Title

Codee: A Tensor Embedding Scheme for Binary Code Search

Permalink
https://escholarship.org/uc/item/5t74h0rn

Journal
IEEE Transactions on Software Engineering, PP(99)

ISSN
0098-5589

Authors
Yang, J
Fu, C
Liu, XY
et al.

Publication Date
2021

DOI
10.1109/TSE.2021.3056139

Peer reviewed

Codee: A Tensor Embedding Scheme for Binary Code Search

Jia Yang, Cai Fu, *Member, IEEE,* Xiao-Yang Liu, *Member, IEEE,* Heng Yin, *Senior Member, IEEE,* and Pan Zhou, *Senior Member, IEEE,*

Abstract—Given a target binary function, the binary code search retrieves top-K similar functions in the repository, and similar functions represent that they are compiled from the same source codes. Searching binary code is particularly challenging due to large variations of compiler tool-chains and options and CPU architectures, as well as thousands of binary codes. Furthermore, there are some pivotal issues in current binary code search schemes, including inaccurate text-based or token-based analysis, slow graph matching, or complex deep learning processes. In this paper, we present an unsupervised tensor embedding scheme, **Codee**, to carry out code search efficiently and accurately at the binary function level. First, we use an NLP-based neural network to generate the semantic-aware token embedding. Second, we propose an efficient basic block embedding generation algorithm based on the network representation learning model. We learn both the semantic information of instructions and the control flow structural information to generate the basic block embedding. Then we use all basic block embeddings in a function to obtain a variable-length function feature vector. Third, we build a tensor to generate function embeddings based on the tensor singular value decomposition, which compresses the variable-length vectors into short fixed-length vectors to facilitate efficient search afterward. We further propose a dynamic tensor compression algorithm to incrementally update the function embedding database. Finally, we use the local sensitive hash method to find the top-K similar matching functions in the repository. Compared with state-of-the-art cross-optimization-level code search schemes, such as Asm2Vec and DeepBinDiff, our scheme achieves higher average search accuracy, shorter feature vectors, and faster feature generation performance using four datasets, OpenSSL, Coreutils, libgmp and libcurl. Compared with other cross-platform and cross-optimization-level code search schemes, such as Gemini, Safe, the average recall of our method also outperforms others.

Index Terms—Function feature extraction; tensor embedding; code search; tSVD.

1 Introduction

Binary code search aims to quantitatively retrieve the similar binary functions from a number of functions in the repository [1]. Moreover, if given two input binary functions, it can precisely measure an accurate similarity score. Hence, code search has recently emerged as a popular topic for solving software security problems, such as finding clone code injection [2], [3], detecting software plagiarism [4], [5], and solving copyright issues [6], [7]. We focus on a cross-platform and cross-optimization level of the binary similarity problem, where we define two similar binary functions that are compiled from the same source code. Inspired by [8], [9], we look for solutions that solve the binary similarity problem using embeddings. Loosely speaking, each binary function is first transformed into a vector of numbers (an embedding), in such a way that code compiled from the same source results in vectors that are similar. Given that the number of binary executable files grows dramatically every year and in most cases the source code is unavailable, it is crucial to design an effective code search scheme at the binary level.

Some recent works [9], [8], [10], [11] propose to convert a piece of binary code (e.g., a function) into a numeric vector (or embedded vector, or simply embedding), such that two pieces of binary code with equivalent semantics would be mapped to two embeddings that are close to each other. The problem of binary code search is then converted into the problem of searching for similar vectors, which can be done in $O(1)$ time by using Locality Sensitive Hashing (LSH) [12]. These embedding-based schemes offer several unique advantages. First of all, most of these schemes are very efficient and scalable, and thus can deal with large code repositories. Second, almost all embedding-based schemes can tolerate (at least to some extent) changes introduced by different compiler options, and some of them can even tolerate changes caused by different instruction set architecture (ISAs) [9], [8]. In general, the latest existing state-of-the-art works can be divided into two categories.

Deep learning-based Works. Some of these schemes (Gemini [9], CDLH [13], and EKLAVYA [14]) leverage deep neural network models. To achieve high search accuracy, it is essential to collect a substantial amount of high-quality training data, which can be challenging in practice. Even with good training data, these deep neural network models may still suffer an overfitting problem. In particular, Gemini [9] empirically demonstrated that a generic model did not work well, and a task-specific model should be trained iteratively, which was a time-consuming and tedious process. Moreover, each basic block feature vector is generated only by manually extracting statistical features of instructions, which may lose some semantic information.

Jia Yang, Cai Fu, and Pan Zhou are with School of Cyber Science and Engineering, Huazhong University of Science and Technology, and Hubei Engineering Research Center on Big Data Security, Wuhan, China, 430074. E-mail: d201780841@hust.edu.cn, fucai@hust.edu.cn
Xiao-Yang Liu is with Electrical Engineering Department, Columbia University, New York, NY, USA, 10027.
Heng Yin is with Department of Computer Science and Engineering, University of California, Riverside, CA, USA, 92521.

CDLH [13] learned the functional similarity of source codes using the lexical and syntactical information by supervised deep learning methods. This technique devotes to the source code analysis and cannot be immediately used on binary code.

Safe [15] generated function embedding based on a self-attentive neural network, which uses a skip-gram method to generate instruction embeddings and fed these instruction embeddings into a bi-directional recurrent neural network. However, the first part of Safe used the NLP-based learning process, and the second part of Safe used a complex bi-directional recurrent neural network to generate function embedding. These two neural networks all learn the sequence information of instructions. Safe only considers the binary function as language, but loses the structural information of binary function. Each basic block embedding generation has to go through complex computations of two neural networks. Massarelli [16] improved the manual extraction part of Gemini as the NLP-based unsupervised feature learning method, which preserved the same Structure2Vec deep learning part as Gemini. This method also has two complex deep learning processes. Thus, the performances of these methods are heavily dependent on the quality of the training data. It is difficult to collect high-quality training dataset due to the diversity and scalability of the binary functions. Moreover, supervised learning methods could suffer the overfitting problem.

NLP-based Works. The latest works (Asm2Vec [10], InnerEye [11], and DeepBinDiff [17]) have leverage Natural Language Processing (NLP) techniques to generate binary function embeddings for the binary code search. Asm2Vec [10] treated a function as a document, and a token (opcode or operand) as a word in the document, and thus adopts the Distributed Memory Model of Paragraph Vectors (PV-DM model) [18] to generate an embedding for each function. InnerEye [11] regarded basic blocks as sentences to train the model by using Neural Machine Translation (NMT). InnerEye [11] has a scalability issue for code search. Each calculation of basic block similarity has to go through a complex LSTM-RNN neural network with a large number of parameters in the current implementation, which significantly affects the performance. While they are intriguing ideas, directly applying NLP techniques on binary codes is not ideal. Binary codes have strict control flow and data flow dependency, and thus is more structured than natural languages. Simply applying random walk and a small sliding window technique will not be able to fully capture the high-level features of a piece of code, leading to degraded search accuracy. Moreover, Asm2Vec cannot deal with binary code from multiple ISAs, and its embedding generation is much slower than deep learning-based schemes, due to its iterative gradient descent optimization process.

DeepBinDiff [17] focused on the basic blocks matching based on the unsupervised program-wide code representation learning technique. However, DeepBinDiff did not support cross-architecture binary code search. Moreover, many hyperparameters need to be determined to maximize the model performance. Only applying NMT techniques in the binary code search has some challenges. In normal NMT, a word embedding model is usually trained once using large corpora, such as Wiki. However, we need to train an instruction embedding model. The out-of-vocabulary (OVV) problem is a well-known problem in NLP-based methods. How to deal with the OOV problem is a challenge. Order Matters [19] used BERT to pre-train the binary code on one token-level task, one block-level task, and two graph-level tasks. They also adopt a convolution neural network (CNN) on adjacency matrices to extract the order information. However, BERT needs an amount of corpus to pre-train the model. BERT has to go through a complex neural network with many parameters, which significantly affects performance and needs strong computation ability. Order Matters is a very large-scale model, which is much larger than the other methods by several orders of magnitude.

In this paper, we propose a tensor-based approach, called **Codee**, to the problem of binary code search. To learn semantic information from basic blocks, we use the NLP-based technique to learn token representation (opcode representation or operand representation). Specifically, we modify a skip-gram model with negative sampling [20] to extract semantic information for tokens (opcodes and operands). After obtaining all token embeddings in an assembly function, we model the basic block embedding generation as a network representation learning problem. We propose an improved algorithm for the basic block embedding generation based on two network embedding algorithms (AANE [21] and LINE [22]). Then we feed the basic block embeddings into tensor to generate function embeddings. To facilitate efficient search, we use a tensor singular value decomposition algorithm (tSVD) [23] to compress all basic block embeddings into a compact function embedding. The tSVD extracts principal features by conducting a circular convolution operation. The tSVD computation can handle more general multilinear data as long as the data is shown to be compressible in the tSVD based representation. Furthermore, we design a dynamic tensor compression method to enable incremental update of the embedding database. When processing new binary programs, this dynamic tensor compression method only needs to generate embeddings for new functions, rather than recomputing the entire database.

Compared with the existing embedding-based schemes, our tensor embedding-based scheme has several advantages:

- No complex training is needed. We do not need to label an amount of data, we only need an one-effort pre-training process, and our embedding-based scheme does not suffer an overfitting problem.
- Higher accuracy. Our tensor embedding-based scheme achieves high accuracy from two aspects. First, the generated basic block embedding not only captures the syntactic information and semantic information of the basic blocks, but also learns the contextual information from the CFG structure. Second, the tensor singular value decomposition (tSVD) algorithm processes feature embeddings across all binary programs to learn the correlation of functions and extract the main features of the function.
- Faster embedding generation. This is mainly due to the parallel processing nature of tSVD tensor decom-

position. After all basic blocks are generated and fed into the tensor, decomposition is performed, and all function embeddings are generated at once.

In summary, we have made the following contributions in this paper.

- **NLP-based token embedding generation.** To improve the semantic-aware ability of our model, we propose a skip-gram pre-training model to generate the token-level embedding. The NLP-based training process is based on the entire ICFG diagram and does not require knowing any prior knowledge of assembly language. We first extract semantic information of basic blocks. we use the node2vecWalk method to scale to the random walk generation time in the ICFG. [20]. Then, we use a skip-gram model with negative sampling to generate the token embeddings from the instruction sequences of ICFG. Negative sampling and node2vecWalk are proposed for effective training.
- **Network representation-based basic block embedding generation.** We devise a novel method to deal with basic block embedding generation in binary code. We propose an optimization function of basic block embedding generation as a network representation learning problem for capturing both semantic information of basic blocks and the structural information of CFG.
- **Tensor-based function embedding generation.** To the best of our knowledge, we propose the first tensor-based approach to the problem of binary code embedding generation, which relies on a circular convolution operation to learn the correlation of functions and extract the main features of the function. It is an essential step to generate function-level embedding for scalable and accurate binary code search. Moreover, a dynamic tensor compression algorithm is proposed to incrementally update the function embedding database.
- We conduct extensive experiments on a large dataset with different compilers, compiler optimization levels, and CPU architectures. Our experimental evaluations demonstrate that Codee outperforms the other latest state-of-the-art embedding-based schemes (Safe [15], DeepBinDiff [17], Gemini [9], and Asm2Vec [10]) concerning higher accuracy, faster embedding generation, and shorter embeddings.

The remainder of this paper is organized as follows. The problem statement and an overview of the proposed scheme are given in Section 2. Section 3 shows how to generate token embedding based on NLP techniques. In Section 4, we propose a basic block embedding generation algorithm. In Section 5, we propose a function embedding method based on tensor computations. Section 6 shows experiments to verify the efficiency and accuracy of the proposed scheme, Codee. Section 7 shows the limitations of our method. Section 8 discusses the related works. Section 9 concludes the paper.

2 Problem Statement and Scheme Overview

2.1 Problem Statement

Binary code search aims to quantitatively measure the similarity between two given binary functions. We define that two binary functions f_1, f_2 are similar if they are compiled from the same original source function with different compiler configurations and different architectures. The source function is written in source code, (e.g, C or C++). In our paper, we evaluate the baseline schemes with different architectures (x86-64, ARM, and MIPS), different compilers (GCC5.4.0 and CLANG3.8.0) and different optimization levels (O0-O3).

Given a target binary function f_t, the code search task is to find functions in the repository that are similar, and achieves high accuracy and high efficiency. The similar functions means that assembly functions may appear different syntactic codes, but have similar functional logic in their source code. To quickly and accurately search binary codes (e.g., a function), we seek to map each function into a low-dimensional feature vector. The goal of the code similarity embedding problem is to find a mapping, which maps the CFG and instructions of a function f_t to a vector representation r. We formally define the search problem as follows:

Definition 1 *Given a target assembly function f, we obtain its representation vector $\mathbf{r}_t$ by $\mathbf{r}_t = E(f)$, where $E()$ represents the function embedding algorithm. Then we use a similarity search algorithm $SIM(\mathbf{r}_t, \mathcal{R})$ to retrieve the top-K function embeddings in the repository $\mathcal{R}$, and these top-K most similar function embeddings are ranked by their similarity values. Retrieve top-K most similar function embeddings by:*

$$SIM(\mathbf{r}_t, \mathcal{R}) = sort(sim(\mathbf{r}_t, \mathbf{r}_{s_1}), sim(\mathbf{r}_t, \mathbf{r}_{s_2}), ..., sim(\mathbf{r}_t, \mathbf{r}_{s_N})), \tag{1}$$

where $\mathbf{r}_{s_i} \in \mathcal{R}$, $i = \{1, 2, ..., N\}$, $sim()$ is the similarity function, such as cosine similarity function, and $sort()$ is the sort function. There are N function embeddings in the repository, and $\mathbf{r}_{s_i}$ is a function embedding in the repository.

The similarity between two assembly functions can be computed using a normal similarity function between two vectors, such as cosine similarity function and others, without incurring the cost of expensive graph matching algorithms. In our problem definition, the repository function stands for the assembly function that is indexed inside the repository; and the target function denotes the assembly function query. Given an assembly function, our goal is to search for its similar functions from the repository $\mathcal{R}$.

2.2 Notations

First, we denote some notations for designing the optimization function of the basic block embedding generation. The i-th column of a matrix $\mathbf{C}$ is denoted by $\mathbf{C}_i$. The (i, j)-th element of matrix $\mathbf{C}$ is denoted by c_{ij}. $\mathbf{C}^H$ denotes the (Hermitian) transpose of $\mathbf{C}$, while $|\mathbf{c}|$ denotes the length of vector $\mathbf{c}$. The symbol $|| \cdot ||_2$ represents the Euclidean norm, and $|| \cdot ||_F$ denotes the Frobenius norm. An asterisk $(*)$ denotes the tensor product [23]. The term vec($\mathbf{C}$) denotes the vectorization operation. We consider a 3-D tensor $\mathcal{F} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, where n_1, n_2, n_3 are tensor dimensions. The i-th frontal slice matrix of tensor $\mathcal{F}$ is denoted as $\mathbf{F}^{(i)}$, and $\mathbf{F}^{(i)} = \mathcal{F}(:, :, i)$. Table 1 lists the main symbols and

TABLE 1: Notations

Notations	Definitions
$N(i)$	The set of adjacent basic blocks
$n = \|V\|$	The number of basic blocks in a CFG
$\mathbf{A} \in \mathbb{R}^{n\times n}$	The adjacency matrix of a CFG
$\mathbf{B} \in \mathbb{R}^{m\times n}$	The basic block feature vector matrix
$\mathbf{S} \in \mathbb{R}^{n\times n}$	The affinity matrix of basic block features
$\mathbf{C} \in \mathbb{R}^{d\times n}$	The basic block embedding matrix
$\mathcal{F} \in \mathbb{R}^{n_1\times n_2\times n_3}$	The tensor representation

their definitions. Let $G = (V, E)$ be the Control Flow Graph (CFG) of a function. $N(i)$ denotes the set of adjacent basic blocks of basic block i.

2.3 Background

In this work, we will use the tensor singular value decomposition (tSVD) algorithm to generate the function embedding and use the locality sensitive hash (LSH) to search for similar assembly functions. Therefore, we introduce the LSH and tSVD algorithms in the background.

LSH [12]. We assume that a set of hash functions $H = h : S \leftarrow U$, and H is (r_1, r_2, p_1, p_2) sensitive. For each function $h \in H$, it satisfies the following two conditions:

$$\Pr[h(O_1) = h(O_2)] \begin{cases} \geq p_1 & \text{if } d(O_1, O_2) \leq r_1 \\ \leq p_2 & \text{if } d(O_1, O_2) > r_2 \end{cases},$$

where $O_1,\ O_2 \in S$ are two high-dimensional objects, $d(O_1, O_2)$ is the difference of two objects. If O_1 is similar to O_2, they are mapped as the same hash value with high probability. If O_1 is different from O_2, they are less likely to be mapped as the same hash value.

Symmetric matrix decomposition.

Given a symmetric matrix $\mathbf{X}$, the goal of the symmetric matrix decomposition is to seek a lower-rank matrix approximation [24], and it can be formulated as

$$\mathbf{X} = \mathbf{G}\mathbf{G}^H.$$

To obtain the decomposed matrix $\mathbf{G}$ of symmetric matrix $\mathbf{X}$, the Frobenius norm is used to measure the distance between $\mathbf{X}$ and $\mathbf{G}\mathbf{G}^H$. Hence, the problem of symmetric matrix decomposition can be formulated as [24]

$$\min_{\mathbf{G}\leq 0} ||\mathbf{X} - \mathbf{G}\mathbf{G}^H||_F^2.$$

The second-order proximity. The general notion of the second-order proximity can be interpreted as nodes with shared neighbors being likely to be similar [22].

Definition 2 *The second-order proximity of a pair of nodes (i, j) represents the similarity between their neighborhood graph structures [22]. Formally, let $p_i = (w_{i,1}, ..., w_{i,|V|})$ denote the connected weight of i with all the other nodes, then the second-order proximity between i and j is calculated according to the similarity between p_i and p_j. If there is no link from/to both i and j, the second-order proximity between i and j is 0.*

The second-order proximity is applicable for both directed and undirected graphs. Each node is also considered as a specific "context", and if nodes with similar "contexts" distributions are assumed to be similar.

Before introducing tSVD, we show some related tensor operations.

Circulant matrices [23]. Circ($\mathcal{F}$) is a circulant matrix that can be diagonalized by multiplying the normalized Fast Fourier Transform (FFT) matrix [25].

Block diagonal matrix [23]. The diag($\mathbf{D}^{(1)}, \mathbf{D}^{(2)}, .., \mathbf{D}^{(n_3)}$) is the block diagonal matrix. Each $\mathbf{D}^{(i)}$ is an $n_1 \times n_2$ matrix, and is also a frontal slice of the tensor $\mathcal{D}$.

Tensor product [23]. If $\mathcal{X} \in \mathbb{R}^{l_1\times l_2\times l_3}$ and $\mathcal{Y} \in \mathbb{R}^{l_2\times l_4\times l_3}$, the tensor product $\mathcal{X} * \mathcal{Y}$ is a $l_1 \times l_4 \times l_3$ tensor, having

$$\mathcal{X} * \mathcal{Y} = \text{fold}(\text{Circ}(\mathcal{X}) \cdot \text{Matvec}(\mathcal{Y})),$$

where Matvec($\mathcal{Y}$) takes the tensor $\mathcal{Y}$ into a block $l_2 l_3 \times l_4$ matrix. Matvec($\mathcal{Y}$)= $[\mathbf{Y}^{(1)}; \mathbf{Y}^{(2)}; ...; \mathbf{Y}^{(l_3)}]$, and the operator fold($\cdot$) makes the matrix into the tensor.

Tensor transpose [23]. $\mathcal{X}$ is the $l_1 \times l_2 \times l_3$ tensor, then we obtain the $l_2 \times l_1 \times l_3$ transposed tensor $\mathcal{X}^\dagger$ by transposing each frontal slice matrix $\mathbf{X}^{(i)} \in \mathbb{R}^{l_1\times l_2}$ and then reversing the order of transposed frontal slices from 2 to l_3.

Orthogonal tensor [23]. An $n_1 \times n_1 \times n_3$ tensor $\mathcal{U}$ is orthogonal if $\mathcal{U}^\dagger * \mathcal{U} = \mathcal{U} * \mathcal{U}^\dagger = \mathcal{I}_{n_1 n_2 n_3}$, where $\mathcal{I} \in \mathbb{R}^{n_2\times n_2\times n_3}$ is an identity tensor. The first frontal slice of $\mathcal{I}$ is an identity matrix of size $n_2 \times n_2$, and other frontal slices are all zero matrices. We have:

$$\sum_{i=1}^{n_3} (\mathcal{U}(:,:,i)\mathcal{U}(:,:,i)^H) = \mathbf{I}_{n_1},$$
$$\sum_{i\neq j} (\mathcal{U}(:,:,i)\mathcal{U}(:,:,j)^H) = \mathbf{0}_{n_1}.$$

Tensor singular value decomposition (tSVD) [23]. The tSVD is defined as follows:

$$\mathcal{F} = \mathcal{U} * \mathcal{S} * \mathcal{V}^\dagger,$$

where $\mathcal{U} \in \mathbb{R}^{n_1\times n_1\times n_3}$ and $\mathcal{V} \in \mathbb{R}^{n_2\times n_2\times n_3}$ are the left and right singular orthogonal tensor, respectively.

In the tSVD process, first, $\widetilde{\mathcal{F}}$ is obtained by taking the FFT along the third dimension of $\mathcal{F}$. That is, we first obtain the circulant matrix Circ($\mathcal{F}$), then we make Circ($\mathcal{F}$) into the Fourier domain. Circ($\mathcal{F}$) is transformed as a block diagonal matrix. Next, we do the truncated SVD decomposition of each diagonal block matrix $\mathbf{D}^{(i)}$, $\mathbf{D}^{(i)} = \widetilde{\mathbf{U}}^{(i)}\widetilde{\mathbf{M}}^{(i)}\widetilde{\mathbf{V}}^{(i)H}$ in parallel. Then we fold each slice matrix $\widetilde{\mathbf{U}}^{(i)}$ into an orthogonal tensor $\widetilde{\mathcal{U}}$, $\widetilde{\mathcal{U}} = \text{fold}([\widetilde{\mathbf{U}}^{(1)}; \widetilde{\mathbf{U}}^{(2)}; ...; \widetilde{\mathbf{U}}^{(n_3)}])$. Then, we obtain $\mathcal{U}$, $\mathcal{S}$, and $\mathcal{V}$ by performing the inverse FFT (IFFT) operation along the third dimension of $\widetilde{\mathcal{U}}$, $\widetilde{\mathcal{S}}$ and $\widetilde{\mathcal{V}}$. Fig. 2 shows a graphical representation for the computation of tSVD algorithm.

We show the tSVD algorithm in the following Algorithm 1 [23]:

2.4 Scheme Overview

Given a binary function, the code search task is to find similar functions in the repository, and achieve high accuracy and high efficiency. To quickly and accurately search binary codes (e.g., a function), we seek to map each function into

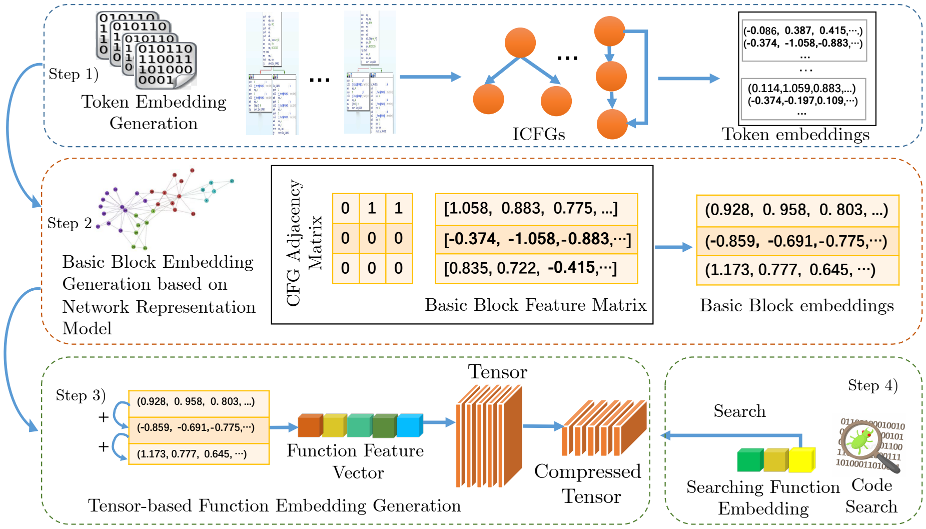


Fig. 1: Overview of the tensor embedding-based scheme.

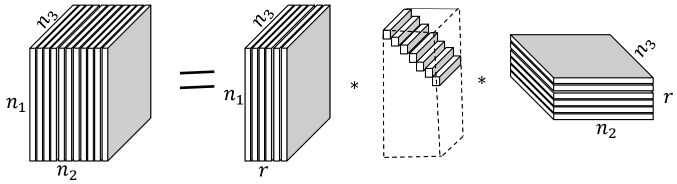


Fig. 2: The (reduced) tSVD for a tensor of size $n_1 \times n_2 \times n_3$ and tubal-rank r.

Algorithm 1: tSVD [23]

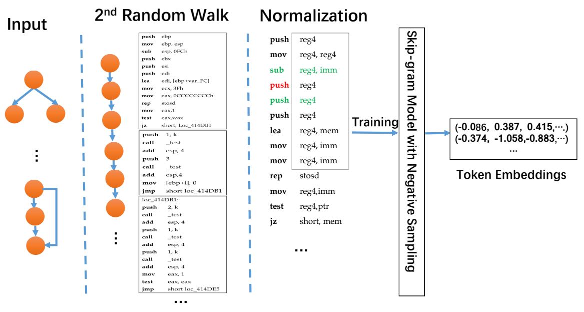
```
Input: $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$.
Output: $[\mathcal{U}, \mathcal{S}, \mathcal{V}]$.
$\widetilde{\mathcal{X}} = \text{fft}(\mathcal{X}, [\,], 3)$,
for $k = 1, 2, 3..., n_3$ do
    $[\widetilde{\mathbf{U}}, \widetilde{\mathbf{S}}, \widetilde{\mathbf{V}}] = \text{SVD}(\widetilde{\mathcal{X}}(:, :, k))$,
    $\widetilde{\mathcal{U}}^{(k)} = \widetilde{\mathbf{U}},\ \widetilde{\mathcal{S}}^{(k)} = \widetilde{\mathbf{S}},\ \widetilde{\mathcal{V}}^{(k)} = \widetilde{\mathbf{V}}$,
$\mathcal{U} = \text{ifft}(\widetilde{\mathcal{U}}, [\,], 3)$, $\mathcal{S} = \text{ifft}(\widetilde{\mathcal{S}}, [\,], 3)$, $\mathcal{V} = \text{ifft}(\widetilde{\mathcal{V}}, [\,], 3)$.
```

a low-dimensional feature vector, and achieve the following design goals:

- **Accuracy**. The generated embedding should accurately represent the binary function to ensure an accurate code search.
- **Robustness**. The generated embedding can tolerate changes caused by different compilers, different compilation optimization levels, and different CPU architectures.
- **Compact representation**. The generated embedding should be reasonably short, such that both search and storage of embeddings are efficient.
- **Fast embedding generation**. The embedding generation process must be fast enough to process the large volume of binary code.

As shown in Figure 1, our scheme follows 4 steps: 1) *Token embedding generation based on NLP technique*, 2) *basic block embedding generation based on network representation technique*, 3) *function embedding based on tensor computation*, and 4) *binary code search*. The first step aims to generate the token embedding based on inter- procedural control-flow graphs (ICFG) of the program using the skip-gram model with negative sampling, which extracts the semantic information of basic blocks. The second step aims to use the generated token embedding and the structural information of CFG to generate basic block embedding. The third is a function embedding method based on the tensor computation, which uses a tensor decomposition algorithm to further simultaneously generate all function embeddings. Moreover, we propose a dynamic tensor compression algorithm for updating the repository. Finally, given a target function, we search top-K similar functions in the repository using LSH.

This proposed scheme can achieve accuracy and robustness for the following reasons: 1) we generate the token embedding based on the ICFG of program, which extracts semantic information and lexical information of basic blocks. 2) We leverage the network representation method with token embeddings to learn the semantic information and the structural information for basic blocks, which simultaneously capture its intra- and inter- basic block features. The intra- basic block feature represents the semantic information and the lexical information of instructions (opcode and operand). The inter- basic block feature represents the semantic information of basic blocks and the structural information (contextual information) between basic blocks. 3) In tSVD process, we rely on a circular convolution operation to extract principal information from all basic block embeddings within a function to generate the function embedding.



```
push   ebp
mov    ebp, esp
sub    esp, 0FCh
push   ebx
push   esi
push   edi
lea    edi, [ebp+var_FC]
mov    ecx, 3Fh
mov    eax, 0CCCCCCCCh
rep    stosd
mov    eax,1
test   eax,wax
jz     short, Loc_414DB1
```

```
push   1, k
call   _test
add    esp, 4
push   3
call   _test
add    esp,4
mov    [ebp+i], 0
jmp    short loc_414DB1
```

```
loc_414DB1:
push   2, k
call   _test
add    esp, 4
push   1, k
call   _test
add    esp, 4
push   1, k
call   _test
add    esp, 4
mov    eax, 1
test   eax, eax
jmp    short loc_414DE5
...
```

```
push   reg4
mov    reg4, reg4
sub    reg4, imm
push   reg4
push   reg4
push   reg4
lea    reg4, mem
mov    reg4, imm
mov    reg4, imm
rep    stosd
mov    reg4,imm
test   reg4,ptr
jz     short, mem
...
```

```
(-0.086, 0.387, 0.415,···)
(-0.374, -1.058,-0.883,···)
...
```

Fig. 3: Overview of the token embedding generation scheme.

3 Token Embedding Generation

Token embedding generation analyzes instructions in the inter- procedural control flow graph (ICFG) of binary functions. To generate token embeddings (e.g. opcode embedding and operand embedding), we use instructions of ICFGs as the input for the skip-gram model. Figure 3 shows the process of token embedding generation.

Given all programs in a repository, we first build an NLP-based learning model to obtain all token embeddings. We do not need to know any prior knowledge of assembly language. We avoid to manually specify all instruction sets of assembly language. To better learn the semantic information and the lexical information of instructions, we use the dependency information of the program, which is extracted from ICFG. The ICFG contains some contextual information of instructions, which can be used to differentiate semantically similar basic blocks in different contexts.

In this section, we proposed an unsupervised NLP method based on the program-level ICFG. Specifically, we modify a skip-gram algorithm with negative sampling to extract semantic information for basic blocks [20]. We take into account the contextual semantic information for learning token embedding by generating token sequences from ICFG.

The number of instructions in the ICFG of a program is much larger than the number of instructions in the CFG of a function, which may lead to a low efficiency problem. To solve this problem, we use the node2vecWalk method to scale to the random walk generation time. Node2vecWalk algorithm can efficiently explore diverse neighbor instructions. Moreover, we use a skip-gram model with negative sampling to train the token embeddings. Negative sampling can effectively reduce the training time of the NLP-based neural network.

Specifically, we have the following steps:

1) Generate the token sequences by using the node2vecWalk random walks method.
2) Normalize the serialized codes by defining some specific notations.
3) Train the skip-gram model with negative sampling to obtain the token embedding.

Before performing a skip-gram training model, we need to generate the sequence of all instructions in a program. This sequence needs to preserve control flow dependency information between basic blocks, and decides the context of instructions. The context of instructions is used to extract the semantics of each token. First, we generate random walks in ICFG by using node2vecWalk method [20]. The node2vecWalk method uses a 2nd-order random walk approach to generate context of nodes. Node2vecWalk generates a set of biased random walks, which can nearly cover all basic blocks. This random walk set contains diverse neighborhood sets for a target node. Each random walk contains one possible binary execution path. In ICFG, the edge from the entrance instruction to the exit instruction is a random walk. We put all random walks together to generate a total instruction sequence for training.

Before training the total instruction sequence, we need to filter some tokens to refine the binary code, Codee performs the normalization after obtaining the instruction sequence in a CFG. We use the following rules to normalize: 1) all immediate values are represented as the special symbol 'imm'; 2) all base memory addresses are represented as the special symbol 'mem'. 3) general registers are renamed according to their lengths; 4) pointers are replaced with string ptr. The 3) and 4) are the same as the normalization of DeepBinDiff [17]. We do this filtering because we believe that there is a small benefit from raw operands.

Codee inputs the generated total instruction sequence into the skip-gram model with negative sampling [18]. In our case, we consider each token (opcode or operand) as a node, and consider generated random walks of ICFG as sentences. The neighbors of each token are as its context. Model training is only an one-time effort. We use a skip-gram algorithm with negative sampling to capture the contextual semantic information of nodes. The contextual information comes from the random walk sequence of instructions in ICFGs.

Skip-gram algorithm with negative sampling defines its objective function as follows [18]:

$$\max_{f} \sum_{i \in V} [-\log Z_i + \sum_{n_j \in N(i)} f(n_j) \cdot f(i)], \tag{2}$$

where $Z_i = \sum_{v \in V} \exp(f(i) \cdot f(v))$, f is the mapping function from nodes to feature representations, and $N(i) \subset V$ is a neighborhood of token i generated through a neighborhood sampling strategy.

As shown in Figure 3, we can see that the target token is "push " (red color), One instruction before and after this target token in the random walk is as the context (e.g., sub reg4, imm and push reg4, shown in green color). If the target instruction is at the block boundary (e.g., entry instruction or exit instruction in the block), it has one adjacent instruction as its context.

4 Basic Block Embedding Generation

After obtaining the token embeddings, we generate the feature vectors of basic blocks. The purpose of basic block embedding generation is to obtain a low dimensional representation based on the generated token embeddings and the structural information of CFG. As shown in the Fig. 3, a binary function contains several basic blocks, each basic block includes an instruction or multiple instructions, and each instruction contain one opcode and one operand or multiple operands. According to token embeddings (opcode embedding and operand embedding) within a basic block, we concatenate the opcode embedding with the average feature vector of the operand embeddings to generate instruction embedding. Then we further sum up all instruction embeddings within a basic block to compute the feature vector of basic block.

We model the basic block embedding generation problem as a network representation learning problem. We feed the basic block feature vectors and function control flow contextual information into the proposed basic block embedding algorithm. Motivated by two efficient network embedding frameworks, AANE [21] and LINE [22], we design a basic block embedding generation algorithm. We combine control-flow graph structural information and semantic information of the basic blocks to generate high-quality basic block embeddings. Hence, the generated basic block embedding contains intra- and inter- basic-block features. In the following contents, we first propose the objective function of the basic block embedding generation. Then we give the iteration steps to compute the basic block embeddings.

4.1 Loss Function

Based on the CFG and the basic block feature vectors generated in the prior steps, we propose an algorithm to generate basic block embeddings. The similar basic blocks can be represented as similar embeddings. First, we use each basic block feature vector within a CFG generated in the prior steps as a row of basic block feature vector matrix $\mathbf{B}$. Then we obtain the affinity matrix of basic block feature vectors $\mathbf{S}$. Each element in $\mathbf{S}$ can be calculated as cosine similarity between two basic block feature vectors, $s_{ij} = \text{cosine}(\mathbf{B}_i, \mathbf{B}_j)$, where the i-th column $\mathbf{B}_i$ of $\mathbf{B}$ is the feature vector of the basic block i. The element s_{ij} shows the similarity between two basic block feature vectors in a function, which approximates the product of the generated basic block embeddings $\mathbf{C}_i$ and $\mathbf{C}_j^H$, where $\mathbf{C}_i$ and $\mathbf{C}_j$ are the i-th and j-th basic block embeddings of $\mathbf{C}$. $\mathbf{S}$ is a symmetric matrix that can be decomposed as $\mathbf{C}^H\mathbf{C}$, where $\mathbf{C} \in \mathbb{R}^{d\times n}$, $d \leq n$, is the basic block embedding matrix. Based on the symmetric matrix decomposition, the key factor is to force the product of two basic block embeddings to be the same as its corresponding similarity s_{ij}. The loss function is defined as:

$$L_S = ||\mathbf{S} - \mathbf{C}^H\mathbf{C}||_F^2 = \sum_{i=1}^{n} ||\mathbf{S}_i - \mathbf{C}^H\mathbf{C}_i||_2^2, \tag{3}$$

where $\mathbf{S}_i$ is the i-th column of $\mathbf{S}$, $\mathbf{S}_i$ shows similarity values between the i-th basic block feature vector and other basic block feature vectors in a CFG.

The generated basic block embedding also needs to consider the transitions between basic blocks, which is the structural information of a CFG. Graph embedding is used to transform the graph into a vector [9], [22]. In the CFG of a binary function, an edge represents the probabilistic dependency relationship between two basic blocks, and the code execution result of one father basic block affects the children basic blocks. In CFG, a basic block is a node, the *second-order* proximity [22] denotes the conditional probability of node j generated by node i using $\mathbf{C}_i$ and $\mathbf{C}_j$ [26], which shows the probability of a random jump from node i to node j. The empirical probability of *second-order* proximity between the node i and j is defined as $\frac{a_{ij}}{o_i}$, where a_{ij} is the weight of the edge (i, j) and o_i is the out-degree of node i. According to the adjacency matrix, we know $a_{ij} = 0$ or $a_{ij} = 1$. For a directed edge (i, j), the node transition probability is defined as follows:

$$p(j|i) = \frac{\exp(\mathbf{C}_j^H\mathbf{C}_i)}{\sum_{l\in V}\exp(\mathbf{C}_l^H\mathbf{C}_i)}.$$

This *second-order* proximity exists between any pair of connected nodes in a graph [22]. To obtain the function feature vector, we use the following loss function by minimizing the KL-divergence distance between the empirical probability of a directed edge and its conditional probability in a CFG, then we simplify the loss function:

$$L_W = -\sum_{j\in N(i)} a_{ij}\log p(j|i). \tag{4}$$

We use L_S and L_W to embed the function based on the influence of the structural information of CFG and the semantic information of basic blocks. We consider the dual influence of Eq. (3) and Eq. (4), and obtain the following loss function:

$$\min_{\mathbf{C}}\ L = \sum_{i=1}^{n} ||\mathbf{S}_i - \mathbf{C}^H\mathbf{C}_i||_2^2 + \lambda(-\sum_{j\in N(i)} a_{ij}\log p(j|i)). \tag{5}$$

4.2 The Proposed ADMM Algorithm

The alternating direction method of multipliers (ADMM) is a popular iteration algorithm, which was efficient to solve optimization problems with multiple non-smooth terms in the objective function [27]. Most existing works show that ADMM is a simple but useful algorithm that is good at distributed convex optimization. We follow the ADMM algorithm to solve the loss function of our proposed basic block embedding algorithm.

Scalar λ denotes a trade-off between the contributions of the structural information of CFG and the semantic information of basic block. The smaller λ is, the less influence weight of CFG topology is considered in the basic block embedding vector $\mathbf{C}_i$. In this work, we choose the $\lambda = 1$, such that the CFG topology and the basic block features have the same influence in $\mathbf{C}$. Section 6.10 evaluates different selections of λ.

Problem (5) is separable for $\mathbf{C}_i$ and reformulated as a bi-convex optimization problem. We copy $\mathbf{C} = \mathbf{H}$. Eq. (5) can be rewritten as

$$\begin{aligned}
&\min_{\mathbf{C}} \sum_{i=1}^{n} ||\mathbf{S}_i - \mathbf{H}^H\mathbf{C}_i||_2^2 - \lambda\sum_{j\in N(i)} a_{ij}\log\frac{\exp(\mathbf{H}_j^H\mathbf{C}_i)}{\sum_{l\in V}\exp(\mathbf{H}_l^H\mathbf{C}_i)} \\
&= \min_{\mathbf{C}} \sum_{i=1}^{n} ||\mathbf{S}_i - \mathbf{H}^H\mathbf{C}_i||_2^2 - \lambda\sum_{j\in N(i)} a_{ij}\mathbf{H}_j^H\mathbf{C}_i \\
&+ \lambda\sum_{j\in N(i)} a_{ij}\log(\sum_{l\in V}\exp(\mathbf{H}_l^H\mathbf{C}_i)) \\
&\text{subject to}\qquad \mathbf{C}_i = \mathbf{H}_i,
\end{aligned} \tag{6}$$

where $\mathbf{H}_i$ and $\mathbf{H}_j$ are the i-th and j-th column of $\mathbf{H}$. Since the Euclidean norm is convex, the first part of Eq. (6) is convex. Since the linear function is convex, the second part of Eq. (6) is convex. Since the exponential linear function is convex, the third part of Eq. (6) is convex too. Therefore Eq. (6) is convex when $\mathbf{C}_i$ is fixed, and $\mathbf{H}_i$ is convex, and vice versa. Motivated by the Alternating Direction Method of Multipliers (ADMM) [27], we use the augmented Lagrangian to formulate Eq. (6):

$$\begin{aligned} L =& \sum_{i=1}^{n} ||\mathbf{S}_i - \mathbf{H}^H \mathbf{C}_i||_2^2 - \lambda \sum_{j \in N(i)} a_{ij} \log \frac{\exp(\mathbf{H}_j^H \mathbf{C}_i)}{\sum_{l \in V} \exp(\mathbf{H}_l^H \mathbf{C}_i)} \\ &+ \frac{\rho}{2} \sum_{i=1}^{n} (||\mathbf{C}_i - \mathbf{H}_i + \mathbf{Z}_i||_2^2 - ||\mathbf{Z}_i||_2^2), \end{aligned} \tag{7}$$

where $\mathbf{S}_i$ is the i-th column of $\mathbf{S}$, columns $\mathbf{Z}_1, \mathbf{Z}_2, ..., \mathbf{Z}_n \in \mathbb{R}^d$ are dual variables, and $\rho > 0$ is the penalty parameter.

To minimize L, it is converted to find the saddle point. Thus finding optimal $\mathbf{H}$ and $\mathbf{C}$. The corresponding optimization problems in terms of each basic block i at iteration $t+1$ are formulated as

$$\begin{aligned} \mathbf{C}_i^{t+1} =& \arg\min_{\mathbf{C}_i} ||\mathbf{S}_i - {\mathbf{H}^t}^H \mathbf{C}_i||_2^2 + \frac{\rho}{2} ||\mathbf{C}_i - \mathbf{H}_i^t + \mathbf{Z}_i^t||_2^2 \\ &- \lambda \sum_{j \in N(i)} a_{ij} \log \frac{\exp({\mathbf{H}_j^t}^H \mathbf{C}_i)}{\sum_{l \in V} \exp({\mathbf{H}_l^t}^H \mathbf{C}_i)}, \end{aligned} \tag{8}$$

and

$$\begin{aligned} \mathbf{H}_i^{t+1} =& \arg\min_{\mathbf{H}_i} ||\mathbf{S}_i^H - \mathbf{H}_i^H \mathbf{C}^{t+1}||_2^2 + \frac{\rho}{2} ||\mathbf{H}_i - \mathbf{C}_i^{t+1} - \mathbf{Z}_i^t||_2^2 \\ &- \lambda \sum_{j \in N(i)} a_{ji} \log \frac{\exp({\mathbf{C}_j^{t+1}}^H \mathbf{H}_i)}{\sum_{l \in V} \exp({\mathbf{C}_l^{t+1}}^H \mathbf{H}_i)}. \end{aligned} \tag{9}$$

To minimize L, it is converted to find the saddle point. Thus finding optimal $\mathbf{H}$ and $\mathbf{C}$. We obtain the derivative of Eq. (7) w.r.t. $\mathbf{C}_i$ and the derivative of Eq. (7) w.r.t. $\mathbf{H}_i$ that are shown as follows:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{C}_i} =& -2\mathbf{H}^t \mathbf{S}_i + 2\mathbf{H}^t (\mathbf{H}^t)^H \mathbf{C}_i^{t+1} - \lambda \sum_{j \in N(i)} a_{ij} \mathbf{H}_j^t + \\ & \lambda \sum_{j \in N(i)} a_{ij} \frac{\sum_{l \in V} \mathbf{H}_l^t \exp((\mathbf{H}_l^t)^H \mathbf{C}_i^t)}{\sum_{l \in V} \exp((\mathbf{H}_l^t)^H \mathbf{C}_i^t)} + \rho(\mathbf{C}_i^{t+1} - \mathbf{H}_i^t + \mathbf{Z}_i^t), \end{aligned} \tag{10}$$

and

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{H}_i} =& -2\mathbf{C}^{t+1} \mathbf{S}_i + 2\mathbf{C}^{t+1} (\mathbf{C}^{t+1})^H \mathbf{H}_i^{t+1} - \lambda \sum_{j \in N(i)} a_{ij} \mathbf{C}_j^t + \\ & \lambda \sum_{j \in N(i)} a_{ij} \frac{\sum_{l \in V} \mathbf{C}_l^{t+1} \exp({\mathbf{C}_l^{t+1}}^H \mathbf{H}_i^t)}{\sum_{l \in V} \exp({\mathbf{C}_l^{t+1}}^H \mathbf{H}_i^t)} \\ & + \rho(\mathbf{H}_i^{t+1} - \mathbf{C}_i^{t+1} - \mathbf{Z}_i^t). \end{aligned} \tag{11}$$

Let their derivatives equal to zero.

Let (10) equal to zero. We obtain the update equations of $\mathbf{C}_i^{t+1}$. Since Eq. (6) is convex, $\mathbf{C}_i$ is the optimal solution if $\mathbf{C}_i^t = \mathbf{C}_i^{t+1}$. When these two vectors are close enough, we stop the iteration.

$$\begin{aligned} \mathbf{C}_i^{t+1} =& \frac{2\mathbf{H}^t \mathbf{S}_i + \lambda \sum_{j \in N(i)} a_{ij} \mathbf{H}_j^t + \rho(\mathbf{H}_i^t - \mathbf{Z}_i^t)}{2\mathbf{H}^t (\mathbf{H}^t)^H + \rho \mathbf{I}} \\ & - \frac{\lambda \sum_{j \in N(i)} a_{ij} \frac{\sum_{l \in V} \mathbf{H}_l^t \exp({\mathbf{H}_l^t}^H \mathbf{C}_i^t)}{\sum_{l \in V} \exp({\mathbf{H}_l^t}^H \mathbf{C}_i^t)}}{2\mathbf{H}^t (\mathbf{H}^t)^H + \rho \mathbf{I}}. \end{aligned} \tag{12}$$

Let (11) equal to zero. We obtain the update equations of $\mathbf{H}_i^{t+1}$. The $\mathbf{H}_i$ (Eq. (13)) follows the $\mathbf{C}_i$ rules,

$$\begin{aligned} \mathbf{H}_i^{t+1} =& \frac{2\mathbf{C}^{t+1} \mathbf{S}_i + \lambda \sum_{(i,j) \in E} a_{ij} \mathbf{H}_j^t + \rho(\mathbf{H}_i^t - \mathbf{Z}_i^t)}{2\mathbf{C}^{t+1} (\mathbf{C}^{t+1})^H + \rho \mathbf{I}} \\ & - \frac{\lambda \sum_{(i,j) \in E} a_{ij} \frac{\sum_{l \in V} \mathbf{C}_l^{t+1} \exp((\mathbf{C}_l^{t+1})^H \mathbf{H}_i^t)}{\sum_{l \in V} \exp((\mathbf{C}_l^{t+1})^H \mathbf{H}_i^t)}}{2\mathbf{C}^{t+1} (\mathbf{C}^{t+1})^H + \rho \mathbf{I}}. \end{aligned} \tag{13}$$

As shown in Algorithm 2, we use the Eq. (12) and Eq. (13) to calculate the final basic block embedding matrix $\mathbf{C}$ after T iterations. Then we feed the $\mathbf{C}$ into the tensor, and perform tensor singular value decomposition (tSVD) to generate the concise function embeddings for all binary functions in the repository. We show the tensor-based function embedding generation in the next section.

Algorithm 2: Basic Block Embedding

```
Input: adjacent matrix of CFG: $\mathbf{A}$, basic block feature
       matrix: $\mathbf{B}$, affinity matrix of basic block feature
       vectors: $\mathbf{S}$, and maximum iteration number $T$.
Output: Basic block embedding matrix: $\mathbf{C}^T$.
Initial basic block embedding matrix $\mathbf{C}^0 \leftarrow$ first $d$
  right singular vectors of $\mathbf{B}$,
$\mathbf{Z}^0 = \mathbf{0}$, $\mathbf{H}^0 = \mathbf{C}^0$,
for $t = 0, 1, 2, ..., T-1$ do
    for $i = 1, 2, ..., n$ do
        Update $\mathbf{C}_i^{t+1}$ using Eq. (12),
    Obtain $\mathbf{C}^{t+1}$, where $\mathbf{C}_i^{t+1}$ is the $i$-th column of
      $\mathbf{C}^{t+1}$,
    for $i = 1, 2, ..., n$ do
        Update $\mathbf{H}_i^{t+1}$ using Eq. (13),
    Obtain $\mathbf{H}^{t+1}$, where $\mathbf{H}_i^{t+1}$ is the $i$-th column of
      $\mathbf{H}^{t+1}$,
    $\mathbf{Z}^{t+1} = \mathbf{Z}^t + (\mathbf{C}^{t+1} - \mathbf{H}^{t+1})$,
```

Time Complexity Analysis Algorithm 2 uses ADMM algorithm to make it converge in a few iterations. The time complexity of initialization is $O(m^2 n)$, the initialization needs to calculate singular vectors of matrix $\mathbf{B}$ of size $m \times n$ using truncated SVD algorithm. Since m is a small defined fixed constant, so the time complexity of initialization is $O(n)$. Before performing the Algorithm 2, the affinity matrix $\mathbf{S}$ also needs to be calculated, which costs $O(n^2)$ for obtaining each element value. At each iteration, the updating time for $\mathbf{C}_i$ should be $O(d^2 n + dn + d|N(i)|)$, since it mainly costs time for matrix multiplication, where $|N(i)|$ is the number of adjacent basic blocks. Since $d << n$, the updating time for $\mathbf{C}_i$ is $O(n)$, and the updating time for $\mathbf{H}_i$ is also $O(n)$. Therefore, the total time complexity of Algorithm 2 is $O(n^2)$.

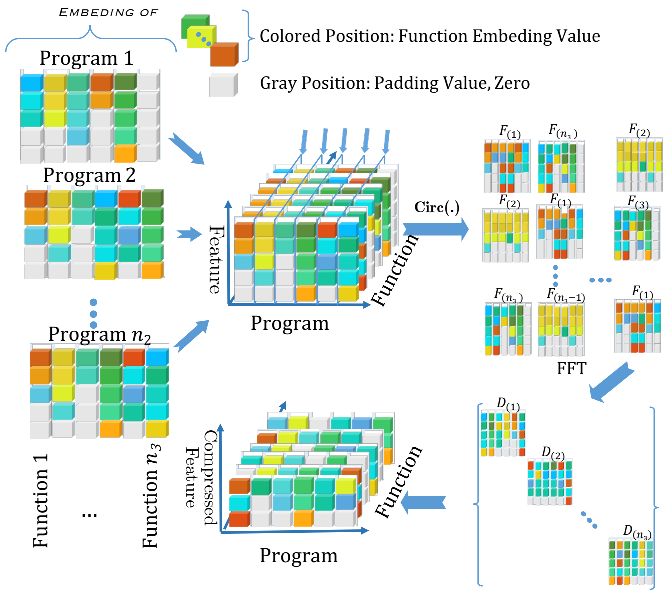


Fig. 4: Function embedding generation process using tSVD algorithm. Left matrices show program representations. They are inserted into the tensor consecutively. Right process shows the tensor embedding.

5 Function Embedding Generation

After obtaining basic block embeddings, we obtain the function feature vector $\mathbf{f}$ based on basic block embedding matrix $\mathbf{C}$. The $\mathbf{f}$ vector can be represented as an aggregation of all basic block embedding vectors in $\mathbf{C}$. $\mathbf{C} \in \mathbb{R}^{d \times n}$ is vectorized as an n_1-dimensional function feature vector $\mathbf{f}$ by column, where $n_1 \geq d \times n$.

Tensor computation is essentially a basic operation in the deep learning and machine learning, and tensor decomposition has emerged as a powerful tool to describe multi-linear relationships between high-dimensional data. When we obtain each function feature vector $\mathbf{f}$, we feed all function feature vectors in a tensor $\mathcal{F}$, and then compress $\mathcal{F}$ to extract the principal features of each function by using the tSVD [28]. As shown in Fig. 4, first, $\widehat{\mathcal{F}}$ is obtained by taking the fft() operation along the third dimension of $\mathcal{F}$. Then, each frontal slice matrix of $\widehat{\mathcal{F}}$ takes truncated SVD: $\widehat{\mathbf{F}}^{(i)} = \widehat{\mathbf{U}}^{(i)}\widehat{\mathbf{M}}^{(i)}(\widehat{\mathbf{V}}^{(i)})^H$. Next, we use the top n_4 columns of $\widehat{\mathbf{U}}^{(i)}$, that is $\overline{\mathcal{U}}(:,:,i) = \widehat{\mathbf{U}}^{(i)}(:,1:n_4)$, and run ifft() operation along the third dimension of $\overline{\mathcal{U}}$ to build the orthogonal tensor $\mathcal{U}$. Finally, we use the $\mathcal{U}$ to compress the tensor $\mathcal{F}$.

5.1 Tensor Representation

We first show how to build a tensor using all function feature vectors generated in the prior steps. The tensor can be represented as $\mathcal{F} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the tensor dimensions n_1, n_2, and n_3 denote the length of a function feature vector, the number of programs, and the number of functions in a program, respectively. A program contains multiple binary functions, then a program feature can be represented as a matrix $\mathbf{F} \in \mathbb{R}^{n_1 \times n_3}$. $\mathbf{F} = [\mathbf{F}_1 \ \mathbf{F}_2 \ ... \ \mathbf{F}_{n_3}]$, a column $\mathbf{F}_i$ of $\mathbf{F}$ is a function feature vector of a program. The order of function feature vectors in $\mathbf{F}$ is based on the order of assembly function sequence number that is extracted by IDA Pro[1].

Different functions have different numbers of basic blocks. We extract n_1 basic blocks. If the number of basic blocks is less than n_1, and we pad 0s at the tail of the function feature vector. Different programs have different numbers of functions. For example, one program has k_1 assembly functions, while another program has k_2 assembly functions, $k_1 \neq k_2$; a program has n_3 assembly functions at most. We use $\mathcal{F}(:,i,:) = [\mathbf{F}_1 \ \mathbf{F}_2 \ ... \ \mathbf{F}_{k_1} \ ... \ \mathbf{0}]$ to denote one program, and $\mathcal{F}(:,j,:) = [\mathbf{F}'_1 \ \mathbf{F}'_2 \ ... \ \mathbf{F}'_{k_2} \ ... \ \mathbf{0}]$ to denote another program.

5.2 Tensor Compression

After building the tensor, we use tSVD to compress the tensor to generate the function embeddings. The tSVD is proposed for high-dimensional data compression, which can grasp the main features and disregard the noise information. If two function feature vectors have mostly similar values, they will be compressed as similar values. We use the tSVD to compress all function feature vectors, which enables us to solve the function misalignment problem and extract the principal function feature. Since tSVD is based on circular convolution operation, we consider that the circular convolution operation can capture the misaligned correlation in function feature vectors. Kilmer and Martin [23] proposed tSVD algorithm to compress the image, and Kuang and Yang [29] showed that 94% of information is concentrated on the first 21% maximum singular values. In the tSVD, the majority of information is concentrated on the top several maximum singular values.

Algorithm 3 shows the function embedding generation process, we build the tensor $\mathcal{F}$, then perform tSVD for $\mathcal{F}$ and obtain the orthogonal tensor $\mathcal{U}$. Next, we obtain the compressed tensor $\mathcal{R}$ by calculating $\mathcal{R} = \mathcal{U}^{\dagger} * \mathcal{F}$, where $\mathcal{U} \in \mathbb{R}^{n_4 \times n_1 \times n_3}$, $\mathcal{R} \in \mathbb{R}^{n_4 \times n_2 \times n_3}$, and $n_4 < \min(n_1, \ n_2)$. The term n_4 is the size of the compressed function embedding vector. Finally, we store every function embedding vector: $\mathcal{R}(:,i,j)$, and its corresponding debug symbol (function name) in the repository, $\mathcal{R}(:,i,j)$ represents the j-th function of the i-th program.

Algorithm 3: Function Embedding Generation

```
Input: Tensor representation $\mathcal{F} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$.
Output: Function embedding tensor $\mathcal{R}$.
$\widehat{\mathcal{F}}$ =fft($\mathcal{F}$, [ ], 3),
for $i = 1, 2, ..., n_3$ do
    $[\widehat{\mathbf{U}}^{(i)}, \widehat{\mathbf{M}}^{(i)}, \widehat{\mathbf{V}}^{(i)}]$ =SVD($\widehat{\mathcal{F}}(:,:,i)$)
    $\overline{\mathcal{U}}(:,:,i) = \widehat{\mathbf{U}}^{(i)}(:,1:n_4)$,
    $\overline{\mathcal{M}}(:,:,i) = \widehat{\mathbf{M}}^{(i)}(1:n_4, 1:n_4)$,
    $\overline{\mathcal{V}}(:,:,i) = \widehat{\mathbf{V}}^{(i)}(:,1:n_4)$,
$\mathcal{U}$ =ifft($\overline{\mathcal{U}}$, [ ], 3), $\mathcal{M}$ =ifft($\overline{\mathcal{M}}$, [ ], 3), and
    $\mathcal{V}$ =ifft($\overline{\mathcal{V}}$, [ ], 3),
$\overline{\mathcal{F}} = \mathcal{U} * \mathcal{M} * \mathcal{V}^{\dagger}$,
$\mathcal{R} = \mathcal{U}^{\dagger} * \mathcal{F}$.
```

[1] https://www.hex-rays.com/products/ida/index.shtml

The circular computation of tSVD shown in Figure 4 solves the basic block misalignment problem in a tensor. That is, the ordering of basic blocks in one function may not match with the ordering of basic blocks in a semantically equivalent or similar function. When we put all the function feature vectors in the tensor, the operator $\mathrm{Circ}(\cdot)$ of the tSVD considers all permutations of the function feature vectors. Most existing works show that the majority of information is concentrated on the top few maximum singular values in tSVD decomposition [23], [29], [30].

5.3 Order Problem

In this section, we show the reason that the tensor-based function embedding generation method can handle the order problem of function feature.

Similarity Computation of Misaligned Functions. Since the function sequences are different in different programs and the different arrangements of vectors form different tensor representations, we need to consider the misalignment problem of function feature vectors in the tensor. The decomposition algorithm needs to solve the function feature misalignment problem. In the tSVD algorithm, $\mathrm{Circ}(\cdot)$ operation and fft algorithm are the circular convolution computation that can capture the misalignment correlation of function feature vectors.

We use a simple example to show how to deal with the order of each function feature vector by tSVD. The initial tensor is represented as follows:

$$\mathcal{F}(:,:,1) = \begin{bmatrix} 1 & 8 & 13 \\ 2 & 6 & 12 \\ 3 & 7 & 11 \end{bmatrix} \quad \mathcal{F}(:,:,2) = \begin{bmatrix} 8 & 11 & 3 \\ 6 & 12 & 2 \\ 7 & 13 & 1 \end{bmatrix}$$

$$\mathcal{F}(:,:,3) = \begin{bmatrix} 11 & 2 & 7 \\ 12 & 3 & 6 \\ 13 & 1 & 8 \end{bmatrix},$$

where $\mathcal{F}(:,:,i)$ represents the i-th function feature vectors of all programs, and $\mathcal{F}(:,j,i)$ represents the i-th function feature vector of the j-th program.

The embedded function feature tensor $\mathcal{R}$ is as follows:

$$\mathcal{R}(:,:,1) = [3.0009\ 7.6717\ -0.9014]$$
$$\mathcal{R}(:,:,2) = [8.2165\ -0.1381\ 1.9324]$$
$$\mathcal{R}(:,:,3) = [0.2867\ 3.3008\ 7.5284],$$

where $\mathcal{R}(:,:,i)$ represents the i-th embedded function feature values of all programs after compressing $\mathcal{F}$, and $\mathcal{R}(:,j,i)$ represents the i-th embedded function feature of j-th program.

If two vectors have the most same values, we consider these two vectors to be similar. We can see that the first function feature vector $\mathcal{F}(:,1,1)$ is similar to $\mathcal{F}(:,3,2)$ and $\mathcal{F}(:,2,3)$, which has most of the same values, although the value sequences of these vectors are different. They are embedded as three most similar embedded values $\mathcal{R}(:,1,1) = 3.0009$, $\mathcal{R}(:,3,2) = 1.9324$, and $\mathcal{R}(:,2,3) = 3.3008$. Besides, $\mathcal{F}(:,2,1)$, $\mathcal{F}(:,1,2)$, and $\mathcal{F}(:,3,3)$ are embedded as three most similar values: $\mathcal{R}(:,2,1) = 7.6717$, $\mathcal{R}(:,1,2) = 8.2165$, and $\mathcal{R}(:\ 3,3) = 7.5284$. In this example, all similar function feature vectors in $\mathcal{F}$ have similar embedded function feature values in $\mathcal{R}$ after compressing the original tensor $\mathcal{F}$. This example demonstrates that the function sequences in different program matrices have less influence in embedded function feature values.

Principal Function Feature Extraction. The tSVD algorithm is proposed for high-dimensional data compression, which can grasp the main features and disregard the noise information. If two function feature vectors have mostly similar values, they will be compressed as a similar value. We use an example to illustrate this issue. The initial tensor is as follows:

$$\mathcal{F}(:,1,:) = [1\ 5\ 11;\ 2\ 6\ 12;\ 3\ 7\ 13;\ 4\ 8\ 14;\ 0\ 0\ 15;\ 0\ 0\ 16]$$
$$\mathcal{F}(:,2,:) = [1\ 5\ 11;\ 3\ 7\ 12;\ 4\ 8\ 13;\ 0\ 6\ 14;\ 0\ 0\ 15;\ 0\ 0\ 16]$$
$$\mathcal{F}(:,3,:) = [1\ 6\ 11;\ 2\ 7\ 12;\ 3\ 8\ 13;\ 0\ 0\ 14;\ 0\ 0\ 15;\ 0\ 0\ 16].$$

After compressing $\mathcal{F}$, a 6-dimensional function feature vector is embedded as an 1-dimensional embedded function feature vector. One function feature vector is $\mathcal{F}(:,1,1) = [1\ 2\ 3\ 4\ 0\ 0]$. Another two function feature vectors are $\mathcal{F}(:,2,1) = [1\ 3\ 4\ 0\ 0\ 0]$ and $\mathcal{F}(:,3,1) = [1\ 2\ 3\ 0\ 0\ 0]$. These similar function feature vectors are embedded as three most similar values, -20.0863, -21.6922, and -20.9557. The tSVD decomposition process thus loses some information but obtains the main features. In our evaluation, if we compress the original function feature values down to 25% (compression ratio p from Eq. (9) is 4), the embedded function feature vector will lose 6% of the information and preserve 94% of the main features.

5.4 Complexity Analysis for Tensor-based Function Embedding Generation

Time Complexity. The running time of tensor singular value decomposition (tSVD) consists of the running time of the tensor unfolding process, the fast Fourier transform (FFT) process, the truncated SVD decomposition process for all block-diagonal matrices, and the compression computation. Let T_{unf}, T_{fft}, T_{svd}, and T_{pro} denote the running time of these four processes, respectively. The total time is the sum of these four times.

Tensor unfolding costs $O(1)$ time. Since $\mathrm{Circ}(\mathcal{F})$ is n_3 blocks of size $n_1 \times n_2$, $T_{\text{fft}} = n_1 n_2 n_3^2 \log n_3 \sqrt{n_1 n_2}$. $T_{\text{svd}} = \sum_{i=1}^{n_3} T_{\text{isvd}}$, where T_{isvd} is the SVD decomposition time consumed by $\mathbf{D}^{(i)}$. We can simultaneously handle the SVD decomposition of all of matrix $\mathbf{D}^{(i)}$, $T_{\text{svd}} = T_{\text{isvd}}$, and $T_{\text{isvd}} = n_1^2 n_2$. $T_{\text{pro}} = n_1 n_2 n_3$ is the compressed time of original tensor by the orthogonal tensor subspace $\mathcal{U}$. The time complexity of the tensor embedding method is $O(1) + O(n_1 n_2 n_3^2 \log n_3 \sqrt{n_1 n_2}) + O(n_1^2 n_2) + O(n_1 n_2 n_3)$, in which n_1 and n_3 can be considered as constants since n_1 is the dimension of the function embedding vector and n_3 is the maximal number of functions in a function in the practical sense. Therefore, the time complexity is $O(n_2 \log \sqrt{n_2})$.

Space Complexity. Let M_{u} denote the space used to store orthogonal tensor bases $\mathcal{U}$, the term M_{fft} and M_{svd} refer to the space usages for the FFT algorithm and the SVD decomposition. $M = M_{\text{u}} + M_{\text{fft}} + M_{\text{svd}}$. The complexity of M is equal to $O(n_1 n_2 n_3)$, $M_{\text{fft}} = O((n_1 + n_2) n_3)$, and $M_{\text{svd}} = O(n_1 n_2 n_3)$. Therefore, the space complexity of Codee is $O(n_1 n_2 n_3)$, in which $n_1,\ n_3$ can be considered as constants, since n_1 is the dimension of function embedding vector and n_3 is the maximal number of functions in a program at the practical sense. Therefore, the memory complexity is $O(n_2)$.

5.5 Dynamic Tensor Compression

We design a dynamic computation algorithm (Algorithm 4) for incrementally generating the function embedding. When updating the database, we need to dynamically calculate newly added function features, rather than compressing an entire new tensor that consists of previous function features and newly increased function features. There is a much greater number of previous function features than that of the newly increased function features. In the dynamic tensor compression, we use a dynamic SVD decomposition algorithm to improve the original SVD decomposition process of every block diagonal matrix in tSVD tensor decomposition. Based on the tSVD decomposition process, we propose the dynamic tensor compression algorithm, which avoids recalculating the entire tensor, and only needs to calculate the newly increased portion.

First, we get the previous SVD tensor decomposition results for each frontal slice in Fourier domain, $\hat{\mathbf{U}}^{(i)}, \hat{\mathbf{M}}^{(i)}$, and $\hat{\mathbf{V}}^{(i)}$. Second, according to newly additive extracted function features, we obtain each frontal slice matrix of additive tensor $\mathbf{F'}^{(i)}, \mathbf{F'}^{(i)} = [\mathbf{F}^{(i)}\ \mathbf{F'}_i]$, the term $\mathbf{F'}_i$ is i-th function feature vector of the additive program and also the n_2+1 column of $\mathbf{F}'$. The new tensor is denoted as $\mathcal{F}'$. In tSVD tensor decomposition [23] [31], the i-th new diagonal block matrix is as follows:

$$\begin{aligned}\mathbf{D'}^{(i)} &= \sum_{j=1}^{n_3}[\mathbf{F}^{(j)}\ \mathbf{F'}_j]\omega_{n_3}^{(j-1)(i-1)} \\ &= \sum_{j=1}^{n_3}[\mathbf{F}^{(j)}\ \mathbf{0}]\omega_{n_3}^{(j-1)(i-1)} + \sum_{j=1}^{n_3}[\mathbf{0}\ \mathbf{F'}_j]\omega_{n_3}^{(j-1)(i-1)} \\ &= [\mathbf{D}^{(i)}\ \sum_{j=1}^{n_3}\mathbf{F'}_j\omega_{n_3}^{(j-1)(i-1)}],\end{aligned}$$

where $\omega_{n_3} = e^{\mathrm{j}\frac{2\pi}{n_3}}$, and $\mathrm{j} = \sqrt{-1}$.

Third, we use an incremental truncated matrix SVD decomposition [32] [33] to obtain each frontal slice matrix of new orthogonal tensor $\mathcal{U}'$ shown in Algorithm 4. Finally, we use the $\mathcal{U}'$ to obtain the new function embedding tensor $\mathcal{R}'$.

```
Algorithm 4: Dynamic tensor compression
Input: $[\mathbf{U}^{(i)},\ \mathbf{M}^{(i)},\ \mathbf{V}^{(i)}] = \mathrm{svd}(\mathbf{D}^{(i)}),\ i \in [1, 2, ..., n_3]$, $\mathbf{f'}_j$ is $j$-th function feature vector of an additive program.
Output: New embedded tensor $\mathcal{R}'$, new orthogonal tensor subspace $\mathcal{U}'$.
$\mathbf{D'}^{(i)} = (\mathbf{D}^{(i)}\ \sum_{j=1}^{n_3}\mathbf{f'}_j\omega_{n_3}^{(j-1)(i-1)})$;
$\mathbf{L} = \mathbf{U}^H\sum_{j=1}^{n_3}\mathbf{f'}_j\omega_{n_3}^{(j-1)(i-1)}$ ;
    $\mathbf{O} = \sum_{j=1}^{n_3}\mathbf{f'}_j\omega_{n_3}^{(j-1)(i-1)} - \mathbf{UL}$ ;
$\mathbf{J}$ is the orthogonal basis of $\mathbf{O}$;
$\mathbf{K} = \mathbf{J}^H\mathbf{O}$ by QR decomposition;
$[\mathbf{U}, \mathbf{J}]\begin{bmatrix} \mathrm{diag}(\mathbf{M}) & \mathbf{L} \\ 0 & \mathbf{K} \end{bmatrix}\begin{bmatrix} \mathbf{V} & 0 \\ 0 & \mathbf{I} \end{bmatrix}^H = \mathbf{D'}^{(i)}$ ;
$\mathbf{Q} = \begin{bmatrix} \mathrm{diag}(\mathbf{M}) & \mathbf{U}^H\sum_{j=1}^{n_3}\mathbf{f'}_j\omega_{n_3}^{(j-1)(i-1)} \\ 0 & \mathbf{K} \end{bmatrix}$;
$[\mathbf{U''}, \mathrm{diag}(\mathbf{M''}), \mathbf{V''}] = \mathrm{svd}(\mathbf{Q})$;
$\mathbf{U'}^{(i)} = [\mathbf{U}^{(i)}, \mathbf{J}]\mathbf{U''}, \mathbf{M'}^{(i)} = \mathbf{M''}$,
    $\mathbf{V'}^{(i)} = \begin{bmatrix} \mathbf{V}^{(i)} & 0; \\ 0 & \mathbf{I}; \end{bmatrix}\mathbf{V''}$;
$\mathcal{U}' = \mathrm{fold}([\mathbf{U'}^{(1)}, \mathbf{U'}^{(2)}, ..., \mathbf{U'}^{(n_3)}])$;
$\mathcal{U}' = \mathrm{ifft}(\mathcal{U}', [], 3), \mathcal{R} = \mathcal{U}'(:, 1 : n_4, :)^{\perp} * \mathcal{F}$.
```

6 EXPERIMENTAL EVALUATION

We now evaluate Codee concerning its effectiveness and efficiency for three different scenarios: cross-architecture, cross-compilers, and cross-optimization-levels. We also consider the accuracy of code search in the mixed dataset of these three scenarios. Furthermore, we conduct a case study to demonstrate the usefulness of Codee in real-world vulnerability analysis. We release the source code of Codee on GitHub[2].

6.1 Implementation

Codee consists of three main components: token-level embedding generation, basic-block-level embedding generation, and function embedding generation. We extract the CFG by using a platform-agnostic binary analysis framework, ANGR. We implement our model in Python, and the tensor computation process in Matlab with tensor toolbox 2.5.

6.2 Experiment Setup

Our experiments are conducted on a server with 32GB memory and 128GB SSD hard drives. In addition, a server with 2 CPU at 3.2GHz and 2 NVIDIA GeForce GTX rtx5000 is used for our token-level embedding generation process, Asm2Vec model, Gemini model, Safe model, and DeepBinDiff model in the comparative experiments.

According to the statistics for our datasets, 98.8% of basic blocks have fewer than 30 instructions, and 97.0% assembly functions have fewer than 200 basic blocks. Compared with other large networks, such as the social network with millions of nodes, most CFGs in our data sets are small graphs with a handful of nodes. The lengths of basic blocks in CFGs are also short, so in the basic block embedding process, we embed a basic block as a value, set $d = 1$. A function feature vector is obtained by vectorizing the basic block embedding matrix by columns; we set $n_1 = 200$. In our data sets, we compile $n_2 = 1,333$ binary libraries, and a binary file has $n_3 = 1000$ assembly functions at most. Choosing the size of final function embedding $n_4 = 20$, expect for discussing n_4 in Figure. 13.

6.2.1 Datasets

We collect binary functions from the following programs as our dataset for evaluation, including OpenSSL[3], Coreutils[4], libgmp[5], and libcurl[6]. We compile the dataset using different architectures (x86-64, ARM, and MIPS), different compilers (GCC5.4.0 and CLANG3.8.0) and different optimization levels (O0-O3). In total, we extract $257,681$ assembly functions using ANGR. Moreover, compiling OpenSSL and bash of different versions (e.g., OpenSSL-1.0.1{a,e,f,g} and bash-{4.2, 4.3}) using two different compilers, GCC 5.4 and CLANG 3.8 with x86-64 and O0. This cross-version dataset contains $14,859$ assembly functions in 12 programs.

Ground Truth. To properly evaluate the experiments, we use the debug symbol information of binary functions to indicate whether two binary functions are empirically matching. Particularly, for each input binary function, we first map the binary function name with the index of this binary function in the repository. We link their assembly functions by using the compiler-output debug symbols and generate an one-to-one mapping between function names and function embeddings. This mapping is written in the feature database, which is used as the ground truth.

To compare our evaluation with state-of-the-art techniques, we use two specific datasets. One is used to compare with Safe and Gemini, which support cross-architectures and cross-optimization-levels code search. Another is used for comparing with Safe, DeepBinDiff, and Asm2Vec, which support cross-versions and cross-optimization-levels code search.

1) **MixedOpenSSL Dataset.** We built the MixedOpenSSL Dataset that consists of a total of $25,837$ functions

[2] https://github.com/ycachy/Codee
[3] https://www.openssl.org/source/
[4] https://ftp.gnu.org/gnu/coreutils/
[5] https://gmplib.org/DOWNLOAD
[6] https://curl.se/download.html

generated from all the binary functions included in Openssl that have been compiled for ARM, MIPS, and x86-64 using GCC-5.4 with 4 optimizations levels (i.e., -O[0-3]). We use ANGR to disassemble this dataset, and discarded all the functions that ANGR is not able to disassemble.

2) **x86-64CrossOptimizations Dataset.** We use four libraries by compiling them for x86-64: OpenSSL, Coreutils, libgmp, libcurl. We use a compiler with GCC-5.4, and 4 optimization levels (i.e., -O[0-3]) to compile these libraries. We use ANGR to disassemble this dataset and obtain $89,795$ functions.

6.2.2 Baseline Comparison Techniques

We select the following baseline methods for comparison:

- **Asm2Vec [10].** This scheme learns the assembly code representation by modeling the CFG as multiple sequences based on the PV-DM model. The embedding size of Asm2Vec is 200, and other training parameters are the same as its original paper. We access its source code in the Github[7], and evaluate it by using our data sets.
- **Gemini [9].** This scheme learns graph embedding based on the Structure2Vec neural network. We access its source code in the Github[8], and evaluate it by handling our data sets. The embedding size of Gemini is 64, and all other training parameters are the same as the original Gemini.
- **Safe** [15] This scheme learns the function embedding using a word2vec model (called i2v model), bi-directional recurrent neural network, and a siamese network. We access its source code in the Github[9], and evaluate it by handling our data sets.
- **DeepBinDiff [17].** This scheme produces the basic block matching based on the word2vec model, and Text-associated DeepWalk algorithm (TADW). We access its source code in the Github[10], and evaluate it by handling our data sets. Since DeepBinDiff was proposed for finding the most similar basic blocks between two binary programs. There are some differences in our code search of the binary function. To show the accuracy in the function-level matching, we define that if two binary functions have at least one similar basic block pair, we think these two binary functions are similar.
- **Order Matters [19].** This scheme generates the function-level embedding based on the BERT model and CNN training. Its source codes are not published in the open web. Since they improved the BERT model that is proposed by google, the improved BERT needs heavily strong computation ability. We can not totally complete their model. We have to use a simple model, a CBOW model to replace the BERT model. Then we evaluate it by handling our data sets.

As the source code of Order Matters is not available, we cannot reproduce the experiments due to the following reasons:

1) Order Matters uses BERT to pre-train the binary code. The BERT model needs a large number of high-quantity datasets and strong computation ability, which is hard to run this model in our laboratory. Besides, The Order Matters adds two tasks (e.g. block inside graph task (BIG) and graph classification task (GC)) in BERT. However, Order Matters does not reveal the detailed technology about how to combine the BERT model and these two tasks.

2) Order Matters also uses the CNN to extract the order information of CFG nodes. This CNN model needs a large number of labeled datasets. However, our method is an unsupervised method, we do not need the labeled dataset, and we do not have a large number of the labeled datasets. Besides, Order Matters does not provide the method how to label the dataset. The normal CNN labels the single data (e.g. a singular image) as a sample, but for binary code search problem, the model needs to label a pair of data (e.g. two CFGs) as a sample. The label denotes whether two CFGs are similar, which is different from the normal labeling method in CNN. We are confused about how to label the samples.

In our experiments, we use the following two steps to perform the Order Matters experiments: 1). We replace the BERT model with the CBOW model, then we pre-process the results of CBOW model to generate the true pair data and false pair data. We label the true-pair data as 1 and the false-pair data as 0. 2). We input all labeled true-pair data and false-pair data into a siamese network. We train the CNN, message passing neural network (MPNN) and multi-layer perception (MLP) in a siamese network to obtain the trained model. Then we use the final function embeddings that are generated by the trained model to perform the code search.

Because experiment results of Order Matters are not ideal in our evaluations, which are different from the original results of their own paper that are pointed. Therefore, we do not totally complete the experiments of this paper. We only test the ROC curves and K-recall curves in MixedOpenSSL Dataset and x86-64CrossOptimizations Dataset, which comparing with our model and other baseline methods.

6.2.3 Performance Metrics

In the following subsections, we will evaluate the performance of Codee in the following metrics.

ROC, F_1 score-precision, and K-Recall. Performance is evaluated by ROC (Receiver Operating Characteristics) curve. The ROC curve is plotted with the recall rate (TPR) against the false positive rate (FPR) where TPR is on the y-axis and FPR is on the x-axis. Recall captures the ratio of assembly functions that are correctly matched. For every query y, suppose there are r matching functions out of a total of X functions in a repository and we select the top K most similar functions as positives, out of which μ functions are correct. In this case, the true positive rate is $Recall = TPR = \frac{\mu}{r}$, the false positive rate is $FPR = \frac{K-\mu}{X-r}$, and the false negative rate is $FNR = 1 - \frac{\mu}{r}$. The precision is calculated by Precision$= \mu/K$, and the F1 score is calculated by F1-score$= 2 * \frac{precision*recall}{precision+recall}$.

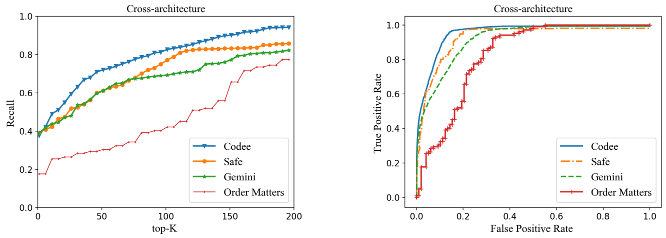
Average recall and mean average precision between two binary programs. Given two binary programs that are compiled from the same source codes but with different optimization levels, different architectures, different versions, or different compilers. We search similar functions in one binary program against another binary program and compute the recall and the precision of the function search, then search similar functions for another binary program against one binary program and compute the recall and the precision of the function search. Then we use the average value of these two recalls, and calculate the mean average precision of these two precision values. Average recall represents the percentage of ground truth function pairs that are confirmed to be correctly matched. The ground truth is collected by examining functions of the two binary programs containing mapped debug symbol information of binary functions. Average precision gives the percentage of correct matching pairs among all the known pairs. In this

[7] https://github.com/McGill-DMaS/Kam1n0-Community

[8] https://github.com/xiaojunxu/dnn-binary-code-similarity/blob/master

[9] https://github.com/gadiluna/SAFE

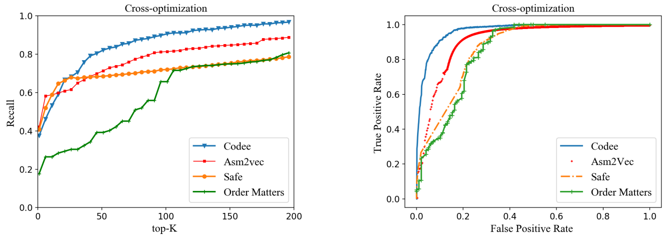
[10] https://github.com/yueduan/DeepBinDiff



(a) Recall across different thresholds top-K.

(b) ROC curves.

Fig. 5: Baseline comparisons for **MixedOpenSSL Dataset** that are compiled in different architectures, compilers and optimization levels.



(a) Recall across different thresholds top-K.

(b) ROC curves.

Fig. 6: Baseline comparisons for **x86-64CrossOptimizations Dataset** that are compiled in different optimization levels from different binary function libraries.

case, if we choose fixed K to compute the correct matching function pairs between two binary programs, and assume that the correct matching function pairs are m, the ground truth function pairs are t and the total function pairs are f, the average recall is m/t, and the average precision is m/f.

Tensor compression metrics. The reconstruction error ratio denoted as e and feature compression ratio denoted as p are used for evaluating tensor compression accuracy and efficiency.

The feature reduction ratio of $\mathcal{F}^{n_1 \times n_2 \times n_3}$ is defined as follows:

$$p = \frac{n_1 n_2}{(n_1 + n_2 + 1)n_4}, \tag{14}$$

where $1 \leqslant n_4 \leqslant \min(n_1, n_2)$. The reconstruction error ratio of tensor $\mathcal{F}$ is defined as

$$e = \frac{||\mathcal{F} - \overline{\mathcal{F}}||_F}{||\mathcal{F}||_F}. \tag{15}$$

Function feature generation time and search time. We use the generation time of the function embedding to evaluate the efficiency of the preparation process. The search time indicates efficiency when searching assembly code on a large-scale dataset.

6.3 Training Model of Token Embedding Generation

We train three NLP-based models using three training corpora. One model is for the instruction set of ARM, one for MIPS, and another for x86-64. With this choice, we try to capture the different lexical and semantic information of these three assembly languages. The model that we use for token embedding generation is the skip-gram implementation of word2vec provided in [20]. Datasets for training the token embedding generation models are collected from a large number of assembly functions. These assembly functions are compiled from all libraries in three architectures using ANGR.

6.4 Code Search against Mixed Binary Functions

We evaluate performances of code search for datasets among different CPU architectures, versions, compilers, and optimization levels. We evaluate whether Codee can precisely search similar assembly functions when the candidate set is from mixed binary functions. We also evaluate its performance with varying retrieval thresholds to inspect whether true positives are ranked at the top. We select all binary functions from the OpenSSL dataset, which are a mixture of functions from OpenSSL-1.0.1 a, e, g, f versions compiled for x86-64, MIPS, and ARM architectures with O0, O1, O2, and O3 optimization levels. There are $53,919$ assembly functions.

Specifically, we collect the test set Y by randomly choosing 3000 functions, and consider the rest functions of the OpenSSL dataset as a search base. For each query y in Y, we search it in the rest to find similar functions. We sort the results and evaluate each of them in sequence. Figure 5(a) plots the recall across different values of K for cross-architecture cross-versions, and cross-optimization-levels OpenSSL dataset. Codee outperforms the other schemes for every K by large margins.

Figure 5(b) shows the ROC curves of the four schemes. ROC curves are computed from queries across 3000 test functions. Figure 5(b) substantiates that Codee can achieve better accuracy than the other state-of-the-art methods. Assembly functions that are compiled from different architectures change more significantly. Both Codee and Gemini can capture the control flow structure in a function. In comparison, Codee performs even better than Gemini and Safe. In Gemini, the CFG of a function is first transformed into an annotated CFG, a graph containing manually selected features, and then embedded into a vector using the graph embedding model (Structure2vec). The manually extracted features used by Gemini may lose some semantic information between basic blocks in a function. Codee replaces manual features with an unsupervised NLP-based pre-trained model for token embedding generation. Codee led to a $2\% - 15\%$ performance improvement than Gemini. Safe uses a skip-gram method-based embedding model i2v and a bi-directional recurrent neural network to train the model. Codee led to a $3\% - 8\%$ performance improvement than Safe. The neural networks of Safe are complex and heavily rely on high-quality dataset. Specifically, for function search in mixed OpenSSL dataset, at $k \in \{5, 15, 25, 35, 45\}$ we have recall values $\{37.72\%, 49.06\%, 54.97\%, 63.12\%, 68.01\%\}$ for Codee, $\{39.2\%, 42.2\%, 47\%, 51.9\%, 52.4\%\}$ for Safe, $\{31.06\%, 35.87\%, 39.09\%, 45.56\%, 46.26\%\}$ for Gemini, and $\{17.64\%, 25.49\%, 26.47\%, 28.43\%, 29.41\%\}$ for Order Matters.

6.5 Code Search across Optimization Levels

In this experiment, we individually evaluate the code search performance against different optimization levels. We use all $1,333$ different binary programs from OpenSSL, libcurl, libgmp and Coreutils libraries that are compiled only by the O0-O3 optimization levels in the x86-64 architecture. In particular, each binary is diffed two times (O0 vs O3, O1 vs O2) and average recall and precision results are reported in Table 2. Table 2 shows the recall ratio and precision at top-$K = 1$ between two binary programs with optimization levels, O0 Vs. O3, and O1 Vs. O2. The results meet our expectations. In general, it is more difficult to search binary code between O0 and O3. Codee achieves the best results, even for a search between O0 and O3: the average recall can still correctly match 82.5% of assembly functions, while Asm2Vec is 74.8%, DeepBinDiff is 70.9%, Gemini is 77.9% and Safe is 81.7%. Binary libraries

TABLE 2: Average recall and average precision between optimization levels

	Optimization O0 vs. O3									
	Recall					Precision				
Baselines	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe
OpenSSL	**0.809**	0.790	0.778	0.774	0.799	**0.789**	0.613	0.531	0.679	0.69
Coreutils	0.712	0.670	0.623	0.780	**0.768**	0.819	0.66	0.618	0.778	**0.869**
libgmp	**0.979**	0.810	0.720	0.889	0.907	**0.828**	0.667	0.730	0.807	0.812
libcurl	**0.798**	0.720	0.714	0.672	0.792	**0.811**	0.778	0.762	0.744	0.792
Average	**0.825**	0.748	0.709	0.779	0.817	**0.854**	0.680	0.660	0.752	0.816
	Optimization O1 vs. O2									
	Recall					Precision				
Baselines	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe
OpenSSL	**0.990**	0.971	0.846	0.924	0.961	**0.864**	0.804	0.798	0.812	0.835
Coreutils	0.912	**0.975**	0.911	0.901	0.924	**0.955**	0.914	0.889	0.905	0.915
libgmp	**0.982**	0.860	0.901	0.916	0.927	**0.937**	0.911	0.896	0.901	0.923
libcurl	0.811	0.765	0.721	0.856	**0.877**	0.821	0.818	0.806	0.872	**0.890**
Average	**0.924**	0.893	0.845	0.899	0.922	**0.894**	0.862	0.848	0.873	0.890

that are compiled by O1 and O2 tend to be more similar than others. The average recall of Codee is 92.40%, which is higher than other schemes. The recall and precision rates in cross-optimization-level binary diffing are lower than that of cross-version diffing, since the compiler optimization techniques can greatly transform the binary codes.

Figure 6(a) plots the recall across different values of K for cross-optimization-levels of four binary libraries. Codee outperforms Asm2Vec, Order Matters and Safe for most K by large margins. Figure 6(b) shows the ROC curves of the four schemes. ROC curves are computed from queries across 3000 test functions. Figure 6(b) substantiates that Codee can achieve better accuracy than the other state-of-the-art methods. Moreover, Figure 7 further presents the Cumulative Distribution Function (CDF) figures of the F1-scores for three diffing techniques on OpenSSL binary libraries in cross-optimization-level diffing settings (o0 vs. o3, o1 vs. o3, o2 vs. o3, and o1 vs. o2). From the CDF figures we can see that Codee, Safe, Gemini, Asm2Vec and DeepBinDiff have somewhat similar F1-scores, while Codee performs much better. Safe uses two neural networks to learn the sequence information of instructions, which only considers the binary function as language, but lose the structural information of binary function. DeepBinDiff, Asm2Vec and Safe heavily rely on NMT-based neural networks, which needs an amount of high-quality train dataset. Codee uses a network embedding-based method and a tensor computation-based method that performs a circular convolution operation. These two parts learn the structural information of code and preserve the semantic information of instructions.

6.6 Code Search across CPU Architectures

Table 3 shows the code search results between different architectures (x86-64, MIPS, and ARM). The results meet our expectation, Codee performs much better than other schemes that support the cross-architecture methods: the average recall of Codee is 85.1%, while Gemini achieves 68.30%, and Safe achieves 81.0% for the comparison between ARM and x86-64. Figure 8 further presents the Cumulative Distribution Function (CDF) figures of the F1-scores for three different techniques on OpenSSL binary libraries in cross-architecture diffing settings (ARM vs. x86-64, MIPS vs. x86-64, and ARM vs. MIPS). Again, from the CDF figures we can see that Codee, performs much better than Safe and Gemini. Since Safe do not support the MIPS architecture, we do the Safe experiment in ARM and x86-64 architectures. Compared with Safe, Codee uses a network embedding-based method and a tensor computation-based method that performs a circular convolution operation. These two parts learn the structural information of code and preserve the semantic information of instructions. Safe uses two neural networks to learn the sequence information of instructions, which only considers the binary function as language, but lose the structural information of binary function.

6.7 Code Search across Compilers

We use all the libraries to evaluate the performance of Codee across compilers. Table 4 lists average recall between different compilers. We compile the functions in each library by x86-64 architecture and the O3 optimization level. In comparison between GCC and CLANG, for each library, we search each function that are compiled by GCC in all functions that are compiled by CLANG, and we search each function that are compiled by CLANG in all functions that are compiled by GCC, and then we calculate the average recall. We can see that Codee performs substantially better than the baseline schemes. On average, Codee achieves 90.1%, Asm2Vec 77.1%, Gemini 74.9%, DeepBinDiff 83.0%, Safe 86.55%. Figure 9(c) further presents the Cumulative Distribution Function (CDF) figures of the F1-scores for Codee, Gemini, Safe, DeepBinDiff and Asm2Vec on OpenSSL binary libraries between GCC and CLANG. We can see that Codee, performs better than other techniques.

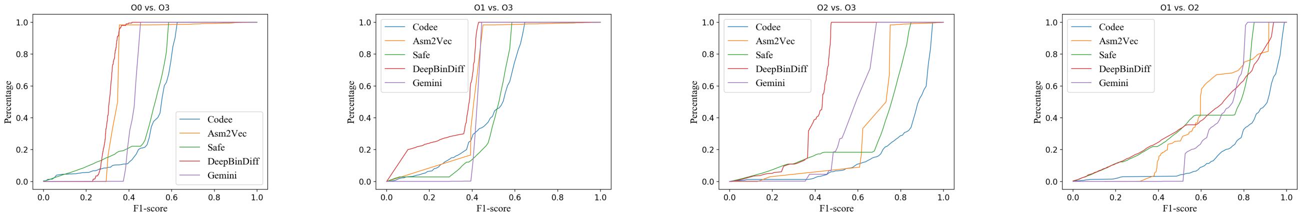
Moreover, we use each function as a query in a known (to be) vulnerable binary programa and try to search other similar functions that contain similar vulnerable binary codes in the target database. This target database differs only in that they are a result of different versions by using different compilation tools. These similar functions are checked suspicious vulnerable functions. The target dataset contains 7 programs. For all functions in each program, the task is to retrieve its variants. The variants are either from different source code versions or generated by different versions of GCC5.4 and CLANG3.8 compilers.

6.8 Code Search across Versions

For each function in a test program (OpenSSL-1.0.1-a-GCC5.4), we search top $K = 5$ similar functions in each target program. We discuss the recall ratio of similar functions between the test program and the target program. Each bar in Figure 10 represents a target program, and the height of the bar represents the recall of similar functions in the target program. The corresponding compiler vendor and source version are noted on the X-axis. Bars represent programs generated by the same source code, they vary in source code versions and compilers (code version OpenSSL-1.0.1e,g,f). "g" is the patched version. Figure 10 shows the search results between different versions and compilers. The different compilers have differing impacts in recall rate. Codee performs better than Gemini and Asm2Vec in most cases.

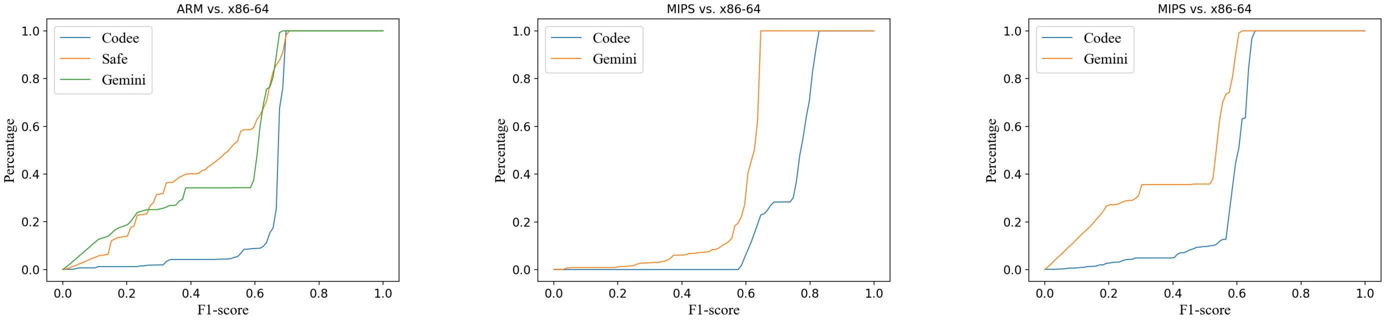
TABLE 3: Average recall and average precision across CPU architectures

Recall	ARM vs. x86-64			ARM vs. MIPS			MIPS vs. x86-64		
Baselines	Codee	Gemini	Safe	Codee	Gemini	Safe	Codee	Gemini	Safe
OpenSSL	**0.843**	0.546	0.788	**0.848**	0.549	-	0.718	**0.728**	-
Coreutils	0.745	0.740	**0.772**	**0.745**	0.720	-	**0.781**	0.746	-
libgmp	**0.979**	0.847	0.894	**0.975**	0.787	-	**0.987**	0.847	-
libcurl	**0.836**	0.600	0.784	**0.682**	0.536	-	**0.710**	0.510	-
Average	**0.851**	0.683	0.810	0.813	0.648	-	**0.799**	0.708	-
Precision	ARM vs. x86-64			ARM vs. MIPS			MIPS vs. x86-64		
Baselines	Codee	Gemini	Safe	Codee	Gemini	Safe	Codee	Gemini	Safe
OpenSSL	**0.937**	0.758	0.885	**0.904**	0.724	-	0.858	**0885**	-
Coreutils	0.802	0.759	**0.814**	**0.808**	0.784	-	**0.811**	0.798	-
libgmp	**0.983**	0.895	0.903	**0.989**	0.829	-	**0.977**	0.832	-
libcurl	**0.845**	0.686	0.798	**0.789**	0.648	-	**0.772**	0.650	-
Average	0.892	0.775	0.850	0.873	0.746	-	0.855	0.791	-



(a) O0 vs. O3 (b) O1 vs. O3 (c) O2 vs. O3 (d) O1 vs. O2

Fig. 7: Cross-optimization-level Diffing F1-score CDF.



(a) ARM vs. x86-64 (b) MIPS vs. x86-64 (c) ARM vs. MIPS

Fig. 8: Cross-architecture Diffing F1-score CDF.

TABLE 4: Average recall and average precision between compilers

	CLANG vs. GCC									
	Recall					Precision				
Baselines	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe	Codee	Asm2Vec	DeepBinDiff	Gemini	Safe
OpenSSL	**0.973**	0.895	0.775	0.714	0.846	**0.981**	0.902	0.818	0.667	0.857
Coreutils	0.803	0.778	0.842	0.755	**0.847**	**0.860**	0.795	0.843	0.612	0.853
libgmp	**0.982**	0.740	0.939	0.704	0.951	**0.989**	0.782	0.942	0.798	0.965
libcurl	**0.847**	0.671	0.762	0.722	0.818	**0.869**	0.741	0.808	0.748	0.832
Average	**0.901**	0.771	0.830	0.749	0.8655	**0.925**	0.805	0.853	0.706	0.877

Moreover, Figure 9(a) and Figure 9(b) further present the Cumulative Distribution Function (CDF) figures of the F1-scores for Codee, Gemini, Safe, DeepBinDiff and Asm2Vec on OpenSSL binary libraries between OpenSSL dataset with different versions. OpenSSL-1.0.1-a is more similar with OpenSSL-1.0.1-e than OpenSSL-1.0.1-g, where “g” is the patched version. The CDF curves of Figures 9(a) and 9(b) are obtained by a set of F1-score values, which is calculated by the corresponding set of Precision-Recall pairs. According to this set of F1-score values, we can plot the CDF curve. We should pay attention to the corresponding abscissa value at the steepest point of CDF curve, which indicates the probability that F1-score value is less than or equal to the corresponding abscissa value. Considering that the larger of this corresponding F1-score value is, the better the model is. That is, if the steep point is at a large abscissa value, it means that the probability of taking a large F1-score value is large, and the model effect is better. From Figure 9(a) and 9(b), we can see that Codee has a larger F1-score value at the steep point of CDF curve than other comparisons methods. Therefore, we can see that Codee, performs better than other techniques.

6.9 Efficiency

We evaluate the efficiency of Codee at a large scale, using $190,090$ real-world assembly functions. Specifically, we test the performance of each phase in Codee. We do not calculate the disassembling time and extraction time of CFG, since disassembling is a necessary step for all schemes. The extraction processes of Codee, Gemini, and Centroid are similar. The extraction time is nearly the same.

Training Time. Training is a one-time effort. We train our token embedding generation model with the binary functions in our dataset. We stop the training for each binary when loss converges or enough epochs (10000 steps). In total, It takes about 24 hours to finish the whole training process for three different instruction sets (e.g. ARM, MIPS, x86-24). Asm2Vec, Safe, and Gemini also need to train their model. Note that the training process could be significantly accelerated if GPUs are used.

Tensor Compression. We evaluate the compression efficiency of the tensor data. The tensor is compressed into a more concise space.

Figure 11 shows the running time of $190,090$ functions against different compression ratios. The running time of the tensor compression increases linearly with an increase in the number of functions. We comprehensively consider the loss ratio and detection time. We set the compression ratio p at 0.25. Before compression, the length of each function feature vector is different. After compression, each function is represented as a vector with a fixed length, $n_4 = 50$. The compression time increases more slowly as the reduction ratio increases.

As shown in Figure 11, we can see different reconstruction error ratios in different compression ratios. The reconstruction error ratio decreases as the feature compression ratio increases. Therefore, we sacrifice only 6.1% of the information in the tensor to save 57.72% of the running time.

Basic Block Embedding Generation Time. We compare basic block embedding generation time with three existing code search schemes: Asm2Vec, Gemini, DeepBinDiff and Safe. Figure 12 shows that function feature generation time increases as CFG size increases. Codee, Gemini, and Safe transform the CFG with basic block raw features of a function to a feature vector. Function feature generation time of Codee includes function feature extraction time and tensor compression time, in which function feature extraction time is 95% of the total time, the tensor compression time is 5% of the total time for a function. Gemini uses a neural network-based scheme to train the model and obtain the function features, which costs time from 25 minutes (5 epochs to achieve reasonable performance) to 4.9 hours (100 epochs to achieve the best performance) by using GPU to train. Asm2Vec learns to construct a semantic feature vector of the assembly code, and it needs time to train sequences like CBOW [34]. The training time of Asm2Vec is nearly 4 hours for 10^5 functions including 2×10^6 basic blocks on average. The embedding generation time of DeepBinDiff takes 1.97 seconds to finish one function embedding generation on average. The embedding generation time of Safe takes 0.51 seconds to finish one function embedding generation on average.

Codee only needs 6 minutes to compress the tensor data with $190,090$ functions. The running time of Codee is mainly spent on the function feature extraction process and the tSVD decomposition. Function feature extraction time costs 95% of total time, while tensor embedding time only costs 5% for each function. Gemini works on GPU, the number of basic blocks has little influence for Gemini. Codee takes less time when the number of basic blocks is less than 1000; 98% of assembly functions have fewer than 200 basic blocks. Thus we can see that Codee runs 5 times faster than Gemini on average, and 10 times faster than Asm2Vec on average.

Search Time. This search time shows the LSH search efficiency when we obtain the all function embeddings in the repository and the target function embedding. We randomly choose functions as codebase that the number of functions in codebase is from $c = 10$ to $c = 1,000,000$. The average search time is close to 4.6×10^{-6} seconds in $90,000$ assembly functions. For the search time of a function, Codee takes 0.043 seconds, Gemini takes 0.1764 seconds for 100 epochs and takes 0.058 seconds for 5 epochs to obtain a reasonable result. Asm2Vec takes 2 seconds. Our method is the most efficient one. DeepBinDiff takes about 42 seconds to finish the function search on average.

6.10 Parameter Selection

We systematically evaluate the accuracy of Codee among different parameter choices. The parameters include λ, the number of extracted instructions for each basic block, and function embedding size (n_4). We use all OpenSSL binary functions to conduct the experiments.

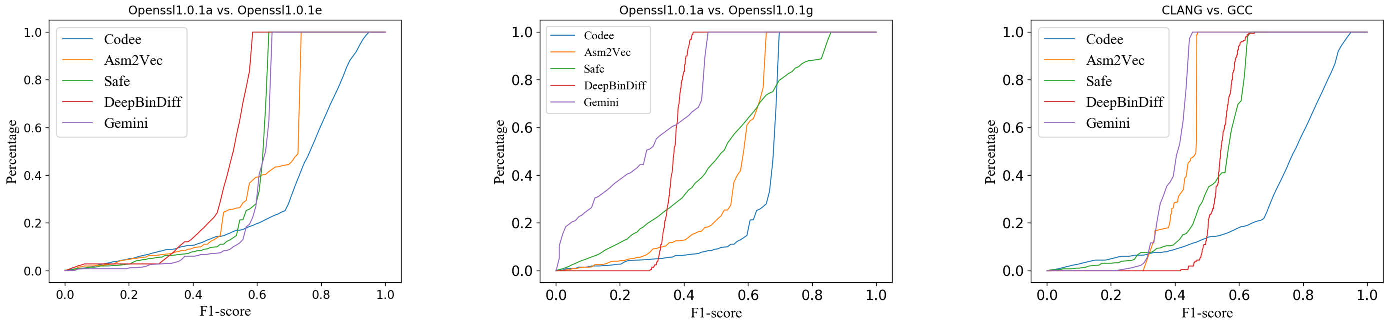
The λ value. To verify the contribution of λ in Eq. (5) of the function feature extraction, we choose different λ values, $\lambda = [0, 0.5, 1, 5, 10]$. We extract 30 instructions for each basic block. As shown in Figure 13(a), the larger λ values represent the higher weight of CFG graph topology structure in the function embedding. For the normal binary file, too large λ or too small λ will have a bad influence on the accuracy. We should balance the influence on both the CFG topology structure and basic block feature. $\lambda = 1$ is a good choice.

Function embedding size. Another important parameter is the final function embedding size. We compress the extracted function feature vector as function embeddings of different sizes. Figure 13(b) shows search results. We can see that the more function embedding values, the better Codee performs, since the size of function embedding vector represents different compressing ratio as shown in Figure 11, the lower compressing ratio leads to the larger function embedding size, the larger function embedding size loses less information of original tensor. If the function embedding size is larger than 50, we can see that the function embedding size seems not to have a significant influence on the accuracy of Codee.

6.11 Case Study

We conduct a case study to evaluate how Codee performs in search of n-day vulnerabilities. In particular, we choose two vulnerabilities: "Heartbleed" (CVE-2014-0160) in OpenSSL and "shellshock" (CVE-2014-6271) in Bash. The two vulnerable functions are compiled using CLANG 3.8. The target dataset is constructed by compiling OpenSSL and bash of different versions (e.g., OpenSSL-1.0.1{a,e,f,g} and bash-{4.2, 4.3}) using two different compilers, GCC 5.4 and CLANG 3.8. In the end, the target dataset contains $14,859$ assembly functions in 12 programs. Each function takes 0.02 seconds on average to extract function features, and takes 38 seconds to do tSVD tensor decomposition for all functions.

When searching the "Heartbleed" vulnerability of OpenSSL-1.0.1.a-CLANG3.8.0, Codee can find two vulnerable functions in other OpenSSL versions generated by CLANG 3.8 at rank 1 and 5, and two vulnerable functions generated by GCC 5.4 at rank 9 and 14. The patched "Heartbleed" functions in version "g" do not appear in the top 20, as the function

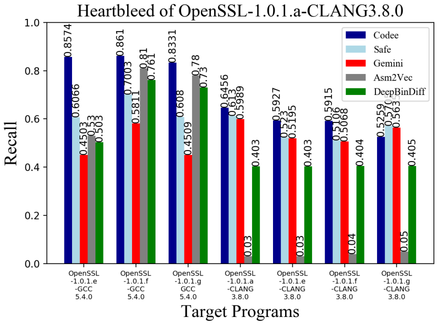


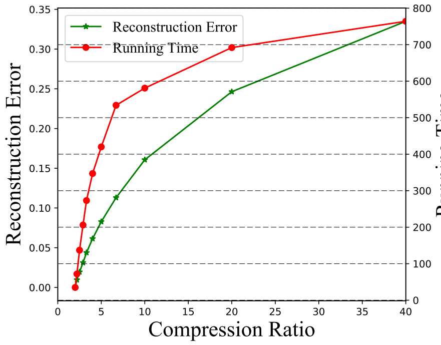
(a) OpenSSL-1.0.1-a vs. OpenSSL-1.0.1-e

(b) OpenSSL-1.0.1-a vs. OpenSSL-1.0.1-g

(c) OpenSSL-CLANG vs. OpenSSL-GCC

Fig. 9: Cross-version Diffing F1-score CDF.





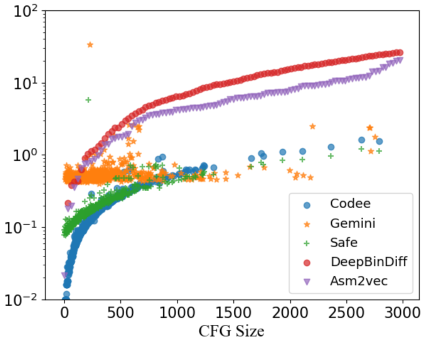
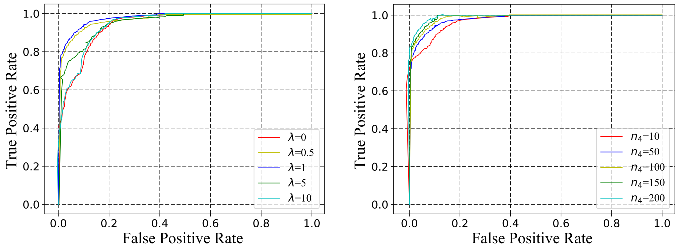


Fig. 10: The recall of similar functions between the OpenSSL-1.0.1.a-GCC5.4.0 and other target programs.

Fig. 11: Running time and compression ratio among different reconstruction errors.

Fig. 12: The function feature generation time (excluding pre-trained time).



(a) ROC among different λs ($n_4 = 50$).

(b) ROC among different sizes of the function embedding ($\lambda = 1$).

Fig. 13: ROC among different parameters

has changed significantly. In comparison, Asm2Vec finds two vulnerable functions generated by CLANG 3.8 at rank 1 and 2, respectively, but cannot find any vulnerable functions generated by GCC 5.4 in at least the top 200. Gemini finds two vulnerable functions generated by CLANG 3.8 at rank 1 and 6, and two vulnerable functions generated by GCC 5.4 at rank 10 and 11.

For the ”Shellshock” vulnerability of bash-4.2-CLANG3.8.0, Codee can find one vulnerable function of another version generated by CLANG 3.8 at rank 1 and two vulnerable functions generated by GCC 5.4 at rank 10 and 18, respectively. In comparison, Asm2Vec finds another vulnerable function generated by CLANG 3.8 at rank 1, but cannot find any vulnerable functions generated by GCC 5.4 in at least the top 200. Gemini also finds another vulnerable function generated by CLANG 3.8 at rank 1, and two vulnerable functions generated by GCC 5.4 do not appear in the top 20.

7 Discussion

7.1 Advantages

Different compiled architectures, different compiler optimization techniques are key problems to perform the code search. Codee is designed to handle optimization techniques and architectures as so to achieve high search accuracy. In experiments, Codee focus on multiple normal architectures and most common compiler optimization techniques. Specifically, we train three token embedding generation models for three different architectures, ARM, MIPS, and x86-64. For function inlining, Codee generates random walks based on ICFG to generate the token embedding, and learn the structural information of CFG to generate basic block embedding. In the tensor-based function embedding, Codee learns the correlation between the functions.

The training process of token embedding generation using an NLP-based model is an one-time effort. We just need to pre-train three NLP-based models to generate token embedding in the different architectures, which improve a considerable speed while learning sufficient semantic information of instructions. The tensor-based function embedding method uses an effective tensor computation to capture the misalignment correlation of function feature vectors based on a circular convolution operation. Our evaluation shows that Codee outperforms other current state-of-the-art approaches on the characteristics of similarity detection accuracy, embedding generation time, and overall search time. Our research is one of the first to demonstrate that the tensor-based data analysis techniques can have unique strengths in the binary code similarity analysis. Codee is both faster and more precise than previous solutions.

7.2 Limitations

However, Codee also has two limitations. If the control flow graph is drastically changed, it influences the accuracy of code search for Codee. This is because our basic block embedding generation relies on CFG to extract graph structural information to differentiate semantically similar functions, a very big change of control flow can significantly affect the results. However, we can fine-tune the parameter λ to change the influence weight of structural information of CFG. If $\lambda = 0$, we do not consider any structural information of CFG. Normally, we choose the $\lambda = 1$, such that the structural information of CFG and semantic information have the same influence in the generated basic block embedding. Moreover, we make an empirical evaluation for discussing the influence of parameter λ, which is shown in Figure 7(a). There is a drawback that we need to know in advance whether the binary function has a big change for its CFG. Consequently, Codee is vulnerable to obfuscation techniques that completely alter the CFG. Packing techniques [35] that encrypt the code can also defeat our system. Nearly no existing learning-based techniques perform better because they also rely on graph control flow information.

Moreover, we add the comparison experiments of obfuscation data to show our limitation for Codee, Asm2Vec, and Gemini. Results show that Codee does not perform better in obfuscated code search. If the CFG topology structure (the adjacency matrix) of a function is changed a lot, the accuracy of our method will be decreased. We show the results in the appendix.

Second, for the revised version, we use an NLP-based learning technique for the token embedding generation. in normal NMT, a word embedding model is usually trained once using large corpora, such as Wiki, and then is reused by other researchers. However, we have to train an instruction embedding model from scratch. Second, if a trained model is used to convert a word that has never appeared during training, the word is called an out-of-vocabulary (OOV) word and the embedding generation for such words will fail. This is a well-known problem in NLP, and it exacerbates significantly in some cases, as constants, address offsets, labels, and strings are frequently used in instructions. We leave this as future work.

8 Related Work

We summarize recent code search methods in the following parts.

Raw features-based code search. Many researchers made a great contribution in binary code search based on various raw features. They directly extracted static features from binary for code similarity matching. David and Yahav proposed trace-based code search approach by using code execution sequences as features [36]. Andrews et al. used static birthmarks to search code based on robust characteristics, such as constant values, call sequences and known classes [37]. However, they cannot tolerate the opcode change by different compilations and are not scalable. Ming et al. proposed a whole-program plagiarism detection method based program equivalence checking approach [38]. However, this approach does not accurately find code clones across different architectures. Pewny et al. used input and output pairs to grasp function semantics at the basic block level [39]. It is very costly to extract features and perform graph matching. DiscovRe used the pre-filtering to handle CFG [40], but Genius showed that pre-filtering is not robust. Hu et al. proposed a semantics-based hybrid approach to detect binary clone functions [5]. Luo [41] proposed an obfuscation-resilient semantically equivalent binary code detection method based on the longest common sub-sequence of basic blocks. They used a set of symbolic formulas to show the input-output relations and the semantics of a basic block based on fuzzy matching. Shirani et al. used simple statistic features of instructions [42]. Instruction-based features fail to consider the relationships between instructions. N-grams or N-perms used the binary function sequence or token code matching [3], [2], they worked poorly on different compilers since the code semantics is not be fully preserved in these types of sequences.

Graph-based code search. Graph techniques have their significance in code search. Bourquin et al. used the expensive graph isomorphism algorithm to measure the similarity between CFGs [43]. Feng et al. proposed a scalable bug search scheme for the cross-platform bug search problem [8], but its training and feature encoding also rely on graph matching. Fu et al. proposed a CFG-based malware detection approach, which extracted “code chunks” features from CFG, then used the classification and regression tree algorithm to classify features [44]. Chandramohan et al. proposed a binary search engine, they use user-defined inlining functions and a relevant library to obtain function semantics [45]. Flake et al. proposed a method to compare CFGs to cope with compiler optimizations [46]. Xue [47] used user-defined functions and an inlining relevant library to capture the fully function semantics based on a selective inlining technique from high-level semantic features and structural features. These solutions rely on expensive graph matching, and are not scalable for searching in large codebases.

Deep learning-based embedding schemes. Recently, there is a line of works that use deep learning to generate embeddings for basic blocks and functions. Gemini employs a graph embedding network, structure2vec, to embed a function [9]. SAFE is a self-attentive neural network model for function embedding [15]. It improves efficiency by avoiding extractions of control flow graphs. InnerEye employs a neural machine translation model for computing basic-block-level embeddings [11]. Massarelli [16] shows a similar method as Gemini [], but replace manual feature extraction with unsupervised-based neural network feature extraction. Yu [19] uses BERT to pretrain the binary code on one token-level task, one block-level task, and two graph-level tasks. Moreover, they adopt convolution neural network (CNN) on adjacency matrices to extract the order information of CFG nodes. However, this method relies on several complex neural networks, which need strong computation ability to train an amount of hyperparameters. Liu et al. employs a convolutional neural network directly on the raw binary code to compute an embedding for a function [48]. While these deep-learning-based solutions are appealing, it is crucial to collect a substantial amount of high-quality training data. In contrast, our tensor-based embedding scheme does not require training.

9 Conclusion

In this paper, we present a tensor embedding-based scheme, called Codee. Codee consists of an NLP-based token embedding generation method, a network representation-based basic block embedding generation method and an effective tensor-based function embedding generation method. The NLP-based token embedding generation method captures the semantic information and lexical information of basic blocks. On this basis, the basic block embedding generation method learns the structural information of CFG while preserving the semantic information of basic blocks. The tensor-based function embedding method captures the misalignment correlation of function feature vectors based on a circular convolution operation. Our evaluation shows that Codee outperforms other current state-of-the-art approaches on the characteristics of similarity detection accuracy, embedding generation time, and overall search time. Our research is one of the first to demonstrate that the tensor based data analysis techniques can have unique strengths in the binary code similarity analysis.

ACKNOWLEDGMENTS

Cai Fu is supported by China NSF (62072200), and Pan Zhou is supported by China NSF (61972448).

REFERENCES

[1] I. U. Haq and J. Caballero, "A survey of binary code similarity," *CoRR*, vol. abs/1909.11424, 2019.

[2] M. E. Karim, A. Walenstein, and A. Lakhotia, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[3] 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.

[4] Y. David and E. Yahav, "Tracelet-based code search in executables," *SIGPLAN Not.*, vol. 49, no. 6, pp. 349–360, Jun. 2014.

[5] Y. Hu, Y. Zhang, and J. Li, "Binmatch: A semantics-based hybrid approach on binary code clone analysis," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, 2018, pp. 104–114.

[6] L. Li, D. Li, T. F. Bissyand, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, June 2017.

[7] P. D. Nicolao, M. Pogliani, and M. Polino, "ELISA: eliciting ISA of raw binaries for fine-grained code and data separation," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, 2018, pp. 351–371.

[8] Q. Feng, R. Zhou, and C. Xu, "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, pp. 480–491.

[9] X. Xu, C. Liu, and Q. Feng, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 363–376.

[10] S. H. Ding, B. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019.

[11] F. Zuo, X. Li, and P. Young, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[12] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[13] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 3034–3040.

[14] Z. L. Chua, S. Shen, and P. Saxena, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 99–116.

[15] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 309–329.

[16] L. Massarelli, G. A. Luna, and F. Petroni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Workshop on Binary Analysis Research (BAR) 2019 24 February 2019, San Diego, CA, USA*, 2019, pp. 1–891 562–58–4.

[17] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[18] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, 2014, pp. 1188–1196.

[19] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 2020, pp. 1145–1152.

[20] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, 2016, pp. 855–864.

[21] X. Huang, J. Li, and X. Hu, "Accelerated attributed network embedding," in *Proceedings of the 2017 SIAM International Conference on Data Mining, Houston, Texas, USA, April 27-29, 2017.*, 2017, pp. 633–641.

[22] J. Tang, M. Qu, and M. Wang, "LINE: large-scale information network embedding," in *Proceedings of the 24th WWW*, 2015, pp. 1067–1077.

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

[24] D. Kuang, H. Park, and C. H. Q. Ding, "Symmetric nonnegative matrix factorization for graph clustering," in *Proceedings of the Twelfth SIAM International Conference on Data Mining, Anaheim, California, USA, April 26-28, 2012*, 2012, pp. 106–117.

[25] G. Golub and C. V. Loan, *Matrix Computation*, 1996.

[26] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, 2018.

[27] S. Boyd, N. Parikh, and E. Chu, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2011.

[28] Z. Zhang, G. Ely, S. Aeron, N. Hao, and M. E. Kilmer, "Novel methods for multilinear data completion and de-noising based on tensor-svd," in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, 2014, pp. 3842–3849.

[29] L. Kuang, L. T. Yang, J. Feng, and M. Dong, "Secure tensor decomposition using fully homomorphic encryption scheme," *IEEE Trans. Cloud Comput.*, vol. 6, no. 3, pp. 868–878, 2018. [Online]. Available: https://doi.org/10.1109/TCC.2015.2511769

[30] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: Models, applications, and scalable algorithms," *ACM TIST*, vol. 8, no. 2, pp. 16:1–16:44, 2017.

[31] B. J. Olson, S. W. Shaw, and C. Shi, "Circulant Matrices and Their Application to Vibration Analysis," *Applied Mechanics Reviews*, vol. 66, no. 4, 2014.

[32] M. Brand, "Incremental singular value decomposition of uncertain data with missing values," in *Computer Vision - ECCV 2002, 7th European Conference on Computer Vision, Copenhagen, Denmark, May 28-31, 2002, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Heyden, G. Sparr, M. Nielsen, and P. Johansen, Eds., vol. 2350. Springer, 2002, pp. 707–720.

[33] L. Kuang and F. Hao, "A tensor-based approach for big data representation and dimensionality reduction," *Emerging Topics Comput.*, vol. 2, no. 3, pp. 280–291, 2014.

[34]

[35] T. Fernique and N. Bdaride, "Density of binary disc packings: The 9 compact packings," 2020.

[36] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 349–360.

[37] 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.

[38] J. Ming, F. Zhang, and D. Wu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Trans. Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.

[39] J. Pewny, F. Schuster, and L. Bernhard, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, 2014, pp. 406–415.

[40] S. Eschweiler, "discovre: Efficient cross-architecture identification of bugs in binary code," in *23rd Annual Network and Distributed System Security Symposium*, 2016.

[41] L. Luo, J. Ming, and D. Wu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Software Eng.*, vol. 43, no. 12, pp. 1157–1177, 2017.

[42] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, Bonn, Germany, July 6-7*, 2017, pp. 301–324.

[43] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, 2013, pp. 4:1–4:10.

[44] D. Fu and Y. Xu, "WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 7 809 047:1–8, 2017.

[45] M. Chandramohan, Y. Xue, and Z. Xu, "Bingo: cross-architecture cross-os binary search," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 678–689.

[46] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings*, 2004, pp. 161–173.

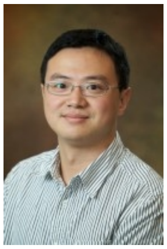
[47] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1125–1149, 2019.

[48] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018.

Jia Yang She received the Master degree in Computer Science and Technology from Huazhong University of Science and Technology, China, in 2017. She is currently a phd degree candidate at Huazhong Science and Technology University, Wuhan, Hubei, China. Her research interests focus on malicious code and data privacy.

Cai Fu Corresponding author, IEEE member, Ph.D, professor in School of Cyber Science and Engineering, Huazhong University of Science and Technology, Hubei Engineering Research Center on Big Data Security, Wuhan, His main research interests include wireless networking security, routing algorithms and distributed system security.

Xiao-Yang Liu IEEE member, he is currently a PhD in the Department of Electrical Engineering, Columbia University. He received the B.Eng. degree in computer science from Huazhong University of Science and Technology, in 2010; PhD. degree in computer science from Shanghai Jiao Tong University, in 2016; and MS. degree in electrical engineering from Columbia University, in 2017. His research interests include tensor theory, non-convex optimization, deep learning, big data analysis and data privacy.



Heng Yin IEEE senior member, professor in the department of Computer Science and Engineering at UC Riverside. He received his Ph.D in Computer Science from the College of William and Mary in 2009. His research interests lie in computer security, program analysis, virtualization, and machine learning/deep learning to solve computer and software security problems, including but not limited to malware detection and analysis, vulnerability discovery, program hardening, and digital forensics.

Pan Zhou IEEE senior member, (S'07M'14-SM'20) is currently a full professor and PhD advisor with Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, P.R. China. He received his Ph.D. in the School of Electrical and Computer Engineering at the Georgia Institute of Technology (Georgia Tech) in 2011, Atlanta, USA. He received his B.S. degree in the Advanced Class of HUST, and a M.S. degree in the Department of Electronics and Information Engineering from HUST, Wuhan, China, in 2006 and 2008, respectively. He held honorary degree in his bachelor and merit research award of HUST in his master study. He was a senior technical member at Oracle Inc., America, during 2011 to 2013, and worked on Hadoop and distributed storage system for big data analytics at Oracle Cloud Platform. He received the Rising Star in Science and Technology of HUST in 2017. He is currently an associate editor of IEEETransactions on Network Science and Engineering. His current research interest includes: security and privacy, big data analytics, machine learning, and information networks.