# A High Efficient Two-Hierarchies Tensor-Embedding-Aware App Clone Detection System

Your N. Here
*Your Institution*

Second Name
*Second Institution*

## Abstract

App homology analysis tasks on binary codes are expected to be scalable and effective. However, most works have a complex training 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 binary codes of the app. In our research, firstly, we implement a novel first-hierarchy embedding prototype for the control flow graph (CFG) of app's binary codes called 5UD-CFG that preserves the global CFG structure. Secondly, we propose the compression, clustering and update algorithms according to the proposed second-hierarchy tensor embedding model to obtain a more concise and accurate feature numeric vector called 3TU-CFG for app's binary codes. This model abandons the complex learning process of existing effective works that are based on the neural network or other learning algorithms. Thirdly, Empirical experiments prove that the effectiveness of two hierarchies CFG embedding on app homology analysis detection. 3TU-CFG run $4\times$ to $76000\times$ faster than the existing works' preprocessing and at least reduce the half of search time. At the same time, the homoloy detection accuracy approximates 99%.

## 1 Introduction

The study of the app homology has recently emerged as a major catalyst for collectively understanding the behavior of complex app in the mobile. 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 the rapidly increasing use of apps, code clone, malware code injection bring several serious finance threats. 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 [44], for example, crack paid apps to bypass payment function, modify the advertisement libraries, or even insert various malicious functions, which lead 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 [36]. The problem here is that the static approach does not work on 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 [5], detection based on token[25][40], detection based on hash [10][41], detection based on the program dependence graph [39] [22] and detection based on abstract syntax tree [18] [24] [28] have shown their scalability to handle thousands and millions of codes. 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 [34], 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:

1. There has been an primitive challenge on designing a feasible model to accurately represent scalable apps at the binary codes level. The binary codes of

app are very complicated source data. They appear on different markets, and most apps' binary codes are confused. We need extract considerable features to denote binary codes.

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 rapidly 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.

Our goal is to address these challenges by proposing a suitable embedding system to transform binary codes of the app to a low-dimensional feature vector. First, Since apps' binary codes are complex, CFG 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 CFG. Second, the embedding features that are extracted from the CFG of app's binary codes are scalable, and tensor computations have a great effect on data reduction. We need to make the feature dimensionality reduced and compressed. Third, based on tensor computations, it is easier to propose an incremental update system of the app features. Moreover, we also need to prove the monotonicity and the validity of embedding features 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 CFG of app's binary codes to handle the app homology analysis. 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 extracting CFG that each vertex has five eigenvalues by decompiling the app, which is called as 5UD-CFG. Moreover, we can theoretically prove that first-hierarchy embedding feature 5UD-CFG can uniquely represent a CFG.

2. We design the second-hierarchy 3TU-CFG tensor embedding model based on the extracting first-hierarchy embedding feature 5UD-CFG. Then we propose an special tensor-SVD factorization algorithm to decompose tensor model, which compress 5UD-CFG as 3TU-CFG. This is second-hierarchy embedding process that compresses and clusters the first-hierarchy embedding feature matrixes to greatly accelerate the efficiency of the match and the classification.

3. We propose an incremental tensor decomposition 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 $4.6 \times 10^{(-9)}$ seconds in a thousand apps and 0.1 seconds in $10^8$ apps.

## 2 Problem Description

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

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

1. Extracting the feature of CFG of app binary codes needs to preserve both 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 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 CFG features. Moreover, the availability of the reducing feature should be proved theoretically and practically.

3. The rapidly increasing novel apps cause the previous feature data that should be updated 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 to compute the novel app features excluding 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.

**Definition 1.** *(CFG extraction with the basic block feature)*. We extract CFG that each vertex is with five unique feature called 5UD-CFG, 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 feature with five numeric elements based on the statistical features of basic blocks. 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, *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 (a) shows a real extracting CFG.

Given a set of training CFGs $C = [c_1,\ c_2,\ ...,\ c_{N_T}] \in R^{m\times 5\times N_T}$ 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 eigenvalues denoted as $G = (V, E)$, CFG embedding aims to learn a mapping function $f : \sum_1^m \vec{x}_i \mapsto y_i \in R^d$, 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 the *first-order call structure* and the *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 $l_{i,j} = 1$, there exists an directed first-order call structure between the vertex $v_i$ and $v_j$. 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 $v_u$ and other vertexes. $|V|$ is the number of neighborhoods. The second-order call structure is determined by the similarity of $N_u$ and $N_v$.

Intuitively, the second-order call structure assumes that if two vertexes share many common basic blocks, they may have more connections. Such an assumption has been proved reasonable in many fields [35]. A father basic block is most frequently called in other basic blocks.

**Definition 5.** *(Second hierarchy embedding tensor model)* To represent all CFGs' features of all apps, 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 total training number of CFGs. $tn$ can be expanded with the development of apps. $I_1$ is the label of the app. Using the label $I_1$, we can partition app communities that can be easier used for the app homology analysis.

## 3 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 our approach, we show binary codes of app represented by 3TU-CFG.

We propose the two-hierarchies embedding mapping model for the extracting CFG from app's binary codes. The two hierarchies embedding model satisfy several highlights: first, it extracts the CFG that each vertex has five unique eigenvalues. This CFG with feature is referred to the raw feature, from a decompiling binary function of app. It remains 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 handle the scalable and considerable CFG feature data, say millions and billions apps' homology 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 embedding compressing and clustering*, 4) *3TU-CFG tensor embedding update*. 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 eigenvalues, and embedding them into a 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 embed the feature vector that is generated by the first-hierarchy embedding process to a more low-dimensionality feature vector, and this model also provide an incremental feature data update approach (Section 5). Finally, given a app, we use the KNN [6] to find its most similarity app in feature data. 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 fast KNN search [3] 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 generates 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. Moreover, the tensor computations make the feature data updated easier in the second-hierarchy tensor embedding process.

## 4 First-hierarchy 5UD-CFG embedding model

In this section, we propose the detailed approaches how the model generates 5UD-CFG from Android apps. The traditional CFG is the common feature used in bug search. Moreover, different from other attributes on basic blocks, such as I/O pairs and statics features [16] [32], are more accuracy matching. Following the idea of the traditional CFG extraction, this paper utilizes CFG with different basic block level attributes with five eigenvalues called 5UD-CFG. We also prove the monotonicity of 5UD-CFG, a 5UD-CFG vector represents a CFG.

Fig. 1(b) shows a primitive extracting CFG from a function. A vertex represents a basic block, and a edge represents a call link between two basic blocks. 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 $A$ represent the the basic block $A$ calls other basic blocks.

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 for representing the CFG. The embedding vector monotonously represent the CFG. We define the feature of a 5UD-CFG based on the 3D-CFG [7].

**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 $k$.

The key factor of feature vector of the 5UD-CFG embedding is the vertex weight $\omega$, and we need to show the monotonicity of 5UD-CFG. For each directed edge $(i,\ j)$ 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 call 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:

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


(a) (b) (c)

Figure 1: (a) shows an example of real CFG; (b) shows extracting CFG with the basic block; (c) shows an example of CFG with embedding feature; (a) is used in this Section 3, (b) and (c) are used in Section 5.

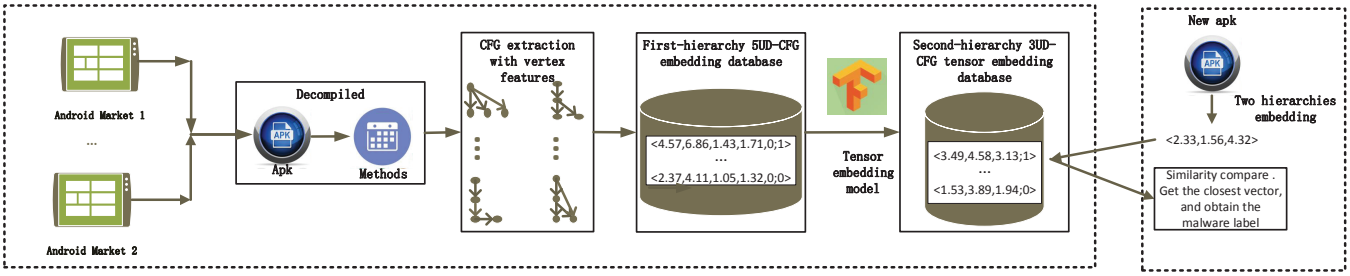



Figure 2: Overview

where $d(.,.)$ is the distance between two distributions. We minimize the KL-divergence [35] of two probability distributions. Therefore, the objective function is:

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

The second-order call structure assumes that the basic block (vertex) are invoked by the another vertexes exclude 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:

$$Link_2(v_j|v_i) = \frac{exp(\vec{x}_j^{\,T}\vec{x}_i)}{\sum_{k=1}^{|V|} exp(\vec{x}_k^{\,T}\vec{x}_i)}, \quad (4)$$

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

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

where $d(.,.)$ is the distance between two distributions. We minimize the KL-divergence [35] of two probability distributions. Therefore, the objective function is as follows:

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

where $l_{i,j}$ is the connection between vertex $v_i$ and vertex $v_j$. If $l_{i,j} = 1$, there exists a edge from 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 parameters $\omega_i$ for each vertex. We adopt Maximum Likelihood Estimate for optimizing Eq (3) and (6). We sample all edges $(i, j)$ in a CFG, the gradient of weight of vertex with 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 probability distributions between original $link_{\{1\ or\ 2\}}$ and embedding $\widehat{link_{\{1\ or\ 2\}}}$. From Eq. (2) and Eq. (4), we know that original $link_{\{1\ or\ 2\}}$ is monotonous. When the gradient $\nabla_1 + \nabla_2 \to 0$, $link_1 \approx \widehat{link_1}$ and $link_2 \approx \widehat{link_2}$. The $link_1$ and $link_2$ are monotonous, so the embedding $\widehat{link_1}$ and $\widehat{link_2}$ are monotonous and $W = [\omega_1,\ ...,\ \omega_n]$ are monotonous. 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 $W = [\omega_1,\ \omega_2,\ ...,\ \omega_n] mod(n-1)$, where $n$ is the number of nodes in the CFG.

Form above analysis, we know that the vertex weight $\omega_j$ 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 $W$. Therefore, they have different embedding vector. Fig.1 (c) shows an example of CFG that contains embedding parameters and basic blocks with features. There is an example for computing 5UD-CFG, which is shown in the appendix.

## 5 Second-hierarchy Tensor Embedding model

According to the extracting 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 is suitable for achieving scalable and accurate app homology analysis. The second-hierarchy embedding process is the tensor embedding, which generates 3UD-CFG based on 5UD-CFG.

### 5.1 Tensor embedding model

We propose a tensor embedding model for generating a 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 Definition 5, we can set $I_1$ of the second-hierarchy embedding tensor model firstly. For example, if the tensor embedding model is used to malware detection, we can set $I_1$ as the malware label of the app, $I_1$ can be defined as $I_1 = 1$ *if this method is from the begin app,* $I_1 = 0$ *if this method is from the malware app,*

We use the accepted notation where an order-3 tensor is indexed by 3 indices and can be represented as a multi-dimensional array of data [29]. 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 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. 3, we cut slices for the tensor embedding model expand in the horizontal orientation.

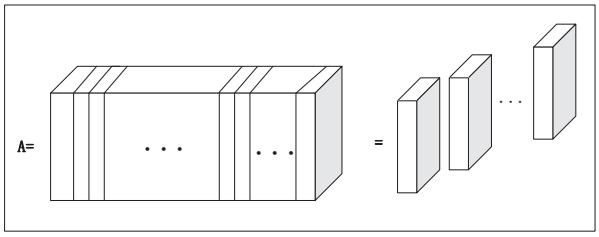



Figure 3: Lateral slices of the tensor model $A$

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 element $t_{i_1 i_2}$ at the position with row number $f_i$ and column number $n_i$, $i_1$ is No. of the feature vector with five eigenvalues. $i_2$ is No. of the CFG.

The following section, we will discuss how the function is decomposed in the tensor embedding model.

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

First we review the Singular Value Decomposition (SVD) [37] [23] as compressing algorithm for 5UD-CFG. An SVD of the tensor model that consists of all apps' 5UD-CFG embedding vector is $A = UDV^T$. 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 [20] 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 $X_l$.

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 $\Psi_l$ 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 unfolding is minimized.

Based on the SVD factorization [21] [9] of proposed tensor model, 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 embedding model. The core tensor and truncated bases of SVD decomposition described in the preliminaries can be employed to make considerable 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 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 decreases 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 app features from app samples as the homology search database. Algorithm 1 shows the compression algorithm for 3TU-CFG tensor embedding model $A$ by the SVD factorization.

**Algorithm 1** T-SVD-Compare

**Require:**

**Input**:
tensor model $A = R^{m \times tn}$, testing app feature set B;
[U,S,V]=SVD(y);
U2=U1(:,1:2);
T=real(U2'*A); compress feature
save T as the comparable object;
V1=clustering(V)
V3=centroid$\{V1\{\}\}$ V3 is the centroid vector in clustering set V1 $\{\}$
A1=T*V3
A2=T*V1,
B1=U2'*B compress detected app feature
B2=B1*V3
Compare (B2, B), get the closest element in the set V1
B3=B1*V1the corresponding element
Compare(B3, A2the corresponding element)
**Output**: The closest comparable result.

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 op-

timal approximation of a tensor is closed to the original tensor. We prove the $A \approx A_k$ in our model as follows:

**Proof 1** *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 three maximum singular values, 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.

## 5.3 Clustering of 3UD-CFG tensor embedding model

After 5UD-CFG embedding model is decomposed as 3UD-CFG tensor embedding model by the SVD algorithm, we need to cluster the decomposing unitary orthogonal matrices $V_k$. The clustering result is generated from a training set of $S_k \times V_k$. 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 collections 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 homology analysis data. 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 $B$ in $SC_i$ or $SC_j$. The process is as follows:

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

Object is the result that we want to get.

Comparison between $SV_c$, $SV_i$ or $SV_j$ and $B$ is a search problem. For 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 [3]. 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 in 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 $t$ distances; 3) Output 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 contains two parts:$O\ (tq)$ multiplication for the $tn \times m$ 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 above methods, 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 2** *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 $U_k \in R^{m\times k\times 1}$*

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

*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.

## 5.4 Incremental tensor embedding for updating

Based on the recursive incremental HOSVD proposed by the Liwei Kuang [30] [42], we propose a incremental tensor model for the expanding app homology analysis model. The special steps is shown in the Algorithm. 2.

**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 $U_{k_{i-1}}$, $Span = U_{k_{i-1}}^T \times C_{i-1}$.
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.

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

# 6 Evaluation

Our model consists of three main components: CFG extractor for each app, 5UD-CFG embedding and the 3TU-CFG tensor embedding work based on the compressing and the clustering. We obtain the CFG extractor for each app, a open project Androguard [1], 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 brute-force KNN searching in nearpy of Python.

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

## 6.1 Dataset

**Dataset I - Game app dataset.** This data set mainly contains all kinds of game. Size of most games is more than 50$MB$, some even is more than 150$MB$. Game apps are 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 20$MB$ to 60$MB$. 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 datasets. Size of entertainment app from 10$MB$ to 50$MB$. We collect $43,885$ entertainment apps from five 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.

**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.

## 6.2 App Homology Analysis Comparison

We compare the 3TU-CFG 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 homology dataset. We use the metric of the false positive rate to evaluate the accuracy. 3TU-CFG Embedding time and the search time respectively represents the preparation efficiency and the search efficiency.

We prepare three representative app clone analysis or bug search techniques to compare our evaluation: Binary search based the Centroid [7], Gemini based Neural Network [31], Genius [17]. We will introduce these three approaches in appendix.

## 6.3 Accuracy Comparison

In this Section, we evaluate the accuracy of a two-hierarchies embedding model and search work. We train embedding feature database from all collective apps as the search database. According to five types' markets, we obtain five basic search databases. We randomly select one or several apps from the testing dataset as the input. We assume that it is unknown about testing 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 basic five databases, and another part is from the novel collective apps. The number of apps 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 testing samples. It is not necessary to find a query for each search sample. Fig. 4 (a) 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 larger, some similarity apps may be defined as the different apps with a large probability. Therefore, TPR is decreasing with the similarity threshold decreasing.

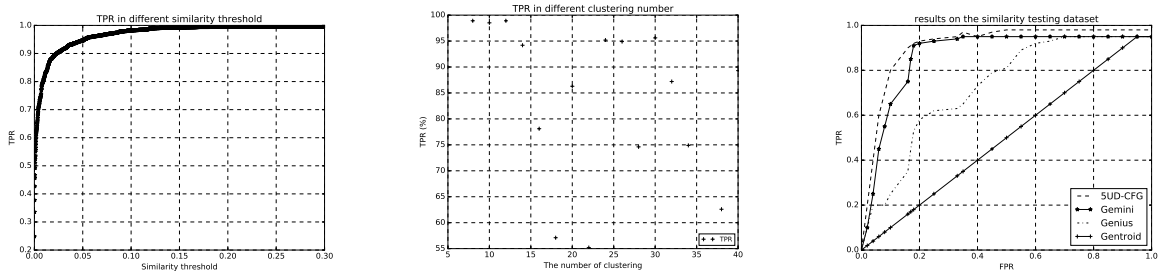
Based on the proposed clustering algorithm in Algorithm 1, the different clustering's number affects TPR. Fig. 4 (b) shows that there is a inflection point in the changing curve of relationship between the clustering number and the TPR. We can see that TPR is the largest if the clustering number is 0.014% of the number of search database.

To compare the efficacy of final 3TU-CFG with centroid, gemini and genius, we use the same testing data to calculate the ROC curve in the same threshold. We use two metrics to evaluate the accuracy of the proposed and compared methodsthe true positive rate (TPR) and the false positive rate (FPR).

We use 2082 apps in the testing database as the queries to match with the target approaches. Fig. 4 (c) shows evaluating 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 FPR is small, TPR is significantly better than other approaches. Four curves show that TPR is increasing with FPR increasing. However, the rate of ROC curve of 3TU-CFG has the fastest growing. The results shows the 3TU-CFG can achieve even better accuracy than other approaches.

We consider search 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 compressing 5UD-CFG. Each function just need a closest candidate to make sure the accuracy for 3TU-CFG embedding process. The Gemini need 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 codebook. Centroid has a great gap between different platforms, 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.




(a) (b) (c)

Figure 4: Accuracy

## 6.4 Efficiency Comparison

We measure the scalability of proposed approach from the following three aspects: the scale of the five collective markets, the performance on app homology detection and update of the basic feature database.

We analyze all apps in the collective five markets to obtain a basic database. Fig. 5 shows 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. We train in total $152,789$ apps are in these five groups. Fig. 5 (a) shows that the size of nearly 94.7% apps is in $0-50MB$. The total size of apps is 7.47 $TB$. The Fig. 5 (b) shows that nearly 69.8% apps have $1-16000$ methods, and nearly 77.6% apps have more than 4000 methods.

We investigate the time consumption for each stage to demonstrate that 3TU-CFG is capable of handling apps at a large scale: 1) CFG with attributes extraction time, 2) embedding generation time, 3) Update time, and 4)app homology search time.

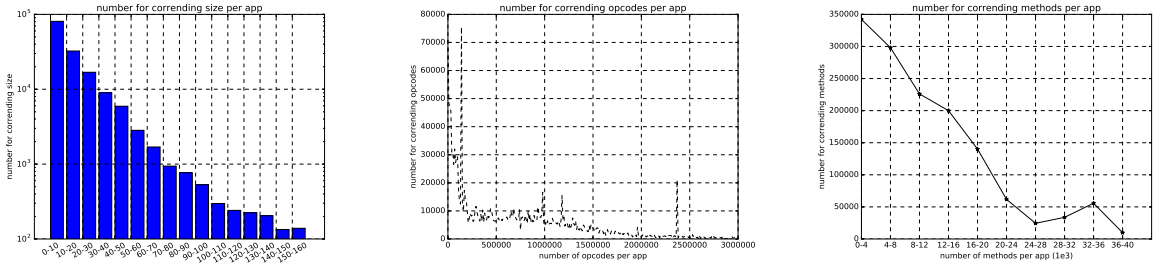
The time of the embedding generation depends on the total size of apps. We need to decompile the app into a series of functions. This step can be done in parallel. We use the androguard to obtain the original CFG structure. The original CFG generation time is the decompile time of the app. We can use several computer to measure the time. At the same time, Fig. 6 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 KNN to search and detection. Then we evaluate the efficiency of Gemini, Genius for embedding time and Centroid for centroid generation time.

### 6.4.1 Performance on CFG with attributes extraction time.

Fig. 6 (a) shows the CFG with attributes extraction times of 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 sizes 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 5UD-CFG is close to Centroid.

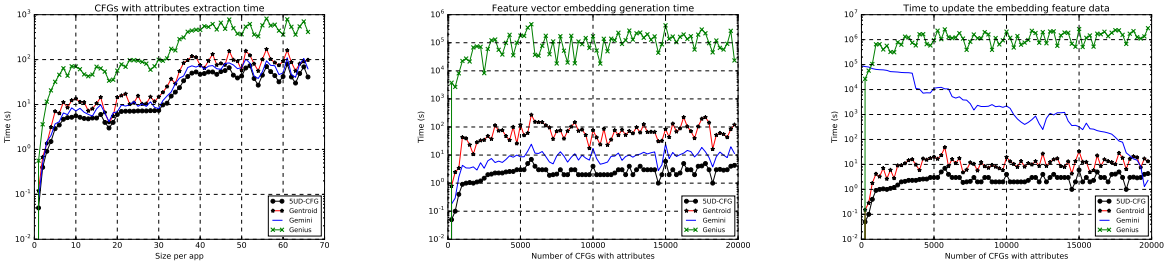
### 6.4.2 Performance on embedding generation time.

Fig. 6 (b) shows the embedding generation time for four approaches with the increasing 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 embedding vector. We can see 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 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 obtaining the CFG embedding feature. Centroid is to compute the centroid of a spatial geometry. This preparation process is simpler than the above



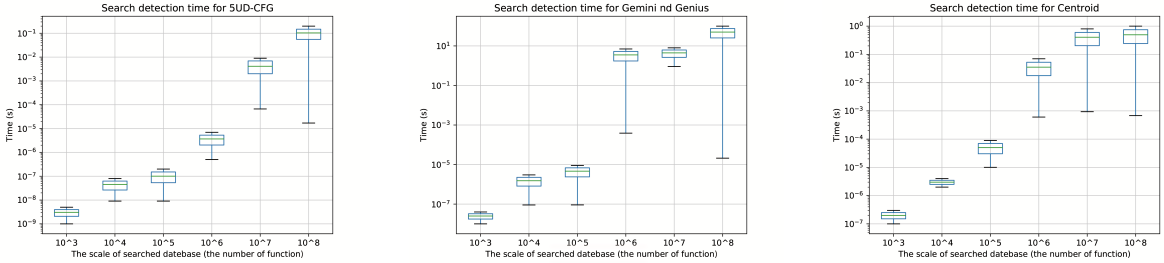

(a) (b) (c)

Figure 5: Scalability of the database.




(a) (b) (c)

Figure 6: Efficiency evaluation.




(a) (b) (c)

Figure 7: Search time in scalable feature data.

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.

### 6.4.3 Performance on Update time.

Fig. 6 (c) shows the update feature 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× to 20000× than Gemini, quicker more than 180000× than Genius, and quicker 2× to 7× 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 incremental algorithm of tenor embedding that just needs to retrain the increasing novel methods rather than retrain all feature vector. If the number of novel methods are same with the number of original feature data. The update embedding time of 3TU-CFG is close to Gemini. Since the Gemini need to embed all ACFGs to get the update feature data although it does not need to regenerate ACFGs. Specifically, Genius needs to retrain the codebook, this is a larger project, which means regenerate all feature vector including scalable graph matching algorithm. It costs

too much time. Centroid just needs to update the increasing apps. However, it's accuracy is lower than the 3TU-CFG. Therefore, 3TU-CFG has a better performance on the update.

#### 6.4.4 App homology search time.

Fig. 7 (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 $4.6 \times 10^{(-9)}$. Fig. 7 (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. We can see the search time of the 3TU-CFG run $2\times$ faster than Genius and Gemini on average. Since the 3TU-CFG finally has a 3-eigenvalues vector, half as many as the feature vector of Genius and Gemini. The search method use the KNN search algorithm. Fig. 7 (c) shows the search time for Centroid, 3TU-CFG run $10\times$ to $100\times$. Since Centroid use the binary search for a 5-eigenvalues vectors, which is slower than KNN with a 3-eigenvalues vector. Therefore, 3TU-CFG has the less search time for scalable database.

## 7 Related Work

We discuss the closest related works 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 function clone detection.** One common approaches is that trace-based approach [11] 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 [27][2] 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 [27] 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] [31] 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.

**App clone detection.** Most existing approaches identify malicious apps [14] [26] typically rely on heavyweight static or dynamic analysis techniques, and cannot detect the unknown malware whose behavior has not been modeled a priori. PiggyApp[43] [8] [12] [4] 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. In dynamic app cone analysis, applications are executed in a virtual environment. Runtime information is recorded to generate dynamic features for malware detection. For example, ANDRUBIS [4][33] 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. [15] [13] [38] 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.

## 8 Conclusion

In this paper, we present a two-hierarchies embedding model to generate a feature vector for extracting 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] Reverse engineering, malware and goodware analysis of android applications ... and more, 2013.

[2] ANDREWS, S., VARUNBABU, B. S., SUBASH, P., AND SWAMINATHAN, M. R. Finding the high probabilistic potential fishing zone by accelerated SVM classification. *IJICT 11*, 4 (2017), 576–585.

[3] ANTOL, M., AND DOHNAL, V. Popularity-based ranking for fast approximate knn search. *Informatica, Lith. Acad. Sci. 28*, 1 (2017), 1–21.

[4] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. 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).

[5] BAKER, B. S. 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.

[6] CHANDRASEKARAN, K., DADUSH, D., GANDIKOTA, V., AND GRIGORESCU, E. 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.

[7] CHEN, K., LIU, P., AND ZHANG, Y. 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.

[8] CHEN, K., WANG, P., LEE, Y., WANG, X., ZHANG, N., HUANG, H., ZOU, W., AND LIU, P. 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.

[9] CHEN, X., AND CANDAN, K. S. 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.

[10] CHEN, Y., GAN, L., ZHANG, S., GUO, W., CHUANG, Y., AND ZHAO, X. 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.

[11] DAVID, Y., AND YAHAV, E. 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.

[12] DU, Y., WANG, J., AND LI, Q. An android malware detection approach using community structures of weighted function call graphs. *IEEE Access 5* (2017), 17478–17486.

[13] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. 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.

[14] ENCK, W., GILBERT, P., CHUN, B., COX, L. P., JUNG, J., MCDANIEL, P. D., AND SHETH, A. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM 57*, 3 (2014), 99–106.

[15] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B., COX, L. P., JUNG, J., MCDANIEL, P. D., AND SHETH, A. N. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst. 32*, 2 (2014), 5:1–5:29.

[16] ESCHWEILER, S., YAKDAN, K., AND GERHARDS-PADILLA, E. 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).

[17] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. 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.

[18] FU, D., XU, Y., YU, H., AND YANG, B. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming 2017* (2017), 7809047:1–7809047:8.

[19] FU, D., XU, Y., YU, H., AND YANG, B. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming 2017* (2017), 7809047:1–7809047:8.

[20] GAO, Y., CONG, X., YANG, Y., WAN, Q., AND GUI, G. A tensor decomposition based multiway structured sparse SAR imaging algorithm with kronecker constraint. *Algorithms 10*, 1 (2017), 2.

[21] GORRELL, G. 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).

[22] HAMID, A., AND ZAYTSEV, V. 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.

[23] HAN, L., WU, Z., ZENG, K., AND YANG, X. Online multilinear principal component analysis. *Neurocomputing 275* (2018), 888–896.

[24] HOVEMEYER, D., HELLAS, A., PETERSEN, A., AND SPACCO, J. 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.

[25] IWAMOTO, M., OSHIMA, S., AND NAKASHIMA, T. 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.

[26] JUST, R., ERNST, M. D., AND MILLSTEIN, S. 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.

[27] KHOO, W. M., MYCROFT, A., AND ANDERSON, R. J. 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.

[28] KIKUCHI, H., GOTO, T., WAKATSUKI, M., AND NISHINO, T. A source code plagiarism detecting method using sequence alignment with abstract syntax tree elements. *IJSI 3*, 3 (2015), 41–56.

[29] KILMER, M. E., AND MARTIN, C. D. Factorization strategies for third-order tensors . *Linear Algebra & Its Applications 435*, 3 (2011), 641–658.

[30] KUANG, L., HAO, F., YANG, L. T., LIN, M., LUO, C., AND MIN, G. A tensor-based approach for big data representation and dimensionality reduction. *IEEE Trans. Emerging Topics Comput. 2*, 3 (2014), 280–291.

[31] LIU, Z., CHEN, C., ZHOU, J., LI, X., XU, F., CHEN, T., AND SONG, L. 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.

[32] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture bug search in binary executables. *it - Information Technology 59*, 2 (2017), 83.

[33] PLATZER, C., LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., AND VAN DER VEEN, V. Andrubis - 1,000,000 apps later: A view on current android malware behaviors, 09 2014.

[34] SPREITZENBARTH, M. *Dissecting the Droid: Forensic Analysis of Android and its malicious Applications (Sezierung eines Androiden)*. PhD thesis, University of Erlangen-Nuremberg, 2013.

[35] TANG, J., QU, M., WANG, M., ZHANG, M., YAN, J., AND MEI, Q. 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.

[36] TEUFL, P., FERK, M., FITZEK, A., HEIN, D. M., KRAXBERGER, S., AND ORTHACKER, C. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). *Security and Communication Networks 9*, 5 (2016), 389–419.

[37] THOMASIAN, A. 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.

[38] YAN, L., AND YIN, H. 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.

[39] YU, X., LIU, J., YANG, Z. J., LIU, X., YIN, X., AND YI, S. 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.

[40] YUAN, Y., AND GUO, Y. 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.

[41] ZHANG, Y., LU, H., ZHANG, L., RUAN, X., AND SAKAI, S. Video anomaly detection based on locality sensitive hashing filters. *Pattern Recognition 59* (2016), 302–311.

[42] ZHAO, Y., YANG, L. T., AND ZHANG, R. A tensor-based multiple clustering approach with its applications in automation systems. *IEEE Trans. Industrial Informatics 14*, 1 (2018), 283–291.

[43] ZHOU, W., ZHOU, Y., GRACE, M. C., JIANG, X., AND ZOU, S. 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.

[44] ZHOU, Y., AND JIANG, X. 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.