Multi-feature based Function Embedding Network for Binary Code Similarity

XIANGYU LI, GUOHAO WU, ZIHUI GUO AND HONGLIANG LIANG, TSIS Lab., Beijing University of Posts and Telecommunications, China

Binary similarity detection determines whether two given binary code snippets are similar or not, usually on function granularity. This task is challenging due to different compilation optimizations and CPU architectures. Recently, deep-learning methods have made great achievements in this field, although most of them use artificially selected features or ignore some important semantic information like code literals or function signatures during feature processing. In addition, random samples and pair loss function are used in similarity training, which only covers limited similarity relations between functions.

In this paper, a new framework MFEN-Sim is proposed to detect similar binary functions. The framework contains three stages: feature extraction and normalization, mutli-feature based function feature embedding network (MFEN) and similarity learning network. Multiple features including assembly instructions, CFG structures and function code literals are extracted from binary functions. Then these features are fed into MFEN composed of three modules: function semantic and structure embedding module, function signature prediction module, and function code literal embedding module. The three modules generate embeddings representing the function semantic and structural features, the function signature prediction features and the function code literal features. Finally, MFEN-Sim utilizes a similarity training network based on contrastive learning to make MFEN recognize more similarity relations between functions. MFEN-Sim is evaluated on 281,601 functions in 144 binaries and 17 CVEs in real-world software. Experimental results show that our work outperforms state-of-the-art systems (*i.e.,* Gemini, FIT and SAFE) by 7.1%, 9.9% and 8.2% on AUC metric in cross-architecture, optimization-level similarity detection, and achieves higher recall than baselines in searching vulnerabilities in real-world applications.

Additional Key Words and Phrases: Binary Code Similarity, Deep Learning, BERT, GAT, Contrastive Learning

1 INTRODUCTION

Binary code similarity detection is to decide whether two given binary code snippets are similar, usually on the function granularity. It has many security applications: code clone detection, malware detection, vulnerability search, etc. Code reuse is common in modern software development though it may bring potential risks. For example, the heart-bleed bug [1] in openssl, a widely used library, which can cause memory leak due to lacking of validation on user input, has affected a large amount of web servers, routers and other devices on the Internet.

Moreover, the fact that most commercial software is closed source makes binary code similarity detection necessary. However, detecting similarity in binary code is facing with two more difficulties than that in source code. First, a binary file is a final product out of a complicated compilation process that involves massive transformations (*e.g.,* optimizations), such as function inlining, instruction replacement and dead code elimination, which eventually discards a majority of high-level program semantics. Second, even the same source code can be compiled into different binaries due to different CPU architectures and compilation optimizations.

In order to detect similarity in binary code, many approaches leverage static analysis [6–8, 12, 13, 21, 25, 31, 38] and dynamic analysis techniques [2, 5, 17, 18, 27, 29, 30, 34, 36], and have achieved good results. However, static analysis methods usually rely on manually selected features or rules, and use complex graph matching algorithm to detect similar functions, which is not effective or computationally inefficient. While dynamic analysis methods often require to run the target binary

Author's address: Xiangyu Li, Guohao Wu, Zihui Guo and Hongliang Liang, TSIS Lab., Beijing University of Posts and Telecommunications, Beijing, China, (pyterware,guohaowu,guozihui,hliang)@bupt.edu.cn.

code, record and analyze its execution trace, which is not efficient or difficult even not possible for some cases like firmware.

Recently, several studies [10, 11, 24, 28, 35, 37, 39] apply deep learning in binary analysis. They leverage control flow graphs or assembly instructions as input, and use neural network to learn semantic embeddings which are then used to detect binary code similarity across architectures and/or optimization levels. These methods have improved the detection accuracy and efficiency, though they are still facing the following problems:

First, code literals such as strings and constants in programs are important semantic information because their values don't change after compilation. However, previous approaches ignore these values and thus drop the semantic information in code literals. For example, Gemini [35] only uses the number of strings and constants as part of features, while word embedding based approaches [11, 24, 28, 39] replace strings or constants with special tokens to avoid the out-of-vocabulary (OOV) issue.

Second, the function signature of a program is an important feature for representing the function semantics and able to improve the performance of binary code similarity detection [22], however, previous works didn't design methods to extract the information in function signatures.

Third, prior approaches[24, 28, 35] leverage supervised metric learning algorithm, which relies on the randomly sampled triplets, *i.e.,* (function1, function2, label), and the pair loss function. However, the sampled triplets and the pair loss function only cover limited function similarity relations, which is not effective and may result in over-fitting.

In this paper, we propose a new framework called MFEN-Sim to detect similarity in binary code on the function granularity. MFEN-Sim contains three stages: feature extraction and normalization, multi-feature based function embedding network (MFEN) and similarity learning network. As the framework's core part, MFEN is composed of three modules: function semantic and structure embedding module, function signature prediction module, and function code literal embedding module. We extract features from assembly instructions, the control flow graph (CFG) structure and code literals of a function in feature extraction and normalization stage, then feed these features into MFEN's modules to generate embeddings which represent the function semantic and structural features, the function signature prediction features and function code literal features, respectively. Finally, we train MFEN using the similarity learning network based on contrastive learning and use the trained MFEN to measure the similarity between functions.

We evaluate MFEN-Sim on 281,601 functions in 144 binaries and 17 CVEs in real-world software. MFEN-Sim outperforms three state-of-the-art systems (*i.e.,* Gemini, FIT and SAFE) by 7.1%, 9.9% and 8.2% on AUC metric on similarity detection in cross-architecture and cross-compilation-optimization binary code, respectively. Moreover, MFEN-Sim achieves higher recall than baselines in searching vulnerabilities in real-world applications, which demonstrates its practicability for binary analysis.

Our contributions are summarized as follows:

- A multi-feature based function embedding network (MFEN) is proposed to generate robust function embedding against diverse CPU architectures and compilation optimizations. MFEN extracts the function semantic and structural embedding, function signature prediction embedding and function code literal embedding respectively, and combines them to construct the final function embedding.
- A similarity learning network based on contrastive learning is put forward to train MFEN. Function batches are randomly sampled during training and MFEN is trained with the contrastive learning loss function that considers both positive and negative function pairs to learn more relations between functions.

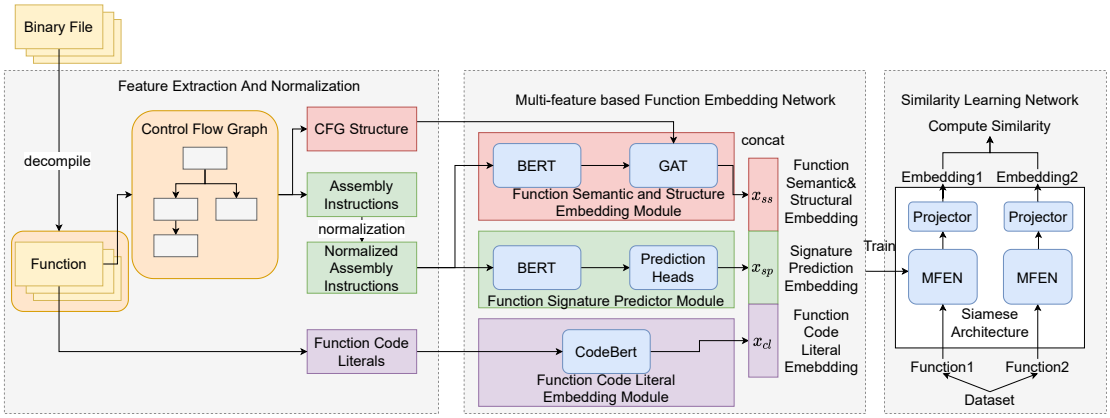

Fig. 1. Overview of MFEN-Sim

- A novel general framework MFEN-Sim is presented for binary similarity detection across architectures and optimization levels. MFEN-Sim is evaluated on over 280,000 functions in 8 open-source projects and two vulnerability datasets with 17 CVEs. The results demonstrate that MFEN-Sim outperforms three state-of-the-art systems, *i.e.,* Gemini, FIT and SAFE, with higher accuracy in similarity detection, and with higher recall in vulnerability search.
- We make our source code, benchmark, and experimental data publicly available[1], in the hope of facilitating further research in the software security field.

2 APPROACH OVERVIEW

As shown in Fig. 1, MFEN-Sim consists of three stages: 1) feature extraction and normalization, 2) multi-feature based function embedding network (MFEN), and 3) similarity learning network.

In feature extraction and normalization, each binary file is decompiled to obtain its functions, which are extracted to get three kinds of features, *i.e.,* normalized assembly instructions, the CFG structure and function code literals.

Then the output features are fed into the three modules of MFEN: function semantic and structure embedding module, function signature prediction module and function code literal embedding module. Specifically, for each function in a binary file, its assembly instruction sequences and CFG structure are fed into a BERT and a graph attention network in the first module to generate the function semantic and structure embeddings. The second module computes the signature prediction embeddings with a BERT and multiple prediction heads. The third module utilizes a pretrained CodeBert to generate the code literal embeddings. Through fusing these three kinds of embeddings, the final embeddings of the input function are generated.

In similarity learning network, the MFEN is combined with a projector and trained with the siamese architecture to maximize semantically similar function's similarity. The trained MFEN can be used to generate functions' embeddings that measure the similarity of functions.

3 FEATURE EXTRACTION AND NORMALIZATION

In this stage, we extract three kinds of function features from binary files using IDA pro [19]: the assembly instructions, the CFG structures and the function code literals including strings and constants.

[1]https://github.com/PyterwareLi/mfen-sim

Table 1. Normalization Rules

Operand	Usage	Description	Normalized Token
Immediate value	call target	function call within a binary function call out of a binary recursive call	innerfunc externfunc self
	jump	jump destination	jumpdst
	reference	string in .rodata section statically allocated variables in .bss section data other than string	dispstr dispbss dispdata
	other	immediate value above the threshold	imm
Pointer	indirect(string) indirect(data) direct	[base+index*scale+dispstr] [base+index*scale+disp] reference or call target of immediate value	[base + index * scale + dispstr] [base + index * scale + dispdata] innerfunc, externfunc, self, dispstr, dispbss or dispdata

Each assembly instruction contains two parts: an operator and an (or several) operand(s). The total number of operators is limited, while the number of operands can be very large due to different compilation options, which may cause the out-of-vocabulary (OOV) problem. To address the OOV problem, it is necessary to normalize certain tokens in operands, *i.e.,* replace them with special tokens to express the meaning of original tokens.

For example, SAFE [28] replaces memory address with `MEM` and immediate values with `IMM`. DeepBinDiff [11] normalizes registers with their sizes, and converts all immediate values into `imme`. These methods using coarse-grained normalization may lose some contextual and semantic information. For example, all call destinations are replaced with the same notation, rendering every call instruction identical. Unfortunately, they also apply a coarse-grained tokenization to assembly instructions. For instance, SAFE treats a whole instruction as a word which ignores its internal details. Though some memory access operands are formed by multiple registers and constant values, they are treated as one token in the coarse-grained tokenization. Moreover, due to the complex and diverse instruction formats, such coarse-grained tokenization leads to a relatively large vocabulary size, which makes it easier to cause OOV problems in untrained software.

To strike a balance between keeping the semantics of binary code and maintaining a reasonable vocabulary size, we adopt a fine-grained tokenization approach and a well-balanced normalization approach on three architectures (x86, ARM, MIPS) based on the normalization rules for x86 [23]. Specifically, each instruction is first decomposed into basic tokens. For instance, an instruction `mov rax, qword [rsp+0x58]` is divided into `mov, rax, qword, [, rsp,+, 0x58`, and `]`. Then some tokens are replaced with normalized tokens according to the rules which are summarized in Table 1. For example, an immediate number may have different usages, *e.g.,* a constant value, a string reference, a jump destination, or a statically-allocated variable and so on. Previous methods drop such an implication, which will make an instruction rarely distinguishable from others. In contrast, we replace immediate numbers with different tokens by their usage. For constants, we use a threshold 500 to judge whether to keep their values or replace them with special tokens, because we believe some constants can help distinguish instructions. For instance, if the threshold is 5,000, the constant `0x58` in the above example will be replaced with its value 88, and the constant `0xFFFFFFFE` in the instruction `mov EAX, 0xFFFFFFFE` will be replaced with `IMM`. The former can be a reference to a variable, which can be distinguished from other variable references like `mov rax [rsp+0x4]`, while the latter can be a constant generated by a compiler, which often appears in assembly instructions. The threshold is also used to control the vocabulary size.

4 MULTI-FEATURE BASED FUNCTION EMBEDDING NETWORK

The multi-feature based function embedding network (MFEN) is composed of three modules: the function semantic and structure embedding module, the function signature prediction module and the code literal embedding module. Given a function f, MFEN can get its semantic and structure embedding x_{ss}, signature prediction embedding x_{sp} and function code literal embedding x_{cl}, which are concatenated to form the function's whole representation embedding x_f.

$$x_f = MFEN(f) = x_{ss} \| x_{sp} \| x_{cl} \tag{1}$$

4.1 Function Semantic and Structure Embedding Module

The function semantic and structure embedding module is built with a BERT model and a GAT model, which is used to capture the semantic features and the graph structural features of functions, respectively.

4.1.1 Input Representation. This module takes normalized assembly instructions and CFGs of functions as inputs. A CFG is denoted as $G =< N, E >$, where N and E is the set of basic blocks and edges respectively. We define the basic block set $N =< BB_1, BB_2, ..., BB_m >$. Each basic block is defined as $BB =< Inst_1, Inst_2, ..., Inst_k >$. The instruction sequence in a basic block is treated as a sentence and all symbols in the instructions as tokens. Each edge in $E =< e_1, e_2, ..., e_n >$ is defined as a tuple (ID of source basic block, ID of target basic block). For instance, the edge from BB_1 to BB_2 is represented as $(1, 2)$.

4.1.2 Semantic Modeling. We utilize a pretrained BERT [9] model to extract the semantic information from assembly instructions. This model is based on a multi-layer Transformer encoder, and designed to pretrain a deep bidirectional representation in unlabeled text through joint adjustment of left and right context in all layers. Furthermore, the task Masked Language Model (MLM) and the task Next Sentence Prediction (NSP) during training are used to learn the word level and sentence level semantic information, respectively. Based on the above tasks, we design two tasks to pretrain the model as shown in the Fig. 2.

The first task to pretrain the model is the MLM task. Fifteen percent of tokens in assembly instruction sequence are randomly masked out. Following the settings in BERT [9], eighty percent and ten percent of the chosen tokens are masked by [MASK] (mask-out tokens) and replaced with other tokens in the vocabulary (corrupted tokens) respectively, while the remain ones are kept unchanged. Then, the transformer encoders in BERT learn to predict the masked-out and corrupted tokens, and output a probability with a softmax layer located on top of the last transformer encoder for predicting a particular token $t = [MASK]$. This task uses a cross entropy loss function:

$$L_1(\theta, \theta_1) = -\sum_{i=1}^{M} logP(m = m_i \mid \theta, \theta_1),\ m_i \in [1, 2, \ldots, V] \tag{2}$$

where θ represents the parameters of the transformer encoder in BERT, θ_1 represents the parameters of output layer connected on the encoder in MLM task, M is the collection of masked tokens during training, and V is the size of the vocabulary of the input sequences.

In each self-attention layer in BERT, an input token attends to other tokens and updates its embedding by computing the attention weights with other tokens' embeddings, thus each token's embedding encodes context-sensitive information. In other words, a token's embedding changes according to its location and context, which is different from a static embedding model such as word2vec. Through this task, the BERT model can learn the semantic information of assembly instructions.

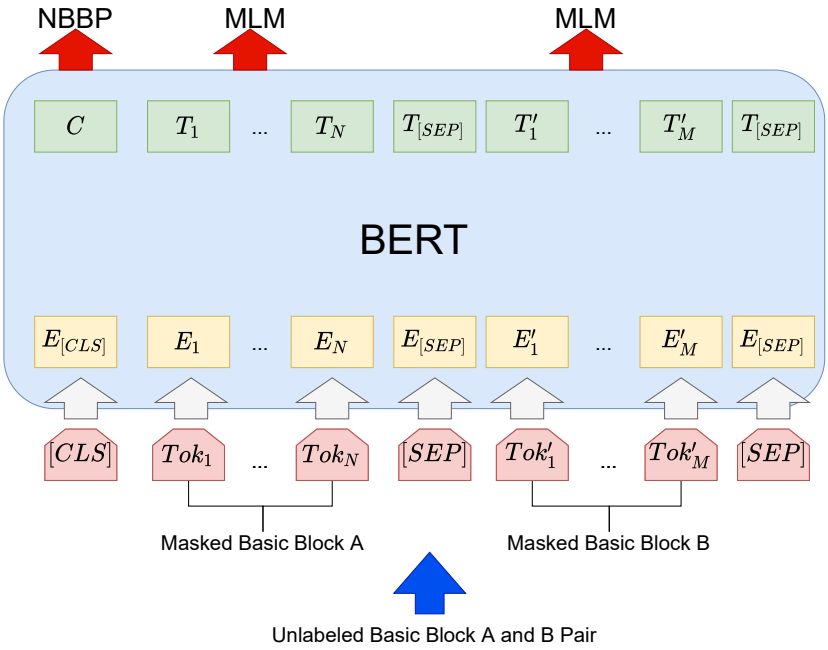

Fig. 2. Semantic Modeling with BERT

The second task is the next basic block prediction (NBBP), similar to the NSP task. It concatenates the instruction sequences of two basic blocks, and predicts whether both blocks satisfy the *next-to* relation through the classification header of BERT model. The loss function is defined as follows:

$$L_2(\theta, \theta_2) = -\sum_{j=1}^{N} logP(n = n_i \mid \theta, \theta_2),\ n_i \in [\text{"IsNext"}, \text{"NotNext"}] \tag{3}$$

where θ_2 represents the parameters of classifier connected to the encoder in NBBP task, and n is the predicted value of NBBP, which means whether the *next-to* relation is satisfied.

Two basic blocks are regarded as a positive sequence if they are on the same edge in a CFG, otherwise as a negative sequence. Through this task, the model can learn the adjacent relations between basic blocks.

By optimizing the joint loss function of the two tasks, the BERT model can learn the word-level and basic block-level semantic information of assembly instructions.

$$L(\theta, \theta_1, \theta_2) = L_1(\theta, \theta_1) + L_2(\theta, \theta_2) \tag{4}$$

4.1.3 Graph Structure Modeling. Edges in CFG represent the execution sequence between basic blocks and reflect the execution semantics of the function. We utilize the graph attention network (GAT) to extract the structural information from CFGs. GAT [33] is a graph neural network based on attention mechanism like Transformer. Its basic idea is to assign different attention weights for neighbor nodes and update the current node's representation by the weighted sum of its neighbor nodes with attention weights. Through the attention mechanism, GAT can adaptively allocate attention weights to different neighbors. Furthermore, GAT allows high parallel computation and has strong generalization ability.

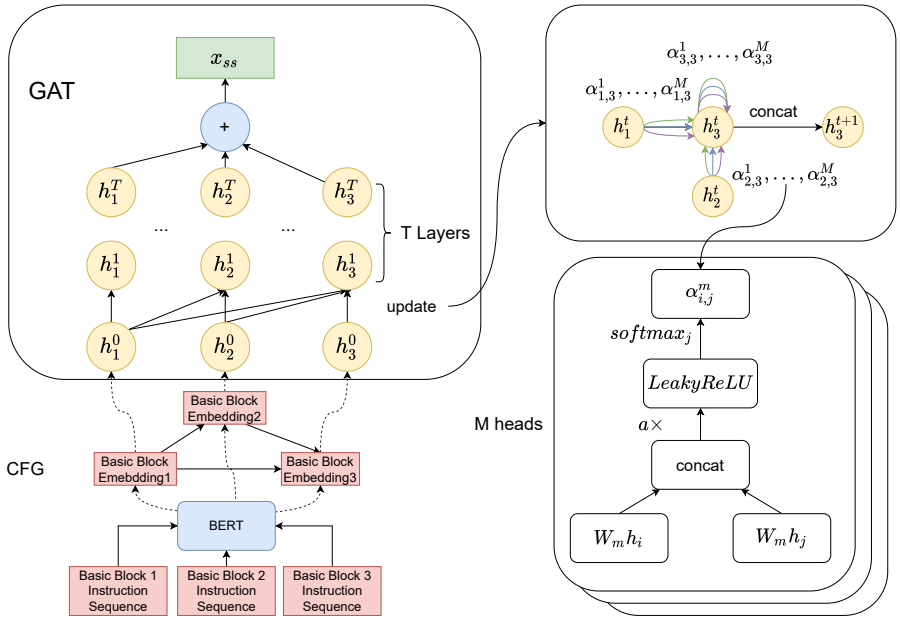

Fig. 3. Graph Structure Modeling with GAT

As shown in the Fig. 3, first, each basic block's instruction sequence in the CFG is fed into BERT to extract the basic block embedding. Then the basic block embedding and the CFG structure are used as the input of GAT to compute the function structure and semantic embedding. GAT has T layers, and every time the nodes of the input CFG go through a layer, GAT passes the semantic embedding of a node to its neighbor nodes. Through the multi-head attention mechanism, each node can focus on neighbor nodes with more semantic information. The attention weight α of two adjacent basic blocks i, j of attention head m is calculated according to the following formula:

$$\alpha_{ij}^{m} = \frac{exp\left(LeakyReLU\left(a\left[W_m h^t{}_i || W_m h^t{}_j\right]\right)\right)}{\sum_{k\in\mathcal{N}_i} exp\left(LeakyReLU\left(a\left[W_m h^t{}_i || W_m h^t{}_k\right]\right)\right)} \tag{5}$$

where h^t is the feature vector of a node at the tth layer, h^0 is the embedding of a basic block, W_m is the learnable weight matrix of mth attention head, a is a learnable single-layer feedforward neural network, $LeakyReLU$ is the activation function of GAT, $||$ is concatenating vectors, and $\mathcal{N}_i$ represents the neighbor nodes of node i.

Then, each node's embedding is updated based on the multi-head attention mechanism as follows:

$$h_i^{t+1} = \Big\|_{m=1}^{M} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{m} W_m h_j\right) \tag{6}$$

where m denotes the mth attention head, the M denotes the number of attention heads, $\alpha_{i,j}^{m}$ denotes the attention weight of the mth attention head between edge i, j, W_m denotes the mth weight matrix.

At last, the node embeddings of the CFG updated by GAT are summed as the function semantic and structure embedding x_{ss}.

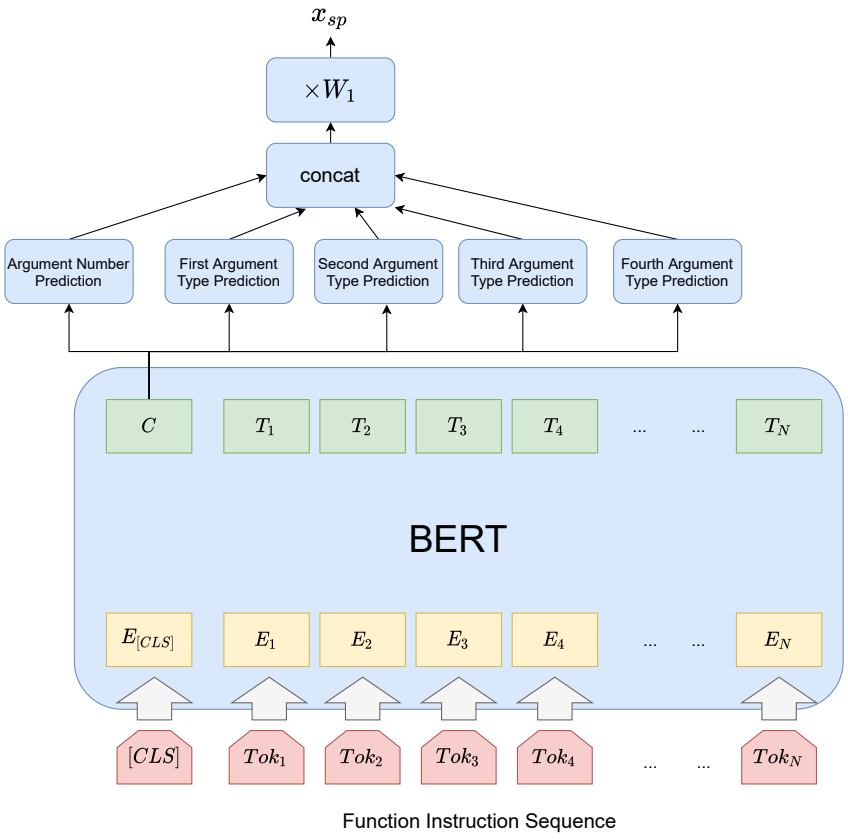

Fig. 4. Function Signature Prediction Module

$$x_{ss} = \sum_{i=0}^{n} h_i^T \tag{7}$$

4.2 Function Signature Prediction Module

If two functions are complied from the same source code, they should have the same function signature, *e.g.,* the type of each argument and return value. In other words, a function signature is robust because compilers won't change the function's input/output behaviors. As such, we combine the function signature inference [4] [22] with binary similarity detection. We design and train a function signature predictor in MFEN to predict the argument number and argument types in function signatures, and use it to generate function signature prediction embeddings. Because a function may have multiple arguments, it is not appropriate to train a model for each argument to predict its type. Furthermore, we observe that most functions (over 85%) in our dataset take less than five arguments, hence, we use five tasks to train the function signature predictor, *i.e.,* a task for argument number prediction and four tasks for argument type prediction.

As shown in the Fig. 4, we train BERT with five prediction heads. For each task t and a function f, the function's assembly instruction sequence is used as input of the task t. For the argument number prediction task, we label the function with more than nine arguments as "9+" to limit the

total number of predication classes. For the kth argument type prediction task, we use nine abstract types as labels, *i.e.,* `char, short, int, float, enum, struct, void, void*` and `none` (none means there is no kth argument). These classification tasks are trained in parallel, and the cross entropy is computed as the loss function of each task t:

$$L_t = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{K_t} y_{ic}\log\left(p_{ic}\right) \tag{8}$$

where y_{ic} is the label indicating whether the ith sample belongs to the cth class, p_{ic} is the prediction score that ith sample belongs to the cth class, and N is the total number of input samples, K_t is the classification number of the current task t. The model's loss function is the sum of the loss function of each task:

$$L = \sum_{t=0}^{TaskNum} L_t \tag{9}$$

The trained model is used to generate signature prediction embedding of a function. Given a function's instruction sequences f_{insts}, we use the prediction score of each class in task t to form a feature vector:

$$x_t = \text{PredictionHead}_t(\phi_{\text{BERT}}(f_{insts})_{cls}) = \left(p_1, \ldots, p_c, \ldots, p_{K_t}\right) \tag{10}$$

where ϕ_{BERT} is the BERT model, f_{insts} is the input function's assembly instruction sequence, $\phi_{\text{BERT}}\left(f_{insts}\right)_{cls}$ is the sentence embedding outputted by BERT, PredictionHead_t is the prediction head that outputs probabilities for task t, p_c is the output probability of class c, K_t is the total class number of task t.

Then we generate the signature prediction embedding x_{sp} with the five vectors:

$$x_{sp} = (x_1 \| x_2 \| x_3 \| x_4 \| x_5) W_1 \tag{11}$$

where W_1 is a linear transformation matrix, and $\|$ is concatenating vectors.

4.3 Function Code Literal Embedding Module

We use code literals, *i.e.,* string and constant values, as additional features because binary functions compiled from the same source code may still share the same strings and constants.

We take the strings and constants extracted from a function as two sequences and concatenate them as one sequence, and feed it into a pretrained CodeBert [14] to obtain the function's code literal embedding x_{cl}, as shown in the Fig. 5. CodeBert is pretrained with source code and natural languages, thus the extracted strings and constants can be fed into the CodeBert directly.

$$x_{cl} = \phi_{\text{CodeBert}}\left(f_{sc}\right)_{<s>} W_2 \tag{12}$$

where ϕ_{CodeBert} is CodeBert model, f_{sc} is the sequence of string and constant, $\phi_{\text{CodeBert}}\left(f_{sc}\right)_{<s>}$ is the sentence embedding outputted by CodeBert, W_2 is a linear transformation matrix.

5 SIMILARITY LEARNING NETWORK

To learn similarities between functions, we propose a similarity learning network based on contrastive learning [3, 16]. The similarity learning network is based on the Siamese architecture and contains the following main components, as shown in Fig. 6.

Data Generation. To learn the similarity relations between functions, we generate syntactically diverse but semantic similar or equivalent functions with different optimization levels and target CPU architectures. Specifically, we compile the same source code with different optimization levels and target CPU architectures to make sure that the compiled functions have similar or equivalent

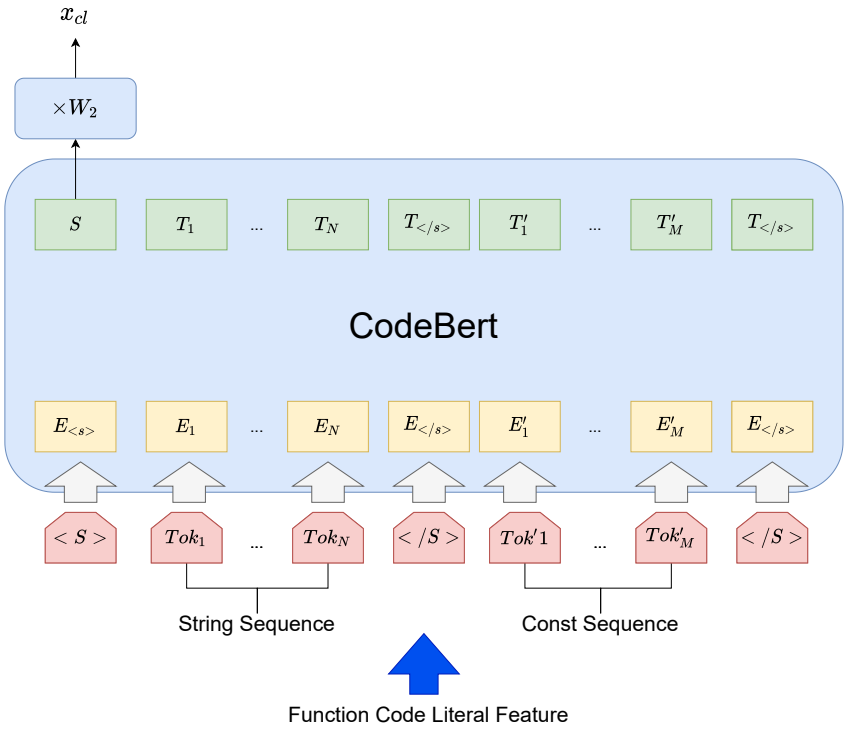

Fig. 5. Function Code Literal Embedding Module

semantic but different syntax in assembly instruction. These functions provide informative diversity in helping the model to capture function's semantic information.

Encoder. The encoder extracts the representation vectors from input function features. The multi-feature based function embedding network (section 4) is used as the encoder. The similarity learning network has two identical encoders that share parameters.

Given a function f, the encoder can get its function embedding x_f (section 4).

$$x_f = \text{Encoder}(f) = MFEN(f) \tag{13}$$

Projector. We use a multi-layer perceptron (MLP) with two layers as the projector. Then we pass the function representation embedding x_f through the projector to get the final embedding z_f.

$$z_f = \text{MLP}(x_f) = W_3\sigma(W_4 x_f) \tag{14}$$

where σ represents the activation function, *i.e.,* ReLU in this paper. W_3 and W_4 are linear transformation matrices. The two projectors in Fig. 6 also share parameters.

Batch Sampler and Loss Function. We random sample N functions that compiled from different source codes, and for each sampled function, we further sample another function that compiled from the same source code but with a different compilation option, such as a different optimization level or CPU architecture, resulting in a mini-batch of $2N$ binary functions. We treat functions compiled from the same source code as the positive pairs and sample negative function pairs from the mini-batch. Given each function, we treat other $2(N-1)$ functions compiled from different source code within the mini-batch as negative samples. We calculate the cosine distance

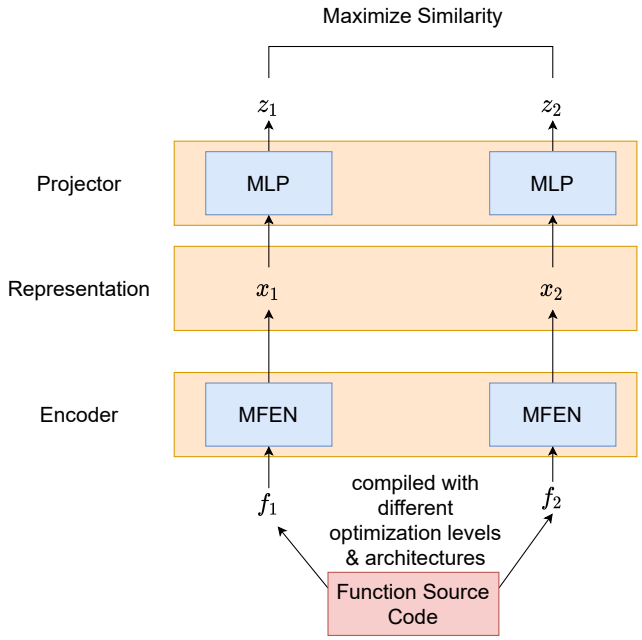

Fig. 6. Similarity Learning Network

between the final embeddings of two functions and take it as their similarity. The loss function for each function pair (i, j) is:

$$l_{i,j} = -log \frac{exp\left(cos\left(z_i, z_j\right)/\tau\right)}{\sum_{k=1,k\neq i}^{2N} exp\left(cos\left(z_i, z_j\right)/\tau\right)} \tag{15}$$

where τ denotes a temperature parameter to adjust the range of cosine similarity.

This loss is computed across all function pairs in the mini-batch and is averaged to get the final loss l_i. As a result, l_i makes those functions compiled from the same source code as close as possible, and other functions as far as possible. So the trained model can make the distances between similar functions lower than dissimilar functions.

The trained MFEN can be used to calculate the similarity between functions as follows and judge whether two functions are similar.

$$\text{Sim}(f_i, f_j) = cos(\text{MFEN}(x_i), \text{MFEN}(x_j)) = \frac{x_i^T x_j}{|x_i||x_j|} \tag{16}$$

6 EVALUATION

6.1 Implementation and Setup

We leverage IDA pro 7.5 [19] and a build-in IDA pro plugin IDApython [20] to implement the feature extraction and normalization, and we implement MFEN and the similarity learning network using PyTorch 1.8.1 [32] in Python. We ran all experiments on a Linux server running Ubuntu 18.04, with an Intel Xeon 6126 CPU at 2.60GHz with 48 virtual cores, 128 GB RAM and a Nivdia RTX 3090 GPU.

Table 2. The details of datasets

Training Set														
Software	Bin.	Functions												Total
		x86				ARM				MIPS				
		O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	
binutils	336	35374	26742	26479	26056	34315	25365	25019	24905	35851	28202	27445	26800	342553
coreutils	2520	15575	10361	10873	12027	15355	10330	10505	11592	15438	10993	11527	13520	148096
findutils	144	1762	1190	1190	1305	1757	1165	1165	1282	1758	1234	1230	1282	16418
Total	3000	52711	38293	38542	39388	51427	36860	36686	37779	53047	40429	40202	41703	507067
Evaluation Set														
Software	Bin.	Functions												Total
		x86				ARM				MIPS				
		O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	
openssl	12	6085	5601	5607	5425	6024	5545	5547	5362	5995	5517	5519	5334	67561
busybox	12	4241	2938	2709	2757	4253	2942	2715	2763	4250	2941	2711	2759	37979
sqlite	12	2190	1523	1380	1183	2285	1616	1475	1280	2199	1531	1392	1197	19251
zlib	12	152	137	132	119	158	143	138	125	154	139	135	121	1653
gmp	12	790	701	698	673	781	691	688	662	769	678	675	650	8456
curl	12	1006	736	734	662	1010	742	736	668	1006	737	733	665	9435
putty	60	11356	7893	7865	7424	11288	7863	7858	7419	11266	7846	7838	7398	103314
Imagemagick	12	4455	2370	2385	2308	4365	2367	2378	2308	4261	2268	2278	2209	33952
Total	144	30275	21899	21510	20551	30164	21909	21535	20587	29900	21657	21281	20333	281601

6.2 Evaluation Setting

6.2.1 Datasets. To evaluate our approach, we collected eleven widely-used open source software projects, *i.e.,* coreutils-8.29, binutils-2.3.0, finduitls-4.6.0, curl-7.71.1, gmp-6.2.1, ImageMagick-7.0.10, openssl-1.0.1f, busybox-1.32, putty-0.76, zlib-1.2.11 and sqlite-3.34.0. To make the evaluation generalized and diverse, the first three projects were compiled by gcc-8.2.0 and clang-7.0 as the training set, and the rest projects were compiled by gcc-7.5.0 as the evaluation set. Moreover, all software projects were compiled with four optimization levels (*i.e.,* O0, O1, O2, O3) and for three CPU architectures (*i.e.,* x86, ARM and MIPS). In this way, for each executable file in each software project, we obtain 24 different binaries in the training set and 12 different binaries in the evaluation set respectively.

We filtered out those small functions that have less than 5 basic blocks or assembly instructions like [12, 13] as these functions contain little semantic information and graph structural information. Finally, we have 507,067 functions for training and 281,061 functions for evaluation, as Table 2 shows. The work and baselines are trained on the training set and evaluated on the evaluation set.

6.2.2 Baselines. We compare our approach with three state-of-the-art approaches, *i.e.,* Gemini, SAFE and FIT, because they are also deep learning based approaches and open source.

Gemini [35] uses a modified graph embedding network structure2vec to compute the embeddings of attribute CFGs of functions, in which each basic block is represented by eight features selected manually.

SAFE [28] utilizes word2vec and bi-RNN with a self-attention mechanism to generate function embeddings. It only uses assembly instructions as input without considering CFGs of functions.

FIT [24] considers the function level statistical features, the basic block level statistical features and the assembly instructions. It utilizes word2vec and structure2vec to generate function embeddings.

6.2.3 Metrics. In order to evaluate the performance of each approach, we use the following metrics:

Precision. $Precision = TP/(TP + FP)$, where the True Positive (TP) represents the number of correctly identified similar function pairs, the False Positive (FP) refers to the number of dissimilar function pairs which are wrongly identified as similar.

Recall. $Recall = TP/(TP + FN)$, where the False Negative (FN) refers to the number of similar function pairs which are wrongly identified as dissimilar.

F1. $F1 = 2 * (Precision * Recall)/(Precision + Recall)$. It is a weighted average of Precision and Recall and takes value in [0,1].

AUC. The area under the curve (AUC) is calculated on the receiver operating characteristics (ROC), which measures the probability that similar pairs are assigned higher similarity scores than dissimilar pairs. It does not require decision thresholds like precision, recall, or F1 measures, and hence is a better metric to evaluate learning models.

Inference time. The time to compute the similarity for 1000 functions. Note that it doesn't include the time to disassemble binaries, which is same for all methods.

In the evaluation, we calculate the precision, recall and F1 under the best threshold obtained from the ROC.

6.2.4 Parameter Setting. For most of the parameters, we follow the recommendations from the literature [9, 33], we set a BERT with hidden_dim=512, attention_head=8, hidden_layer=4, max_length=512 for semantic modeling, and a GAT with hidden_dim=512, layer=2, head=8 for graph structure modeling, for the sake of efficiency and training costs. Moreover, we used a trial-and-error procedure to adjust output dimension of each module in MFEN as 512 (thus the total one of MFEN is 1536) and the temperature parameter in similarity learning network as 0.05. We use the Adam optimizer and set learning rate 1×10^{-5} for similarity learning to avoid over-fitting.

6.3 Cross compilation optimization level

In this experiment, we measure the performance of MFEN-Sim and baselines under cross compilation optimization levels (O0-O3) on the evaluation dataset. The function pairs in the evaluation dataset come from different optimization levels but have the same architectures. The evaluation results are shown in Table 3.

Experimental results show that the features and deep learning models MFEN-Sim adopts are more robust against compiler optimizations than Gemini, FIT and SAFE on untrained projects. On average, MFEN-Sim outperforms Gemini by 4.0%, FIT by 5.2% and SAFE by on 3.2% on AUC respectively, Gemini by 5.5%, FIT by 6.9% and SAFE by 4.2% on F1 respectively. The reasons behind is that compared with these baseline tools, MFEN-Sim can extract more features (*i.e.,* function signatures and function code literal) and has better learning ability, *i.e.,* the pre-trained BERT and CodeBert in MFEN-Sim excel at capturing long-distance dependency in basic block sequences and learn the proper representation of function.

6.4 Cross CPU architecture

In this section, we evaluate the performance of MFEN-Sim and baselines to detect similarity in binaries with different architectures (*i.e.,* x86, ARM, MIPS). The function pairs in the evaluation dataset come from different architectures but have the same optimization levels.

As Table 4 shows, MFEN-Sim outperforms Gemini, FIT and SAFE on all projects in cross-architecture cases. On average, MFEN-Sim outperforms Gemini by 5.5% on AUC, by 5.0% on F1, outperforms FIT by 11.3% on AUC, by 14.4% on F1 and outperforms SAFE by 6.8% on AUC, by 9.2% on F1.

The evaluation results show that MFEN-Sim's pre-trained BERT can approximately learn function semantics explicitly and generalize to match semantically similar functions with different

Table 3. Evaluation results on function pairs across optimization levels (O0-O3)

Software	Tool	AUC	Precision	Recall	F1
openssl	Gemini	0.926	0.886	0.863	0.874
	FIT	0.921	0.876	0.845	0.860
	SAFE	0.933	0.887	0.867	0.877
	MFEN-Sim	**0.968**	**0.925**	**0.918**	**0.921**
busybox	Gemini	0.952	0.896	0.878	0.887
	FIT	0.947	0.897	0.870	0.884
	SAFE	0.940	0.877	0.850	0.863
	MFEN-Sim	0.976	0.925	0.921	0.923
sqlite	Gemini	0.891	0.818	0.801	0.809
	FIT	0.899	0.833	0.811	0.822
	SAFE	0.913	0.861	0.836	0.848
	MFEN-Sim	**0.965**	**0.912**	0.898	**0.905**
zlib	Gemini	0.905	0.847	0.818	0.832
	FIT	0.876	0.813	0.788	0.800
	SAFE	0.944	0.897	0.887	0.892
	MFEN-Sim	**0.952**	**0.911**	**0.898**	**0.917**
gmp	Gemini	0.942	0.890	0.857	0.873
	FIT	0.925	0.868	0.818	0.842
	SAFE	0.963	0.913	0.895	0.904
	MFEN-Sim	**0.972**	**0.922**	**0.911**	**0.917**
curl	Gemini	0.923	0.859	0.846	0.853
	FIT	0.917	0.859	0.831	0.845
	SAFE	0.935	0.882	0.865	0.873
	MFEN-Sim	**0.978**	**0.935**	**0.924**	**0.930**
putty	Gemini	0.936	0.886	0.857	0.872
	FIT	0.932	0.888	0.850	0.868
	SAFE	0.920	0.869	0.847	0.858
	MFEN-Sim	**0.964**	**0.914**	**0.902**	**0.908**
ImageMagick	Gemini	0.955	0.901	0.888	0.894
	FIT	0.920	0.872	0.849	0.860
	SAFE	0.952	0.898	0.874	0.886
	MFEN-Sim	**0.979**	**0.935**	**0.922**	**0.929**
Average	Gemini	0.929	0.873	0.851	0.862
	FIT	0.917	0.863	0.833	0.848
	SAFE	0.937	0.885	0.865	0.875
	MFEN-Sim	**0.969**	**0.923**	**0.912**	**0.917**

architectures, while Gemini, SAFE and FIT may lose part of instruction semantics and dependency information. So MFEN-Sim is robust against different CPU architectures in untrained projects.

Table 4. Evaluation results on function pairs across architecture (x86, ARM, MIPS)

Software	Tool	AUC	Precision	Recall	F1
openssl	Gemini	0.947	0.898	0.867	0.883
	FIT	0.910	0.850	0.816	0.833
	SAFE	0.946	0.888	0.864	0.876
	MFEN-Sim	**0.987**	**0.951**	**0.944**	**0.948**
busybox	Gemini	0.939	0.884	0.860	0.872
	FIT	0.841	0.786	0.749	0.767
	SAFE	0.870	0.798	0.781	0.790
	MFEN-Sim	**0.989**	**0.953**	**0.942**	**0.947**
sqlite	Gemini	0.947	0.889	0.862	0.875
	FIT	0.922	0.847	0.822	0.834
	SAFE	0.900	0.845	0.818	0.831
	MFEN-Sim	**0.989**	**0.957**	**0.948**	**0.952**
zlib	Gemini	0.938	0.867	0.823	0.845
	FIT	0.909	0.845	0.822	0.834
	SAFE	0.954	0.902	0.881	0.891
	MFEN-Sim	**0.973**	**0.915**	**0.896**	**0.905**
gmp	Gemini	0.821	0.771	0.760	0.765
	FIT	0.722	0.668	0.623	0.645
	SAFE	0.885	0.818	0.797	0.808
	MFEN-Sim	**0.949**	**0.889**	**0.875**	**0.882**
curl	Gemini	0.932	0.870	0.845	0.857
	FIT	0.906	0.833	0.804	0.819
	SAFE	0.941	0.883	0.865	0.874
	MFEN-Sim	**0.988**	**0.955**	**0.943**	**0.949**
putty	Gemini	0.955	0.903	0.862	0.882
	FIT	0.932	0.888	0.850	0.868
	SAFE	0.907	0.841	0.827	0.834
	MFEN-Sim	**0.990**	**0.959**	**0.948**	**0.953**
ImageMagick	Gemini	0.915	0.853	0.841	0.847
	FIT	0.845	0.777	0.765	0.771
	SAFE	0.894	0.832	0.809	0.834
	MFEN-Sim	**0.975**	**0.928**	**0.913**	**0.921**
Average	Gemini	0.925	0.866	0.839	0.852
	FIT	0.867	0.804	0.772	0.788
	SAFE	0.912	0.851	0.830	0.840
	MFEN-Sim	**0.980**	**0.938**	**0.926**	**0.932**

6.5 Cross compilation optimization level and Cross CPU architecture

Functions may be compiled with any compilation optimization level and on any architecture. In this experiment, we evaluate the performance of MFEN-Sim and baselines under different compilation optimizations and architectures on the evaluation dataset. The function pairs in the evaluation

Table 5. Evaluation results on function pairs across architecture (x86, ARM, MIPS) and optimization levels (O0-O3)

Software	Tool	AUC	Precision	Recall	F1
openssl	Gemini	0.914	0.852	0.810	0.831
	FIT	0.905	0.863	0.786	0.823
	SAFE	0.902	0.843	0.825	0.834
	MFEN-Sim	**0.962**	**0.913**	**0.900**	**0.906**
busybox	Gemini	0.908	0.853	0.825	0.839
	FIT	0.853	0.804	0.757	0.780
	SAFE	0.851	0.781	0.765	0.773
	MFEN-Sim	**0.969**	**0.919**	**0.904**	**0.911**
sqlite	Gemini	0.885	0.829	0.799	0.814
	FIT	0.887	0.831	0.791	0.811
	SAFE	0.850	0.784	0.765	0.774
	MFEN-Sim	**0.957**	**0.898**	**0.885**	**0.892**
zlib	Gemini	0.882	0.838	0.810	0.824
	FIT	0.876	0.824	0.782	0.803
	SAFE	0.912	0.865	0.846	0.856
	MFEN-Sim	**0.952**	**0.901**	**0.888**	**0.894**
gmp	Gemini	0.764	0.719	0.694	0.706
	FIT	0.742	0.697	0.642	0.668
	SAFE	0.840	0.776	0.757	0.766
	MFEN-Sim	**0.931**	**0.869**	**0.853**	**0.861**
curl	Gemini	0.908	0.853	0.826	0.839
	FIT	0.895	0.842	0.799	0.820
	SAFE	0.899	0.835	0.818	0.827
	MFEN-Sim	**0.969**	**0.922**	**0.906**	**0.914**
putty	Gemini	0.916	0.863	0.810	0.836
	FIT	0.826	0.774	0.722	0.747
	SAFE	0.845	0.778	0.763	0.770
	MFEN-Sim	**0.958**	**0.905**	**0.891**	**0.898**
ImageMagick	Gemini	0.911	0.859	0.830	0.844
	FIT	0.875	0.833	0.792	0.812
	SAFE	0.902	0.838	0.825	0.831
	MFEN-Sim	**0.959**	**0.899**	**0.886**	**0.892**
Average	Gemini	0.886	0.835	0.804	0.819
	FIT	0.858	0.807	0.762	0.784
	SAFE	0.875	0.813	0.795	0.804
	MFEN-Sim	**0.957**	**0.903**	**0.889**	**0.896**

dataset come from both different optimization levels (*i.e.,* O0, O1, O2, O3) and different architectures (*i.e.,* x86, ARM, MIPS). Table 5 shows the evaluation results. MFEN-Sim still outperforms Gemini, FIT and SAFE on all projects. On average, MFEN-Sim is better than Gemini by 7.1% on AUC, by

Table 6. Ablation study results

	AUC	Precision	Recall	F1	Inference time
Gemini	0.886	0.835	0.804	0.819	3.8s
FIT	0.858	0.807	0.762	0.784	4.3s
SAFE	0.875	0.813	0.795	0.804	2.9s
MFEN-pl-ss	0.919	0.850	0.847	0.848	4.7s
MFEN-pl-ss-cl	0.928	0.856	0.852	0.854	5.8s
MFEN-pl-ss-sp	0.935	0.863	0.857	0.860	6.3s
MFEN-pl-all	0.941	0.873	0.866	0.870	7.2s
MFEN-Sim	**0.957**	**0.903**	**0.889**	**0.896**	**7.8s**

7.7% on F1, better than FIT by 9.9% on AUC, by 11.2% on F1, and better than SAFE by 8.2% on AUC, by 9.2% on F1. The evaluation results indict that MFEN-Sim is robust against different CPU architectures and different optimization levels in untrained projects.

6.6 Ablation study

In this section, we evaluate the contribution of each important component in MFEN-Sim for binary code similarity detection. We conducted experiments in the case of cross-optimization level and cross-architecture using the same experimental setting as in section 6.2. We compare MFEN-Sim with its following variants.

MFEN-pl-all. We discard the similarity learning network and train MFEN with metric learning algorithm like Gemini, FIT and SAFE. For each binary function, we randomly select a positive function compiled from the same source code and a negative function compiled from different source code. The pair loss function is:

$$l = \sum_{i=1}^{i=K} \left(cos(f_i, f_i^{'}) - y_i\right)^2 \tag{17}$$

where y_i is the label indicating similar (+1) or dissimilar (-1), and $f_i^{'}$ is the selected binary function for f_i to form a function pair.

MFEN-pl-ss. We only leverage the function semantic and structure embedding as the final function embedding and train the model with the metric learning algorithm.

MFEN-pl-ss-cl. We use the function semantic and structure embedding and the function code literal embedding to generate the final function embedding, and train the model with the metric learning algorithm.

MFEN-pl-ss-sp. We exploit the function semantic and structure embedding and the signature prediction embedding to generate the final function embedding, and train the model with the metric learning algorithm.

As shown in Table 6, the function signature prediction embedding module can improve 1.6% on AUC, 1.2% on F1, while the function code literal embedding module can improve 0.9% on AUC, 0.6% on F1. With both modules, the performance is improved 2.2% on AUC and 2.2% on F1, because code literals and function signatures are important features for representing function semantics, and extracted by the BERT and CodeBert models in MFEN-Sim. Furthermore, the similarity learning network improves 1.7% on AUC, 2.6% on F1, which is better than that of training with the metric learning algorithm, because by using contrastive learning MFEN-Sim considers both positive and negative pairs to learn more function relations and reduce the probability of overfitting.

Table 7. CVEs in the first dataset

CVE-ID	Description	Target Function	#Vul. Fun.
2014-0160	data leak	dtls1_process_heartbeat	12
2014-3508	data leak	OBJ_obj2txt	12
2014-3566	data leak	ssl_cipher_list_to_bytes	12
2014-3572	data leak	ssl3_get_key_exchange	12
2015-0292	DoS	EVP_DecodeUpdate	12
2016-0705	DoS	dsa_priv_decode	12
2016-0798	DoS	SRP_VBASE_get_by_user	12
2016-2182	DoS	BN_bn2dec	12
2016-2842	DoS	doapr_outch	12
2016-6303	DoS	MDC2_Update	12

6.7 Efficiency

In this section, we evaluate the efficiency of MFEN-sim and other baselines under different compilation optimizations and architectures. We recorded inference time for each approach and reported the average values in Table 6. SAFE is the fastest among all the approaches, Gemini and FIT use a modified graph embedding network structure2vec to compute the embeddings of attribute CFGs of functions which incurs a little overhead. MFEN-Sim and its variants have larger inference time as they extract function features using BERT and CodeBert. The more components MFEN-sim has, the better its effectiveness, while the larger its inference time. In a word, there is a trade-off between the effectiveness and efficiency for MFEN-sim.

7 VULNERABILITY SEARCH

In this section, we apply MFEN-Sim to vulnerability search to evaluate its practicality in finding the reuse of vulnerable functions in two datasets. Given a query function q, vulnerability search is to find those similar ones in target functions. In other words, we treat it as a rank problem instead of only giving the similarity score between q and another function, thus we use $recall@k$ to evaluate each approach's performance:

$$recall@k = \frac{TP@k}{N} \tag{18}$$

where k represents first k candidate functions, $TP@k$ is the number of the target functions in k candidate functions, N is the total number of target functions. So a higher $recall@k$ means that an approach has a better performance.

The first dataset is built on the openssl-1.0.1f in Table 2, which contains total 67,561 functions. Ten CVEs in openssl-1.0.1f are selected and each CVE corresponds to 12 vulnerable functions due to our 12 compilation combinations, as Table7 shows. For each vulnerable function, we try to find similar functions from all functions in openssl-1.0.1f and calculate $recall@k$. Finally, we average the $recall@k$ of all vulnerable functions to measure the performance of MFEN-Sim and baselines. The maximum value of k is set to 200 as in SAFE [28].

In Fig. 7, the results show that MFEN-Sim outperforms Gemini, FIT and SAFE. For k=50, MFEN-Sim achieves 90.8% recall, while Gemini gets 42.6% recall, FIT gets 42.4% recall and SAFE gets 18.6% recall. For k=200, MFEN-Sim reaches the recall of 96.3%, while the recall of Gemini, FIT and SAFE is 58.3%, 58.9% and 31.6% respectively. The results demonstrate that MFEN-Sim can effectively

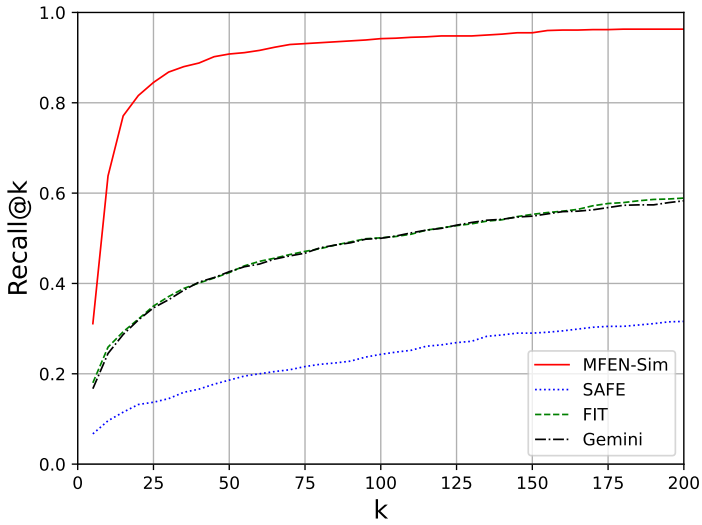

Fig. 7. Recall@k on OpenSSL dataset

Table 8. CVEs in the second dataset [5]. DoS: deny of service; ACE: arbitrary code execution

CVE-ID	Software	Description	Target Function	Vul. Fun.
2011-0444	wireshark	DoS, ACE	snmp_usm_password_to_key_sha1	7
2014-0160	openssl	data leak	dtls1_process_heartbeat	13
2014-4877	wget	ACE	ftp_retrieve_glob	3
2014-6271	bash	ACE	initialize_shell_variables	6
2014-7169	bash	ACE	initialize_shell_variables	3
2014-9295	ntp	ACE	configure	7
2015-3456	QEMU	DoS, ACE	fdtrl_handle_drive_specification_command	6
2015-6826	ffmpeg	DoS	ff_rv34_decode_init_thread_copy	7

search vulnerable functions with cross architectures and optimization levels in a large number of untrained functions.

The second dataset is a public vulnerability dataset presented in [5] This dataset contains seven open source software with different versions, including coreutils, openssl, bash, ntpd, QEMU, wireshark and wget. These software are compiled with 11 compilers of clang, gcc, icc families on x86 architecture. After feature extraction, we get 3,005 functions from this dataset. This dataset contains 8 CVEs, as Table 8 shows. Each CVE corresponds to at least 3 and at most 13 vulnerable functions in the dataset. The average number of vulnerable functions for all CVEs is 7. In fact, different versions of a software project are often compiled with default optimization options, and different software projects may use different compilers, so this dataset is close to a real world scenario of binary software analysis. We use the same settings and metrics as the experiment on the first dataset.

Fig. 8 shows the experimental results. MFEN-sim also achieves the best result among four approaches on this dataset. For k=50, MFEN-Sim achieves 75.8% recall, while the recall of Gemini, FIT and SAFE is 45.5%, 47.4% and 62.1% respectively. In the case of k=200, MFEN-Sim, Gemini,

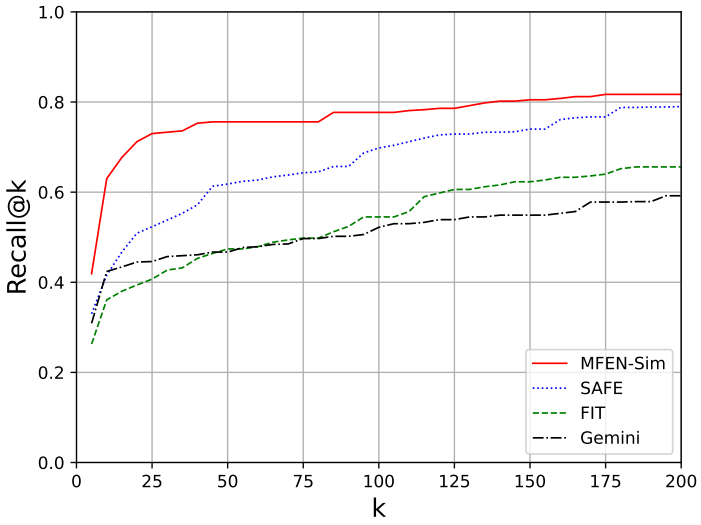

Fig. 8. Recall@k on the vulnerability dataset [5]

FIT and SAFE achieves 81.7%, 55.9%, 65.6% and 79.0% recall respectively. The experimental results show that MFEN-Sim can retrieve the vulnerable functions in real-world software. Obviously SAFE performs better on this dataset than on the first dataset. We think that the reason may be that SAFE only utilize a function's assembly instructions as its input and the differences in assembly instructions introduced by different compilers are less than that brought by different CPU architectures and compilation optimizations.

Although MFEN-Sim outperforms all baselines on two datasets, for k=10, MFEN-Sim achieves 73.5% and 63.4% recall on two datasets respectively, and for k=200, MFEN-Sim doesn't reach 100% recall. That's probably because of the limited size of the training set, and the differences between the vulnerability set and the training set (*e.g.*, different software, different compilers), MFEN-Sim may not learn the similarity relations between some functions in the vulnerability set, and assigns relatively low similarity scores to these similar functions. This may be improved by further training on the vulnerability set, which we leave for future work.

8 LIMITATIONS

MFEN-Sim has limitations. First, it cannot cope well with the obfuscated binaries because obfuscation techniques bring significant changes in binary code. Thus it's better to leverage anti-obfuscation techniques before applying our approach. Second, MFEN-Sim has a complicated architecture, causing more time cost. For example, it takes about 7.8s for generating 1000 function embeddings, while SAFE takes about 2.9s. However, MFEN-Sim is flexible and its models can be modified or replaced to balance effectiveness and efficiency. In addition, MFEN-Sim only considers internal features of a function as inputs, considering inter-function features like function call is a future work.

9 RELATED WORK

We discuss the related research efforts by classifying them into three groups: binary code similarity detection based on static analysis, dynamic analysis and deep learning.

9.1 Static analysis based methods

Static approaches directly extract code features from binaries, usually syntactic features and structural features. For example, Rendezvous [21] builds a statistical model comprising N-grams and N-perms of instruction mnemonics, control flow sub-graphs and data constants to match functions. Tracy [8] extracts tracelets from function control flow graphs to capture the function execution semantics. TEDEM [31] matches the expression tree of basic blocks. Bindiff [40] adopts expensive graph isomorphism algorithm to match the control flow graphs (CFG) of functions, which is not suitable for large scale search. To perform the cross-architecture detection, some studies [12, 13, 25, 38] utilize across-architecture numeric features and construct attribute control flow graphs (ACFG). DiscovRe [12] first selects statistical features such as number of instructions, number of constants from functions and utilizes these features to filter out candidate functions based on KNN, then applies a graph isomorphism algorithm to match functions. Lin et al. [25] and Zhao et al. [38] further extract function-level features and apply SVM in the pre-filtering stage to improve the performance. Considering the cost of the graph isomorphism algorithm, Genius [13] first generates codebook from ACFGs, and encodes ACFGs into vectors based on the codebook to measure the similarity between functions. TikNib [22] utilizes statistical features in CFGs and CGs, and leverages a greedy feature selection algorithm to select appropriate statistical features for binary code similarity analysis. Some studies also utilize data flow features. Gitz [6] lifts assembly instructions to intermediate representation and compares functions based on data-flow slices. Frimup [7] also uses data-flow slices and a game algorithm to match functions. Static approaches can achieve high scalability and relative high accuracy. However, the features utilized by these approaches are easily influenced by different CPU architectures, compilers and optimizations. MFEN-Sim is based on deep learning, which is an end-to-end methods, thus don't need the manual effort to select and extract features. Furthermore, MFEN can be re-trained for different purposes.

9.2 Dynamic analysis based methods

Dynamic approaches usually extract robust features based on emulations executions, such as input/output pairs, values written to stack/heap or system calls. For example, BinGo [2] utilizes a selective-inlining technique and partial traces extracted from CFGs for cross-architecture and cross-OS function search. CoP [27] utilizes symbolic execution to check whether two basic blocks have equivalent semantics, and uses the longest common subsequence algorithm to calculate the similarity between basic block paths. Esh [5] leverages data-flow slices called strands from basic blocks and utilizes symbolic execution to calculate the similarity between strands. Multi-MH [30] builds assign formulas for each basic block and uses concrete input values to capture the I/O pairs of basic blocks. IMF-SIM [34] utilizes in-memory fuzzing to obtain the execution traces of functions from which they extract feature vectors and feed them into a machine learning model to predict whether functions are similar. Dynamic approaches can detect semantically similar functions and have some ability to resist the influence of optimizations and obfuscations. However, they are based on execution or emulation techniques and hence introduce considerable performance cost.

9.3 Deep learning based methods

Some approaches use neutral networks to search similar binary code on single-architecture and multi-architecture. These approaches learn a binary code representation that is supposed to encode the binary code's syntax and semantics into low dimensional vectors, thus they can compute the similarity through these embeddings. They usually learn a model that takes the structural information (like a function's CFG) or instruction sequences to construct embeddings and train the model so that the similar binary code's embeddings are closer.

Gemini [35] is the first approach to combine the CFG with the graph embedding network, it treats ACFGs as input and utilizes structure2vec with Siamese network to generate function embeddings. Vulseeker [15] adds data dependency information to ACFGs to learn data flow information of functions. Alpha-diff [26] utilizes CNN to learn directly from the images generated by original function bytes. These methods use raw information or manual-selected features from binary codes as the model's input, which needs expert efforts.

Some studies apply NLP techniques to binary code similarity detection. Asm2vec [10] trains a Paragraph Vector-Distributed Memory model to generate function embeddings. Innereye [39] uses word2vec and LSTM to compare the similarity of basic blocks. DeepBinDiff [11] uses word2vec for semantic information and text-associated DeepWalk algorithm for program-wide contextual information, it works at basic block granularity while MFEN-Sim performs at function granularity. SAFE [28] generates function embedding with bi-RNN and a self-attention mechanism. FIT [24] utilizes word2vec and LSTM to generate the basic block embedding, then utilizes graph embedding network to generate function embeddings. Yu et al. [37] build semantic-aware modeling, structural-aware modeling and order-aware modeling with BERT, MPNN and Resnet, respectively. Koo et al. [23] propose a well-balanced instruction normalization to minimize the out-of-vocabulary problem and hold semantic information in assembly instructions, and utilize BERT model to generate the semantic-aware embedding for downstream tasks like binary similarity comparison.

Most of these works are based on semantic features like assembly instructions and structure features like CFGs, but they ignored the important semantic information in code literals and function signatures which are used in our approach. Moreover, MFEN-Sim uses a similarity learning network based on contrastive learning to learn more relations between functions than prior methods.

10 CONCLUSION

In this paper, we propose a novel framework MFEN-Sim for binary code similarity detection, which contains three stages: feature extraction and normalization, multi-feature based function embedding network and similarity learning network. Besides the semantic feature like assembly instructions and the structure feature like CFGs, MFEN-Sim also extracts function code literal features and function signature features. To capture these features, we propose the function semantic embedding module based on BERT and GAT, the function signature prediction module based on BERT and classification heads, the function code literal embedding module based on CodeBert. Furthermore, a similarity learning network based on the contrastive learning is used to capture more similarity relations between functions than prior methods. The evaluation results on eight open source projects demonstrate that MFEN-Sim outperforms three state-of-the-art methods (*i.e.,* Gemini, FIT and SAFE) in binary similarity detection, and can achieve higher recall than baselines in the case of searching vulnerabilities.

MFEN in MFEN-Sim can be used as a general function encoder. It can be pretrained and fine-tuned for different downstream tasks, such as vulnerability classification, malware family classification, and bug patch detection.

CONFLICT OF INTEREST

The authors declare that they have no conflict of interest.

REFERENCES

[1] 2014. The heartbleed bug, https://heartbleed.com/.

[2] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.

[3] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[4] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 99–116.

[5] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[6] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 79–94.

[7] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices* 53, 2 (2018), 392–404.

[8] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[11] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*.

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*, Vol. 52. 58–79.

[13] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 896–899.

[16] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *arXiv preprint arXiv:2104.08821* (2021).

[17] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. 2019. A semantics-based hybrid approach on binary code similarity comparison. *IEEE Transactions on Software Engineering* (2019).

[18] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 88–98.

[19] IDA. [n. d.]. https://hex-rays.com/.

[20] IDAPython. [n. d.]. https://github.com/idapython/src.

[21] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 329–338.

[22] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *arXiv preprint arXiv:2011.10749* (2020).

[23] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. 2021. Semantic-aware Binary Code Representation with BERT. *arXiv preprint arXiv:2106.05478* (2021).

[24] Hongliang Liang, Zhuosi Xie, Yixiu Chen, Hua Ning, and Jianli Wang. 2020. FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. *Computers & Security* 99 (2020), 102032.

[25] Hong Lin, Dongdong Zhao, Linjun Ran, Mushuai Han, Jing Tian, Jianwen Xiang, Xian Ma, and Yingshou Zhong. 2017. Cvssa: Cross-architecture vulnerability search in firmware based on support vector machine and attributed control flow graph. In *2017 International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 35–41.

[26] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 667–678.

[27] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 389–400.

[28] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329.

[29] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 253–270.

[30] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724.

[31] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 406–415.

[32] PyTorch. [n. d.]. https://pytorch.org/.

[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[34] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.

[35] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. 363–376.

[36] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149.

[37] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

[38] Dongdong Zhao, Hong Lin, Linjun Ran, Mushuai Han, Jing Tian, Liping Lu, Shengwu Xiong, and Jianwen Xiang. 2019. CVSkSA: cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. *Software Quality Journal* 27, 3 (2019), 1045–1068.

[39] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).

[40] Zynamics. 2019. *Bindiff, https://www.zynamics.com/bindiff.html.*