it does not occur its a's context.

4.8 Model dynamics

The trainable parameters of our model are $\Phi^{\mathbb{A}}, \Phi^{\mathcal{L}}, \phi^v$, and p^v. Recall, $\Phi^{\mathbb{A}}$ and $\Phi^{\mathcal{L}}$ are regular embeddings as they are in the input layers and ϕ^vs are hash embeddings. Also, the total number of tokens in the input and output layers would be $|\mathbb{A}|+|\mathcal{L}|$ and $\sum_v K^v$, respectively. Hashing (which is applicable only to ϕ^v) reduces the number of parameters in the output layer from $\sum_v K^v$ to $K^v k + kB^v$ where $B^v << K^v$ (typically, we set $k = [2, 4]$ and $K^v > B^v \cdot 100$).

From the explanations above, it is evident that the computational overhead of using hash embeddings instead of standard embeddings is in the embedding lookup step. More precisely, a multiplication of a $1 \times k$ matrix (obtained from p^v) with a $k \times \delta$ matrix (obtained from ϕ^v) is required instead of a regular matrix lookup to get $1 \times \delta$ subgraph embedding. When using small values of k, the computational overhead is therefore negligible. In our experiments, hash embeddings are marginally slower to train than standard embeddings on datasets with small vocabularies.

5 EVALUATION

We evaluate the efficacy of `apk2vec`'s embeddings with several tasks involving various learning paradigms that include supervised learning (batch and online), unsupervised learning and link prediction. The evaluation is carried out on five different datasets involving a total of 42,542 Android apps. In this section, we first present the experimental design aspects, such as research questions addressed, datasets and tasks chosen pertaining to the evaluation. Subsequently, the results and relevant discussions are presented.

Research Questions. Through our evaluations, we intend to address the following questions:

- How accurate do `apk2vec`'s embeddings perform on various app analytics tasks and how do they compare to state-of-the-art approaches?
- Do multi-view profiles offer better accuracies than single-view profiles?
- Does semi-supervised RL help improving the accuracy of app profiles?
- How does `apk2vec`'s hyperparameters affect its accuracy and efficiency?

Evaluation setup. All the experiments were conducted on a server with 40 CPU cores (Intel Xeon(R) E5-2640 2.40GHz), 6 NVIDIA Tesla V100 GPU cards with 256 GB RAM running Ubuntu 16.04.

Comparative analysis. To provide a comprehensive evaluation, we compare our approach with four baseline approaches, namely, `WLK` [19], `graph2vec` [25], `sub2vec`[9] and `GE-FSG` [45]. Refer to §1 and §3 for brief explanations on the baselines. The following evaluation-specific details on baselines are noted: (i) Since all baselines are unimodal they are incapable of leveraging all the three DGs to yield one unified *apk* embedding. Hence, to ensure fair comparison, we merge all three DGs into one graph and feed them to these approaches. (ii) `sub2vec` has two variants, namely, `sub2vec_N` (which leverages only neighborhood information for graph embedding) and `sub2vec_S` (which leverages only structural information). Both these variants are included in our evaluations, and (iii) For all baselines except `GE-FSG`, open-source implementations provided by the authors are used. For `GE-FSG`, we reimplemented it by following the process described in their original work. Our reimplementation could be considered faithful as it reproduces the results reported in the original work.

Hyperparameter choices. In terms of `apk2vec`'s hyperparameters, we set the following values: $\mathcal{E} = 100, \delta = 64, \alpha = 0.1$ (with decay) and $\eta = 2$. When hash embedding is used $k = 2, B^v = \frac{K^v}{10}$ (for all v). To ensure fair comparison, in all experiments, the hyperparameters of all baseline approaches are maintained same as those of `apk2vec` (e.g., the embedding dimensions of all baselines are set to 64, etc.). In all experiments, for datasets where class labels are available, 25% of the labels are used for semi-supervision during embedding (unless otherwise specified).

5.1 `apk2vec` vs. state-of-the-art

In the following subsections we intend to evaluate `apk2vec` against the baselines on two classification (i.e., batch and online malware

Table 1: Datasets used for evaluations

Task	Data source	# of apps	Avg. nodes			Avg. edges		
			ADG	PDG	SDG	ADG	PDG	SDG
Batch malware detection	Malware: [3, 27] Benignware: [1]	19,944 20,000	783.03	131.73	80.12	2604.28	174.64	94.04
Online malware detection	Malware: [27] Benignware: [1]	5,560 5,000	365.18	63.11	37.885	744.99	66.96	34.67
Malware familial clustering	Drebin [27]	5,560	229.82	69.96	43.41	464.73	72.34	44.57
Clone detection	Clone apps [29]	280	674.71	179.09	94.69	1553.29	182.24	76.64
App recommendation	Googleplay [1]	2,318	2168.88	242.87	154.14	5137.47	348.67	150.87

detection), two clustering (i.e., app clone detection and malware familial clustering) and one link prediction (i.e., app recommendation) tasks. To this end, the datasets reported in Table 1 are used. It is noted that, DGs used in our experiments are much larger than benchmark datasets (e.g., datasets used in [45]) and even some large real-world datasets (e.g., used in [11]). It is noted that some baselines do not scale well to embed such large graphs and they run into *Out of Memory* (OOM) situations.

5.1.1 *Graph classification.*

Dataset & experiments. For batch learning based malware detection task, 19,944 malware from two well-known malware datasets [27] and [3] are used. To form the benign portion, 20,000 apps from Google Play [1] have been used. To perform detection, we first obtain profiles of all these apps using `apk2vec`. Subsequently, a Support Vector Machine (SVM) classifier is trained with 70% of samples and is evaluated with the remaining 30% samples (classifier hyperparamters are tuned using 5-fold cross-validation). This trial is repeated 5 times and the results are averaged.

For online malware detection task, 5,560 malware from [27] and 5,000 benign apps from Google Play are used. In this experiment, the real-world situation where apps stream in over time is simulated as follows: First, *apk*s are temporally sorted according to their time of release (see [23] for details). Thereafter, the embeddings of first 1,000 *apk*s are used to train an online Passive Agressive (PA) classifier. For the remaining 9,560 *apk*s, their embeddings are obtained in an online fashion using `apk2vec` as and when they stream in. These embeddings are fed to the trained PA model for evaluation and classifier update.

For both batch and online settings, to evaluate the efficacy, standard metrics such as precision, recall and f-measure are used.

Results & discussions. The batch and online malware detection results are presented in Table 2. The following inferences are drawn from the tables.

- In batch learning setting, as it is evident from the f-measure, `apk2vec` outperforms all baselines. More specifically, with just 25% labels it is able to outperform the worst and best performing baselines by more than 20% and nearly 2%, respectively. Clearly, this improvement could be attributed to `apk2vec`'s multimodal and semi-supervised embedding capabilities.
- `apk2vec`'s improvements in f-measure are even more prominent in the online learning setting. More specifically, it outperforms the worst and best performing baselines by nearly 35% and more than 5%, respectively. Clearly, this improvement could be attributed to `apk2vec`'s hash embedding capabilities through which it handles dynamically expanding vocabulary of subgraphs.

Table 2: Malware detection (graph classification) results

Technique	Batch			Online		
	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)
`apk2vec`	88.07	**90.41**	**89.22**	**87.90**	**89.73**	**88.81**
`WLK`[19]	**88.15**	86.38	87.25	84.13	82.32	83.22
`graph2vec`[25]	76.96	82.48	79.63	82.55	84.21	83.37
`sub2vec_N`[9]	68.31	69.65	68.98	52.20	56.65	54.33
`sub2vec_S`[9]	66.94	68.36	67.64	53.13	54.98	54.04

- Looking at the performances of baselines, one could see all of them perform reasonably better in the batch learning setting than the online setting. This is owing to their inability to handle vocabulary expansion which renders their models obsolete over time. Besides, none of them posses multi-view and semi-supervised learning potentials which could explain their overall substandard results.
- Due to poor space complexity, `GE-FSG` [45] is unable to handle the large graphs used in this experiment and went OOM during the FSG extraction process. Hence, its results are not reported in the table.

5.1.2 *Graph clustering.*

Dataset & experiments. We now evaluate `apk2vec` on two different graph clustering tasks. Firstly, in the malware familial clustering task, 5,560 apps from Drebin [27] collection are used. These apps belong to 179 malware families. Malware belonging to the same family are semantically similar as they perform similar attacks. Hence, we obtain the profiles of these apps and cluster them into 179 clusters using k-means algorithm. Profiles of samples belonging to same family are expected to end up in the same cluster.

The next task is clone detection which uses 280 apps from Chen *et al.*'s [29] work. The apps in this dataset belong to 100 clone groups, where each group contains at least two apps that are semantic clones of each other with slight modifications/enhancements. Hence, in this task, we obtain the app profiles and cluster them into 100 clusters using k-means algorithm with the expectation that cloned apps end up in the same cluster.

Adjusted Rand Index (ARI) is used as a metric to determine the clustering accuracy in both these tasks.

Results & discussions. The clustering results are presented in Table 3. The following inferences are drawn from the table.

- At the outset, it is clear that `apk2vec` outperforms all the baselines on both these tasks. For familial clustering and clone detection, the improvements over the best performing baselines are 0.07 and 0.01 ARI, respectively.
- Interestingly, unlike malware detection not all the baselines offer agreeable performances in these two tasks. For instance, the ARIs of `sub2vec` and `GE-FSG` are too low to be considered as practically viable solutions. Given this context, `apk2vec`'s performances show that its embeddings generalize well and are task-agnostic.

5.1.3 *Link prediction.*

Dataset & experiments. For this task, we constructed an app recommendation dataset consisting of 2,318 apps downloaded from Google Play. We build a recommendation graph $\mathcal{R}$, with these apps as nodes. An edge is placed between a pair of apps in $\mathcal{R}$, if Google Play recommends one of them while viewing the other. With this graph, we follow the procedure mentioned in [8] to cast app

Table 3: Malware familial clustering and clone detection (graph clustering) results

Technique	Familial clustering (ARI)	Clone detection (ARI)
apk2vec	**0.5124**	**0.8360**
WLK[19]	0.3279	0.7766
graph2vec[25]	0.4441	0.8272
sub2vec_N[9]	0.0374	0.1801
sub2vec_S[9]	0.0945	0.0454
GE-FSG [45]	OOM	0.0171

Table 4: App recommendation (link prediction) results

Technique	AUC (P = 10%)	AUC (P = 20%)	AUC (P = 30%)
apk2vec	**0.7187**	**0.7347**	**0.7236**
WLK[19]	0.6643	0.6865	0.6805
graph2vec[25]	0.6830	0.7043	0.6876
sub2vec_N[9]	0.5446	0.5808	0.5403
sub2vec_S[9]	0.5206	0.5632	0.5631

recommendation as a link prediction problem. That is, P, a subset of edges (chosen at random) are removed from $\mathcal{R}$, while ensuring that this residual graph $\mathcal{R}'$ remains connected. Now, given a pair of nodes in $\mathcal{R}'$, we predict whether or not an edge exists between them. Here, endpoints of edges in P are considered as positive samples and pairs of nodes with no edge between them in R are considered as negative samples. We perform the experiment for P = {10%, 20%, 30%} of total number of edges in $\mathcal{R}$.

Area under the ROC curve (AUC) is used as a metric to quantify the efficacy of link prediction.

Results & discussions. The results of the app recommendation task are presented in Table 4 from which the following inferences are drawn.

- For all values of P, apk2vec consistently outperforms all the baselines. Also, apk2vec's margin of improvement over baselines is consistent and much higher in this task than graph classification and clustering tasks. For instance, it improves best baseline performances by 3 to 4% across all P values.
- It is noted that link prediction task does not involve any semi-supervision and hence all this improvement could be attributed to apk2vec's multi-view and data-driven embedding capabilities.

In sum, apk2vec consistently offers the best results across all the five tasks reported above. This illustrates that apk2vec's embeddings are truly task-agnostic and capture the app semantics well.

5.2 Single- vs. multi-view profiles

In this experiment, we intend to evaluate the following: (i) significance of three individual views used in apk2vec, (ii) significance of concatenating app profiles from individual views (i.e. linear combination), and (iii) whether non-linear combination of multiple views is better than (i) and (ii).

Dataset & experiments. To this end, we use the clone detection experiment reported in §5.1.2. First, we build the app profiles (i.e., 64-dimensional embedding) with individual views (i.e., only one output layer is used in skipgram). Clone detection is then performed with each view's profile. Also, we concatenate the profiles from three views to obtain a 192-dimensional embedding and perform clone detection with the same. Finally, the regular 64-dimensional multi-view embedding from apk2vec is also used for clone detection. Due to space constraints, from this experiment onwards, only WLK is considered for comparative evaluation, as it offers the most consistent performance among the baselines considered.

Table 5: Clone detection results: single- vs. multi-view *apk* profiles

Technique	Views				
	APIs(ARI)	Perm.(ARI)	Src-sink(ARI)	concat.(ARI)	multi-view(ARI)
apk2vec	0.8208	0.7855	0.7953	0.8325	**0.8360**
WLK[19]	0.8078	0.7382	0.7479	0.7766	-

Table 6: Impact of semi-supervised embedding on malware detection efficacy

Labels(%)	apk2vec			WLK[19]		
	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)
0	92.97	95.43	94.19	95.83	91.04	93.38
10	95.44	97.11	96.27	-	-	-
20	95.91	96.76	96.33	-	-	-
30	**96.59**	**97.28**	**96.93**	-	-	-

Results & discussions. The results of this experiment are reported in Table 5. The following inferences are drawn from the table:

- At the outset, it is evident that individual views are capable of providing reasonable accuracies (i.e., 0.70+ ARI). This reveals that individual views possess capabilities to retain different, yet useful program semantics.
- Out of the individual views, as expected, API view yields the best accuracy. This could be attributed to fact that this view extracts much larger number of high-quality features compared to the other two views. Owing to this well-known inference, many works in the past (e.g., [23, 27, 32, 36, 37]) have used them for a variety of tasks (incl. malware and clone detection). Also, source-sink view extract too few features to perform useful learning. In other words, it ends up underfitting the task. These observations are inline with the existing work on multi-view learning such as [24].
- In the case of WLK, API profiles gets a very high accuracy and when they are concatenated with other views, the accuracy is reduced. We believe this is due to the inherent linearity in this mode of combination i.e., views do not complement each other.
- Interestingly, in the case of apk2vec, concatenating profiles from individual views yields higher accuracy than using just one view. This reveals that using multiple views is indeed offering richer semantics and helps to improve accuracy. However, concatenation could only facilitate a linear combination of views and hence is yielding slightly lesser accuracy than apk2vec's multi-view profiles. This illustrates the need for performing a non-linear combination of the semantic views.

5.3 Semi-supervised vs. unsupervised profiling

In this experiment, we intend to study the impact of using the (available) class labels of apps during profiling.

Dataset & experiments. Here, we use the same dataset which was used for online malware detection reported in §5.1.1. However, in order to study the impact of varying levels of supervision, we use

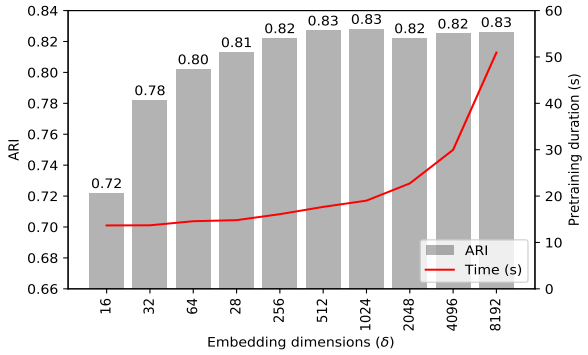

Figure 3: Sensitivity w.r.t embedding sizes

labels for the following percentages of samples: 10%, 20%, and 30%. Profiles for apps are built with aforementioned levels of supervision and for each setting an SVM classifier is trained and evaluated for malware detection (other settings such as train/test split are similar to §5.1.1).

Results & discussions. The results of this experiment are reported in Table 6, from which the following inferences are made:

- One could observe that leveraging semi-supervision during apk2vec's embedding is indeed helpful in improving the accuracy of the downstream task. For instance, by using labels for merely 10% of samples help to improve the accuracy for malware detection by more than 2%.
- Clearly, using 30% labels yields better results than using just 10% and 20% labels. This illustrates the fact that the more the supervision is, the better the accuracy would be.
- In the case of WLK which uses handcrafted features, one could not use labels or other form of supervision to obtain graph embeddings. Hence, without any supervision, it performs reasonably well to obtain an f-measure of more than 93%. However, apk2vec with even just 10% supervision is able to outperform WLK significantly i,e., by nearly 3% f-measure.

5.4 Parameter Sensitivity

The apk2vec framework involves a number of hyperparameters such as embedding dimensions (δ), number of hash buckets (B) and number of hash functions (k). In this subsection, we examine how the different choices of δ affects apk2vec's accuracy and efficiency, as it is the most influential hyperparameter. For the sensitivity analysis with respect to B and k, we refer the reader to the online appendix at [22].

Dataset & experiments. Here, the clone detection experiment reported in §5.1.2 is reused. The sensitivity results are fairly consistent on the remaining tasks reported in §5.1. Except for the parameter being tested, all other parameters assume default values. Embeddings' accuracy and efficiency are determined by ARI and pretraining durations (averaged over all epochs), respectively.

Results & discussions. These results are reported in Figure 3 from which the following inference are drawn.

- Unsurprisingly, the ARI values increase with δ. This is understandable as larger embedding sizes offer better room for learning more features. However, the performance tends to saturate once the δ is around 500 or larger. This observation is consistent with other graph substructure embedding approaches [7, 8, 11].
- Also, the average pretraining time taken per epoch increases with δ. This is expected, since increasing δ would result in an exponential increase in skipgram computations. This is reflected in the exponential increase in pretraining time (especially, when $\delta > 500$). This analysis helps in understanding the trade-off between apk2vec's accuracy and efficiency for a given dataset and picking the optimal value for δ.

6 CONCLUSIONS

In this paper, we presented apk2vec, semi-supervised multimodal RL technique to automatically build data-driven behavior profiles of Android apps. Through our large-scale experiments with more than 42,000 apps, we demonstrate that profiles generated by apk2vec are task agnostic and outperform existing approaches on several tasks such as malware detection, familial clustering, clone detection and app recommendation. Our semi-supervised multimodal embeddings also prove to provide significant advantages over their unsupervised and unimodal counterparts. All the code and data used within this work is made available at [22].

REFERENCES

[1] Google Play Market. https://play.google.com/store.
[2] App brain Android market statistics. https://www.appbrain.com/stats.
[3] Virus share malware repository. https://virusshare.com.
[4] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.
[5] Svenstrup, Dan Tito, Jonas Hansen, and Ole Winther. "Hash embeddings for efficient word representations." Advances in Neural Information Processing Systems. 2017.
[6] Le, Quoc, and Tomas Mikolov. "Distributed representations of sentences and documents." Proceedings of the 31st International Conference on Machine Learning (ICML-14). 2014.
[7] Perozzi, Bryan, et al. "Deepwalk: Online learning of social representations." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2014.
[8] Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016.
[9] Adhikari, Bijaya, et al. "Sub2Vec: Feature Learning for Subgraphs." Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, Cham, 2018..
[10] Narayanan, Annamalai, et al. "subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs." International Workshop on Mining and Learning with Graphs. (2016).
[11] Yanardag, Pinar, and S. V. N. Vishwanathan. "Deep graph kernels." Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2015.
[12] Yanardag, Pinar, and S. V. N. Vishwanathan. "A structural smoothing framework for robust graph comparison." Advances in Neural Information Processing Systems. 2015.
[13] Ribeiro, Leonardo FR, et al. "struc2vec: Learning node representations from structural identity." Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2017.
[14] Pan, Shirui, et al. "Tri-party deep network representation." in IJCAI, 2016, pp. 1895–1901.
[15] Goyal, Palash, and Emilio Ferrara. "Graph Embedding Techniques, Applications, and Performance: A Survey." arXiv preprint arXiv:1705.02801 (2017).
[16] Niepert, Mathias, et al. "Learning convolutional neural networks for graphs." Proceedings of the 33rd annual international conference on machine learning. ACM. 2016.
[17] Lee, John Boaz, et al. "Deep Graph Attention Model." arXiv preprint arXiv:1709.06075 (2017).
[18] Lee, John Boaz, and Xiangnan Kong. "Skip-graph: Learning graph embeddings with an encoder-decoder model." (2016).
[19] Shervashidze, Nino, et al. "Weisfeiler-lehman graph kernels." Journal of Machine Learning Research 12.Sep (2011): 2539-2561.

[20] Shervashidze, Nino, et al. "Efficient graphlet kernels for large graph comparison." Artificial Intelligence and Statistics. 2009.
[21] Wei, Fengguo, et al. "Deep Ground Truth Analysis of Current Android Malware." Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment. 2017.
[22] apk2vec website: https://sites.google.com/view/apk2vec/home
[23] Narayanan, Annamalai, et al. "Context-aware, adaptive, and scalable android malware detection through online learning." IEEE Transactions on Emerging Topics in Computational Intelligence 1.3 (2017): 157-175.
[24] Narayanan, Annamalai, et al. "A multi-view context-aware approach to Android malware detection and malicious code localization." Empirical Software Engineering (2017): 1-53.
[25] Narayanan, Annamalai, et al. "graph2vec: Learning Distributed Representations of Graphs." International Workshop on Mining and Learning with Graphs. (2017).
[26] Shi, Qinfeng, et al. "Hash kernels for structured data." Journal of Machine Learning Research 10.Nov (2009): 2615-2637.
[27] Arp, Daniel, et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." Ndss. Vol. 14. 2014.
[28] Aggarwal, Charu C., and Chandan K. Reddy, eds. Data clustering: algorithms and applications. CRC press, 2013.
[29] Chen, Kai, Peng Liu, and Yingjun Zhang. "Achieving accuracy and scalability simultaneously in detecting application clones on android markets." Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.
[30] Au, Kathy Wain Yee, et al. "Pscout: analyzing the android permission specification." Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012.
[31] Arzt, Steven, Siegfried Rasthofer, and Eric Bodden. "Susi: A tool for the fully automated classification and categorization of android sources and sinks." University of Darmstadt, Tech. Rep. TUDCS-2013-0114 (2013).
[32] Zhang, Mu, et al. "Semantics-aware android malware classification using weighted contextual api dependency graphs." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.
[33] Gascon, Hugo, et al. "Structural detection of android malware using embedded call graphs." Proceedings of the 2013 ACM workshop on Artificial intelligence and security. ACM, 2013.
[34] Tian, Ke, et al. "Detection of repackaged android malware with code-heterogeneity features." IEEE Transactions on Dependable and Secure Computing (2017).
[35] Watanabe, Takuya, et al. "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps." Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on. IEEE, 2017.
[36] Fan, Ming, et al. "DAPASA: detecting android piggybacked apps through sensitive subgraph analysis." IEEE Transactions on Information Forensics and Security 12.8 (2017): 1772-1785.
[37] Fan, Ming, et al. "Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis." IEEE Transactions on Information Forensics and Security (2018).
[38] Chen, Kai, et al. "Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale." USENIX Security Symposium. Vol. 15. 2015.
[39] Chen, Kai, et al. "Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios." Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016.
[40] Gärtner, Thomas, Peter Flach, and Stefan Wrobel. "On graph kernels: Hardness results and efficient alternatives." Learning Theory and Kernel Machines. Springer, Berlin, Heidelberg, 2003. 129-143.
[41] Borgwardt, Karsten M., and Hans-Peter Kriegel. "Shortest-path kernels on graphs." Data Mining, Fifth IEEE International Conference on. IEEE, 2005.
[42] Li, Wenchao, et al. "Detecting similar programs via the weisfeiler-leman graph kernel." International Conference on Software Reuse. Springer, Cham, 2016.
[43] Narayanan, Annamalai, et al. "Contextual Weisfeiler-Lehman graph kernel for malware detection." Neural Networks (IJCNN), 2016 International Joint Conference on. IEEE, 2016.
[44] Wold, Svante, Kim Esbensen, and Paul Geladi. "Principal component analysis." Chemometrics and intelligent laboratory systems 2.1-3 (1987): 37-52.
[45] Nguyen, Dang, et al. "Learning graph representation via frequent subgraphs." Proceedings of the 2018 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2018.
[46] Ivanov, Sergey, and Evgeny Burnaev. "Anonymous Walk Embeddings." International conference on machine learning. 2018.
[47] Yan, Xifeng, and Jiawei Han. "gspan: Graph-based substructure pattern mining." Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on. IEEE, 2002.
[48] Rehurek, Radim, and Petr Sojka. "Software framework for topic modelling with large corpora." In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. 2010.
[49] Rousseau, François, Emmanouil Kiagias, and Michalis Vazirgiannis. "Text categorization as a graph classification problem." Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Vol. 1. 2015.