A high efficient Two-Hierarchies tensor-embedding -aware App clone detection System

Abstract—App homology analysis tasks on binary codes is expected to be scalable and effective. However, most works have a complex trained process or are slow and less capable of catching new increasing apps. We use two hierarchies' embedding model to obtain concise and accurate feature vector, which represents the binary code of CFG in the app. In our research, firstly, we implement a novel first-hierarchy embedding prototype for the compiling CFG of app's binary code called 5UD-CFG that preserves the global CFG structures. Secondly, we propose a compact, clustering and update algorithm according to the proposed second-hierarchy tensor embedding model to obtain a more concise and accurate feature numeric vector 3TU-CFG for app's binary code. Thirdly, Empirical experiments prove that effectiveness of two hierarchies CFG embedding on the malware app detection. 3TU-CFG can accelerate the existing works' preprocessing time from $4\times$ down to $76000\times$ and at least reduce the half of search time. At the same time, the detected accuracy approximates 99%.

Index Terms—CFG embedding, Malware detection, tensor embedding model

1 Introduction

The study of the app homology and the app classification has recently emerged as a major catalyst for collectively understanding the behavior of complex app in the mobile. With the rapidly increasing use of smartphone, code clone, malware code injection bring several serious finance threats. Thousands, and millions of apps have been used around the world from various app markets, such as PP assistant, Android, Google Play, SnapPea and tens of smaller third-party markets. With this prosperity, app homology can be used in the malware detection, plagiarism detection, etc. More than 85% of Android malware rely on app clone to spread to a large number of genuine users [1], for example, crack paid apps to bypass payment function, modify the advertisement libraries, or even insert various malicious functions, which leads to a serious security problem for the mobile application. Specifically, app developers and security researchers urgently need to find an effective app homology method to satisfy the requirement of fast increasing apps.

However, discovering the similar app or derivative app is like distinguishing beans from a mount of jumble beads, even when we are dealing with the known app. To address this critical problem, primitive app researchers statically scans an app for known codes and operations for suspicious activities [2]. The problem here is that the static approach does not work on the new threats. Recent researchers have been actively developing techniques to automatically analyze and detect app homology in the mobile markets. There are several studies that propose the app clone technology and scalable app classification. Detection based on string [3], detection based on token[4][5], detection based on hash [6][7], detection based on the program dependence graph [8] [9] and detection based on abstract syntax tree [10] [11] [12] have shown their scalability to handle thousands and millions of code. However, some approaches generate too many false negative at handling the classification of malware with imperceptible change, such as sting-based detection, token-based detection. Some approaches are not scalable that can not handle billions of opcodes in multiple app market, such as the program dependence graph. The dynamic method can be circumvented by an app capable of fingerprinting the testing environment [13], but the dynamic analysis can be heavyweight, which makes it hard to explore all execution paths of an app.

There has been several specific challenges must be tackled:

1) There has been an primitive challenge on designing a feasible model to accurately represent the scalable apps at the binary codes level. The binary code of app is a very complicated source data. They appear on different markets, and most apps' binary codes are confused. We need extract more considerable features of the binary code to denote the binary code.
2) How handle the millions and billions features of app binary codes in multiple markets to accurately analysis the app homology? There is an challenging that we need to accurately find homologous apps in the mass quantity of features of apps.
3) There has been facing an challenge to update the feature data for the rapid increasing apps.

Since the update of app feature data needs to relearn all apps' feature including preceding app features in the existing works. However, most existing update approaches are expensive works, and they cost too much additional time and space. We should design a update system that can incrementally and timely update the app feature data to analysis more novel app's homology.

Our goal is to address these challenges by proposing a suitable embedding system to transform the binary code of app to a low-dimensional feature vector. Since apps' binary codes are complex, we need to extract more concise and specific storable features represented the apps' binary codes to be easier for the app homology analysis. First, We know that the main content of a app is its smali functions. The control flow graph of the function can greatly preserve the opcode structure. We need to design a embedding approach to preserve both the opcode structure and numeric characteristics of original binary code of the app based on the control flow graph. Second, the embedding features that are extracted from the CFG of app's binary codes are scalable, we need to make the feature dimensionality reduced and compressed. The tensor computations have a great effect on data reduction. We need to propose a tensor embedding model and a compress approach to get more concise and accurate app 's feature data. Third, based on the proposed tensor embedding model, it is easier to propose a incremental update system of the app feature data. Moreover, we also need to prove the monotonicity and validity of the embedding feature theoretically and practically.

In this paper, our interdisciplinary study focuses on leveraging two hierarchies embedding model to obtain the feature numeric vector based on the control flow graph (CFG) of app's binary code to handle the app homology analysis and the app classification. The two hierarchies embedding model shows a more effective and accurate app homology strategy than current other works. The main contributions of our work are summarized as follows:

1) We develop the first-hierarchy line embedding model for the extracted CFG with five vertex features by decompiling the app, which is called as 5UD-CFG. Moreover, we also prove the monotonicity of the proposed embedding feature 5UD-CFG. We can theoretically prove that first-hierarchy embedding feature can uniquely represent a CFG.
2) We design the second-hierarchy 3TU-CFG tensor embedding model based on the extracted first-class embedding feature 5UD-CFG. Then we propose an special tensor-SVD factorization algorithm to decompose tensor model, which compress the 5UD-CFG as the 3TU-CFG. This is second-hierarchy embedding process that compressing and clustering the first-hierarchy embedding feature matrixes to greatly accelerate the efficiency of the match and the classification.
3) We propose an incremental tensor decomposed algorithm to match the compressing and clustering learning process of tenor embedding model. Compared with the existing work for the update process of the app homology analysis data, the efficiency is greatly improved $2\times$ to $18000\times$.
4) Our evaluation shows that two hierarchies embedding process obtained feature vector can accelerate the preparation process $4\times$ to $76000\times$ faster than prior works, i.e., Gemini, Genius, Centroid. At the same time, the search and detection time run at least 1 to 2 orders of magnitude faster than prior time even though the prior search time is very minor. Evolution also shows that finding a unknown function less than $4.6\times10^{(-9)}$ seconds in a thousand apps and 0.1 seconds in 10^8 apps.

The remainder of the paper is structured as follows. Background of the app classification and basis tensor computation are given in Section 2. In Section 3, we introduce the problem definition that we need to solve. In Section 4, we show the overview solution for proposed two-hierarchies embedding model. In Section 5, we propose the first-hierarchy line embedding model, and prove the monotonicity of the proposed first-hierarchy embedding feature 5UD-CFG. In Section 6, we propose the second-hierarchy tensor embedding model based on the first-hierarchy embedding feature 5UD-CFG. Moreover, we give a learning algorithm to compress and clustering the first-class embedding feature unitary matrix to obtain a more concise embedding feature 3TU-CFG, which can be effective to handle the app homology match and detection. Section 7 illustrates the experiments to verify the efficiency and accuracy of two hierarchies embedding process. We conclude with remarks on future work in Section 8.

2 Related Work

We discuss the closest related work in this Section. We focus on approaches using code similarity to search for known malware apps. There are many other approaches that aim at finding unknown malware.

Binary-level malware detection. One common approaches is that trace-based approach [14] captures execution sequences as features for code similarity checking, which can detect the CFG changes. However, this approach does not accurately find the malware codes across different architectures. Static birthmarks [15][16] are usually the characteristics in the code that cannot easily be modified such as constant values in field variables, a sequence of method calls, an inheritance structure and used classes. Rendezvous

[15] first explored the code search in binary code. however, it has two limitations. It relies on ngram features to improve the search accuracy. Secondly, it decomposes the whole CFG of a function into subgraphs. Jannik [17] [18] propose a system to derive bug signatures for known bugs. They compute semantic hashes for the basic blocks of the binary. When can then use these semantics to find code parts in the binary that behave similarly to the bug signature, effectively revealing code parts that contain the bug. However, it can not be scalable. Deqiang Fu et al. [19] proposed a control flow graph-based malware detection method. They extract features named "code chunks" from the control flow graph, then use the classification and regression tree (CART) algorithm to classify these features. The method can obtain 96.3% classification accuracy.

Malicious app detection. Most existing approaches identify malicious apps [20][21] typically rely on heavyweight static or dynamic analysis techniques, and cannot detect the unknown malware whose behavior has not been modeled a priori. PiggyApp[22][23] utilize the features (permissions, API, etc.) identified from a major component shared between two apps to find other apps also including this component, then clusters the rest part of these apps' code and samples from individual clusters to manually determine whether the payloads are indeed malicious. Yao [24] present a new malware detection method that automatically divides a function call graph into community structures. The features of these community structures can then be used to detect malware. Arp et al. [25] proposed a lightweight detection method called DREBIN. This method can extract eight types of static features from Android applications. These features are then input to vector space models for classification by a support vector machine (SVM) algorithm. Their method achieved 94% detection accuracy on $123,453$ Android applications.

Dynamic malware detection Another effective approach for malware detection that is resistant to code confusion mechanisms is dynamic analysis. In dynamic analysis, applications are executed in a virtual environment. Runtime information is recorded to generate dynamic features for malware detection. For example, ANDRUBIS [25][26] dynamically examined the operations for over 1 million apps in four years, which is an off-line analyzer for recovering detailed behavior of individual malicious apps. Enck et al. [27] proposed TaintDroid, a virtualization-based malware detection method that can trace the flow of sensitive information. In an evaluation of 30 Android applications, TaintDroid found 20 applications had misused users private information. However, it has a worse limitation that can not be scalable to handle millions and thousands of apps. Blanket-execution [28] uses the dynamic run-time environment of the program as features to conduct the code search. This approach can defeat the CFG changes, but it is only evaluated in a single architecture. DroidScope [29] is also a virtualization-based malicious code detection method. It can identify malware by analyzing semantics features of operating system level and Java level.

3 Problem Decription

In this paper, we aim at the problem: *How to achieve accuracy and scalability simultaneously in analyzing the app homology on Android Market?*

In order to obtain an effective the app homology analysis, we need to obtain an informative low-dimensional representation from the extracted CFG data. Refine such representations is a non-trivial task due to the following difficult points:

1) Extracted feature of CFG of app binary codes need to preserve both the characteristics of basic blocks and the call structure. Since CFG is a directed graph in most cases, and their corresponding vertexes have own unique features, which needs to be extracted from the basic blocks. How can we carefully deal with such graph property in the representation learning process.
2) The mass quantity of apps have the millions and billions CFG features that leads to the scalable graph match problem. Graph match essentially is a $NP-problem$. We need to find a practicable approach to accurately reduce the CFG feature. Moreover, the availability of the reduced low-dimension feature should be proved theoretically and practically.
3) The increasing novel apps leads to update the previous feature data timely. Since if app samples are changed, the feature data also needs to be re-obtained. We need to find a incremental computation algorithm that just needs compute the novel app features rather than compute pervious app features.

To address those critical problems, we extract the numeric and structural features in the binary code of Android apps based on the following basic definition. We formally define the problem of large-scale app homology analysis using the CFG extraction with basic-block feature, the first-hierarchy CFG embedding model considering the first-order call structure of CFG and the second-order call structure of CFG, and the second-hierarchy embedding tensor model based on the first-hierarchy embedding CFGs' feature vector.

Definition 1. *(CFG extraction with basic-block feature).* We extract the control flow graph that each vertex is with five unique feature, or 5UD-CFG in short, is a directed graph $G =< V, E >$, where V is a set of basic blocks in a function$E \subseteq V \times V$ is a set of edges representing connections between these basic blocks. Each vertex has a unique five numeric elements based on the statistical feature of the basic block. The corresponding feature of a vertex represents as a vector $v = \vec{x} =< num,\ squ,\ in,\ out,\ loop >$. num is the sequence number of the basic block in the CFG, squ is the number of opcodes for a basic block, the in is the number of calls, out is the number of basic blocks that is called by other vertexes. $loop$ is the number of loops of a vertex. Each vertex has a unique weight $\omega \in \aleph$. Fig.1 shows a extracted real CFG.

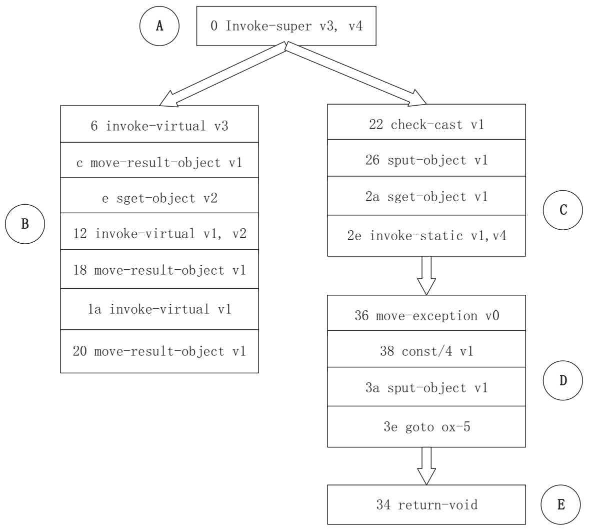


Fig. 1. An example of real CFG

Given a set of training CFGs $C = [c_1,\ c_2,\ c_3,\ ...,\ c_{N_T}] \in R^{m\times 5\times N_T}$ where $c_i \in R^{m\times 5}$ is a CFG, m is the number of vertexes in a CFG, and N_T is the number of all CFGs. Embedding aims to map the graph data into a low-dimensional latent space, where combines all vertexes of a CFG represented as a low-dimensional vector. As this explained, both vertex features and the graph structure are essential to be preserved.

Then we define the first-hierarchy 5UD-CFG embedding model.

Definition 2. *(First-hierarchy 5UD-CFG embedding)* Given a CFG that each vertex has five features denoted as $G = (V, E)$, CFG embedding aims to learn a mapping function $f : \sum_i^m x_i \mapsto y_i \in R^d$, where $d << |V|$. The objective of function is to distinguish the similarity between the feature of a CFG y_i and the feature of another CFG y_j, explicitly preserve the first-order and second-order call structure of v_i and v_j. We denote the result of first-hierarchy CFG embedding as 5UD-CFG.

We consider *first-order call structure* and *second-order call structure* to show CFG structure based on the definition of the first-hierarchy 5UD-CFG embedding.

Definition 3. *(First-Order Call Structure)* The first-order call structure describes the directed pairwise structure between vertexes. For any pair of vertexes, if $l_{i,j} = 1i,\ j = \{1,\ 2,\ ...,\ n\}$, there exists an directed first-order call structure between the vertex v_i and v_j. n is the total number of original CFG. Otherwise, the first-order call structure between the vertex v_i and v_j is 0.

The first-order call structure is necessary for the CFG embedding to imply the first-order inherit or invocation between two real basic blocks in real binary codes. The result of the basic block will be directed used by the directional vertexes.

Definition 4. *(Second-Order Call Structure)* The second-order call structure between a pair of vertexes describes the call structure of neighborhoods. Let $N_u = l_{u,1}$, ..., $l_{u,|V|}$ denote the first-order call structure between $v_u,\ u = \{1,\ 2,\ ...,\ |V|\}$ and other vertexes. $|V|$ is the number of neighborhoods. The second-order call structure is determined by the similarity of N_u and N_v, $v = \{1,\ 2,\ ...,\ n\}$.

Intuitively, the second-order call structure assumes that if two vertexes share many common basic blocks, they tend to be have more connections. Such an assumption has been proved reasonable in many fields [30]. A father basic block is most frequently called in the other basic blocks.

Definition 5. *(Second hierarchy embedding tensor model)* To represent all CFGs' features of app samples, the proposed tensor embedding model can be written as a three-order tensor model $A^{I_f * I_c * I_1}$, which I_f, I_c indicate the feature vector of CFG and the number of all CFGs. For embedding CFG, $I_f = 5$ and $I_c = tn$, tn is the training number of CFG. tn can be expanded with the development of apps. I_1 is the label of the app.

4 Solution Overview

In this section, we present the key idea of our solution to the app homology embedding problem, which can be scalable and accurate. In this approach, we assume the binary code of app's function f can be represented by our proposed 3TU-CFG.

We propose the two hierarchies embedding mapping model for the extracted CFG from app's binary codes. The two hierarchies embedding model satisfy several highlights: first, it extracts the traditional control flow graph that each vertex has five unique features. This CFG with feature is referred to the raw feature, from a decompiled binary function of app. It remains the characteristics of raw binary code of app. Second, the first-hierarchy 5UD-CFG embedding is able to preserve both the *first-order call structure* and the *second-order call structure* between vertices of CFG. Third, the second-hierarchy tensor embedding can scale for very large database, say millions and billions apps detection in very short time.

The proposed methods including the following main steps, as shown in Fig. 2: 1) *original CFG extraction with vertex features*, 2) *5UD-CFG embedding feature vector generation*, 3) *3TU-CFG tensor compress* *embedding*, 4) *3TU-CFG tensor clustering embedding*. The first step and the second step are the first-hierarchy embedding process. They aim at extracting the original control flow graph that each vertex of CFG has five features, and embedding them into monotonous feature vector (Section 4). The third step and the fourth step are the second-hierarchy embedding process. They use the tensor factorization learning algorithm to further embedding the feature vector that is generated by the first-hierarchy embedding process to a more low-dimensionality feature vector (Section 5). Finally, given a app, we use the Locality Sensitive Hashing (LSH) [31] to find the its most similarity app in the trained databases. Since each app is decompiled into several functions, and each function is embedded into a low-dimensionality unique and accurate feature vector in two hierarchies embedding process. We can directly apply LSH to conduct efficient matches and searches. The detailed of each steps will be discussed in the following section.

In the first-hierarchy 5UD-CFG embedding process, we show the monotonicity of 5UD-CFG feature vector, which give a high scalability and accurate. When a function changes a little, its 5UD-CFG will not change a lot. Make sure two similarity apps have a little different, and two different apps have a large different. In the second-hierarchy 3TU-CFG learning embedding process, we show the embedding reversibility and the monotonicity of 3TU-CFG, which is a more concise embedding feature vector to monotonously map the original apps' function. The compress and clustering algorithm based on the tensor embedding model give a further significant scalability and accurate.

5 First-hierarchy 5UD-CFG embedding model

In this section, we introduce the detailed approaches how to generate 5UD-CFG from Android apps. The traditional CFG is the common feature used in bug search. Moreover, different from other attributes on the basic blocks, such as I/O pairs and statics features [32] [33], are more accuracy matching. Following the idea of the traditional CFG extraction, this paper utilizes the control flow graph with different basic-block level attributes about five features based, it called as the 5UD-CFG. We also prove the monotonicity of the feature of the 5UD-CFG, a feature of 5UD-CFG represents a CFG.

5.1 Disassembly of Android Application and original CFG extraction

In our system, the pretreatment of an application consists of disassembling the application and extracting opcode sequences. A code file is a dex file that can be transformed into smali files, where each smali file represents a single class and contains methods of such a class. Each method contains instructions and each instruction consists of a single opcode and multiple operands.

After preparation, including downloading all apps from multiple markets and extracting methods from the apps, we encode a projection form of CFG to get the unique feature of the function of an app.

CFG is the control flow graph of a method. Each vertex in a CFG corresponds to a basic block in the method. A basic block is a straight-line piece of code with one entry point and one exit point. Jump targets start a block, and jump end a block. Directed edges are used to represent jumps in the control flow.

Before discussing the CFG embedding, we extract the feature of basic block for each vertex in a CFG.

Fig. 3 shows real CFG of a function in a class. A vertex represents a basic block, and a edge represents a call Link between two basic block. A basic block is a set of opcodes. The outgoing edges of a vertex A represents the basic block A is called by other basic blocks. The input edges of a vertex A represents the the basic block A calls other basic blocks.

5.2 5UD-CFG embedding

The main idea of 5UD-CFG embedding is to encode a CFG as a 5-dimensional vector. We use a generic nonlinear mapping to update the embedding result based on the CFG topology.

A 5UD-CFG can be viewed as a set of vertexes connected by edges based on the CFG topology. We need to train the vertex weight of each vertex (basic block). The vertex weight of each vertex is used to combine all vertex to represent the CFG. The embedding vector monotonously represent the CFG. We define the feature of a 5UD-CFG based on the 3D-CFG [34].

Definition 6. The feature of 5UD-CFG is a vector $< f_n, f_s, f_i, f_o, f_l >$.

$$
\begin{aligned}
f_n &= \frac{\sum_{e(j,k)\in CFG}(\omega_j n_j + \omega_k n_k)}{\omega},\\
f_s &= \frac{\sum_{e(j,k)\in CFG}(\omega_j s_j + \omega_k s_k)}{\omega},\\
f_i &= \frac{\sum_{e(j,k)\in CFG}(\omega_j i_j + \omega_k i_k)}{\omega},\\
f_o &= \frac{\sum_{e(j,k)\in CFG}(\omega_j o_j + \omega_k o_k)}{\omega},\\
f_l &= \frac{\sum_{e(j,k)\in CFG}(\omega_j l_j + \omega_k l_k)}{\omega},\\
\omega &= \sum_{e(j,k)\in 5UD-CFG}(\omega_j + \omega_k).
\end{aligned}
$$

where $e(j, k)$ is an edge in CFG. This edge connects two vertexes j and k, $j,\ k = \{1,\ 2,\ ...,\ n\}$.

The key factor of feature vector of the 5UD-CFG embedding is the vertex weight ω, and we need to

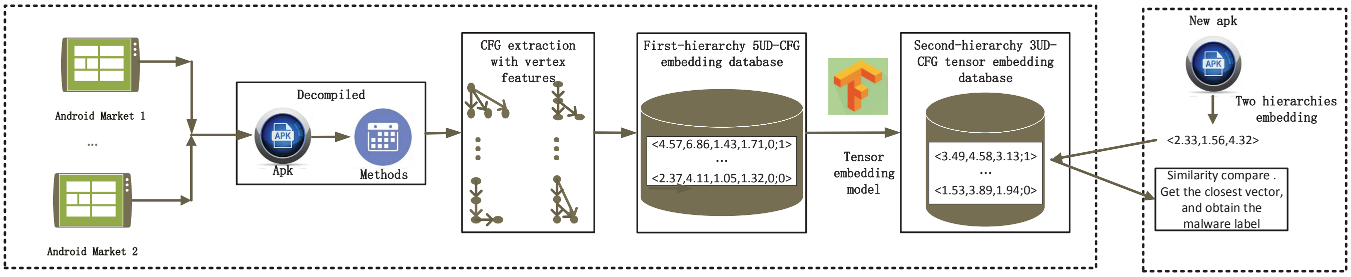
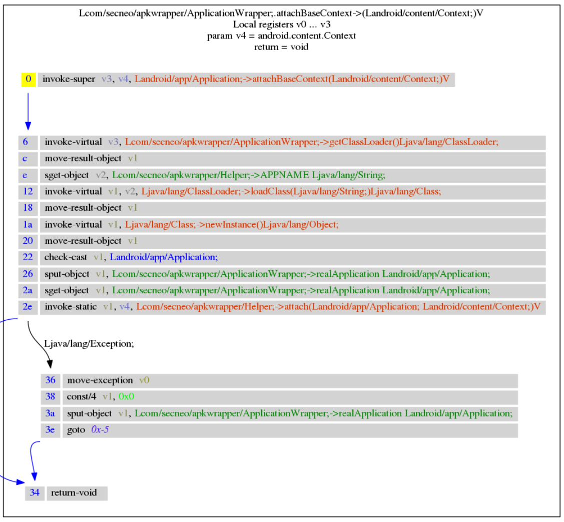

Android Market 1
...
Android Market 2
Decompiled
Apk
Methods
CFG extraction with vertex features
First-hierarchy 5UD-CFG embedding database
<4.57,6.86,1.43,1.71,0;1>
...
<2.37,4.11,1.05,1.32,0;0>
Tensor embedding model
Second-hierarchy 3UD-CFG tensor embedding database
<3.49,4.58,3.13;1>
...
<1.53,3.89,1.94;0>
New apk
Two hierarchies embedding
<2.33,1.56,4.32>
Similarity compare . Get the closest vector, and obtain the malware label

Fig. 2. Overview



```
Lcom/secneo/apkwrapper/ApplicationWrapper;.attachBaseContext->(Landroid/content/Context;)V
Local registers v0 ... v3
param v4 = android.content.Context
return = void

0   invoke-super v3, v4, Landroid/app/Application;->attachBaseContext(Landroid/content/Context;)V

6   invoke-virtual v3, Lcom/secneo/apkwrapper/ApplicationWrapper;->getClassLoader()Ljava/lang/ClassLoader;
c   move-result-object v1
e   sget-object v2, Lcom/secneo/apkwrapper/Helper;->APPNAME Ljava/lang/String;
12  invoke-virtual v1, v2, Ljava/lang/ClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
18  move-result-object v1
1a  invoke-virtual v1, Ljava/lang/Class;->newInstance()Ljava/lang/Object;
20  move-result-object v1
22  check-cast v1, Landroid/app/Application;
26  sput-object v1, Lcom/secneo/apkwrapper/ApplicationWrapper;->realApplication Landroid/app/Application;
2a  sget-object v1, Lcom/secneo/apkwrapper/ApplicationWrapper;->realApplication Landroid/app/Application;
2e  invoke-static v1, v4, Lcom/secneo/apkwrapper/Helper;->attach(Landroid/app/Application; Landroid/content/Context;)V

    Ljava/lang/Exception;

    36  move-exception v0
    38  const/4 v1, 0x0
    3a  sput-object v1, Lcom/secneo/apkwrapper/ApplicationWrapper;->realApplication Landroid/app/Application;
    3e  goto 0x-5

34  return-void
```

Fig. 3. An example of a method CFG in a real apk

show the monotonicity of 5UD-CFG. For each directed edge $(i,\ j)$, $i,\ j\ =\ \{1,\ 2,\ ...,\ n\}$ ($(i,\ j)$'s scope are the same in this Section) in CFG, we define the joint probability of the first-order call structure between vertex v_i and vertex v_j as follows:

$$Link_1(v_i,\ v_j) = \frac{1}{1+exp(\vec{x_i}^T\vec{x_j})}, \tag{1}$$

where $\vec{x_i} \in R_d$ is feature vector of vertex v_i, $\vec{x_i} = <\ num_i,\ squ_i,\ in_i,\ out_i,\ loop_i\ >$. In Eq.(1), we use the sigmod function as the embedding activation function. Eq.(1) defines a joint probability distribution of the vertex call over a CFG, and its empirical probability of two vertexes can be defined as $\widehat{Link_1} = \frac{\omega_j}{\omega_i \times W}$, where $W = \sum_{i\in V}\omega_i$ is related with the number of vertexes that calls the vertex v_i by the first-order call structure. To preserve the first-order call structure, we use the distance between two probability distributions to be the objective function. A straightforward way is to minimize the following objective function:

$$y_i^{(1)} = d\ (\widehat{Link_1},\ Link_1), \tag{2}$$

where $d(.,.)$ is the distance between two distributions. We minimize the KL-divergence [30] of two probability distributions. Therefore, the objective function is as follows:

$$y_i^{(1)} = -\sum_{(i,j)\in E} s_{i,j}\ log\ Link_1\ (\omega_i v_i,\ \omega_j v_j). \tag{3}$$

The second-order call structure assumes that the basic block (vertex) are invoked by the another vertexes expect neighbors. Each basic block is also treated as a specific "contexts" of other vertexes. We define the joint probability of the second-order call structure between vertex v_i and vertex v_j as follows:

$$Link_2(v_j|v_i) = \frac{exp(\vec{x_j}^T\vec{x_i})}{\sum_{k=1}^{|V|} exp(\vec{x_k}^T\vec{x_i})}, \tag{4}$$

where $|V|$ is the number of other vertexes expect neighbor vertexes. To preserve the second-order call structure, we need to make the conditional distribution of the "contexts" be closed to the empirical distribution $\widehat{Link_2}\ (v_j|v_i) = \frac{\omega_j}{\omega_i \times d_i}$, where the d_i is the degree of the vertex i. Therefore, we minimize the following objective function:

$$y_i^{(2)} = \sum_{i\in V} d_i d(\widehat{Link_2},\ Link_2), \tag{5}$$

where $d(.,.)$ is the distance between two distributions. We minimize the KL-divergence [30] of two probability distributions. Therefore, the objective function is as follows:

$$y_i^{(2)} = -\sum_{(i,j)\in E} l_{i,j}\ log\ Link_2\ ((\omega_j v_j)|(\omega_i v_i)), \tag{6}$$

where $l_{i,j}$ is the connection between the vertex v_i and the vertex v_j. If $l_{i,j} = 1$, there exists a edge from the vertex i to j. If $l_{i,j} = -1$ for vertex i, there exists an edge from vertex j to i. Otherwise there is no directed edge between v_i and v_j.

To generate the CFG feature vector by preserving both the first-order and second-order call structure, we train the parameters ω_i for each vertex. We adopt Maximum Likelihood Estimate for optimizing Eq (2) and (3). We sample all edges (i,j) in a CFG, the gradient of the weight of vertex with the embedding vector $\vec{x_i}$ will be calculated as:

$$\nabla_1 = \frac{\partial y_i^{(1)}}{\partial \omega_i} = \frac{\partial(-\sum_{(i,j)\in E} s_{i,j}\ log\ Link_1(\omega_i v_i,\ \omega_j v_j))}{\partial \omega_i}$$
$$\nabla_2 = \frac{\partial y_i^{(2)}}{\partial \omega_i} = \frac{\partial(-\sum_{(i,j)\in E} s_{i,j}\ log\ Link_2((\omega_j v_j)|(\omega_j v_j)))}{\partial \omega_i}.$$

The embedding vector space of all vertexes in a CFG is spanned by the vertex weight W.

In our case, we only need to ensure that it is monotonicity to embed a CFG as a vector. The gradient is used to calculate the distance between two vertexes. Therefore, for ensuring the monotonicity of the embedding vector, we combine the gradient $\nabla = \nabla_1 + \nabla_2 = 0$ by the Maximum Likelihood Estimate formulas. let

$$\frac{\partial y_i^{(1)}}{\partial \omega_i} + \frac{\partial y_i^{(2)}}{\partial \omega_i} = 0,$$

we train the parameter of the vertex weight sequence $W = [\omega_1,\ \omega_2,\ ...,\ \omega_n]mod(n-1)$.

Form above analysis, we know that the vertex weight ω_j is trained by the CFG topology $\sum_{e(j,k)\in 5UD-CFG}$, which is unique when the CFG is determined. As the view of the Definition 6, $< f_n,\ f_s,\ f_i,\ f_o,\ f_l >$ is related with $\sum_{e(j,k)\in 5UD-CFG}$. The different CFG has the different vertex weight sequence W.

5.3 An example of 5UD-CFG embedding

we give an example to show how to calculate ω_i. As shown in the Fig. 4, vertex A with the first sequence number is called by the vertex B and the vertex C directly. The vertex C is called by the vertex D directly, and vertex D is called by the vertex E directly. This outgoing degree of vertex A is $A^{\nearrow C \longrightarrow D \longrightarrow E}_{\searrow B}$. A is related with B, C, D and E. The result of vertex A will be passed to vertex B and vertex C, which have a first-order call structure with vertex A. This process will also influence the vertex D and E by influencing the vertex C, which are the second-order call structure.

Based on the above first-order call structure and the second-order call structure, the particular trained process of W is as follows:

$$\begin{cases} \frac{\partial\ Link_1^{(1)}}{\partial\ \omega_1} + \frac{\partial\ Link_2^{(1)}}{\partial\ \omega_1} = 0 \\ \frac{\partial\ Link_1^{(2)}}{\partial\ \omega_2} + \frac{\partial\ Link_2^{(2)}}{\partial\ \omega_2} = 0 \\ \frac{\partial\ Link_1^{(3)}}{\partial\ \omega_3} + \frac{\partial\ Link_2^{(3)}}{\partial\ \omega_3} = 0 \\ \frac{\partial\ Link_1^{(4)}}{\partial\ \omega_4} + \frac{\partial\ Link_2^{(4)}}{\partial\ \omega_4} = 0 \\ \frac{\partial\ Link_1^{(5)}}{\partial\ \omega_5} + \frac{\partial\ Link_2^{(5)}}{\partial\ \omega_5} = 0. \end{cases}$$

let $log(1 + exp\{k \times \omega_i \times \omega_j\}) = k \times \omega_i \times \omega_j$ and the weight of final vertex $\omega_5 = 0$, we have

$$\begin{cases} 9\omega_2 + 10\omega_4 + 6\omega_5 = 0 \\ 35\omega_3 + 37\omega_4 + 18\omega_5 - 18\omega_1 = 0 \\ 20\omega_5 + 35\omega_2 - 18\omega_1 = 0 \\ 25\omega_5 + 37\omega_2 + 10\omega_1 - 60\omega_3 = 0 \\ 20\omega_3 + 6\omega_1 + 18\omega_2 - 50\omega_4 = 0 \\ W = \omega_1, \omega_2, \omega_3, \omega_4, \omega_5\ mod\ 4. \end{cases}$$

We obtain the $W = [1,\ 2,\ 3,\ 1,\ 0]$.

Based on the Definition 6, we calculate the feature of a 5UD-CFG as follows:

$$\begin{cases} \omega = 1+2+3+1+0 = 7. \\ f_n = \frac{1\times2\times1+2\times1\times2+3\times2\times3+4\times2\times1+5\times1\times0}{1+2+3+1+0} = 4.57 \\ f_s = \frac{1\times2\times1+7\times1\times2+4\times2\times3+4\times2\times1+1\times1\times0}{1+2+3+1+0} = 6.857 \\ f_i = \frac{0\times2\times1+1\times1\times2+1\times2\times3+1\times2\times1+1\times1\times0}{1+2+3+1+0} = 1.429 \\ f_o = \frac{0\times2\times1+2\times1\times2+1\times2\times3+1\times2\times1+0\times1\times0}{1+2+3+1+0} = 1.714 \\ f_l = \frac{0\times2\times1+0\times1\times0+0\times2\times2+0\times2\times1+0\times1\times0}{1+2+3+1+0} = 0. \end{cases}$$

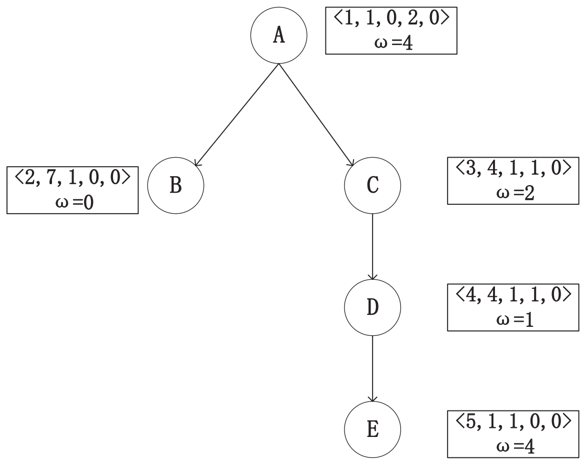


Fig. 4. An example of feature of 5UD-CFG

6 Second-hierarchy Tensor Embedding model

According to the extracted 5UD-CFG in the Section 4, this section discusses how we utilize the feature vector of 5UD-CFG embedding, compress and cluster them into a more concise feature that are suitable for achieving scalable and accurate app homology analysis. The second hierarchy embedding process is the tensor embedding, which generate 3UD-CFG based on the 5UD-CFG.

6.1 Tensor embedding model

We propose a tensor embedding model for generating more concise and effective feature vector. We review a well-studied factorization and compact model that are based on the first hierarchy CFG embedding. In the definition 5 of Section 3, we can set I_1 of the second-hierarchy embedding tensor model. For example, if the tensor embedding model is used to malware detection , we can set I_1 as the malware label of the app, using the label I_1, we can partition the app community that can be easier for the app homology analysis. For example, we can define

$$\begin{cases} I_1 = 1, \; if \; this \; method \; is \; from \; the \; begin \; apk, \\ I_1 = 0, \; if \; this \; method \; is \; from \; the \; malware \; apk. \end{cases}$$

We use the accepted notation where an order-3 tensor is indexed by 3 indices and can be represented as a multidimensional array of data [35]. That is, an order-3 tensor, A, can be written as

$$A = (a_{i_1 i_2 i_3}) \in R^{n_1 \times n_2 \times n_3} (n_3 = 1). \tag{7}$$

A third-order tensor can be pictured as a "cube" of data. It is convenient to refer to its slices. We use terms lateral slices to specify which two indices are held constant. Using the MATLAB notation, $A(:, k, 1)$ corresponds to the $k-th$ lateral slice. A tube of a third-order tensor is defined by holding the first two indices fixed and varying the third. In particular, the third orientation of our tensor embedding model indicates a label. It does not join in the calculation of the tensor embedding model.

As shown in the Fig. 4, we cut slices for the tensor embedding model expand in the horizontal orientation.

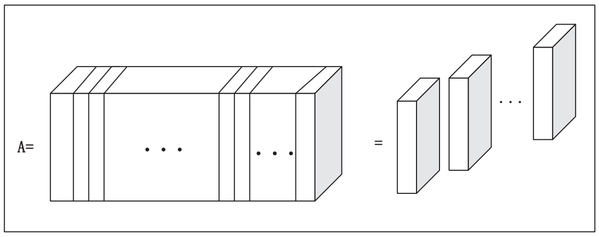


Fig. 5. Lateral slices of the tensor model A

For the third-order tensor embedding model, $A \in R_{n_1 \times n_2 \times n_3}$, we have the representation that the tensor unfolding $A_{(3)} \in R^{I_f \times (I_n * 1)}$ contains the element $t_{i_1 i_2}$ at the position with row number f_i and the column number n_i, $i_1$is number of one of five features. i_2 is number of the CFG.

The following section, we will discuss how the function f decomposes the tensor embedding model.

6.2 Compact 5UD-CFG to 3UD-CFG Based on tensor embedding model

First we review the Singular Value Decomposition (SVD) [36] [37] as compress algorithm for 5UD-CFG: For an SVD in form of 5UD-CFG tensor model $A = UDV^T$. We interpret the matrix U to consist of latent embedding vectors in rows, for each entity represented in the first dimension of A. The matrix A is constructed by a linear combination of the embedding U with weights defines as rows of $(DV^T)^T$.

Then we consider $U = A(DV^T)^\dagger$, which is an inverse-relation, to be a mapping function from A to the latent embedding in U. It is generally assumed that this mapping relation also holds for a new observation which is not present in A, i.e.

$$u_{new}^T = x_{new}^T (DV^T)^\dagger. \tag{8}$$

The first step in the proposed method is to unfold the scalable embedding tensor model. Similarly, we anchor the MatVec command [38] to the lateral slices of the tensor, $MatVec(A)$ takes an $n_1 \times n_2 \times n_3$ tensor and returns a block $n_1 * n_3 \times n_2$ matrix, in which model, $n_3 = 1$,

$$MatVec(A) = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n_2} \end{bmatrix}.$$

The operation that takes $MatVet(A)$ back to tensor form is the fold command:

$$fold \; (MatVet(A)) = A. \tag{9}$$

The tensor mapping model defines a function $f(\bullet; \psi_l)$ that maps each row in tensor unfolding A^l to the corresponding row in the embedding matrix X^l as

$$\widehat{A^l} = f(X^l; \psi_l) \; \forall l \in [1, .., L]. \tag{10}$$

Note that in the input of mapping function, each arbitrary row i in X_l.

The mapping cost function: we define the mapping cost function as

$$c_M = \sum_{l=1}^{L} d_M(A_l, \widehat{A_l}) = \sum_{l=1}^{L} d_M(A_l, f(X_l; \Psi_l)). \tag{11}$$

Optimizing that mapping cost function involves adjusting Ψ_l for each l with a given $f(\bullet)$ so that the distance between the learned embedding A_l from factorization and mapped embedding $\widehat{A_l}$ from the corresponding tensor unfoldings is minimized.

Based on the SVD factorization [39] [40], we define $f(\bullet) = (DV^T)^T$. We embed all 5UD-CFGs into a smaller embedding vector called 3UD-CFG, which means compress 5UD-CFG tensor embedding model. The core tensor and truncated bases of SVD decomposition described in the preliminaries can be employed to make scable data smaller. In particular, for our tensor embedding model, the SVD factorization is as follows:

Definition 6 [Singular Value Decomposition (SVD)] for our model: Let $A \in R^{m \times tn \times 1}, m < tn$ denote a matrix, the factorization

$$A = USV^T, \tag{12}$$

is called the SVD of A. Matrices U and V refer to the left singular vector space and the right singular vector space of matrix A respectively. Both U and V are unitary orthogonal matrices. Matrix $S = diag(\delta_1, \delta_2, ..., \delta_k, ...\delta_m)$ is a diagonal matrix that contains the singular values of A. In Section 3, we know that the number of extracted embedding vector is $m = 5$, therefore, $k < 5$, the smaller k has more effective match, but the larger k has more accurate match, we choose $k = 2$ or $k = 3$, the compression tensor model is as follows:

$$M_k = U_k S_k V_k^T, \tag{13}$$

is called the $rank - k$ truncated SVD of A, where $U_k = [u_1, u_2, ..., u_k]$, $V_k = [v_1, v_2, ..., v_k]$, $S_k = diag(\delta_1, \delta_2, ..., \delta_k)$. The truncated SVD of M is much smaller to store and faster to computer. We decompose the tensor model to classify the app, so we need to make the unitary orthogonal matrices U as the projection and compressed orientation.

We change the tensor model $A \in R^{m\times tn\times 1}$ as a compressed tensor model $A \in R^{k\times tn\times 1}$. This process decrease the original m feature to particular k feature $k < m$, which is called the feature compression. It is shown as the following equation:

$$U_{k\times m}^T A_{m\times tn} \approx S_{k\times k} V_{k\times tn}^T. \tag{14}$$

We obtain a mount of the app feature of app samples as the homology search database. Algorithm 1 shows the compression algorithm for 5UD-CFG tensor embedding model A by the SVD factorization. Before decomposing the tensor model by the SVD algorithm, we make the spare matrix centering by the fast Fourier transform.

In we present a compression strategy based on Algorithm 1, the compression is based on the assumption that the terms $||S(m,m,1)||_F^2$ decay rather quickly. A particularly nice feature of SVD compress proves that an optimal approximation of a tensor is closed to the original tensor. We prove the $A \approx A_k$ in our model as follows:

Proof:

First, we adopt the definition of the Frobenius norm of a tensor used in the literature:

Definition 7: Suppose $A = a_{ij1}$ is size with $n_1 \times n_2 \times 1$, Then

$$||A||_F = \sqrt{\sum_{i=1}^{n_1}\sum_{j=2}^{n_2} a_{ij}^2}. \tag{15}$$

We calculate he Frobenius norm between the original tensor and the approximate tensor as the mapping cost function c_M.

$$\begin{aligned} &||A - A_k||_F^2 = ||USV^T - U_k S_k V_k^T||_F^2 \\ &= ||S(k+1:m, K+1:m, 1)||_F^2 \\ &= ||F_m \times S(k+1:m, k+1, m)||_F^2 \\ &= ||\delta_{k+1}||_F^2 + ||\delta_{k+2}||_F^2 + ... + ||\delta_m||_F^2. \\ &\leq T_{error} \quad (Error \;\; Threshold) \end{aligned}$$

Since the singular value is decreasing in order and the difference between two neighbors is also greatly decreasing in order, we choose the first two maximum singular value, the approximate tensor $U_k S_k V_k^T$ is closest to the original tensor model A. □

According to a mount of experiments, we can know the SVD factorization has the minimal error. The most major information are concentrated on the first several maximum singular values. Therefore, $A \approx A_k$, when compressing the feature of 5UD-CfG, $(U' * A)(1:k, 1:k, 1) \approx (U_k' * A)$. The five feature of the 5UD-CFG is projected as major two or three features to nearly loselessly represent the original tensor model.

Algorithm 1 T-SVD-Compare

```
Require:
1: Input:
2: tensor model $A = R^{m\times tn}$, detected apk feature set B;
3: y=fft(A,[],1);
4: [U,S,V]=SVD(y);
5: U1=ifft(U,[],1);
6: U2=U1(:,1:2);
7: T=real(U2'*A); compress feature
8: save T as the comparable object;
9: V1=clustering(V)
10: V3=centroid{$V1\{\}$} V3 is the centroid vector in clustering set V1 {}
11: A1=T*V3
12: A2=T*V1,
13: B1=U2'*B compress detected app feature
14: B2=B1*V3
15: Compare (B2, B), get the closest element in the set V1
16: B3=B1*V1the corresponding element
17: Compare(B3, A2the corresponding element)
18: Output: The last comparable result.
```

6.3 Clustering of 3UD-CFG tensor embedding model

After the 5UD-CFG tensor embedding model is decomposed as 3UD-CFG tensor embedding model by the SVD algorithm, we need to cluster the decomposed unitary orthogonal matrices V_k. The clustering result is generated from a training set of $S_k \times V_k$ by an unsupervised learning algorithm. The clustering result is as the app homology analysis basis. Each cluster comprises a number of closed 3UD-CFGs, which is community of similar CFG for apps.

In this paper, we use a k-means clustering as the unsupervised learning algorithm to generate the clustering result of the $S_k \times V_k$. Formally, the k-means clustering partition the training 3UD-CFG tensor embedding model into t sets $SV = SV_1, SV_2, ..., SV_t$ so as to minimize the sum of the distance of every 3UD-CFG to its cluster center. $c_i \in SV_c$ is the centroid for the subset SV_i, and the collection of all centroid nodes constitute the clustering center of the projected tensor SV_c.

The number t of the clustering will affect the search accuracy. Tensor factorization is effectively handle to the scalable data. We cluster all column vectors in SV_k. Then we get the above centroid set SV_c and each clustering set $SV_c = SV_{c_1}, SV_{c_2}, ...Sv_{c_t}$ as the comparable database. First, we calculate the distance between the detected sample B and the centroid set SV_c. We find indexes of the minimum two vector as the comparable candidate. This two indexes is denoted as i, j. Second, we calculate distance between each vector of the SC_i, the SC_j and the detected sample B. Therefore, we get the closest distance with the B in the SC_i or the SC_j. The process is as follows:

$$\{SV_i, SV_j\} = arg\ min_{i,j \in \{1,2...,t\}} distance\ (B, SV_c)$$
$$\longrightarrow object = arg\ min_{SV_g \in (SV_i \bigcup SV_j)} distance\ (B, SV_g).$$

Object is the result that we want to get.

Comparison between the SV_c, the SV_i or the SV_j and the B is a search problem. For the high-dimension features in the machine learning, the most effective method to find the nearest neighbor is the randomized k-d forest and the priority search k-means tree, which is more effective than the LSH algorithm. This section introduces a scalable solution by the fast KNN search [41]. The principle and major steps of the brute-force KNN search are as follows:

Considering a set SV_c of t reference points in a $d-$dimensional space $SV_c = \{SV_{c_1}, SV_{c_2},\ ...,\ Sv_{c_t}\}$, and a set B of q query points int the same space $B = \{b_1, b_2, ...b_q\}$, for a query point $b \in B$, the brute-force algorithm is composed of the following steps:

1) Compute the distance between b and t reference points of SV_c:
2) Sort the t distances;
3) Output the distances in the ordered of increasing distance.

When applying this algorithm for the q query points with considering the typical case of large sets, the complexity of this algorithm is as follows:

- $O\ (tq)$ multiplication for the $ttn \times m$ distances computed
- $O\ (tq\ logt)$is for t sorting processes

The brute-force kNN search method is by nature highly parallelizable and perfectly suitable for the high-dimension feature search.

In the above clustering method, we use the clustering result of columns of S_k multiply by V_k to represent the clustering result of columns of original tensor model. Next we prove why the clustering result of columns of $S_k \times V_k$ can represent the clustering result of columns of A.

Proof:

We know that $U_k^T \times A = S_k \times V_k$ from the SVD decomposition of A. This means that we need to prove the clustering result of columns of $U_k^T \times A$ represents the clustering result of columns of A.

Based on the SVD decomposition, we know that the U_k is an orthogonal tensor $U_k \in R^{m \times k \times 1}$

$$||A||_2^F = trace((A * A^T)_{(:,:1)}) = trace((A * A^T)_{(:,:,1)}), \tag{16}$$

where $(A * A^T)_{(:,:,1)}$ is the lateral slice of $A * A^T$ and $(A * A^T)_{(:,:,1)}$ is the lateral slice of $A^T * A$. Therefore,

$$\begin{aligned}||U_k^T * A||_2^F &= trace([(U_k^T * A)^T * (U_k^T * A)]_{(:,:,1)})\\ &= trace([A^T * U_k * U_k^T * A]_{(:,:,1)})\\ &= ||A||_F^2.\end{aligned}$$

□

We are finally in a position to consider tensor factorizations of 3UD-CFG tensor embedding model that are effectively handle the cluster and classification.

6.4 Incremental tensor embedding for updating

Based on the recursive incremental HOSVD proposed by the Liwei Kuang [42] [43], we propose a incremental classification tensor model for the expanding learning model. The special steps is shown in the Algorithm. 2.

We compute the incremental SVD decomposition method for streaming 5UD-CFG feature data dimensionality reduction. We just need the new decomposed unitary orthogonal matrix U, V. This incremental algorithm only needs to decompose the incremental part C_{i-1} rather than decomposing all matrix $[A_{i-1}, C_{i-1}]$.

7 Evaluation

Our model consists of two main components: CFG extractor for each app, 5UD-CFG embedding and the 3TU-CFG tensor embedding work based on the compact and the clustering. We obtain the CFG extractor for each app, a open project Androguard [44], which is disassembly tool to transform APK files to SMALI code. It is a disassembler for Android's DEX format. We implement 5UD-CFG embedding and 3TU-CFG tensor embedding work in numpy of Python, and achieve LSH searching in nearpy of Python.

We evaluate our approach on five typical third-party Chinese Android markets: PP market, Xiaomi markets, Baidu markets , Tencent markets , Huawei markets. We performed a cross-market analysis to

Algorithm 2 T-SVD-Incremental

```
Require:
 1: Input:
 2: Initial tensor model $A_{i-1}$ and incremental 5UD-CFG feature matrix $C_{i-1}$.
 3: Decomposition and compression results $U_{k_{i-1}}$, $S_{k_{i-1}}$, $V_{k_{i-1}}$ of tensor model $A_{i-1}$.
 4: Project $C_{i-1}$ on the orthogonal space spanned by $U_{k_{i-1}}$, $Span = U^T_{k_{i-1}} \times C_{i-1}$.
 5: Compute $H$ which is orthogonal to $U_j$, $H = C_{i-1} - U_{k_{i-1}} \times Span$
 6: Obtain the unitary orthogonal basis $J$ from matrix $H$;
 7: Compute the coordinates of matrix $H$, $K = J^T \times H$;
 8: $[A_{i-1}, C_{i-1}] = [U_{k_{i-1}}, J] \begin{bmatrix} S_{k_{i-1}} & J \\ 0 & K \end{bmatrix} \begin{bmatrix} V & 0 \\ 0 & I \end{bmatrix}$
 9: Obtain the unitary orthogonal basis $Uo, Vo$ from matrix $\begin{bmatrix} S_{k_{i-1}} & J \\ 0 & K \end{bmatrix}$;
10: Obtain new decomposition results, $U = [U_{k_{i-1}} J] \times Uo, V = \begin{bmatrix} V & 0 \\ 0 & I \end{bmatrix} Vo$
11: Output: U, V.
```

analysis the app homology. Our experiments are conducted on a server with 16 GB memory, 12 core at 1.6GHz and 7 TB hard drives. All the evolutions are conducted based on five app datasets: 1) game app dataset; 2) social app dataset; 3) entertainment app dataset; 4) finance app dataset; 5) other app dataset. We let each type's app dataset as a kind of app community.

7.1 Dataset

Dataset I - Game app dataset. This data set mainly contains all kinds of game. Size of most games is more than $50MB$, some even is more than $150MB$. Game apps nearly larger than other apps. We collect $32,554$ apps from five Android markets, including 37% hot game apps according to the rank and download rate.

Dataset II - Social app dataset. This data set is the chat app, which has more links with others. This kind of app has more hidden threatens. Size of social app is from $20MB$ to $60MB$. The social app in other app types is the middle size. We collect $23,295$ social apps from five Android markets.

Dataset III - Entertainment app dataset. Entertainment app dataset includes sing apps, photography apps, video apps, reading apps and etc. This kind of app enriches people's life, which is smaller than other app dataset. Size of entertainment app from $10MB$ to $50MB$. We collect $43,885$ entertainment apps from five Android markets.

Dataset IV - Finance app dataset. Finance app dataset includes shopping apps, banking apps and etc. Many malware advertisements can be added in repacking finance apps. We collect $22,846$ finance apps from five Android markets.

Dataset V - Other app dataset. Other app dataset includes utility apps, life apps and etc. Other apps mostly are from the unknown developer. This kind of app exists more serious threatens. We collect $30,209$ other types' apps from five Android markets.

7.2 Cross-Market App Homology Analysis Comparison

We compare the tensor embedding work based on the 5UD-CFG embedding result with existing methods. All evaluations are conducted under the above five app datasets. We use the above five app datasets to train the dataset. We use the metric of the false positive rate to evaluate the accuracy. We perform two time metrics to indicate the efficiency of the proposed and the other compared works. 5UD-CFG Embedding time and the search time respectively represents the preparation efficiency and the search efficiency.

The number of apps in the same type group and the number of app class, which can also be called the app community, give a direct impression on the app detection in Android markets. The different community groups nearly does not share the same code. We also need to define a threshold for the app homology degree, impacts the classification results. We use the threshold $\delta = 0.91$ to perform the cross-market app homology analysis. When $\delta = 0.91$, the measured false positive rate through 120 manual examination of randomly selected the app community group is 1%.

We prepare three representative app clone analysis or bug search techniques to compare our evaluation: Binary search based the Centroid [34], Gemini based Neural Network [18], Genius [17]. We introduce the main idea for these three solutions as follows:

1) **Centroid [34]:** The centroid-based approach shows the scalability and accuracy for the app clone. We implemented a centroid bug search method for the five markets. This paper used a geometry characteristic, centroid, of dependency graphs to measure the similarity between methods in two apps.
2) **Gemini [18]:** Gemini shows a novel neural network-based approach to compute the embedding based on the control flow graph of each binary function. We set the same iteration number of neural network as 5. According to the proposed method by this paper, we use the above five datasets to train the embedding database.
3) **Genius [17]:** Genius is a bug search system based on the CFG, which can scalable search bugs in the cross-platform. Genius's source code is not available. We use the proposed method to generate 16 codebooks, and then complete the search database. We compare accuracy and efficiency with our method.

7.3 Accuracy Comparison

In this Section, we evaluate the accuracy of two hierarchies embedding model and search work. We construct a testing dataset by randomly selecting a certain number of apps or collecting several novel samples. We train embedding numeric database as the search database from all collective apps. According to five types' markets, we obtain five basic searched databases. We randomly select one or several apps from the testing dataset as the input. Based on the app decompilation and two hierarchies embedding process, we obtain 3TU-CFG embedding vector as the searched databases according to Algorithm. 1 and Algorithm. 2. We assume that it is unknown about searched apps' types. We respectively search the input as five basic search databases.

For the five search databases, we train $152,789$ apps in total, which includes $7,491,123,901$ methods. Database I has the $1,596,096,888$ methods. Database II has the 1142135436 methods. Database III has the $2,151,646,861$ methods. Database IV has $1,120,121,321$ methods. Database V has $1,481,123,391$ methods.

For the testing dataset, we have two parts: a part of apps from the search database, and another part is from the novel collective apps. The number of app that are collected in the search database is 1495. The number of novel collective apps is 587.

According to the proposed method, we can obtain one closest candidate for searched samples. It is not necessary to find a query for each search sample. Fig. 6 shows the true positive rate (TPR) when we set different similarity thresholds in the method-level. If the similarity thresholds is less than 0.0057, TPR in the method-level is more than 98.5%. If the similarity thresholds is less than $6.9 \times 10^{(-5)}$, TPR in the method-level is more than 68.7%. The larger the similarity threshold is, the TPR is larger. When the distance is more than the similarity threshold, we judge those two methods are not same. According to the proposed method, we find that different apps are with a larger distance. Make sure the similarity apps have the smaller distance is more important. If the similarity threshold is set too largersome similarity apps may be defined as the different apps with a large probability. Therefore, TPR is decreased with the similarity threshold decreasing.

Based on the proposed clustering algorithm in Section 5.3, the different clustering's number affects TPR. Fig. 7 shows that there is a inflection point in the changed curve of relationship with the clustering number and the TPR. We can see that the TPR is the largest if the clustering number is 0.014% of the trained database.

To compare the efficacy of final 3TU-CFG with centroid, gemini and genius, we use the same app database to calculate the ROC curve in the same threshold. We use the two metrics to evaluate the accuracy of the proposed and compared methodsthe true positive rate (TPR) and the false positive rate (FPR). For the searched samples query q, there are m matching functions out of total of L functions. If we set the first F results as the positives, the total number of the correct matched functions c, which are true positive. The remaining number of functions in the F functions, that is $F - c$, are false positives. We set the TPR as $TPR = \frac{c}{m}$ and the false positive rate is $FPR = \frac{F-c}{L-m}$.

We use 2082 apps in the testing database as the queries to match with the target approaches. Fig. 8 shows evaluated results of the ROC curve. we can see that 3TU-CFG outperforms another three approaches: Gemini, Genius, and Centroid. The size of ROC curve of 3TU-CFG is the largest, which shows the 3TU-CFG is the most accurate. When the false positive rate is small, the true positive rate is significantly better than other approaches. Four curves show that the true positive rate is increasing with the false positive rate increasing. However, the rate of ROC curve of the

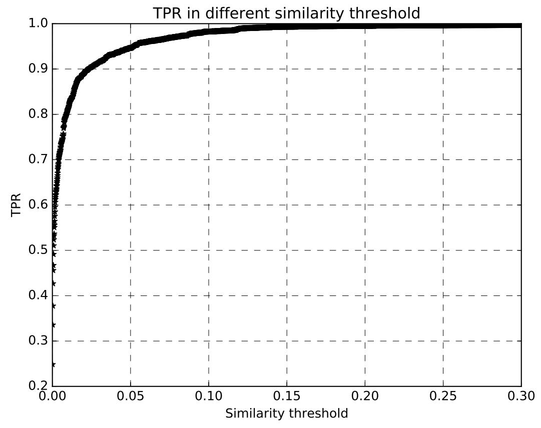


Fig. 6. TPR with different threashold

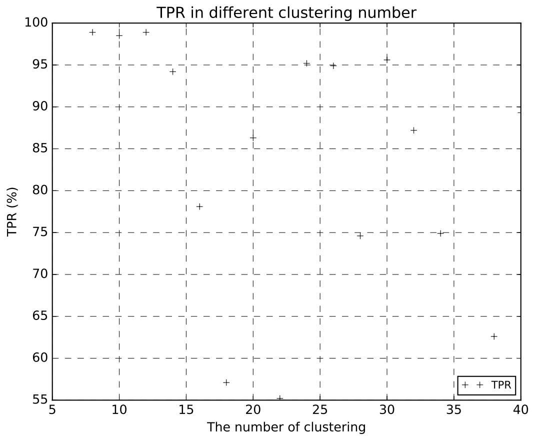


Fig. 7. TPR with different clustering number

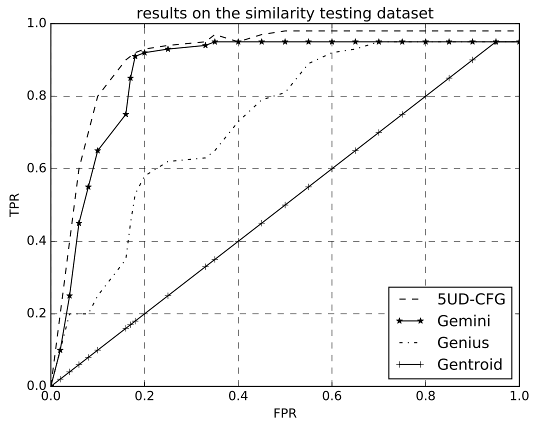


Fig. 8. ROC on the similarity testing dataset

3TU-CFG has the fastest growing. The results shows the 3TU-CFG can achieve even better accuracy than other approaches.

We consider searched results and observe the advanced performance of 3TU-CFG is mainly because the 5UD-CFG can monotonously represent a function (proved in Section 4) and 3TU-CFG is obtained by monotonously compress 5UD-CFG. Each function just need a closest candidate to make sure the accuracy for 3TU-CFG embedding process. The Gemini need to train several iterations to get the embedding vector of the function. The model that is trained by the neural network needs a mount of samples to make sure the accuracy. Genius has several candidates that the accuracy depends on the accuracy of the obtained codebook. Centroid have a great gap between different platform, which is not monotonously represent the function. Our method is adept in representing entire structure of CFG, distinguishing the change, and thus shows the better results.

According to our tensor model, we know that each benign function in a benign app have a benign label that denoted as 1. If there is a malware function in a app, we consider this app is a suspicious app. For the suspicious apps found by our model, we validated them through online virus detection system and manual evaluations to judge the validity of our detection.

7.4 Efficiency Comparison

We measure the scalability of proposed approach from the following three aspects: the scale of the five collective markets, the performance on cross-market app detection and updating of the basic trained database.

We analyze all apps in the collective five markets to obtain a basic database. Fig. 9 show the distribution of size per app and the number for this app, the distribution of opcodes per app and the number for this kind of opcodes, and the distribution of the number of methods per app and the number for this app. For the five markets, we train in total $152,789$ apps are in these five groups. Fig. 9 (a) shows that nearly 94.7% of a app is in $0-50MB$. The total size of this app is $7.47\ TB$. The Fig. 9 (b) shows that nearly 69.8% apps have $0-16000$ methods, and nearly 77.6 apps have more than 4000 methods.

The performance of the app detection can be divided into three steps: 1) embedding generation time, 2) update embedding time, and 3) app homology search ttime.

The time of the embedding generation depends on the total size of apps. We need to decompile the app into a series of functions. This step can be done in parallel. We use the androguard to obtain the original CFG structure. The original CFG generation time is the decompile time of the app. We can use several computer to measure the time. At the same time, Fig. 10 compare the 3TU-CFG embedding generation time with other three approaches, which including the decompile time. We store the CFG embedding vector in the database, and then use the LSH to search and detection. Then we evaluate the efficiency of Gemini, Genius for embedding time and Centroid for centroid generation time.

7.4.1 Performance on CFG with attributes extraction time.

Fig. 10 (a) shows the CFG with attributes extraction times of the 5UD-CFG can improve upon Genius by $8\times$ to $15\times$ on average on CFG embedding time. The 5UD-CFG needs to extract 5 basic-block attributes along with the structure feature of the CFG. However, Genius needs to additionally extract the betweenness attributes, in total 8 attributes. The different becomes larger for the app with a large size. A app with the large size has more methods. There are more nodes in a CFG. It needs more time to compute the betweenness attributes for larger size's app with a amount of nodes. The preparation times of the 5UD-CFG can improve upon Gemini by $1.1\times$ to $1.7\times$ on average for different size of apps. Gemini needs to extract 6 basic-block attributes and the number of offspring time. However, Gemini aggregate the graph structural information through iterations of embedding update. Therefore, the CFG with attributes extraction times of the 3TU-CFG is close to Gemini. Centroid also extract 5 basic-block attributes with the structure feature, the process is similar to the CFG with attributes extraction process of the 5UD-CFG. The CFG with attributes extraction times of the 3TU-CFG is close to Centroid.

7.4.2 Performance on embedding generation time.

Fig. 10 (b) shows the embedding generation time for four approaches with the increased number of methods. The number of method denotes the scalable of the CFG. Obviously, more methods need more embedding time. 3TU-CFG, Genius, Gemini and Centroid all transform the CFG of the function to the

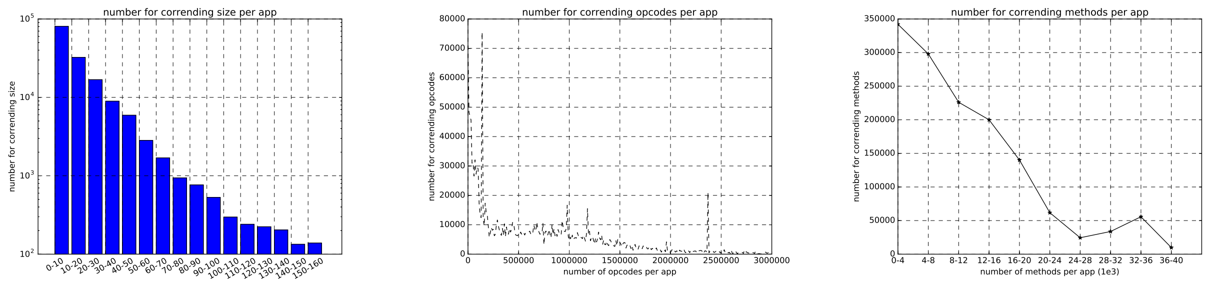


Fig. 9. Scalability of the database.

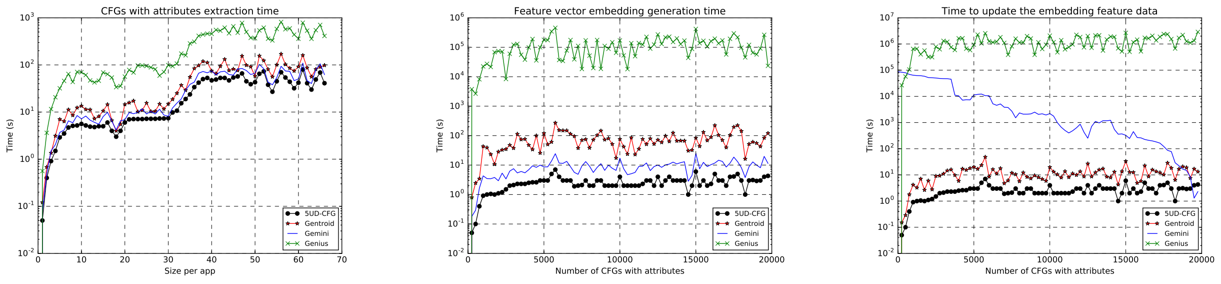


Fig. 10. Efficiency evaluation.

embedding vector in embedding process. We can see the two hirearchies embedding process of the 3TU-CFG run $4700\times$ to $76000\times$ faster than Genius, run $2.1\times$ to $5\times$ faster than Gemini, and run $7\times$ to $51\times$ faster than Centroid on average. Since the proposed 3TU-CFG embedding process avoids the complexed neural network learning process, the expensive graph matching and the scalable clustering algorithm. Genius needs extra time to generate the codebook for all CFGs by the complex bipartite graph matching. Gemini needs extra time to learn five iterations for embedding CFG. Centroid is to compute the centroid of a spatial geometry. This preparation process is simpler than the above two approaches. Therefore, the preparation time of Centroid is lower than another two approaches. However, the embedding process of the 3TU-CFG is more concise than other three approaches. The line embedding and the tensor embedding have much simpler matrix operations than that in the neural network iteration computation. However, Gemini and 3TU-CFG belong to the matrix operation, which can be parallelized.

7.4.3 *Performance on Updating time.*

Fig. 10 (c) shows the updating embedding database time. Adding new apps is very common in android markets. If the new app is not included in the database, we need to update the database. We can see that the update time of 3TU-CFG is quicker $4\times$ to $20000\times$ than Gemini, quicker more than $180000\times$ than Genius, and quicker $2\times$ to $7\times$ than Centroid. If we just update few apps at once, the different of time between the 3TU-CFG and the Gemini is larger. Since we propose a increased algorithm of tenor embedding that just needs to retrain the increased novel methods rather than retrain all feature vector. If the number of retrained novel methods are same with the number of original feature database. The update embedding time of 3TU-CFG is close to Gemini. Since the Gemini need to embedding all ACFG to get the updating feature vector although it does not need to regenerate the ACFG. Specifically, Genius need to retrain the codebook, this is a larger project, which means regenerate all feature vector including scalable graph matching algorithm. It cost too much time. Centroid just needs to update the increased apps. However, it's accuracy is lower than the 3TU-CFG. Therefore, the 3TU-CFG has a better performance on the updating.

7.4.4 *App homology search time.*

We evaluate the scalability of 3TU-CFG on the collective five app markets, which consist of in total $152,789$ apps. We investigate the time consumption for each stage to demonstrate that 3TU-CFG is capable of handling apps at a large scale.

Fig. 11 (a) shows the search time for 3TU-CFG in the larger scale codebase. We choose five collective dataset into six codebases of different scales from $c = 10^3$ to $c = 10^8$, where c is the total number of functions in the codebase. We choose the 1 to 10000 sequentially submitted queries. As we can see, the search time grows like linearly according to the increase of the codebase size, and the average search time is close to $4.6\times10^{(-9)}$. Fig. 11 (b) shows the search time for Gemini and Genius, these two approaches use the same propose LSH search method and the number of the feature vector are also same. Therefore, they have the same distribution of the search time.

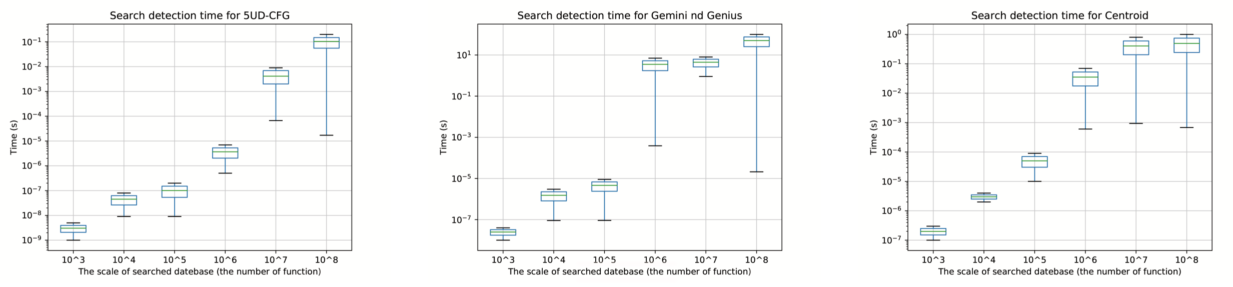


Fig. 11. Search time in scalable datbase.

We can see the search time of the 3TU-CFG run 2× faster than Genius and Gemini on average. Since the 3TU-CFG finally has 3 feature vector, half as many as the feature vector of Genius and Gemini. The search method also use the LSH search algorithm. Fig. 11 (c) shows the search time for Centroid, 3TU-CFG run 10× to 100×. Since Centroid use the binary search for 5 feature vectors, which is slower than LSH with 3 feature vector. Therefore, 3TU-CFG has the less search time for scalable database.

There have been several existing works for the function clone and other works for app clone:

8 Conclusion

In this paper, we present a two-hierarchies embedding model to generate a feature vector for extracted CFG of app's functions. We propose a prototype called 3TU-CFG, which has a scalable and effective power on the app homology analysis. The embedding feature vector is carefully designed objective functions that preserve both the first-order and second-order proximities of CFG for functions, which is called 5UD-CFG. At the same time, we can prove the monotonous of first-hierarchy embedding 5UD-CFG. An effective and efficient compress and clustering algorithm based on the second-hierarchy tensor embedding model is proposed for generating 3TU-CFG. Our extensive evaluation shows that 3TU-CFG outperforms the current other approaches by large margins with respect to similarity detection accuracy, embedding generation time and overall search time. The real dataset demonstrate that 3TU-CFG significantly improve detection accuracy and search effective.

Acknowledgment

The paper is supported by China NSF (61572222, 61272405,61272033, 61272451,61472121) and China University Innovation Foundation (2013TS102, 2013TS106).

References

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 95–109. [Online]. Available: https://doi.org/10.1109/SP.2012.16

[2] P. Teufl, M. Ferk, A. Fitzek, D. M. Hein, S. Kraxberger, and C. Orthacker, "Malware detection by applying knowledge discovery processes to application metadata on the android market (google play)," *Security and Communication Networks*, vol. 9, no. 5, pp. 389–419, 2016. [Online]. Available: https://doi.org/10.1002/sec.675

[3] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *2nd Working Conference on Reverse Engineering, WCRE '95, Toronto, Canada, July 14-16, 1995*, 1995, pp. 86–95. [Online]. Available: https://doi.org/10.1109/WCRE.1995.514697

[4] M. Iwamoto, S. Oshima, and T. Nakashima, "Token-based code clone detection technique in a student's programming exercise," in *2012 Seventh International Conference on Broadband, Wireless Computing, Communication and Applications, Victoria, BC, Canada, November 12-14, 2012*, 2012, pp. 650–655. [Online]. Available: https://doi.org/10.1109/BWCCA.2012.113

[5] Y. Yuan and Y. Guo, "Boreas: an accurate and scalable token-based approach to code clone detection," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 286–289. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351725

[6] Y. Chen, L. Gan, S. Zhang, W. Guo, Y. Chuang, and X. Zhao, "Plagiarism detection in homework based on image hashing," in *Data Science - Third International Conference of Pioneering Computer Scientists, Engineers and Educators, ICPCSEE 2017, Changsha, China, September 22-24, 2017, Proceedings, Part II*, 2017, pp. 424–432. [Online]. Available: https://doi.org/10.1007/978-981-10-6388-6_35

[7] Y. Zhang, H. Lu, L. Zhang, X. Ruan, and S. Sakai, "Video anomaly detection based on locality sensitive hashing filters," *Pattern Recognition*, vol. 59, pp. 302–311, 2016. [Online]. Available: https://doi.org/10.1016/j.patcog.2015.11.018

[8] X. Yu, J. Liu, Z. J. Yang, X. Liu, X. Yin, and S. Yi, "Bayesian network based program dependence graph for fault localization," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2016, Ottawa, ON, Canada, October 23-27, 2016*, 2016, pp. 181–188. [Online]. Available: https://doi.org/10.1109/ISSREW.2016.35

[9] A. Hamid and V. Zaytsev, "Detecting refactorable clones by slicing program dependence graphs," in *Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014.*, 2014, pp. 37–48. [Online]. Available: http://ceur-ws.org/Vol-1354/paper-04.pdf

[10] D. Fu, Y. Xu, H. Yu, and B. Yang, "WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 7 809 047:1–7 809 047:8, 2017. [Online]. Available: https://doi.org/10.1155/2017/7809047

[11] D. Hovemeyer, A. Hellas, A. Petersen, and J. Spacco, "Control-flow-only abstract syntax trees for analyzing students' programming progress," in *Proceedings of the 2016 ACM Conference on International Computing Education Research, ICER 2016, Melbourne, VIC, Australia, September 8-12, 2016*, 2016, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/2960310.2960326

[12] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino, "A source code plagiarism detecting method using sequence alignment

with abstract syntax tree elements," *IJSI*, vol. 3, no. 3, pp. 41–56, 2015. [Online]. Available: https://doi.org/10.4018/IJSI.2015070104

[13] M. Spreitzenbarth, "Dissecting the droid: Forensic analysis of android and its malicious applications (sezierung eines androiden)," Ph.D. dissertation, University of Erlangen-Nuremberg, 2013. [Online]. Available: http://www.opus.ub.uni-erlangen.de/opus/volltexte/2013/4528/

[14] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594343

[15] W. M. Khoo, A. Mycroft, and R. J. Anderson, "Rendezvous: a search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 329–338. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624046

[16] S. Andrews, B. S. Varunbabu, P. Subash, and M. R. Swaminathan, "Finding the high probabilistic potential fishing zone by accelerated SVM classification," *IJICT*, vol. 11, no. 4, pp. 576–585, 2017. [Online]. Available: https://doi.org/10.1504/IJICT.2017.10008318

[17] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978370

[18] Z. Liu, C. Chen, J. Zhou, X. Li, F. Xu, T. Chen, and L. Song, "POSTER: neural network-based graph embedding for malicious accounts detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 2543–2545. [Online]. Available: http://doi.acm.org/10.1145/3133956.3138827

[19] D. Fu, Y. Xu, H. Yu, and B. Yang, "WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 7 809 047:1–7 809 047:8, 2017. [Online]. Available: https://doi.org/10.1155/2017/7809047

[20] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Commun. ACM*, vol. 57, no. 3, pp. 99–106, 2014. [Online]. Available: http://doi.acm.org/10.1145/2494522

[21] R. Just, M. D. Ernst, and S. Millstein, "Collaborative verification of information flow for a high-assurance app store," in *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, 2015, p. 77.

[22] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, 2013, pp. 185–196. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435377

[23] 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 *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 659–674. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-kai

[24] Y. Du, J. Wang, and Q. Li, "An android malware detection approach using community structures of weighted function call graphs," *IEEE Access*, vol. 5, pp. 17 478–17 486, 2017. [Online]. Available: https://doi.org/10.1109/ACCESS.2017.2720160

[25] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket

[26] C. Platzer, M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, and V. van der Veen, "Andrubis - 1,000,000 apps later: A view on current android malware behaviors," 09 2014.

[27] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014. [Online]. Available: http://doi.acm.org/10.1145/2619091

[28] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 303–317. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[29] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, 2012, pp. 569–584. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan

[30] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "LINE: large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, 2015, pp. 1067–1077. [Online]. Available: http://doi.acm.org/10.1145/2736277.2741093

[31] K. Chandrasekaran, D. Dadush, V. Gandikota, and E. Grigorescu, "Lattice-based locality sensitive hashing is optimal," in *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, 2018, pp. 42:1–42:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ITCS.2018.42

[32] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[33] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," *it - Information Technology*, vol. 59, no. 2, p. 83, 2017. [Online]. Available: https://doi.org/10.1515/itit-2016-0040

[34] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[35] M. E. Kilmer and C. D. Martin, "Factorization strategies for third-order tensors ," *Linear Algebra & Its Applications*, vol. 435, no. 3, pp. 641–658, 2011.

[36] A. Thomasian, "Singular value decomposition, clustering, and indexing for similarity search for large data sets in high-dimensional spaces," in *Big Data - Algorithms, Analytics, and Applications.*, 2015, pp. 39–70. [Online]. Available: http://www.crcnetbase.com/doi/abs/10.1201/b18050-5

[37] L. Han, Z. Wu, K. Zeng, and X. Yang, "Online multilinear principal component analysis," *Neurocomputing*, vol. 275, pp. 888–896, 2018. [Online]. Available: https://doi.org/10.1016/j.neucom.2017.08.070

[38] Y. Gao, X. Cong, Y. Yang, Q. Wan, and G. Gui, "A tensor decomposition based multiway structured sparse SAR imaging algorithm with kronecker constraint," *Algorithms*, vol. 10, no. 1, p. 2, 2017. [Online]. Available: https://doi.org/10.3390/a10010002

[39] G. Gorrell, "Generalized hebbian algorithm for incremental singular value decomposition in natural language processing," in *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, April 3-7, 2006, Trento, Italy*, 2006. [Online]. Available: http://aclweb.org/anthology/E/E06/E06-1013.pdf

[40] X. Chen and K. S. Candan, "LWI-SVD: low-rank, windowed, incremental singular value decompositions on time-evolving data sets," in *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, 2014, pp. 987–996. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623671

[41] M. Antol and V. Dohnal, "Popularity-based ranking for fast approximate knn search," *Informatica, Lith. Acad. Sci.*, vol. 28, no. 1, pp. 1–21, 2017. [Online]. Available: https://content.iospress.com/articles/informatica/inf1130

[42] L. Kuang, F. Hao, L. T. Yang, M. Lin, C. Luo, and G. Min, "A tensor-based approach for big data representation and dimensionality reduction," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 3, pp. 280–291, 2014. [Online]. Available: https://doi.org/10.1109/TETC.2014.2330516

[43] Y. Zhao, L. T. Yang, and R. Zhang, "A tensor-based multiple clustering approach with its applications in automation systems," *IEEE Trans. Industrial Informatics*, vol. 14, no. 1, pp. 283–291, 2018. [Online]. Available: https://doi.org/10.1109/TII.2017.2748800

[44] "Reverse engineering, malware and goodware analysis of android applications ... and more," 2013. [Online]. Available: https://code.google.com/p/androguard/,2013