Cross-architecture Binary Function Similarity Detection based on Composite Feature Model

Xiaonan Li[1,2], Guimin Zhang[1*], Qingbao Li[1], Ping Zhang[1], Zhifeng Chen[1], Jinjin Liu[2] and Shudan Yue[1]

[1] Information Engineering University
Zhengzhou, Henan 450001 China
[e-mail: zh.guimin@163.com]
[2] School of Computer Science, Zhongyuan University of Technology
Zhengzhou, Henan 450007 China
[e-mail: lxn@zut.edu.cn]
[*]Corresponding author: Guimin Zhang

Received April 12, 2023; accepted July 25, 2023; published August 31, 2023

Abstract

Recent studies have shown that the neural network-based binary code similarity detection technology performs well in vulnerability mining, plagiarism detection, and malicious code analysis. However, existing cross-architecture methods still suffer from insufficient feature characterization and low discrimination accuracy. To address these issues, this paper proposes a cross-architecture binary function **s**imilarity **d**etection method based on **c**omposite **f**eature **m**odel (SDCFM). Firstly, the binary function is converted into vector representation according to the proposed composite feature model, which is composed of instruction statistical features, control flow graph structural features, and application program interface calling behavioral features. Then, the composite features are embedded by the proposed hierarchical embedding network based on a graph neural network. In which, the block-level features and the function-level features are processed separately and finally fused into the embedding. In addition, to make the trained model more accurate and stable, our method utilizes the embeddings of predecessor nodes to modify the node embedding in the iterative updating process of the graph neural network. To assess the effectiveness of composite feature model, we contrast SDCFM with the state of art method on benchmark datasets. The experimental results show that SDCFM has good performance both on the area under the curve in the binary function similarity detection task and the vulnerable candidate function ranking in vulnerability search task.

Keywords: Binary Similarity, Composite Feature Model, Cross-Architecture, Graph Embedding Network, Vulnerability Detection.

This research was supported by the National Key Research and Development Program of China under Grant 2021YFB3101804.

http://doi.org/10.3837/tiis.2023.08.008 **ISSN : 1976-7277**

1. Introduction

With the rapid expansion of embedded devices and the widespread application of Internet of Things (IoT), security concerns about firmware vulnerabilities are rising. Reusing code, which results in the rapid spread of the same or similar vulnerabilities in firmware built on different architectures, is one of the most important reasons for the high incidence of firmware attacks. Due to the inability to accurately obtain the relationship between firmware suppliers, subcontractors and developers, it is difficult to trace a vulnerability in firmware across different architectures. Therefore, studying the method to detect similar vulnerabilities accurately in existing firmware for different architectures is crucial for device security [1, 2].

Existing code similarity detection methods can be divided into source code based [3, 4] and binary code based [5, 6]. Since most firmware source codes are often unavailable in practice, researchers prefer to investigate cross-architecture similarity detection of binaries. In recent, many researchers have proposed to convert binary functions into embeddings (i.e., numeric vectors) and then use the distance between the two embeddings to measure the similarity between a pair of functions [7-11]. Among various function similarity detection approaches, graph embedding based methods have outstanding performance both in accuracy and speed [12]. Especially, Genius [2] and Gemini [7] are the most representative and state-of-the-art works in these studies, making the embedding method a research hotspot in code similarity detection. However, in terms of feature characterization and discrimination accuracy of the embedding model, existing methods still need to be further improved.

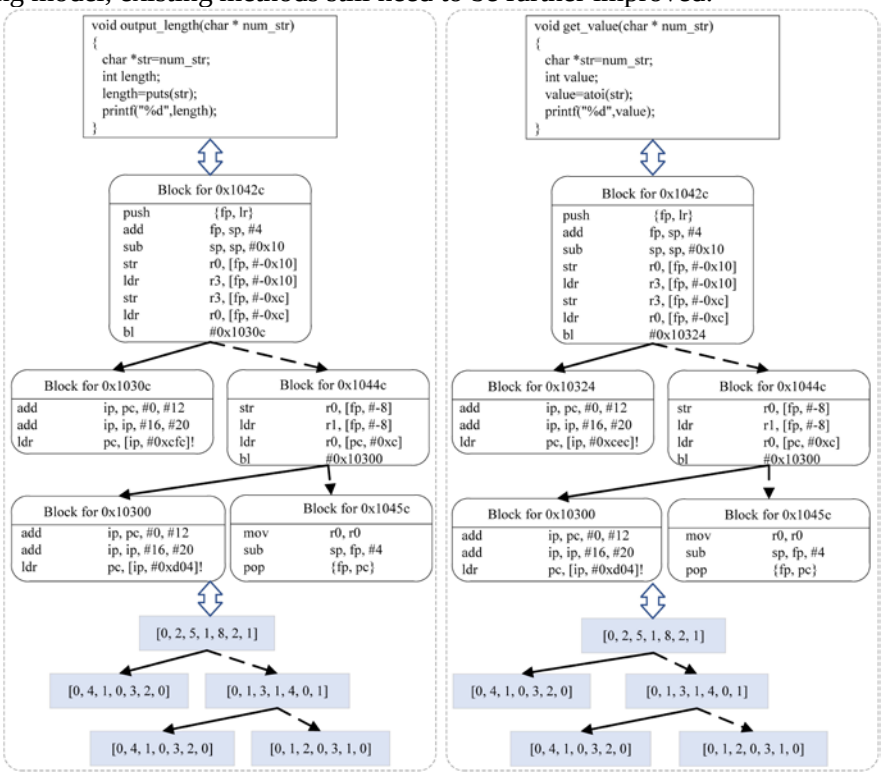

Fig. 1. Code example. The upper part shows the source code. The middle part shows the assembly code and control flow graph corresponding to the source code above. The lower part shows the feature vectors extracted from the assembly code according to the feature construction method of Gemini.

For example, according to the feature description method of Gemini, the two functions "output_length" and "get_value", as shown in **Fig. 1**, are described by identical embeddings, which means that Gemini will consider the two functions to be similar. However, there is a big difference between the two functions. The function "output_length" on the left calls the library function "puts" to write a string to the standard output. In contrast, the function "get_value" on the right calls the library function "atoi" to convert the numeric string into an integer. Gemini cannot distinguish such functions because it only extracts the number of constants, different types of instructions, and successor nodes to describe function features (shown at the bottom of **Fig. 1**). No behavioral features are considered in Gemini. Other methods based on graph embedding also suffer from the above problems and cannot precisely reflect the difference in the behavior of different functions [8, 11]. Additionally, some researchers adopt natural language processing methods to detect binary similarity [9, 10]. However, they only consider the opcodes of the assembly instructions when extracting words and ignore the difference of the operands, which means that they also cannot distinguish differences in behavior due to different operands (such as the two functions "output_length" and "get_value" as shown in **Fig. 1**).

In response to these issues, this paper proposes a cross-architecture binary function **s**imilarity **d**etection method based on **c**omposite **f**eature **m**odel, abbreviated as SDCFM. SDCFM adopts composite feature model (CFM) to describe the binary function, which is composed of instruction statistics in a basic block (i.e., statistical features), control flow graph (CFG) (i.e., structural features), and application program interface (API) call information (i.e., behavioral features). Based on a comprehensive analysis of the attack surface and cross-architecture applicability, CFM currently only focuses on Glibc API call information. Then, SDCFM proposes a hierarchical embedding method to generate embeddings for cross-architecture binary functions. To the best of our knowledge, SDCFM is the first attempt to analyze binary code similarity by combining API call features with statistical features of basic block instructions and structural features exhibited by CFG. The contribution of this paper involves the following aspects:

(1) We propose a CFM to characterize features of the binary function. The CFM utilizes the behavioral features of the API function call and combines them with statistical features of basic block instructions and CFG structural features to describe the cross-architecture features of functions more precisely.

(2) We propose a hierarchical embedding method to take full advantage of the features extracted based on the CFM. In this way, the statistical features and behavioral features can be embedded independently, and the three types of features in CFM can be fused into more valuable feature vectors.

(3) During the update process of embeddings in the graph embedding network, we analyze the directionality of GNN and choose the embeddings of predecessor nodes to modify that of the current node. Empirical study proves that this manner can improve the accuracy of similarity discrimination and enhance the stability of the model.

(4) We perform multiple cross-architecture similarity detection tests based on open-source programs. For the testing dataset consisting of binary functions with Glibc API calls, SDCFM gets a higher area under the curve (AUC) value (=0.983) than that of the baseline method Gemini (=0.958). Furthermore, SDCFM places real vulnerable functions at the first place for 51 times among the 120 constructed cross-architecture vulnerability search tasks, which outperforms Gemini (=15) by a large margin.

The remainder of this paper is organized as follows. Section 2 presents some existing studies for code similarity detection. In Section 3, the overall framework, the raw feature

representation, and the hierarchical embedding method are presented. Section 4 presents our experimental results and analysis. Section 5 discusses the limitation and future work. Section 6 concludes this paper.

2. Related Work

There are two critical challenges in the cross-architecture similarity analysis of binary code: finding the raw features that can represent the semantics of code and converting the raw features into vector representations that are easy to compare. Researchers have conducted in-depth analysis.

2.1 Raw feature of binary code for similarity detection

Chua et al. [13] propose to represent instructions using word embedding, which learns embeddings directly from a large set of assembly instructions. Ding et al. [14] decompose CFG into instruction sequences, each representing a potential execution trace, and then generate sequence embeddings based on instruction embeddings. Zuo et al. [9] optimize this type of feature by abstracting the operands of assembly instructions and utilizing the longest common subsequence in units of basic blocks to characterize codes. Yu et al. [10] introduce an adjacency matrix to reserve the order information of the basic blocks in CFG. However, all the above methods require large-scale datasets to train instruction embedding models. Some methods may face the issue of out-of-vocabulary when dealing with assembly codes.

To obtain lightweight features, Eschweiler et al. [15] propose syntax-level features (e.g., the number of logic instructions and function calls) and simplify function-level features before performing graph matching. But this pre-filtering leads to a decrease in search accuracy. To improve the matching accuracy, Feng et al. [2] add two structural features (the number of offspring and betweenness) to build an ACFG to model functions. Due to the high time consumption of extracting betweenness, Xu et al. [7] eliminate this attribute when constructing ACFG. Ji et al. [11] and Gao et al. [8] also use feature description approaches similar to [2] and [7].

Aiming at the problem that the existing methods lack the ability to describe function behavior features, this paper adopts the CFM to represent the binary function features. On the basis of making full use of the ACFG, the API call features are considered to improve the ability of precise function characterization further. The API function call features itself have cross-architecture robustness, which has been widely proved in program behavior analysis research [16-19].

2.2 Embedding methods based on neural networks

In recent years, neural network-based embedding methods have gradually become the mainstream of binary similarity detection. Among them, the graph embedding approaches and NLP-based methods have been well applied in this field.

The graph embedding approaches usually target graph-structured objects (e.g., control-flow graphs and function call graphs) and propagate node attributes according to the edges in the graph. Feng et al. [2] are the first to introduce graph embeddings to the similarity detection of binary functions. However, when computing similarity, it is still necessary to perform bipartite graph matching, which results in high computational complexity. Xu et al. [7] construct a Siamese network with two identical graph neural networks (GNN) and obtain a graph embedding model through end-to-end training. Moreover, the similarity is obtained by calculating the cosine distance between embeddings, which reduces the time consumption.

Gao et al. [8] take the effects of data flow into account during the iterations of GNN, and Ji et al. [11] construct a Triplet-Loss network with GNN to enlarge the distance of embeddings of different codes. However, they both lead to further increases in training costs. Li et al. [20] perform graph representation calculations based on paired graphs and increase the cross-graph communication to make the matching model more sensitive to the difference in graph pairs. But it is expensive for large graphs due to the graph matching. Such methods do not perform differential learning on basic block-level and function-level features, resulting in the loss of feature information.

The NLP-based methods typically treat instructions as words and sequences of instructions as sentences, then leverage NLP techniques to generate embeddings for binary code. Zuo et al. [9] utilize the skip-gram model to generate embeddings for assembly language and adopt the LSTM and Siamese network to build a cross-language basic block embedding model. Ding et al. [14] utilize the PV-DM model to generate embeddings of instruction sequences. But it is designed for one assembly language type and cannot be directly applied to cross-architecture semantic clone detection. Yu et al. [10] adopt BERT, MPNN, and Resnet to capture semantic features, structural information, and the node order information, respectively, and fuse these three parts to generate the final embedding at last. Such methods usually use different models or techniques at various levels, such as assembly instructions, instruction sequences, and basic block sequences.

In response to the feature loss in the current graph embedding methods, this paper adopts a hierarchical embedding method. It propagates and aggregates basic block features according to the CFG topology by GNN, then concatenates the block-level features with function-level features and fuses them into the final function embedding at last.

3. Methodology

3.1 Overview

The purpose of SDCFM is to determine whether the semantics of the functions in the binary are similar to those of the functions in the target function database. Its overall framework is shown in **Fig. 2**.

SDCFM mainly comprises two key modules: the raw feature extractor (①) and the function embedding generator (②). The raw feature extractor extracts the statistical features, structural features, and behavioral features of a binary function. The three types of features are combined into the composite feature. According to the CFM, each binary function is characterized by an attributed control flow graph (ACFG, composed of statistical features and structural features) and a set of behavioral features (i.e., API feature) (Section 3.2). The function embedding generator encodes the composite features of a function into an embedding vector in a high-dimensional space (Section 3.3). Specifically, the statistical features of basic blocks are firstly propagated and aggregated along the CFG topology by using a graph neural network (GNN), which can generate partial embedding. Then, the partial embedding and the behavioral features are fused to generate the final embedding of the binary function by the fully-connected layer.

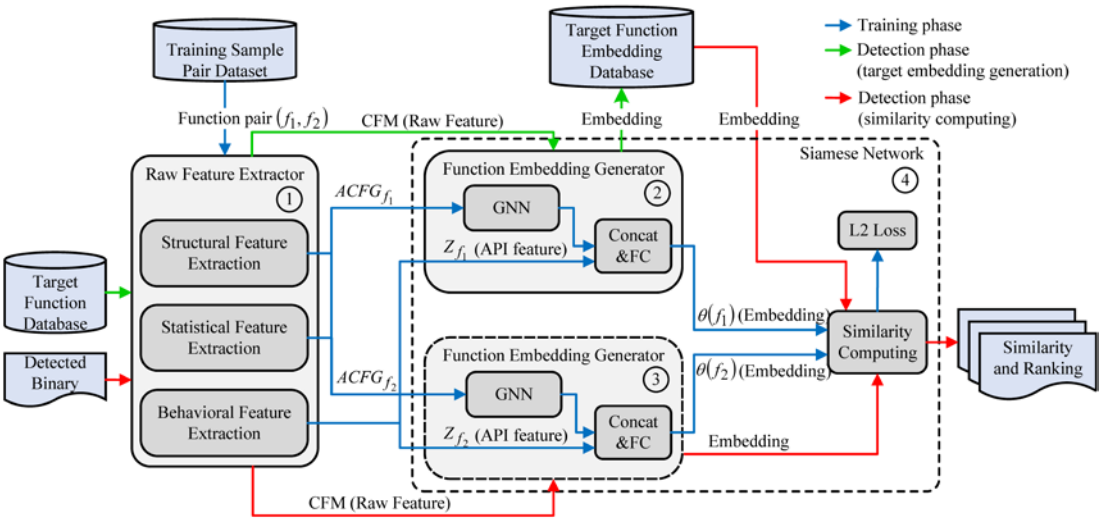

Fig. 2. Framework of SDFCM.

The workflow of SDCFM includes two phases: the training phase and the detection phase.

In the training phase, SDCFM utilizes large-scale sample pairs with ground truth to train a neural network model, i.e., the function embedding generator, which should generate similar embedding for the functions with similar semantics. Our approach uses the default policy that binary functions compiled from the same source code are homologous and analogous regardless of architecture or compiler optimization level. Therefore, such function pairs are used to construct positive samples, with the label 1. Meanwhile, the negative samples are constructed from function pairs in the same binary but with different names, with the label -1. Subsequently, the composite features of the two functions in each sample pair are extracted by the raw feature extractor (①) and then fed to a Siamese network [21] (④), which is composed of two identical function embedding generators (②and③). The Siamese network updates and adjusts related parameters through the back-propagation algorithm until the L2 loss in the training dataset reaches a relatively small value. Finally, a fairly optimal function embedding generator (②) is obtained.

In the detection phase, SDCFM detects the similarity between the binary to be detected and the functions in the target function database. We first construct the target function database by collecting binary code of functions that we care about (such as vulnerable functions). Then, all binary functions in the target function database are processed by the raw feature extractor (①) and the function embedding generator (②) to obtain the target embeddings, which will be stored in the target function embedding database. When detecting whether a binary (i.e., detected binary in **Fig. 2**) contains functions in the target function database, we utilize (①) and (②) to extract composite features of each function in the binary and generate their embeddings. Then, the similarity scores between the embeddings of functions in the binary and that in the target function embedding database are calculated, and the similarity ranking is obtained finally. When applied to vulnerability search tasks, SDCFM needs to estimate whether the binary contains functions similar to a known vulnerable function. In this case, the target function is the vulnerable function, and the output is the similarity and ranking of all functions in the binary compared with the vulnerable function.

3.2 Raw Feature Representation

In response to insufficient feature description in existing methods, this paper proposes the CFM to describe the raw features of binary functions. The CFM consists of statistical features, structural features, and behavioral features. The statistical features describe ten categories of statistics for a basic block, which are block-level features. The structural features, i.e., the CFG, present the location of each basic block in the CFG. The behavioral features, i.e., the API call information, represent function semantics. The structural and behavioral features all belong to function-level features. The specific raw features information is shown in **Table 1**.

Table 1. The raw features information

Level	Type	Feature name
Block-level features	Statistical features	No. of String Constants
		No. of Numeric Constants
		No. of Transfer Instructions
		No. of Instructions
		No. of Logical Instructions
		No. of Arithmetic Instructions
		No. of Bit Operation Instructions
		No. of Branch Instructions
		No. of Subroutine Calls
		No. of Offspring
Function-level features	Structural features	CFG
	Behavioral features	Glibc API

The API call has been proven an effective feature for describing the behavior of functions, both in source code semantic analysis [16, 17] and in dynamical binary similarity analysis [18, 19]. Although there are various APIs, the CFM currently only concerns the Glibc API and extracts the Glibc API call information as the behavioral features for the following three reasons. Firstly, a large number of known vulnerabilities (such as buffer overflow, format string, and information disclosure vulnerabilities) are related to Glibc API function calls (such as "strcpy" and "memcpy"). Secondly, programs can control and operate many system resources (such as the network, standard input, and memory) by exploiting the Glibc API, which means that Glibc API functions are closely related to the behavioral semantics of one function. Last but not least, Glibc API is one of the most basic and widely used dynamic link libraries for lots of applications in Linux-like operating systems. In general, taking the Glibc API function call information as the behavioral features for one function has strong expression ability and broad applicability.

The raw feature extractor is implemented based on angr [22]. Since each function call instruction is regarded as the end of the current basic block in angr, there will be at most one Glibc API call in a basic block. However, the called Glibc API function in a single basic block can only reflect the behavior of this block. To reflect the behavioral semantics of the entire function, we need to utilize the Glibc API call information of all basic blocks in the function. Meanwhile, due to the numerous Glibc API functions, simply using the combined information of the Glibc API calls in a function as function-level features will lead to high-dimensional sparse features, which is not conducive to model convergence. Therefore, Glibc API functions are categorized based on their features. By comprehensively considering safety and functional characterization, CFM only concerns some typical and essential Glibc API function types, as shown in **Table 2**.

Table 2. Glibc API function types

TypeID	Glibc Function Type	Example
1	Copy	memcpy, memmove, strncpy
2	Compare	memcmp, strncasecmp, strncmp
3	Memory Read	strlen, getenv, times
4	Memory Write	memset, setvbuf, write
5	Memory Management	free, malloc, calloc
6	Data Type Transform	strchr, atoi, strrchr
7	File Management	rename, fileno, stat
8	File Write	fwrite, fputs, fprintf
9	File Read	fgets, fread, fread_chk
10	Output	perror, puts, printf
11	Computing	sin, cos, pow
12	Program Management	exit, assert_fail, stderr
13	System Management	shutdown, ioctl, access
14	Network Contact Management	setsockopt, connect, htons
15	Network Data Capture	recvfrom, recvmsg, recv
16	Network Data Send	sendto, sctp_senmsg, rdma_post_send
17	Execution Control	exec, fexecve, dlsym
18	Pipe Communication	pipe, pclose, popen

This classification can bring two benefits. Firstly, it facilitates the discovery of potential vulnerabilities. Since the Glibc API functions of the same category represent similar behavioral semantics, binaries which call Glibc API functions of the same category are semantically similar. Secondly, it improves the scalability of the detection model. When the behavioral features are extended by taking new API functions into account, they can be appended to a specific category according to its behavioral semantic without retraining the deep learning model.

Given a binary function f, for each basic block $b \in B$, where B is the basic block set of f, its function call attribute is denoted as α. If there is a Glibc API call in b, the value of α will be set to the Glibc API function ID (each concerned Glibc API function is assigned a unique ID number); otherwise, α is given as 0. Then, the numeric vector z_f is used as the function-level behavioral features of f. Formally, z_f is a discrete numeric vector and $z_f = \{k_1, k_2, ...k_d\}$, where k_j represents the total times that functions of the j-th type (i.e., typeID = j) are called in f. Currently, the CFM only focuses on 18 types of Glibc API functions, so d equals 18.

Besides, the block-level features (i.e., the top 10 attributes in **Table 1**) which are encoded as a numerical vector and the structural features (i.e., CFG of the binary function) work together to form an ACFG. It should be noted that the description of block-level features of ACFG in SDCFM differs from that of Genius [2] or Gemini [7].

Therefore, according to the design of the CFM, each function is characterized by an ACFG and a set of function-level behavioral features.

3.3 Hierarchical Embedding Method

As aforementioned, each binary function f is characterized by an ACFG and a set of function-level behavioral features. The ACFG corresponding to f is denoted as $g_f = \langle V, E \rangle$,

where V and E are the node set and the edge set respectively. Each node $v \in V$ represents a basic block in the ACFG, and the attribute of v is denoted as a c-dimensional feature vector x_v. The function-level behavioral feature corresponding to f is denoted as z_f. The objective of the function embedding generator is to generate embedding $\theta(f)$ of f for subsequent similarity computation. The network architecture of the function embedding generator is shown in **Fig. 3**.

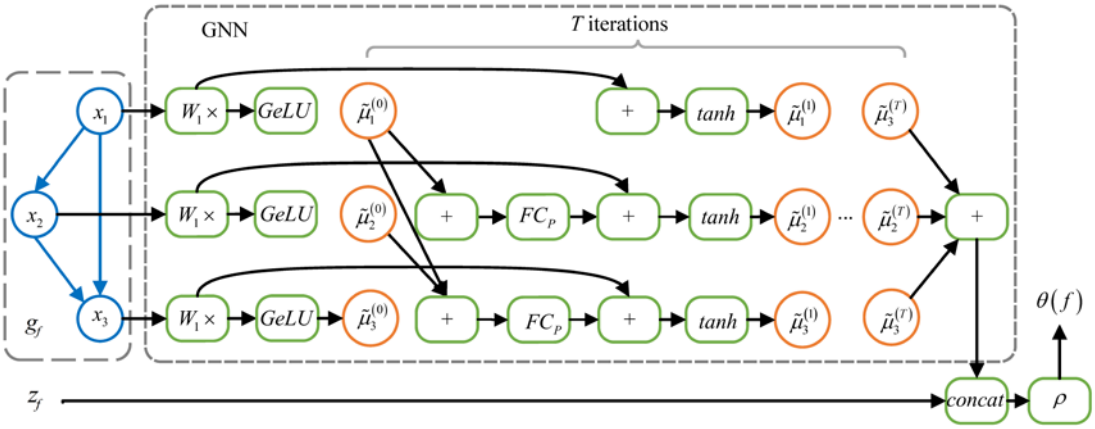

Fig. 3. Function embedding generation network.

In converting composite features into embedding, SDCFM adopts a hierarchical embedding method. Specifically, based on CFG topology, SDCFM utilizes GNN to embed the ACFG and gets a partial embedding. Then, SDCFM splices the partial embedding and the behavioral features to generate the final embedding by fully-connected layers.

The reason for hierarchical embedding is that if the Glibc API call information participates in the propagation and aggregation of GNN directly with the ACFG, the role of behavioral features in characterizing functions will be greatly limited. By applying the proposed hierarchical embedding method, SDCFM maximizes the role of behavioral features in binary function similarity detection.

3.3.1 Embedding Generation Network

The GNN used in SDCFM is adapted from Structure2vec [23], and the instantiation method is inspired by Gemini. **Fig. 3** visualizes the improved network architecture. Structure2vec is an embedding network for structured data. It facilitates using stochastic gradient descent to learn parameters and can handle large-scale datasets. Therefore, many subsequent kinds of research refer to Structure2vec [8, 24, 25].

Structure2vec calculates a m-dimensional embedding $\tilde{\mu}_v$ for each node $v \in V$ in the graph g_f and then aggregates the embeddings of all nodes as the embedding vector of the graph g_f. Its embedding algorithm of mean-field initializes the embedding $\tilde{\mu}_v^{(0)}$ at each node as 0, and updates the embeddings at each iteration as:

$$\tilde{\mu}_v^{(t)} = \tilde{T}\left(x_v, \left\{\tilde{\mu}_u^{(t-1)}\right\}_{u\in N(v)}, \left\{x_u\right\}_{u\in N(v)}\right) \tag{1}$$

Where $\tilde{T}$ is an arbitrary nonlinear function mapping, $N(v)$ represents the set of neighbors of node v in the graph g_f, and $\tilde{\mu}_v^{(t)}$ denotes the embedding of node v in the t-th iteration. The update formula (1) indicates that the update process of the embedding is based on the topology of the graph. In addition to the node itself, the embeddings of adjacent nodes in the previous round and their attributes are used to update the node embedding in the current round.

Based on our observations, the execution of one node (i.e., one basic block) in CFG is only affected by the execution results of its predecessor nodes and not by its successor nodes. Therefore, we improve the original Structure2vec network by replacing adjacent nodes $N(v)$ with predecessor nodes $P(v)$ in ACFG to update the node embedding. The embedding update process is parameterized as follows:

$$\tilde{\mu}_v^{(t)} = \tanh\left(W_1 x_v + P_h \times GeLU\left(P_{h-1} \times ...GeLU\left(P_1 \sum\nolimits_{u \in P(v)} \tilde{\mu}_u^{(t-1)}\right)\right)\right) \tag{2}$$

Where W_1 is a $m \times c$ matrix, P_i is the i-th coefficient of the fully connected layer, and h is the number of fully-connected layers (also known as the embedding depth). *GeLU* is the Gaussian error linear unit activation function [26]. In particular, the initial value of the node embedding $\tilde{\mu}_v^{(0)} = GeLU(W_1 x_v)$.

Afterward, the embeddings of all nodes obtained after T iterations are aggregated by addition. In the end, the aggregation of node embeddings is spliced with function-level features and fused into the final function embedding:

$$\theta(f) = \rho\left(\sum\nolimits_{v \in V} \tilde{\mu}_v^{(T)} \| z_f\right) \tag{3}$$

Where z_f represents the function-level features of the function f, and ρ is a fully-connected network. The input of ρ is the connection of the m-dimensional aggregated embedding and the d-dimensional function-level features. The output is the final e-dimensional function embedding. The number of layers and neurons in hidden layers can be adjusted according to the training results.

3.3.2 Similarity Calculation

Given two binary functions f_1 and f_2, the embedding $\theta(f_1)$ and $\theta(f_2)$ can be obtained by the improved function embedding generation network shown in **Fig. 3**. In SDCFM, we compare the embeddings of the two functions using the cosine similarity [11], which is effective in binary code similarity detection. It is described as: $\cos(\theta(f_1), \theta(f_2)) = \dfrac{\theta(f_1) \cdot \theta(f_2)}{\|\theta(f_1)\| \cdot \|\theta(f_2)\|}$.

In the training phase, the positive sample pair has the ground-truth label $y = 1$, and the negative sample pair has the label $y = -1$. Then, the samples with ground truth are used to perform end-to-end training on the Siamese network composed of two function embedding generation networks. The stochastic gradient descent algorithm minimizes the mean square error $\frac{1}{n}\sum_{i=1}^{n}(y_i - \cos_i(\theta(f_1), \theta(f_2)))^2$, n is the total number of sample pairs and i stands for the i-th sample pair. This way, all parameters of the function embedding generation network can be learned.

4. Experiment and Evaluation

In this section, the effectiveness, stability, and cross-architecture vulnerability detection ability of SDCFM are evaluated quantitatively to answer the following four Research Questions (RQs):

RQ1: Is it reasonable and effective to adopt the categorical behavioral features and the hierarchical embedding method?

RQ2: Is it possible to improve the stability of the embedding generation model by exploiting the embeddings of predecessors to modify that of the current node in the iterative process of the graph neural network?

RQ3: How is the binary code similarity detection accuracy of SDCFM compared with the baseline method Gemini [7]?

RQ4: How is the cross-architecture vulnerability detection ability of SDCFM compared with Gemini?

4.1 Experimental settings

We leverage angr [22], a platform-agnostic binary analysis framework of Python 3 libraries, and use its API to implement the raw feature extractor. It extracts all the raw features that conform to the CFM of binary functions, including CFG, Glibc API call information, and statistics for various types of instructions in the basic blocks, according to **Table 1**.

Besides, a hierarchical embedding network is implemented in Python based on TensorFlow [27]. According to the CFM, the dimensions of the statistical and behavioral features are 10 and 18, respectively. To ensure the best performance of baseline, we utilize the optimal parameter settings of Gemini, in which the embedding size is 64, the embedding depth is 2, and the number of iterations is 5.

All experiments are conducted on a server equipped with two Intel Xeon Gold 5218 CPUs @ 2.3 GHz (64 cores in total), 256 GB memory, 1 TB usable HDD, and one NVIDIA Tesla P100 GPU. All training and testing are performed with GPU acceleration.

4.2 Dataset

In the experiments, three datasets are constructed for evaluation.

Dataset I: This dataset provides large-scale data with ground truth for training neural networks and evaluating the accuracy of models. As the ground truth data is limited in most tasks, we select some open-source programs and compile them into binaries for different architectures and optimization levels. The binary functions compiled from the same source code are considered similar, otherwise dissimilar. Specifically, we compile OpenSSL (version 1.0.1f and 1.0.1u) using GCCv7.5. The compiler is set to generate binaries for ARM, MIPS, and PowerPC architectures, with optimization levels O0-O3. In this way, we get 143046 binary functions corresponding to 6324 source functions. According to the function names, Dataset I is divided into three disjoint subsets at a ratio of 8:1:1 for training, validation, and testing, respectively. Functions with the same name will be assigned to the same subset. The details of Dataset I are presented in **Table 3**.

Table 3. Details of binary functions in Dataset I and Dataset II

	Total	Training set	Validation set	Testing set
Dataset I	143046	113317	15378	14351
Dataset II	21783	17573	2092	2094

For each binary function f in the training set, a function f_1 with the same function name is randomly selected from the same subset to form a positive sample pair $(f, f_1, 1)$. Meanwhile, a function f_2 with a different name is selected to construct a negative sample pair $(f, f_2, -1)$. The sample generation methods on the validation set and testing set are the same as that on the training set. Since the testing set, training set, and validation set are disjoint, the performance of the model on unseen functions can be verified.

Dataset II: To evaluate the effectiveness of the CFM and hierarchical embedding method for cross-architecture binary similarity detection, we extract a subset from Dataset I. It contains 21,873 binary functions corresponding to 1,291 source functions containing Glibc API calls. Dataset II is split into three disjoint subsets in the same proportion as Dataset I, as shown in **Table 3**. The labeled samples on Dataset II also adopt the same construction method as Dataset I.

Dataset III: This dataset is used to evaluate the vulnerability detection ability of the SDCFM. Five published vulnerabilities in three open-source programs are selected from the Common Vulnerabilities and Exposures (CVE) [28]. The details are shown in **Table 4**.

Table 4. Details of vulnerable functions and affected programs

Vulnerability function	CVE No.	Affected program	Affected version
OBJ_obj2txt	CVE-2014-3508	OpenSSL	1.0.1f
MDC2_Update	CVE-2016-6303		
ssl3_get_new_session_ticket	CVE-2015-1791		
url_parse	CVE-2017-6508	Wget	1.19.1
parse_datetime	CVE-2014-9471	Coreutils	8.13

The compiler GCCv7.5 is used to generate binary code for the three affected open-source programs in **Table 4**. The objective binary architectures include ARM, MIPS, and PowerPC, with four optimization levels O0-O3, respectively. Therefore, 36 different binaries are constructed. The statistics of functions in binaries in Dataset III are presented in **Table 5**.

Table 5. Details of functions in binaries in Dataset III

Architecture	OpenSSL 1.0.1f				Wget 1.19.1				Coreutils 8.13			
	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
ARM	1037	952	904	933	1037	826	802	708	1474	1401	1367	1197
MIPS	949	877	762	807	949	825	801	695	1420	1404	1369	1187
PowerPC	988	919	880	912	988	825	801	694	1417	1401	1367	1183

4.3 Evaluation Metrics

This paper adopts the receiver operating characteristic (ROC) curve and AUC to measure the performance of binary code similarity detection models. The ROC curve is a powerful tool for studying the generalization performance of a learner. The closer the ROC curve is to the coordinate (0, 1), the better the performance of the learner. When the ROC curves of two learners cross or are relatively close, we use AUC to compare their accuracy.

Then, we utilize Range and Standard Deviation to evaluate the stability of embedding models. They are common metrics for measuring variations. The range reflects the difference between the maximum and minimum values, and Standard Deviation demonstrates the degree of dispersion of the data. For the detection results of the similarity detection model on different testing sets, the smaller the variability of AUC values, the more stable the model detection effect is.

In addition, to conveniently find the most similar function to the vulnerable function, functions should be ranked based on descending order of their similarity scores to the vulnerable function. The ranking of detected vulnerable functions is an intuitive measure to evaluate the performance of similarity-based vulnerability detection methods. Besides, if the real vulnerable function is included in top-k candidates, we call this a top-k hit.

4.4 Experimental Results & Implications

4.4.1 Experiments for Answering RQ1

In order to evaluate the reasonableness and effectiveness of adopting the categorical behavioral features and the hierarchical embedding method, four types of different features are constructed for experiments.

- Without_API: The statistical features and structural features are used, without behavioral features (i.e., Glibc API call features), to characterize the binary function.
- API_Id_In_Block: Based on the first type, the IDs of the Glibc API function called in each basic block are added as block-level features.
- API_Type_In_Block: Based on the first type, the categorical behavioral features of Glibc API calls in each basic block are added as block-level features.
- API_Type_In_Function: Based on the first type, the categorical behavioral features of Glibc API calls in a function are added as function-level features, which are processed according to the hierarchical embedding method.

We extract the above four types of features of binary functions in Dataset II. Then, we feed them to the function embedding generation network respectively and train the networks for 100 epochs. Hence, we obtain four models last. The AUC and loss of these models on the same validation set are illustrated in **Fig. 4**.

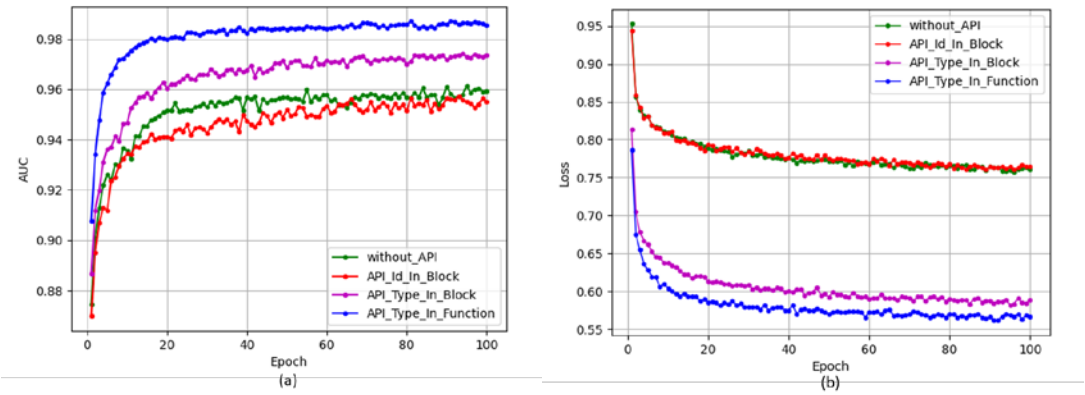

Fig. 4. The AUC and loss of the different models on the validation set.
(a) AUC vs. the number of epochs. (b) Loss vs. the number of epochs.

According to **Fig. 4(a)**, in the same epoch, the AUC values of API_Type_In_Block and API_Type_In_Function are both higher than those of Without_API and API_Id_In_Block. It demonstrates the effectiveness of the categorical behavior features in detecting the similarity of binary function across architectures. Meanwhile, the loss values of API_Type_In_Block and API_Type_In_Function are much smaller than those of Without_API and API_Id_In_Block, as shown in **Fig. 4(b)**. It indicates that the hierarchical embedding method can generate much more similar embeddings for binary functions corresponding to the same source codes.

Furthermore, the AUC value of API_Type_In_Function is always higher than API_Type_In_Block. Moreover, the curve becomes stable in the 40th epoch, which converges faster than that of API_Type_In_Block. This result shows that it is more effective to utilize categorical behavior features of Glibc API call as function-level features. Meanwhile, it confirms that too few Glibc API calls in a single basic block result in sparse classification features, which is not conducive to model convergence.

Besides, the AUC value of API_Id_In_Block is close to that of Without_API after 60 epochs. This result reflects that simply using the Glibc API function ID as the block-level feature will degenerate behavioral features into statistical features, which makes the Glibc API feature fail to play a discriminative role in similarity detection. Due to the numerous Glibc API functions, the large gap in function ID affects the convergence speed. Additionally, the IDs of functions used less frequently may become noise and affect the accuracy of the model.

The above experimental results demonstrate the reasonableness and effectiveness of the categorical behavioral features and the hierarchical embedding method.

4.4.2 Experiments for Answering RQ2

In the process of generating a particular node embedding using GNN, the node embeddings of its adjacent nodes will be used to update its embedding. In order to evaluate the impact of exploiting different nodes on the accuracy and stability of the model, this paper considers three network structures.

- Neighbors: All the embeddings of adjacent nodes are utilized to update the current node, including predecessors and successors.
- Successors: Only the embeddings of the successor nodes are used.
- Predecessors: Only the embeddings of the predecessor nodes are used.

Due to the randomness of sample generation, we perform ten tests with different samples on Dataset II to ensure the unbiasedness of the evaluation results. In each trial, the composite features extracted from the same sample set are fed to three neural networks. We finally obtain three different similarity detection models and evaluate the accuracy of these models on the testing set. The results of the ten tests are presented in **Table 6**.

Table 6. The result of the ten tests

	1(%)	**2(%)**	**3(%)**	**4(%)**	**5(%)**	**6(%)**	**7(%)**	**8(%)**	**9(%)**	**10(%)**	**Mean (%)**	**Range**	**Standard Deviation**
Neighbors	97.17	97.16	98.05	98.23	98.52	98.08	97.73	98.51	97.69	98.27	97.94	1.36	0.47
Successors	97.54	97.38	97.50	98.18	98.35	97.72	97.27	98.58	97.79	98.45	97.88	1.31	0.45
Predecessors	97.86	97.54	98.17	98.25	98.67	98.08	97.93	98.67	98.17	98.74	98.21	1.2	0.37

As shown in **Table 6**, the model Predecessors (row 3) consistently achieves the highest AUC compared with the model Neighbors (row 1) and Successors (row 2). The average AUC value of the Predecessors is 0.27% higher than that of the Neighbors and 0.33% higher than that of Successors. The results demonstrate the better performance of the model that uses the embeddings of predecessor nodes in the previous round to update the current node.

The Range (the difference between the maximum and minimum values) and the Standard Deviation (the square root of means of the squared deviations from the arithmetic mean) of Predecessors are also the smallest among the three models. It indicates that only using the embeddings of predecessors to update the current node makes the model more stable.

4.4.3 Experiments for Answering RQ3

To further evaluate the accuracy of SDCFM, comparison experiments with the baseline are conducted on Dataset I and Dataset II.

Baseline method. There are two reasons for choosing Gemini as the baseline. Firstly, the construction of ACFG in the CFM refers to the design method of Gemini. The expansion of statistical features and the addition of function-level behavior features can be regarded as an improvement to Gemini's original ACFG. Therefore, it is more pertinent to compare SDCFM with Gemini. Secondly, although Genius [2] extracts the original features of binary functions in the form of ACFG for the first time, Gemini follows Genius' ACFG. But Gemini eliminates the attributes with high computational cost and proves that its performance far exceeds Genius. Several later GNN-based binary code similarity detection researches [8, 11, 29] have improved Gemini. However, they only partially change the embedding process or loss function, and do not change the method to characterize functions. Therefore, the comparison with Gemini is more convincing.

In order to compare the performance more fairly and effectively, we adopt the same raw feature extractor based on angr to construct ACFG according to the respective definition of each method. Meanwhile, we utilize the optimal parameter settings of Gemini to ensure its best performance.

Results. **Fig. 5** shows the ROC curves of SDCFM and Gemini on the test set of Dataset I and Dataset II. The ROC curves of SDCFM in **Fig. 5(a)** and **Fig. 5(b)** are both closer to the point (0,1). The AUC values of SDCFM on Dataset I and II are 0.964 and 0.983, while the AUC values of Gemini are 0.952 and 0.958, respectively. Therefore, SDCFM outperforms Gemini in terms of accuracy.

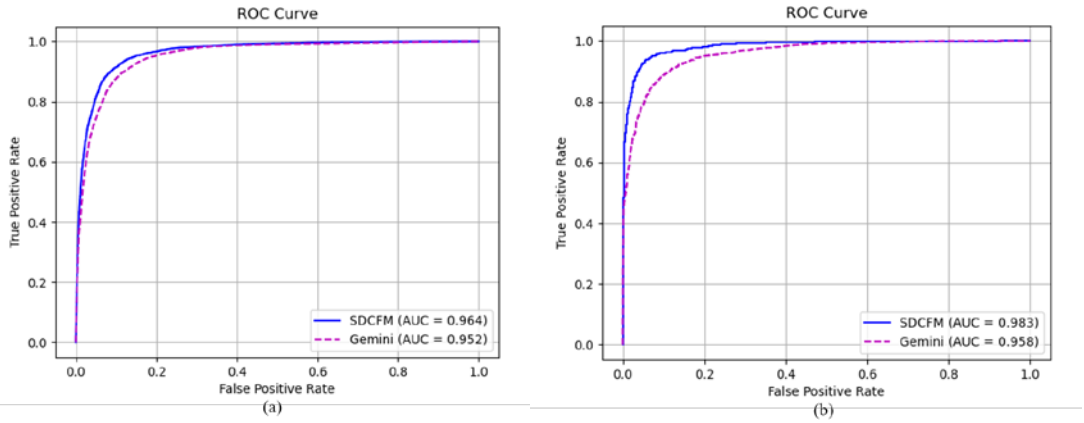

Fig. 5. ROC curves of different methods on the testing set.
(a) Results on Dataset I. (b) Results on Dataset II.
(b)

By comparing test results on Dataset I and Dataset II, we find that the advantage in the accuracy of SDCFM on Dataset I is not as many as that on Dataset II. The reason for this result may be related to the composition of the dataset. In Dataset I, only 20.4% of all functions contain calls to Glibc API (the number of functions in Dataset I is 6324, of which 1291 functions have calls to Glibc API), which dilutes the role of categorical Glibc API call feature in binary code similarity detection. However, as mentioned in Section 3.2, many vulnerabilities are closely related to Glibc API calls, so this attempt to improve the discriminability of function embedding by focusing on Glibc API calls has great significance.

Table 7 demonstrates the time overhead on testing samples of Dataset I and Dataset II. Due to the introduction of behavioral features and increased the neural network parameters, SDCFM takes more time to generate embeddings for binary functions. However, the increased time overhead is acceptable relative to the improvement in accuracy.

Table 7. Time overhead of Gemini and SDCFM on testing samples

	No. of sample pairs for testing	Time Overhead (s)		The difference on Average (μs)
		Gemini	SDCFM	
Dataset I	28476	7.56788	8.543317	34.2547
Dataset II	4382	2.673718	2.933303	59.2389

4.4.4 Experiments for Answering RQ4

Two experiments are constructed on Dataset III to evaluate the cross-architecture vulnerability detection ability of the SDCFM. We use two scenarios to simulate a practical cross-architecture binary vulnerability search task. The first is that the detected object and the vulnerability sample have the same architecture but different compilation optimization levels (Test 1). The second is that the detected object and the vulnerability sample have different architectures (Test 2). We compare SDCFM with Gemini in terms of the average ranking of the detected vulnerable functions in various search tests and the top-k hits (the times that the real vulnerable function is contained in top-k candidates over multiple tests). It is important to note that the Gemini and SDCFM models used in these experiments are general. That is, the models are not retrained with specific vulnerable functions. Moreover, some vulnerable functions in Dataset III do not appear in the training set. It is more conducive to testing the generalization ability of the model.

For the 36 binaries in Dataset III, we utilize the function embedding generator trained on Dataset II to generate embeddings of the functions. The vulnerable binary functions with the highest optimization level (i.e., O3) are used as query functions.

Test 1. In this evaluation, the query functions are searched from the binaries in the same architecture but with different compilation optimization levels. For example, the five known vulnerable functions at the O3 optimization level in ARM architecture are taken as the query functions. The task is to search similar functions with query functions from the affected binaries of ARM architecture with O0-O2 optimization levels. Thus, we have 15 vulnerability search tests in one architecture. In this way, a total of 45 different tests are formed for the three architectures of ARM, MIPS, and PowerPC. The rankings for vulnerable functions in the 45 search tests by Gemini and SDCFM are presented in **Table 8**.

Table 8. The search rankings of Gemini and SDCFM for five vulnerabilities at different optimization levels

Architecture	CVE No.	Gemini				SDCFM			
		O0	O1	O2	Avg	O0	O1	O2	Avg
ARM	CVE-2014-3508	2	2	1	2	2	1	1	1
	CVE-2016-6303	18	8	1	9	3	1	1	2
	CVE-2015-1791	75	3	1	26	1	1	1	1
	CVE-2017-6508	61	1	1	21	50	1	1	17
	CVE-2014-9471	10	1	3	5	28	1	1	10
MIPS	CVE-2014-3508	1	2	1	1	1	1	1	1
	CVE-2016-6303	2	2	1	2	6	6	1	4

	CVE-2015-1791	147	9	1	52	14	7	1	7
	CVE-2017-6508	124	21	2	49	124	1	1	42
	CVE-2014-9471	39	8	4	17	84	3	1	29
PowerPC	CVE-2014-3508	1	1	1	1	1	1	1	1
	CVE-2016-6303	1	5	1	2	2	1	1	1
	CVE-2015-1791	94	4	1	33	17	2	1	7
	CVE-2017-6508	77	8	2	29	84	1	1	29
	CVE-2014-9471	1	39	1	17	11	1	1	4
Avg		44	8	1	**18**	28	2	1	**10**

As shown in column 9 of **Table 8**, the SDCFM ranks the real vulnerable functions first in all the 15 queries at the O2 optimization level. The top-1 hit rate reaches 100%. It can significantly reduce the workload of manual analysis and facilitate the implementation of large-scale vulnerability searches. Among the top-1 candidates for all 45 searches, SDCFM identifies 29 real vulnerable functions, which is 61% higher than that of Gemini (=18). In addition, in terms of the average performance of all tests, ten functions on average need to be analyzed to find the real vulnerable function, and Gemini needs to analyze 18 functions on average.

Fig. 6 presents the results of the top-k hits for 15 search tests. The search results for different binaries with O0, O1, and O2 optimization levels are plotted in **Fig. 6(a)**, **Fig. 6(b)**, and **Fig. 6(c)**, respectively. SDCFM has eight more top-1 hits than Gemini at the O1 optimization level and detects all 15 real vulnerable functions in the top-10 candidates, as shown in **Fig. 6(b)**. **Fig. 6(c)** shows that the top-1 hit rate of SDCFM for binaries compiled with O2 optimization level reaches 100%. Overall, SDCFM outperforms Gemini in the vulnerability search across compilation optimization levels at the same architecture.

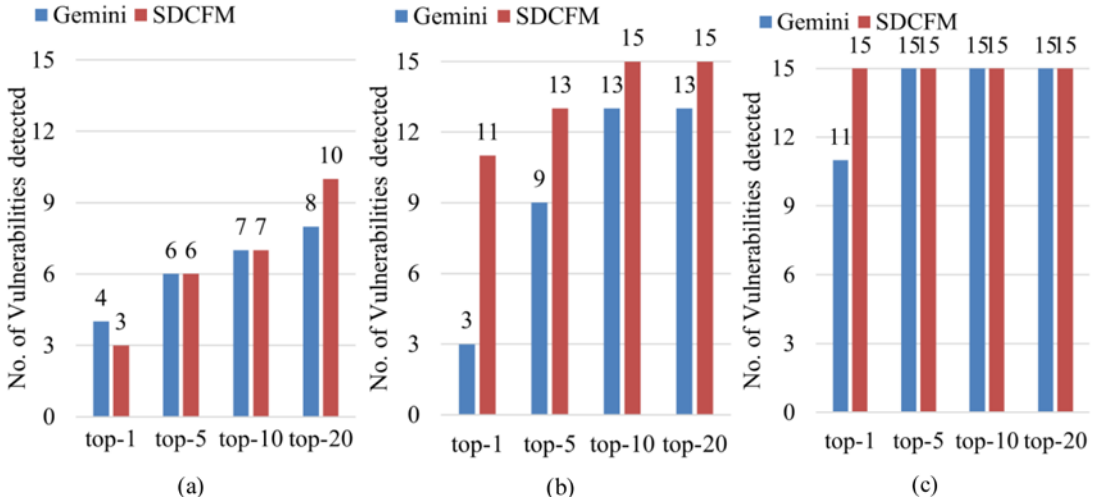

Fig. 6. Top-k hits results.
(a) O0 optimization level. (b) O1 optimization level. (c) O2 optimization level.

Test 2. In this evaluation, the query functions are searched from binaries in different architectures with the queries. For example, the vulnerable functions in PowerPC with O3 optimization level are taken as the query functions. The task is to search the vulnerable functions from the binaries in ARM and MIPS, regardless of the optimization level. Thus, eight different searches are performed for one vulnerable function, and 40 different searches in

total for five vulnerable functions in one architecture. The cross-architecture search results of the vulnerable functions in PowerPC architecture by Gemini and SDCFM are shown in **Table 9**.

Table 9. The cross-architecture search results of the vulnerable functions in PowerPC

Architecture	CVE No.	Gemini					SDCFM				
		O0	O1	O2	O3	Avg	O0	O1	O2	O3	Avg
MIPS	CVE-2014-3508	16	33	6	5	15	1	3	1	1	2
	CVE-2016-6303	9	11	4	4	7	3	1	29	27	15
	CVE-2015-1791	223	11	4	4	61	22	2	3	3	8
	CVE-2017-6508	152	29	11	1	48	121	1	1	1	31
	CVE-2014-9471	17	20	4	16	14	42	1	6	1	13
ARM	CVE-2014-3508	2	9	1	3	4	1	12	2	2	4
	CVE-2016-6303	3	4	18	21	12	3	1	1	2	2
	CVE-2015-1791	284	150	39	40	128	33	24	19	21	24
	CVE-2017-6508	77	8	19	35	35	75	1	1	1	20
	CVE-2014-9471	21	89	107	69	72	23	1	1	1	7
Avg		80	36	21	20	**39**	32	5	6	6	**13**

According to the results, SDCFM ranks the real vulnerable functions 13th on average, while Gemini ranks them 39th on average. In the top-5 candidate functions, SDCFM identifies 27 real vulnerable functions with an accuracy of 67.5%, which is 2.25× higher than Gemini (the value is 12). More importantly, SDCFM ranks the real vulnerable function in the first place 18 times, while Gemini correctly ranks that only 2 times.

In Test 2, there are 120 different cross-architecture searches in total for three architectures. The results are shown in **Fig. 7**. There are 51 times (i.e., the sum of 17, 18, and 16) that SDCFM ranks the real vulnerable functions in the top-1, and Gemini only has 15 times (i.e., the sum of 6, 2, and 7). Among other top-k candidates produced by SDCFM, it still contains more real vulnerable functions than Gemini. These results prove that SDCFM has a stronger ability for cross-architecture vulnerability detection than Gemini.

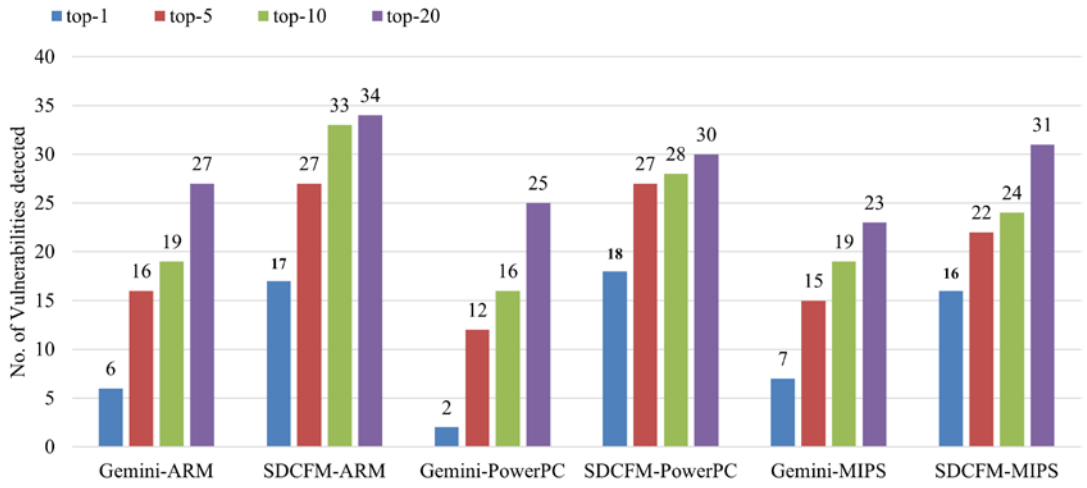

Fig. 7. The number of hits in the top-k candidates of Gemini and SDCFM on 120 vulnerability search tests across architectures. Specifically, SDCFM-ARM indicates using the O3 optimization level vulnerable function of ARM architecture as a query and searching it from the binaries of MIPS or PowerPC architecture with O0-O3 optimization levels. Others are all expressed in this way.

There are situations where the ranking of vulnerable functions detected by SDCFM lags behind Gemini. For example, as shown in **Table 8**, when searching for the CVE-2016-6303 vulnerable function at the O1 optimization level in MIPS architecture, SDCFM ranks the real vulnerable function 6th, while Gemini ranks it in the second place. By analyzing and checking those functions that are placed before the real vulnerable function, we attribute this to two main reasons. One is that, although different functions may have different statistical features, structural features, and behavioral features, it is still possible to generate the same embedding for these functions through neural network fusion. The other reason is that SDCFM classifies Glibc API functions, hence, the functions containing similar Glibc API calls will have the same behavioral features, which may also result in the same embedding for different functions.

5. Discussion and Future Work

We have demonstrated the effectiveness and high accuracy of SDCFM in cross-architecture binary code similarity detection and vulnerability search. However, this method still faces some challenges.

SDCFM uses static analysis methods for binaries to extract all the raw features according to CFM. Therefore, SDCFM cannot handle the obfuscated code in binaries currently, which may affect the detection accuracy.

The extraction of CFG and other features currently used by CFM depends on angr completely. Therefore, the accuracy of feature extraction is determined by the analysis capability of angr.

At present, SDCFM only focuses on the Glibc API and has not considered the other APIs. It is necessary to extend the scope of behavioral features to other public basic libraries in the future to improve the characterization ability further, which is exactly the focus of our future work.

6. Conclusion

This paper proposes a cross-architecture binary function similarity detection method based on composite feature model called SDCFM. The CFM covers behavioral features, statistical features, and structural features, to improve the ability of characterizing binary functions. Besides, SDCFM adopts the hierarchical embedding method to fuse statistical and behavioral features, and selects the attributes of predecessor nodes in CFG to iteratively update the graph embedding network. Experimental results show that the AUC value of SDCFM reaches 0.964 on the benchmark dataset. Furthermore, when tested on the dataset consisting of binary functions with Glibc API function calls, a higher AUC (=0.983) can be reached. Meanwhile, SDCFM has a stronger ability to locate vulnerable functions accurately. Among the 120 constructed cross-architecture vulnerability search tasks, SDCFM accurately places the target vulnerable function in the first candidate 51 times, which is 2.4× more than that of Gemini (only 15). In general, SDCFM further improves the accuracy and stability of firmware binary similarity detection.

References

[1] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. of the 23rd USENIX Security Symposium*, San Diego, CA, USA, pp. 95-110, August 2014. Article (CrossRef Link)

[2] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-based Bug Search for Firmware Images," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, pp. 480-491, October 2016. Article (CrossRef Link)

[3] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224-2236, 2021. Article (CrossRef Link)

[4] Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable Secure Computing*, 19(4), 2244–2258, 2022. Article (CrossRef Link)

[5] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "VulSeeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation," in *Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, pp. 803-808, October 2018. Article (CrossRef Link)

[6] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," in *Proc. of the 27th Network and Distributed System Security Symposium*, San Diego, California, USA, February 2020. Article (CrossRef Link)

[7] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, pp. 363-376, October 2017. Article (CrossRef Link)

[8] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 896-899, Montpellier, France, September 2018. Article (CrossRef Link)

[9] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs," in *Proc. of the 26th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2019. Article (CrossRef Link)

[10] 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 *Proc. of the 34th AAAI Conference on Artificial Intelligence*, New York, NY, USA, vol. 34(01), pp. 1145-1152, February 2020. Article (CrossRef Link)

[11] Y. Ji, L. Cui, and H. H. Huang, "BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network," in *Proc. of the 2021 ACM Asia Conference on Computer and Communications Security*, Virtual Event, Hong Kong, pp. 702-715, June 2021. Article (CrossRef Link)

[12] Marcelli A, Grazizno M, Ugarte-Pedrero X, Fratantonio Y, Mansouri M, and Balzarotti D, "How Machine Learning Is Solving the Binary Function Similarity Problem," in *Proc. of 31st USENIX Security Symposium*, Boston, MA, USA, pp. 2099–2116, August 10-12, 2022. Article (CrossRef Link)

[13] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural Nets Can Learn Function Type Signatures From Binaries," in *Proc. of the 26th USENIX Security Symposium*, Vancouver, BC, Canada, pp. 99–116, August 2017. Article (CrossRef Link)

[14] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. of the 40th IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, pp. 472-489, May 2019. Article (CrossRef Link)

[15] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proc. of the 23rd Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2016. Article (CrossRef Link)
[16] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing API usages through semantic cross-checking," in *Proc. of the 25th USENIX Security Symposium*, Austin, TX, USA, pp. 363-377, August 2016. Article (CrossRef Link)
[17] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Proc. of the 25th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2018. Article (CrossRef Link)
[18] Y. Liao, R. Cai, G. Zhu, Y. Yin, and K. Li, "MobileFindr: Function Similarity Identification for Reversing Mobile Binaries," in *Proc. of the 23rd European Symposium on Research in Computer Security*, Barcelona, Spain, pp. 66-83, September 2018. Article (CrossRef Link)
[19] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "DMalNet: Dynamic malware analysis based on API feature engineering and graph learning," *Computers & Security*, vol. 122, 102872, 2022. Article (CrossRef Link)
[20] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph Matching Networks for Learning the Similarity of Graph Structured Objects," in *Proc. of the 36th International Conference on Machine Learning*, Long Beach, California, USA, pp. 3835-3845, June 2019. Article (CrossRef Link)
[21] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature Verification Using A "Siamese" Time Delay Neural Network," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 4, pp. 669-688, 1993. Article (CrossRef Link)
[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proc. of the 2016 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, pp. 138-157, May 2016. Article (CrossRef Link)
[23] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. of the 33rd International Conference on Machine Learning*, New York City, NY, USA, pp. 2702-2711, June 2016. Article (CrossRef Link)
[24] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. of the 31st Annual Conference on Neural Information Processing Systems*, Long Beach, CA, USA, pp. 6351-6361, December 2017. Article (CrossRef Link)
[25] H. Liang, Z. Xie, Y. Chen, H. Ning, and J. Wang, "FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching," *Computers & Security*, vol. 99, 102032, 2020. Article (CrossRef Link)
[26] D. Hendrycks, and K. Gimpel, "Gaussian Error Linear Units (GELUs)," Jun 2016. Article (CrossRef Link)
[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, pp. 265-283, November 2016. Article (CrossRef Link)
[28] Common Vulnerabilities and Exposures. [Online]. Available: https://cve.mitre.org/.
[29] J. Gao, X. Yang, Y. Jiang, H. Song, K.-K. R. Choo, and J. Sun, "Semantic Learning Based Cross-Platform Binary Vulnerability Search For IoT Devices," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 2, pp. 971-979, 2021. Article (CrossRef Link)

Xiaonan Li received the M.S. degree from Beijing University of Posts and Telecommunications, China, in 2007. She is presently pursuing the Ph.D. degree at Information Engineering University, Zhengzhou, China. She has been with the School of Computer Science, Zhongyuan University of Technology, as a lecturer. Her research interests include binary analysis and machine learning.

Guimin Zhang received the Ph.D. degree from Information Engineering University in 2018. He is currently a lecturer at Information Engineering University. His research interests include cyberspace security and trusted computing.

Qingbao Li is currently a professor at Information Engineering University. His research interests include information security, software protection theory and trusted computing.

Ping Zhang is currently a professor at Information Engineering University. Her research interests include information security and parallel compilation.

Zhifeng Chen received the Ph.D. degree from Information Engineering University in 2016. He is currently a lecturer at Information Engineering University. His research interests include information security, software protection theory and trusted computing.

JinJin Liu received the Ph.D. degree from Information Engineering University in 2021. She is a lecturer now in the School of Computer Science, Zhongyuan University of Technology. Her research interests include artificial intelligence, computer vision, and pattern recognition.

Shudan Yue is currently pursuing the Ph.D. degree at Information Engineering University. Her research interests include vulnerability detection, network and information security and machine learning.