electronics

Article

AAPFE: Aligned Assembly Pre-Training Function Embedding for Malware Analysis

Hairen Gui, Ke Tang, Zheng Shan, Meng Qiao, Chunyan Zhang, Yizhao Huang and Fudong Liu *

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China; guihairen@163.com (H.G.); tuck3r@foxmail.com (K.T.); zzzhengming@163.com (Z.S.); kopo.code@gmail.com (M.Q.); iecyzhang@163.com (C.Z.); huangyizhao1996@163.com (Y.H.)
* Correspondence: lwfydy@126.com

Abstract: The use of natural language processing to analyze binary data is a popular research topic in malware analysis. Embedding binary code into a vector is an important basis for building a binary analysis neural network model. Current solutions focus on embedding instructions or basic block sequences into vectors with recurrent neural network models or utilizing a graph algorithm on control flow graphs or annotated control flow graphs to generate binary representation vectors. In malware analysis, most of these studies only focus on the single structural information of the binary and rely on one corpus. It is difficult for vectors to effectively represent the semantics and functionality of binary code. Therefore, this study proposes aligned assembly pre-training function embedding, a function embedding scheme based on a pre-training aligned assembly. The scheme creatively applies data augmentation and a triplet network structure to the embedding model training. Each sub-network extracts instruction sequence information using the self-attention mechanism and basic block graph structure information with the graph convolution network model. An embedding model is pre-trained with the produced aligned assembly triplet function dataset and is subsequently evaluated against a series of comparative experiments and application evaluations. The results show that the model is superior to the state-of-the-art methods in terms of precision, precision ranking at top N (p@N), and the area under the curve, verifying the effectiveness of the aligned assembly pre-training and multi-level information extraction methods.

Keywords: malware analysis; function embedding; aligned assembly; self-attention; graph convolution network

Citation: Gui, H.; Tang, K.; Shan, Z.; Qiao, M.; Zhang, C.; Huang, Y.; Liu, F. AAPFE: Aligned Assembly Pre-Training Function Embedding for Malware Analysis. *Electronics* **2022**, *11*, 940. https://doi.org/10.3390/electronics11060940

Academic Editor: Arkaitz Zubiaga

Received: 16 February 2022
Accepted: 14 March 2022
Published: 17 March 2022

1. Introduction

Malicious code of great variety and scale exists as binary programs. Binary program analysis technology is an important means of studying malware; however, to improve the efficiency of malware, technologies such as cross-referencing, automatic splicing, and polymorphic mutation are illegally applied on a large scale with malicious intent. Traditional manual analysis [1] and machine learning-based methods [2–4] have difficulty coping with the increasingly complex binary analysis requirements.

With the rapid development of deep learning, an increasing number of studies have focused on introducing natural language processing (NLP) into binary analysis. These studies use multi-layer neural networks to automatically extract the features of a certain binary code. Vectors containing these features are applied to various binary analysis tasks, such as binary search [5], similarity comparison [6–10], and code reuse detection [11,12]. They are also used in malware analysis tasks, such as vulnerability searches, malware classification [6], and anomaly detection in multiple IoT scenarios [13].

However, the application of NLP methods to binary analysis faces two key problems. First, embedding binary into numerical vectors with fixed dimensions that can be received by neural networks is difficult. These vectors should represent each binary in an

n-dimensional Cartesian coordinate space. Secondly, the quality of the embedded vector with a limited binary corpus must be maintained. The resulting binary vector should effectively represent the semantics of the binary code.

To solve the aforementioned problems, researchers have proposed several binary embedding approaches. Many NLP-based techniques [7,8] use instruction as input, generating representation vectors with the context information of the instruction sequence to extract the binary feature. However, these schemes neglect the relationships among basic blocks, resulting in a loss of structural information. Some researchers use graphs, such as control flow graphs (CFGs) [5] and inter-procedural CFGs [9], produced in the disassembly process to generate a binary representation vector. These graph methods use word embedding to obtain instruction vectors, mining the little knowledge inside basic blocks. In addition, the processing cost of the graph model is high. Other studies [11,12] have combined representation methods with multiple granularities and used different types of information at different levels; however, these end-to-end feature extraction methods have a significant correlation with specific applications and are not generalizable. To maintain the availability of datasets for the feature extraction model, most schemes fail to fully utilize the binary corpus. Moreover, partial corpus information is lost during disassembly, causing the representation capability of the resulting vectors to be relatively low.

To overcome these problems, this study draws on the data augmentation mechanism [14,15] commonly used in the NLP field to propose an augmentation method for a binary corpus. Assembly files derived from the same source code in different ways, called homologous assembly, should be equivalent in semantics and functionality. When the assembly is embedded into the same dimensional space, the distance between the vectors of the homologous assembly should be closer than that of the non-homologous assembly. Because the function name is preserved during disassembly, certain segments of the homologous assembly can be aligned by the function name. In this paper, such segments are defined as aligned assemblies and are described in detail in Section 3.2.1. Therefore, in this study, the assembly function was selected as the binary embedding granularity.

Meanwhile, to improve the representation capability, a triple-network-based structure was designed to train the function embedding model through a binary distance comparison task. Inspired by pre-training in deep learning, pre-training is used in augmented datasets to advance the generality of different malware analysis applications. The vector embedded by the obtained model can provide better performance with fine-tuning according to specific tasks further downstream.

The major contributions of this work are as follows:

- A data augmentation method with aligned assembly functions to generate a triple dataset aligned assembly triplet function (AATF) for binary function embedding was proposed.
- A multi-level embedding network framework that can simultaneously capture sequence information at the instruction level and structure information at the block level was designed.
- Aligned assembly pre-training function embedding (AAPFE) was pre-trained based on the created triple dataset AATF and triplet loss function.
- Extensive experiments were conducted, and the results demonstrated that the proposed model outperformed the state-of-the-art models in terms of precision, p@N, and area under the curve.

The rest of this paper consists of four sections, which are organized as follows. In Section 2 we present related work. In Section 3 we describe the overall architecture of AAPFE and the internal mechanism of each component. Experiments and evaluations are presented in Section 4. Finally, a conclusion is given in Section 5.

2. Related Work

A binary contains not only the sequence information of its instruction but also the structural information of graphs, such as the control flow graph and data dependency graph. Therefore, we can classify binary embedding methods into the following three categories.

2.1. Sequence-Based Embedding Methods

Sequence-based embedding methods have been proposed in many studies, with most using raw binary programs as inputs for the embedding models. For example, Zhang et al. [16] adopted the Sequence2Sequence model MIRROR to transform the basic block into a fixed-dimensional vector, regardless of the architecture. However, the proposed method could only extract information within a basic block, losing the relationship among different basic blocks. Li et al. [17] proposed a pre-training-based assembly language embedding method, PalmTree, to capture the different features and learn an embedding model for instruction sequences. However, the pretraining-based method requires well-labeled function samples.

Some studies have used assembly code as the input. Steven et al. [5] proposed an assembly representation learning model, Asm2Vec, that uses the PV-DM model in NLP to extract and fuse the semantic information of the assembly code. However, this approach cannot be transferred to other applications. Massarelli et al. [6] proposed a general function embedding scheme, SAFE, to embed instructions and a bidirectional recurrent neural network (biRNN) to embed functions on the disassembled binary functions. It can simultaneously generate embeddings on a variety of architectures. However, their i2V-based instruction embedding scheme cannot extract behavioral function information. Li and Jin [18] proposed a simple function embedding model based on the word2Vec algorithm and biRNN, and verified this model without considering the graph information. However, this method cannot obtain enough control flow and semantic information feature for other applications.

Asm2Vec and SAFE directly use the NLP model to predict the sequence relationship of instructions or basic blocks, avoiding the calculation of complex graph structures. The optimization of this model is relatively simple; however, these sequence methods utilize instruction information to represent the binary assembly file through a one-dimensional linear relationship, whose graph structure information is lost.

2.2. Graph-Based Embedding Methods

Several studies have directly utilized basic block control flow graphs for binary representation. For example, Duan et al. [9] proposed a method of using the generated CFG graph to extract contextual information. This embedding method obtains the semantic feature of the instruction and embeds the two types of information as basic blocks to construct a comparison model to handle program-wide binary diffing. However, since the input granularity is program level, the model is highly dependent on data preprocessing. Feng et al. [19] utilized attribute control flow graphs (ACFGs) to embed a binary for firmware images. The model was shown to outperform others in terms of effectiveness and time consumption; however, it requires a large scale of dataset to guarantee the robustness of application.

These methods extract features through graphs, such as CFGs or ACFGs, that represent binary assembly files, extracting basic inter-block structural information and ignoring the instruction information within a block. Moreover, a model based on a purely graphical approach is typically heavier than a sequence-based model. To obtain sufficient performance advantages, large-scale datasets are often trained, which causes efficiency bottlenecks in some scenarios.

2.3. Embedding Methods Based on a Hybrid Structure

In order to gain the advantages of both sequence and graph methods, researchers have begun to adopt hybrid structure-based methods. Qiao et al. [11] designed an embedding

scheme of cross-architecture basic blocks. This method combines sequence and graph structure information to embed the basic blocks in the labeled dataset MISA; however, the model depends on a large number of preprocessed samples for training. Xu et al. [10] proposed a novel neural network (NN)-based approach called Gemini that computes the embedding based on the block sequence and CFG of every binary function. However, this model generates embeddings with cosine distance, which may result in a remarkably time-consuming process. Yu et al. [20] adopted the masked language model (MLM) and adjacency node prediction (ANP) tasks inspired by BERT [21] into a pre-trained token and block embedding to encode the semantic feature. A graph algorithm and convolutional neural network (CNN) models were adopted to extract the structure and order information of CFG; however, this representation method is computationally complex and thus requires considerable time consumption.

These methods are complex for researchers without substantial computing power or a large amount of data. In addition, they are not task-transfer models for other applications. The characteristics of the binary embedding approaches proposed in recent years are summarized in Table 1.

Table 1. Comparison among binary embedding methods. For Boolean columns: √—supported; ×—unsupported. Embedding method can be sequence based (S), graph based (G), or a hybrid structure (H). Input granularity and approach granularities can be instruction (I), basic block (BB), function (F), or program (P). Computation complexity can be high (H), moderate (M), or low (L).

Approach	Year	Embedding Method	Input Granularity	Approach Granularities	Semantic Capturable	Task Transferable	Large-Scale Dataset	Model Complexity	Computation Complexity
GENIUS [19]	2016	G	F	BB	×	×	√	√	H
GEMINI [10]	2017	H	F	BB, F	√	×	√	×	M
ASM2Vec [5]	2019	S	F	I	×	×	×	√	M
SAFE [6]	2019	S	F	I, BB	√	×	×	×	M
Li's [18]	2020	S	BB	I, BB	×	×	×	×	L
DEEPBINDIFF [9]	2020	G	P	BB, F	√	×	√	√	H
Qiao's [11]	2020	H	BB	I, BB	√	×	√	×	M
MIRROR [16]	2020	S	BB	I	√	×	√	√	M
Yu's [20]	2020	H	BB	I, B	√	√	√	×	H
PALMTREE [17]	2021	S	F	BB	√	√	√	√	H

3. Proposed Approach

This study proposes a function embedding model called AAPFE, which uses an assembly function as input to automatically construct an assembly representation network through deep learning, introducing as little human bias as possible. In downstream application scenarios, the binary can be embedded into function vectors with high representation abilities, and after fine-tuning, the malware analysis task can achieve a high accuracy rate. AAPFE takes aligned assembly functions as input and must learn the semantic and functionality information between different assembly files with high quality; the vector distance of homologous alignment functions is closer than that of other non-homologous functions. The vectors of the aligned functions Entity2 and aligned functions Entity7 derived from the libssl-1_1.dll program in OpenSSL are projected into the same three-dimensional vector space, as shown in Figure 1.

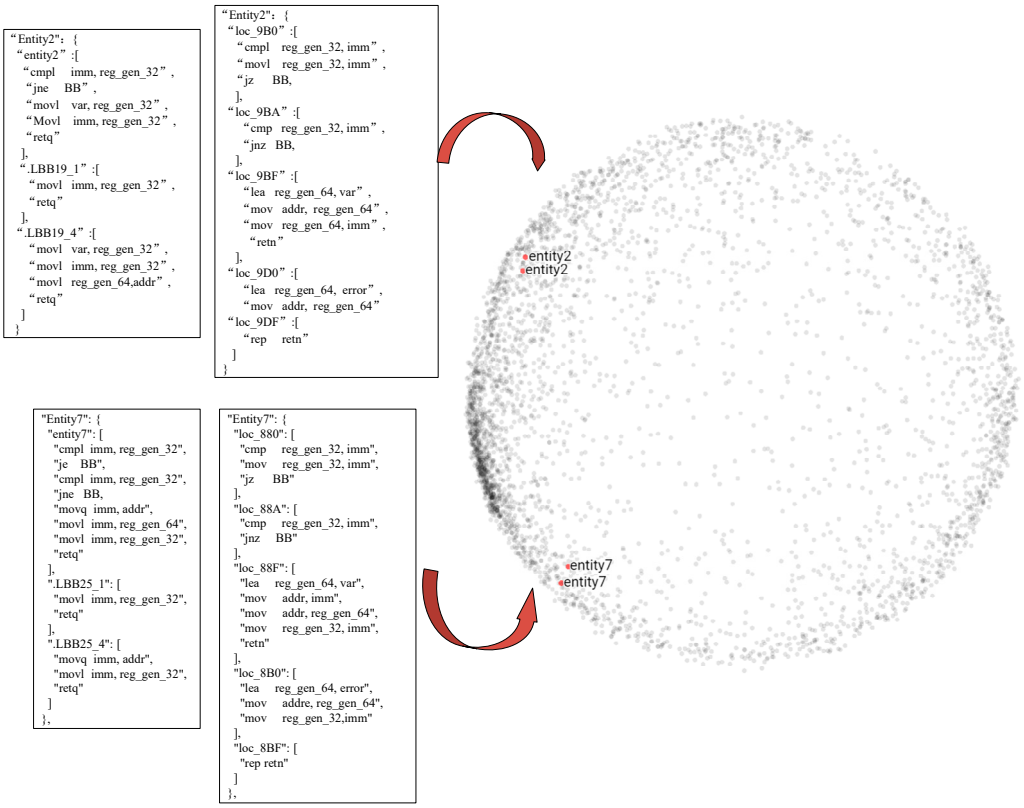


Figure 1. Assembly function embedding vector space T-SNE visual distribution. The classical functions Entity2 and Entity7 in the OpenSSL project were selected as examples. The function vectors of aligned assembly generated by homologous assembly should be close, whereas the function vectors of unaligned assembly should be far apart and distributed discretely.

3.1. Overview

AAPFE, whose overall structure is shown in Figure 2, is composed of three parts:

1. The first part is the aligned assembly generation. The anchor assembly ASM_a is generated by the compilation of a source code, and the positive assembly ASM_p is obtained by the disassembly of the binary file Bin. Bin is derived from the same code. ASM_a and ASM_p are homologous assemblies with the same functionality and semantics; however, their syntax and structure are different. Next, the name is used as the index for alignment and preprocessing to obtain the aligned assembly functions. Meanwhile, random sampling from ASM_n is adopted to obtain the negative functions to generate the assembly function triplet.
2. The second part involves the function embedding net, which accepts triplets. Using multi-granularity embedding fusion, these three embedding nets have the same structures. The instruction sequence and basic block jump relationship information is embedded as a real-valued vector representing the assembly function at the instruction level and the basic block levels. After summing and normalizing the function vectors, the vector representing the assembly function is output.
3. The final part is the learning target, which compares the output of the embedded vectors in pairs. The objective is to have the distance of the vector of the aligned assembly be lower than that of the unaligned assembly. The distance is used as part of the loss function for the gradient propagation. After training, the converged model parameters are obtained, and the AAPFE model is finally stored.

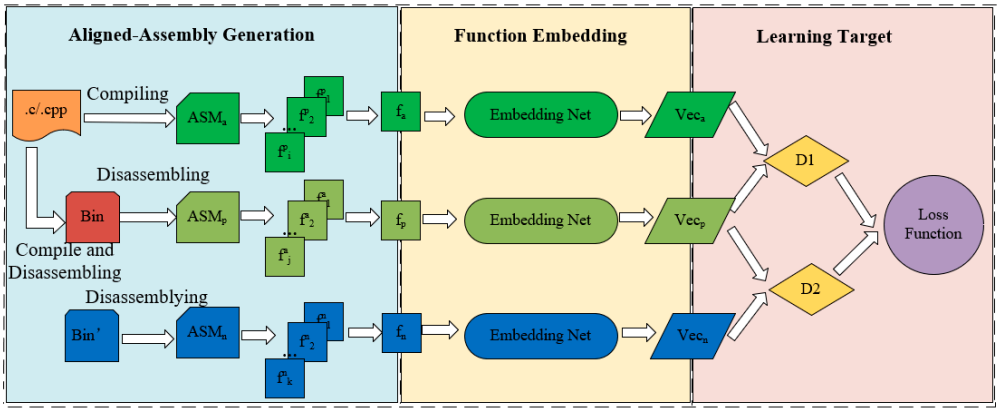


Figure 2. AAPFE overview.

AAPFE is a function embedding pre-training model based on similarity comparison tasks. The purpose is to learn an assembly embedding model to effectively extract the semantic and functionality features of malicious binary programs. This process involves converting binary program inputs into fixed-dimensional numerical vectors, thus providing a useful data source for malware analysis tasks using deep neural networks. Inspired by [22], the triplet network is employed as the overall training structure of AAPFE, and each embedding network in the triplet network is a sub-network of the same feed-forward structure with shared parameters.

The following sections introduce the various components of AAPFE based on the triple structure, including the aligned assembly generation, function embedding network, training network goals, and deployment methods.

3.2. Dataset Collection

3.2.1. Aligned Assembly Generation

The field of NLP digs deep into the corpus of datasets. For example, applications such as event detection based on a multilingual corpus [13,14] have achieved improvements. In malware analysis, the labeled binary corpus itself is relatively scarce, and some source program syntax and semantic information have been lost in the process of code lowering, such as compilation and linking. After preprocessing and optimization by the compiler, information in the form of arrays, structures, and enumerations in high-level coding languages no longer exists. Logical structures, such as loops and selections, are transformed into "jmp," even when lifting binary code to the assembly layer. With this background, a function pair called an aligned assembly is proposed.

The generation process of the aligned assembly is shown in Figure 3 and Algorithm 1. Firstly, the LLVM is utilized to compile several open-source C/C++ projects into assembly codes, which are treated as anchor files (ASM_a). In the next step, the binary file Bin is generated by compiling and assembling the source file with GCC. Then, IDA-Pro is utilized to disassemble the binary file Bin, and the generated assembly files are treated as positive samples (ASM_p). Such a homologous assembly file is regarded as an equivalent assembly program pair, $ASM_p - ASM_a$. Next, random sampling is employed, where ASM_p is found in order to obtain the non-homologous assembly file ASM_n, which is used as a negative assembly file.

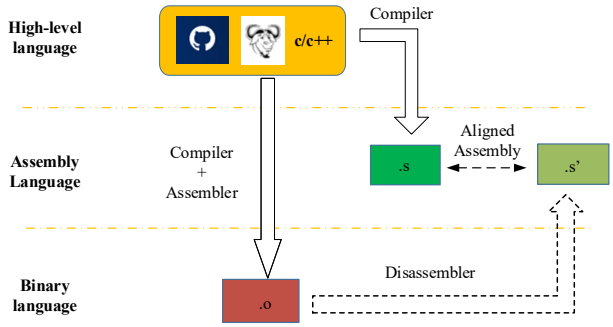


Figure 3. Schematic of aligned assembly generation.

Subsequently, the assembly functions are mapped one by one based on the previous steps, according to the function name. Here, the corresponding anchor function sample f_a and positive function sample f_p are aligned. Then, another random sampling method is processed to select a function sample from the functions generated from ASM_n as the negative function sample f_n. Algorithm 1 below details the generation of AATF. According to the algorithm, a large-scale aligned assembly triplet function dataset can be created using a moderate-scale open-source project set. Currently, a triple function input unit based on an aligned assembly, has been manufactured. Compared to simply utilizing the binary corpus for training, the aligned assembly is expected to provide better training benefits, as will be verified by the experiments described in Section 4.

Algorithm 1. Aligned Assembly Triplet Functions (AATF) Generation.

Input: Anchor function sets S_a, positive function S_p, negative function S_n
Output: Aligned assembly triplet functions dataset,
$D\ (AATF) = \{(f_a, f_p, f_n, aligned(f_a, f_p)) \mid f_a \in S_a, f_p \in S_p, f_n \in S_n\}$

$D \leftarrow EmptySet(\ )$;
for $func \in S_p$ **do**
  $candidates_a = \{f \mid f$ has same name with func, $f \in S_a\}$;
  $r \leftarrow RandomSample\ (candidate_a)$;
  $D \leftarrow D \cup \{(r, func, +1)\}$;
  $r \leftarrow RandomSample\ (S_a - candidate_a)$;
  $D \leftarrow D \cup \{(r, func, -1)\}$;
end
for $func \in S_a$ **do**
  $candidates_p = \{f \mid f$ has same name with func, $f \in S_p\}$;
  $r \leftarrow RandomSample\ (candidate_p)$;
  $D \leftarrow D \cup \{(func, r, +1)\}$;
  $r \leftarrow RandomSample\ (S_p - candidate_p)$;
  $D \leftarrow D \cup \{(func, r, -1)\}$;
end
for $func \in S_p$ **do**
  $candidates_n = \{f \mid f$ has different name with func, $f \in S_n\}$;
  $r \leftarrow RandomSample\ (candidate_n)$;
  $D \leftarrow D \cup \{(func, r, +1)\}$;
  $r \leftarrow RandomSample\ (S_n - candidate_n)$;
  $D \leftarrow D \cup \{(func, r, -1)\}$;
end

3.2.2. Preprocess

The resulting aligned assembly cannot be directly used by the embedding model. Thus, preprocessing in advance is essential. First, normalization and expression are applied to each instruction in every assembly file. During normalization, the instructions are split into opcodes and operands. Opcodes are used to describe machine language instructions, specifying which part of the instruction field performs an operation. The instruction formats and specifications that constitute an opcode are specified by the processor's instruction set specification. Operands participate in the execution of instructions, which specify the objects that perform mathematical operations in the instructions. They contain various constants and registers, which may cause out-of-vocabulary (OOV) problems. The distinction of constant names or register names, which have no semantics, can only introduce noise into the embedding model; thus, the categorization of processing operands is a critical step in instruction normalization. A rule-based instruction normalization scheme (Table 2) is utilized to alleviate the OOV problem.

Table 2. Instruction normalization rule list.

Type	Character	Normalization
Constant	Address	addr
	Variant name	var
	Immediate value	imm
	Basic block label	BB
	Function name	Func
Register	Pointer type	reg_pointer
	Float type	reg_float
	General type (8bit)	reg_gen_8
	General type (32bit)	reg_gen_32
	General type (64bit)	reg_gen_64
	Address type (32bit)	reg_addr_32
	Address type (64bit)	reg_addr_64
	Data type (8bit)	reg_data_8
	Data type (32bit)	reg_data_32
	Data type (64bit)	reg_data_64

Taking the function Entity2 as an example, the assembly code obtained by compiling the source code contains three basic blocks: Entity2, LBB19_1, and LBB19_4. Without normalization, it is easy for the model to fail to extract information effectively, as a token does not exist in the dictionary. A schematic of the assembly function before and after normalization is shown in Figure 4.

In AAPFE, instead of simply deleting subsequent instructions by sequence, the term frequency-inverse document frequency (TF-IDF) method is used to weight all instructions in the basic block. When an instruction appears more frequently in a basic block and is less common, its TF-IDF value is higher, and the instruction is more important. The top instructions with the highest TF-IDF values define the maximum length of the basic block. The remaining instructions are discarded to preserve the information of the basic block as much as possible. This parameter is specified along with others in Section 4.1.

After normalization and compression, tokenization is performed. Here, the torch.nn class in the PyTorch framework is used to transform instruction tokens directly into embeddings. This embedding is simply an initiatory vector, which is randomly initialized as a unique real-valued vector. The initiatory vector has no meaning and can represent neither the semantics of the instruction token nor the relationship between different tokens. At this point, data preprocessing is complete, and the vectors of the instruction tokens are fed into the triplet sub-network as input.

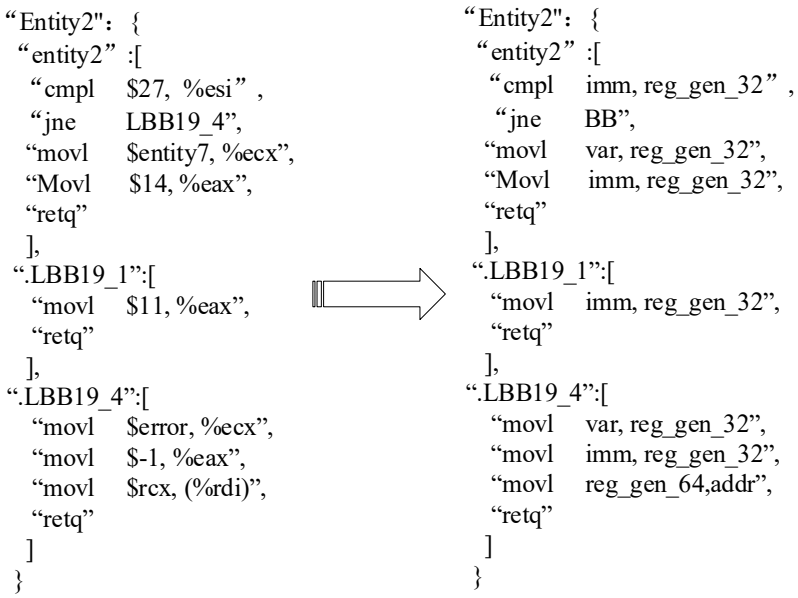


Figure 4. Schematic of normalization on function Entity2.

3.3. Function Embedding Sub-Network

In AAPFE, an embedding network is designed at both the instruction and block levels. A multiheaded self-attention [23,24] (hereinafter referred to as self-attention) mechanism is utilized to calculate the context information of each token in an instruction sequence and to update the weight of the token vector in a basic block. All vectors are summed in one basic block to obtain a basic block vector BB^i_k.

Subsequently, by capturing the jump instructions in the basic block, an adjacency matrix of the CFG can be generated. The adjacency matrix describes whether there is a jump relationship between basic blocks and is utilized to extract the neighboring information and update the block-level embedding. In the triplet sub-network of the AAPFE architecture, the structure of each embedded network is the same, as shown in Figure 5. Only one subnetwork is introduced.

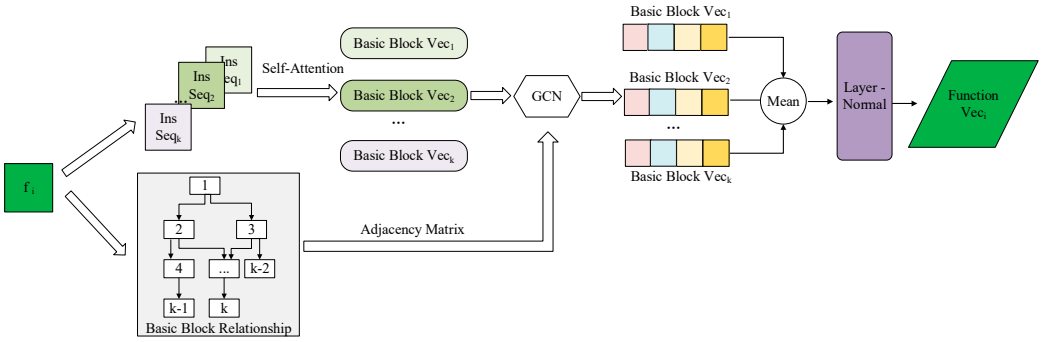


Figure 5. Embedding network structure that performs two tasks: the extraction of the instruction sequence information based on self-attention and the embedding of the basic block jump adjacency matrix using a graph convolution network (GCN). The generated basic block vectors compute the average and then connect to the layer normalization.

Each embedding network consists of two parts: instruction and basic block embedding. The input assembly function is characterized using two types of embedding information as the output of each subnetwork. The embedding networks are described in detail below.

3.3.1. Self-Attention-Based Instruction Embedding

Firstly, the obtained instruction token is processed through the position-wise connection layer and the position information is embedded into each instruction as an instruction sequence. The embedding of the instruction sequence is an information embedding of a two-dimensional sequence structure, which is generally processed by a recurrent neural network (RNN). However, it is difficult for an RNN to extract semantic information far from the target word, and LSTM developed from an RNN cannot perform parallel computing. The self-attention mechanism is used at the instruction level.

Before calculating the self-attention layer, the instruction token is fed into the position-wise feed-forward network to embed the position information. The perception layer updates the vector value according to the position information of each token in the instruction sequence. The output of this position layer is connected to the residuals. The residual connection can not only avoid a single fitting of position information but can also reduce the gradient vanishing or exploding. In this process, an instruction token containing the sequence information of the positional relationship is obtained.

The self-attention mechanism is utilized to calculate and update the weight of each instruction token based on the context information, which uses key-value pairs as the representation of input words and regards the query vector as the representation of the target token. The self-attention weight calculation process for "reg_gen_32" involves its context tokens, as shown in Figure 6 below.

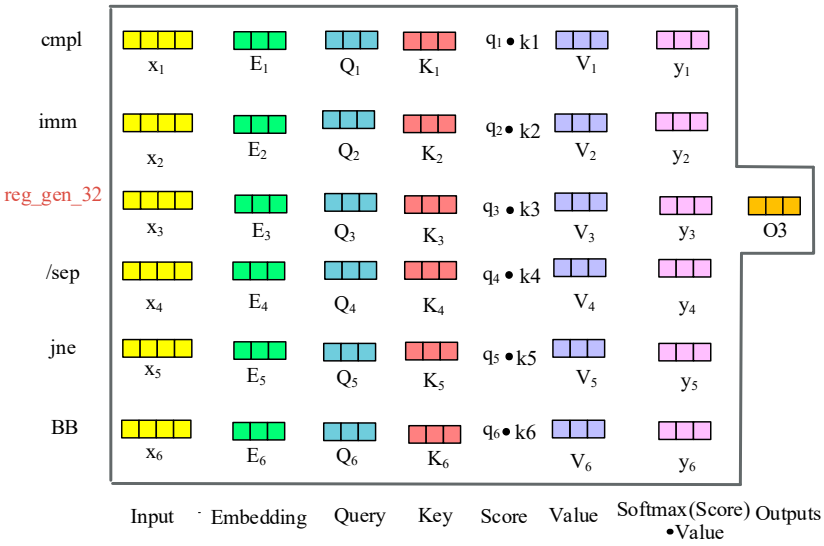


Figure 6. Calculation of the self-attention of token "reg_gen_32" based on the context token.

For example, in the two instructions "cmpl imm reg_gen_32" and "jne BB," the embedding vector of "reg_gen_32" is calculated through the self-attention mechanism. Firstly, position information is added to each token in the instructions, after which each token is converted into a vector and the query, key, and value of each token are calculated. Next, the query of "reg_gen_32" and the key of context tokens are multiplied to obtain the corresponding score value, and the SoftMax function is used to calculate the score to obtain the self-attention weight of "reg_gen_32." Finally, the generated self-attention weight and value of "reg_gen_32" are multiplied to obtain the self-attention output of the target token in the "cmpl imm, reg_gen_32" instruction.

The output and input of the self-attention processing layer are also connected by a residual network followed by a layer normalization, and all the token vectors are summed up to aggregate the instruction sequence as a block-level embedding.

3.3.2. GCN-Based Basic Block Embedding

The graph convolutional network (GCN) [25–27] is a model that performs convolution operations on graphs. Marcheggiani et al. [28] and Huang et al. [29] demonstrated that sequence models and GCNs have complementary modeling capabilities; therefore, based on the instruction sequence vector obtained earlier, the GCN is used to fuse the edge information between basic blocks into block-level information. Based on this basic block intermediate representation vector, the main discussion is how to extract the jump relationship information between the CFG basic block nodes and generate basic block embeddings. First, the jump instructions in each basic block are filtered out. If the instruction opcode is a jump function instruction, such as "jmp," "jnc," or "jc," an edge is established between these two basic blocks. The element of the corresponding position in the adjacency matrix is assigned a value of one; otherwise, the elements are assigned a value of zero. By traversing the individual instructions of the instruction sequence, the CFG basic block adjacency matrix, which contains the structure information of an assembly function, can be built. For example, sample *Opt_path_end* is a function disassembled from the dynamic library file *libss.lib* in the OpenSSL project, as shown in Figure 7.

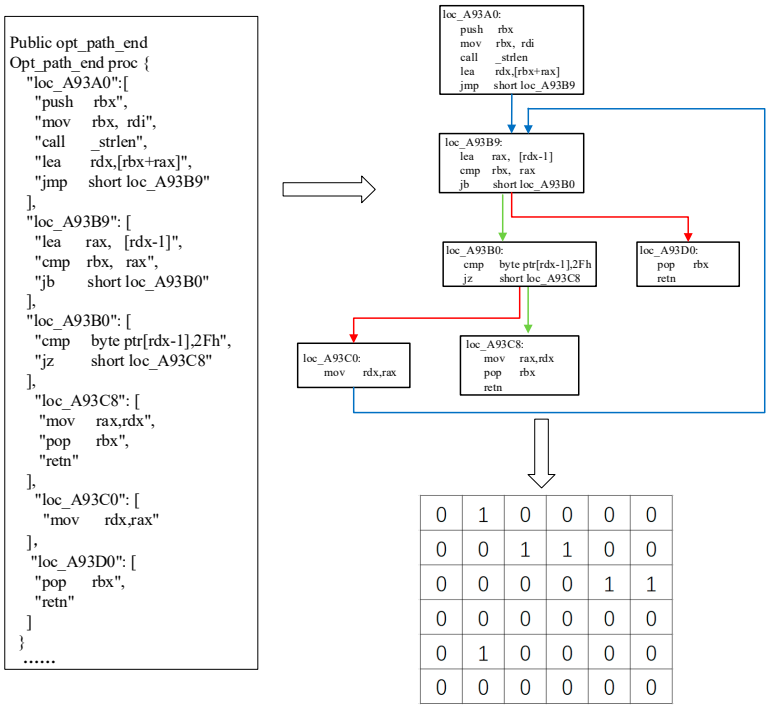


Figure 7. Schematic of basic block adjacency matrix establishment. The assembly segment is selected from function "Opt_path_end" in the OpenSSL project as an example to show a detail matrix of the establishment process.

Based on the characteristics of the above information, the GCN model is used to extract block-level information. The GCN model uses a multi-layer graph neural network to update the embedding according to the layer-by-layer propagation concept of the CNN [30]. Given a graph G (V, E), where V represents a node, $||V|| = N$, and E represents an edge, the GCN utilizes the convolution of the two functions, as shown in Equation (1).

$$g \times x \approx \theta(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})x \quad (1)$$

The adjacency matrix is represented by $A \in R^{N \times N}$ and the degree of a node is given by $D_{ij} = \sum_j A_{ij}$. Because $I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \in [0, 2]$, the multi-layer neural network often makes the gradient easily explode or disappear.

To alleviate this problem, re-regularization is employed. Let $I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = \widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}$ and $\widetilde{A} = A + I_N, \widetilde{D}_{ij} = \sum_j \widetilde{A}_{ij}$, where $\widetilde{A}$ is the adjacency matrix with self-connections and $\widetilde{D}$ is the degree of each node. Equation (2) is approximated, and the activation function *Relu* is added to obtain the GCN model calculation formula, as shown in Equation (2).

$$H^{(l+1)} = Relu(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2}$$

In the above formula, l represents the number of layers of the GCN and $H^{(l)} \in R^{N\times C}$ represents the input vector of the node in the lth layer. The input layer $H^{(0)} = X$ is the initial input vector of the model, whereas N is the number of vectors and C is the matrix dimension. $W \in R^{C\times F}$ is a parameter of NNs and F represents the number of feature mappings, which is the number of kernel functions. $D^{-\frac{1}{2}}\widetilde{A}D^{-\frac{1}{2}}$ can be regarded as a normalized adjacency matrix and $H^{(l)}W^{(l)}$ is equivalent to performing a linear transformation on the embedding of the lth layer node. The left-multiplied adjacency matrix indicates that the feature of the node is the result of adding the features of neighboring nodes.

In this study, the jump information between basic blocks is extracted through the GCN model using Equation (2), after which the representation vectors of the basic blocks are updated. After embedding the basic block, the corresponding basic block vector is obtained by fine-tuning the multi-layer neural network, which corresponds to Equation (3).

$$E_g = MLP(GCN(Info_g, \mathrm{Att}(b))) \tag{3}$$

Xu [31] proved that the sum function is the best choice for fusing both semantic feature and structure information; hence, all block vectors are summed, and the function vectors are obtained. Thus far, a function embedding model that simultaneously extracts sequence information at the instruction level and graph structure information at the block level has been constructed.

3.4. Model Training

3.4.1. Training Objective

In binary analysis, the binary code needs to be converted into a vector through the function embedding model to achieve a similarity comparison with another binary. The binary similarity comparison problem is transformed into a comparison of Euclidean distances between vectors. In the feature space, the distance between the representation vectors can be inversely proportional to the similarity between the binaries; this can be measured directly, as shown in Equation (4).

$$\begin{aligned} \mathrm{Sim}(F_i, F_j) &= \exp(-\frac{D(E_i,E_j)}{d}) \\ &= \exp(-\frac{||E_i - E_j||^2}{d}) \end{aligned} \tag{4}$$

Here, E_i and E_j are the representation vectors of F_i and F_j, respectively. d represents the embedding dimension, and D is the distance between two vectors.

Next, we will describe the way in which the constructed model can be trained. Because data augmentation and the triplet network structure are employed, the resulting vectors are more complex than pairwise direct comparisons. Here, utilizing the triplet loss [22] is an effective means of training a suitable embedding model for each function. In the embedding space, functions of the same functionality and semantics should be clustered closely together to form well-separated groups. A margin-based triplet loss function is built, and reliance is placed on minimizing the loss to train the AAPFE. In the model training process, we attempt to distinguish the distance difference between the aligned and unaligned function vectors as the training objective. The Euclidean distance is used to measure the difference between two assembly function vectors. In the metric learning method, the distance between the positive sample and the anchor sample vector is smaller

than that between the negative sample and the anchor sample vector. The training loss function can be calculated using Equation (5).

$$\begin{aligned} \text{Loss} &= \max\{D(\vec{F}_a, \vec{F}_p) - D(\vec{F}_a, \vec{F}_n) + \alpha\ ,\ 0\} \\ D(\vec{F}_a, \vec{F}_p) &= D(E_a, E_p) = ||E_{ga} - E_{gp}||^2 \\ D(\vec{F}_a, \vec{F}_n) &= D(E_a, E_n) = ||E_{ga} - E_{gn}||^2 \end{aligned} \tag{5}$$

In the projection space, it should be an absolute distance instead of a relative distance. Therefore, the Euclidean distance is used. Meanwhile, the Euclidean distance does not cause all samples of the same type to map to the same point, which is meaningless. α represents the margin, which measures the difference between the two groups of distance comparisons. The larger α is, the greater the difference between the distances of the aligned and unaligned assemblies, the stronger the distinguishability, and the greater the corresponding training difficulty. If α is zero, the condition is extremely loose, and the training difficulty is extremely low. This parameter will be discussed further in Section 4.1.

3.4.2. Model Deployment

As mentioned in the model overview in Section 3.1, this process adopts pre-training, which obtains an embedded model AAPFE of assembler functions in advance through a large-scale dataset, and then deploys the obtained model in the analyzer training of downstream tasks. For downstream applications, AAPFE has two deployment patterns, the first of which is the function embedding generation pattern. The AAPFE is applied as an assembly function embedder, which can provide an input vector for downstream task models. Downstream models use the vectors generated by the embedder directly without tuning. The embedded vector is more compatible than the features extracted by the end-to-end method. The function embedding generation pattern is useful when hardware resources are limited, for example, in embedded devices that do not have sufficient computing power.

The second deployment pattern is a fine-tuning pattern. The resulting model parameters are first frozen and then used to initialize the embedding models for downstream tasks. The new model already has a certain ability to discriminate, although this ability needs to be adjusted further by small-scale training with other datasets. In this approach, the AAPFE-initialized models are fine-tuned and trained together in the downstream process. When sufficient computing resources and training budget are available, this pattern usually provides additional benefits and is task independent. There are several fine-tuning strategies [32], such as two-stage fine-tuning and multi-task fine-tuning.

In this study, the fine-tuning pattern is adopted, employing fine-tuning AAPFE for model deployment.

4. Experiments and Evaluation

4.1. Experimental Setup

The experimental dataset was derived from dozens of famous C/C++ open-source projects in different fields, including 10,989 programs in total. The dataset was employed to pretrain and evaluate the AAPFE. Table 3 lists the relevant projects and their classifications.

Table 3. Open-source projects used in the dataset.

Project	Version	Description
Cmake	3.19.8	Cross-platform build tool
libtomcrypt	1.18.2	Cryptographic toolkit
micropython	1.14	Python for microcontrollers
opencv	4.5	CV and ML library
Binutils	2.30	Binary tool
gdb	10.1	Debug tool
Redis	4.0.13	Database of key value
FFmpeg	4.2.2	Multimedia process tool
Libuv	1.x	Asynchronous I/O library
Libpng	1.6.38	Graphic r/w library
VTK	9.0.1	Visualization toolkit
Curl	7.37.1	Data transmission tool
CoreUtils	4.5.1	GNU core library
Glibc	2.33	C runtime library of Linux
valgrind	3.17.0	Dynamic detection toolkit
OpenSSL	1.1.1b	Cryptographic tool

AAPFE is suitable for assembly embedding at the functional level, but not the program level. The method described in Section 3.2 was used to generate the aligned assembly derived from the open-source project. The anchor sample assembly was compiled using LLVM (v4.0) with no compiler optimization (O0). The binary was assembled by GCC (v7.5.0) and then disassembled by IDA Pro (7.4 sp1) into a positive sample assembly. After alignment, normalization, and compression, the previously described AATF, which consists of 45,360 triple samples, was obtained. Each triplet contained an aligned anchor-positive sample pair and a negative sample. In this study, 80% of the triplets were used for training and 20% for testing.

PyTorch was applied as the deep-learning framework. The experiment was built on a ubuntu18.04 server in the laboratory, equipped with a 2×Xeon Gold 6154 CPU and a 2×Titan-Xp GPU with a memory size of 128 G; the disk was an SSD with a capacity of 4 T.

The hyperparameters for model training after multiple performance comparisons are listed in Table 4 below.

Table 4. Hyperparameters during model pre-training.

Hyperparameter	Value	Description
d_model	256	Embedding dimension
GCN_depth	5	Number of GCN layer
α	120	Margin
Max_len	45	Maximum length of basic block
Max_size	100	Maximum length of function
lr	0.001	Learning rate
Dropout	0.1	Dropout coefficient
Opt	SGD	Optimization algorithm
B_size	32	Batch size
d_ff	256	FC-feed-forward dimension

After 40 epochs of pre-training, both the training and testing losses of AAPFE were near convergence, as shown in Figure 8. The trained model was used as an embedder for function vectors.

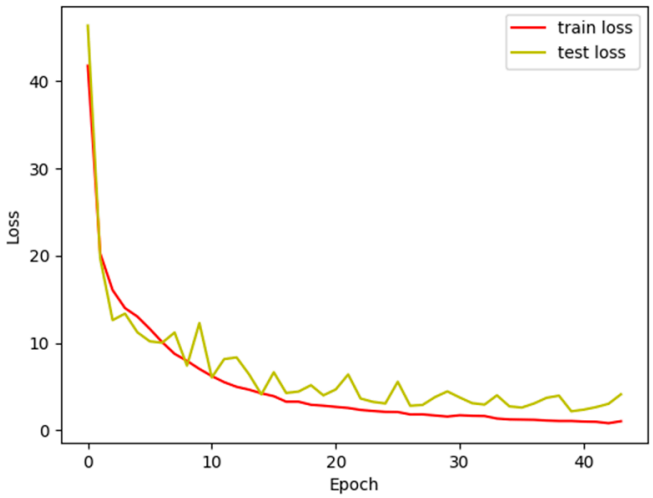


Figure 8. AAPFE pre-training loss curve.

4.2. Performance Evaluation

To verify the performance of the function embedding model, SAFE and Gemini were used as baselines for comparative experiments. SAFE embeds functions through i2V-based instruction embedding and a biRNN with a self-attention mechanism. Gemini utilizes Structure2Vec to capture the ACFG structure information of basic blocks. SAFE and Gemini are state-of-the-art sequence information extraction and graph information embedding approaches, respectively.

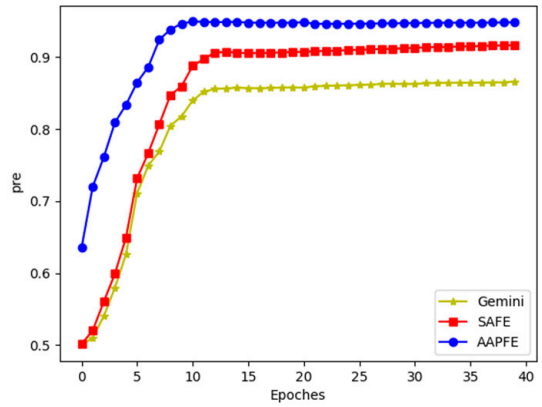
Because the AAPFE parameters are updated by self-supervised learning distance discrimination between different functions, two tasks were established to verify the performances of the three models for fairness:

1. Similarity comparison task: The objective of this task was to have the Euclidean distance of the aligned assembly functions be lower than that of the unaligned assembly functions. Precision and accuracy were used as indicators to measure task performance; the dataset used was AATF.
2. Function search task: This task was consistent with the pre-training task; given a source function, the target was to rank the aligned function as far as possible. The evaluation metric of the task was p@N, which represents the precision ranking of the objective function equivalent to the source function in this function set. The aligned assembly triplet function extension (AATFX) dataset with 10,000 function sets was obtained by augmenting AATF. Each function set contained a pair of aligned assembly functions in the form of triplet data from AATF; the other 99 functions were randomly selected samples, which were treated as negative samples. The source function was equivalent to the aligned objective function. The distance to the positive sample vector was the smallest. The source function was not equivalent to an unaligned assembly function; therefore, the distance should be greater than that of the aligned function.

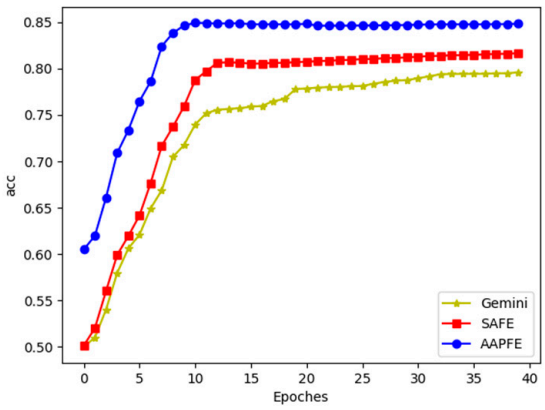
It is worth noting that a function search experiment was designed for two scenarios. The first scenario utilized the compiled function as the source to search for the equivalent target function in the set where 99 negative samples were disassembled functions, which was called the #disassembling asm set. The second used the disassembled function as the source to search for the equivalent target in the function set where all negative samples were compiled functions, which was called the #compiling asm set.

The results shown in Figure 9 and Table 5 indicate that the precision, accuracy, and p@N of AAPFE outperformed those of SAFE and Gemini. The corresponding results of the three models were close to p@1, which was 51.02% of AAPFE, 47.29% of SAFE, and 45.75% of Gemini when searching in the disassembled sample set, and 52.58%, 49.9%, and 46.67% when searching in the compiled sample set. The p@1 indicator is relatively harsh. It is used to rank the assigned function from 100 samples as the first; if the sampled functions are

close, there will be a certain amount of noise. The results indicate that after pretraining, AAPFE could fit a data conversion model with better performance than the Gemini and SAFE models.



(**a**)



(**b**)

Figure 9. Precision and accuracy curve in similarity comparison task: (**a**) precision curve; (**b**) accuracy curve.

Table 5. Comparison with the baseline using p@N metric.

Model	# Disassembling Asm Set			# Compiling Asm Set		
	p@1	p@3	p@10	p@1	p@3	p@10
Gemini	45.75%	50.22%	74.3%	46.67%	52.05%	75.93%
SAFE	47.29%	63.73%	86.08%	49.9%	63.01%	87.61%
AAPFE	51.02%	77.09%	91.01%	52.58%	79.36%	94.16%

The reason for this may be that SAFE adopts i2V to embed the entire instruction information as a vector, and then connects to the biRNN and self-attention network to generate a function vector, which lacks the graph structure information that characterizes the basic block of the function. However, Gemini uses an ACFG graph to represent functions and embeds graph structure information as a function vector through Structure2Vec, without considering the instruction information inside the node. The embedding network in AAPFE is composed of self-attention and a GCN, which is not only more suitable for embedding the node context instruction information of fine-grained preprocessed tokens, but also integrates CFG graph structure information into the embedding, showing high-quality embedding for multi-granularity information extraction performance.

The results also demonstrate that the #compiling asm set values were slightly better than those of the #disassembling asm set, verifying that in the feature space, the compiled function was more contrastable than the disassembled function. The reason for this is that the assembly function vector obtained by directly compiling the source codes extracts more semantic and structural information than the function vector of the assembly produced by the disassembler.

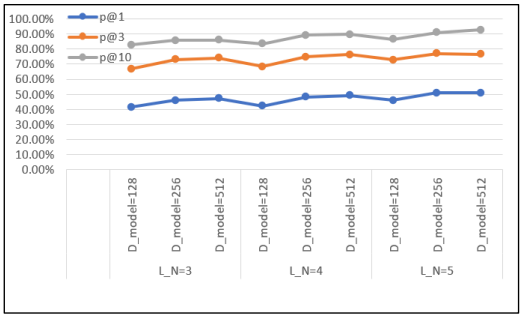
4.3. Training Evaluation

Hyperparameter tests and ablation experiments were performed on the model. These tests were intrinsically benchmark evaluations that provided a generic assessment of the quality and consistency of vector spaces, independent of their performance in downstream applications. The different properties and composition structures of the model can be evaluated in this way.

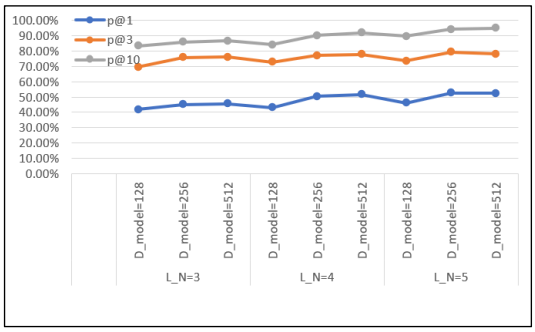
The dataset used was AATFX, as proposed in Section 4.2, and the evaluation index was p@N.

4.3.1. Hyperparameter Test

Based on the changes in the model's hyperparameters, controlled experiments were conducted. The main hyperparameters of the model structure design include the number of GCN convolution layers, L_N, and the dimension of the model, D_model. In the two search scenarios, 128, 256, and 512 were selected as the dimensions; the number of GCN layers was set to three to five; and 40 rounds of training were conducted. Figure 10 shows the results of the hyperparameter testing.



(**a**)



(**b**)

Figure 10. Comparison of training hyperparameters: (**a**) p@N in the disassembling set; (**b**) p@N in the compiling set.

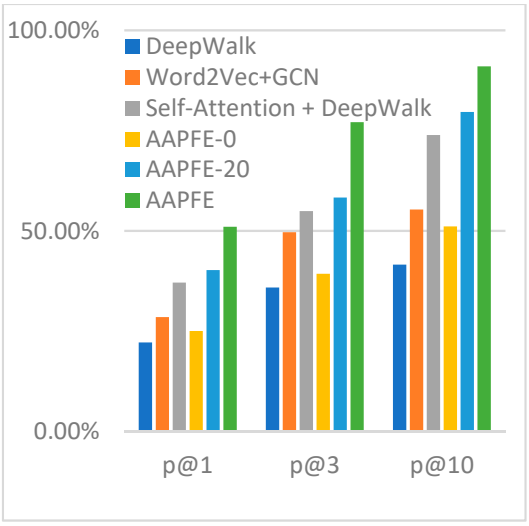
These results indicate that when the number of GCN layers was five, the accuracy ranking index p@10 reached 92.66%, and when the vector dimension was set to 256, p@1 and p@3 were higher than 128. When the embedding dimension was 512, p@10 was the highest, and the improvement was limited compared to p@3. Considering the time complexity factor, the embedding model was set to have 256 dimensions and five GCN

layers to ensure the performance of the AAPFE model without causing excessive memory overhead. Therefore, these two parameters were presented in advance during the pre-training process.

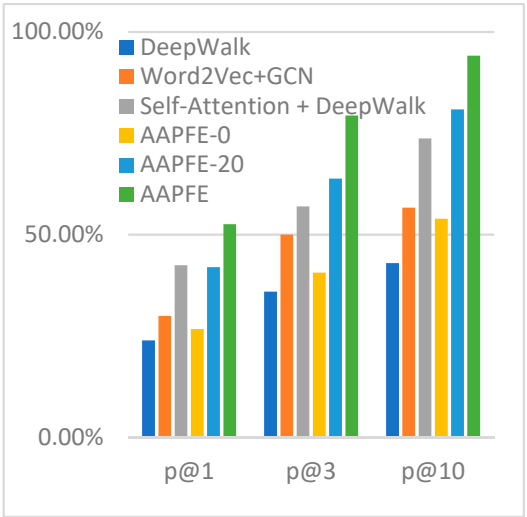
4.3.2. Ablation Evaluation

AAPFE uses the self-attention mechanism and GCN model to build each embedding network. Ablation experiments were conducted, and the different components of the embedding net were compared. The sequence embedding under the self-attention mechanism was replaced with Word2Vec to evaluate the embedding efficiency at the instruction level using DeepWalk as the reference algorithm of the GCN to compare the graph embedding effect at the basic block level. Similarly, the models of different training epochs were also compared to verify the effects of pre-training on embedder construction.

In addition, comparisons between the fully trained AAPFE (40 training epochs), a half-trained AAPFE (AAPFE-20; 20 training epochs), and an AAPFE without pre-training (AAPFE-0) were made to demonstrate the effect of pre-training. The results are shown in Figure 11.



(**a**)



(**b**)

Figure 11. Comparison of different model compositions and pretraining epochs: (**a**) p@N in the disassembling set; (**b**) p@N in the compiling set.

DeepWalk [33] is a graph structure path construction method based on the RandomWalk [34] algorithm. It embeds the basic block jump relationship as a vector through neighbor nodes, and the obtained graph vector loses a great deal of structural information. Word2Vec is a classic word embedding method proposed by Mikolov [35]. The word embedding framework is a classic learning function library for word embedding. The obtained sequence information is limited by the size of the slicing window, and the embedding of different instructions cannot be calculated in parallel. In the instruction-level and basic block-level multi-granularity extraction methods, the Word2Vec + GCN scheme cannot solve the problems of semantic information representation, such as polysemy, because Word2Vec is still a one-to-one mapping between words and vectors. In contrast, because DeepWalk is limited by the number of paths, the self-attention + DeepWalk scheme cannot express multi-path information and intra-node information simultaneously. The proposed multi-granularity extraction method adopts self-attention and the GCN, considering the context information of each instruction simultaneously, and fuses the information of each node in all the constructed paths.

The effects of pre-training on the results are evident, as the performance of AAPFE-0 was lower than that of Word2Vec + GCN because Word2Vec embeds instruction information within a slicing window. However, AAPFE-0 extracted very little sequence information. The p@N index of AAPFE-20 was close to that of the self-attention + DeepWalk scheme, reflecting the slow parameter convergence of the GCN and the effect of graph convolution embedding, which can only be implemented in the subsequent 20 epochs, unlike in DeepWalk. In AAPFE-0, the instruction vector is randomly assigned by the torch.nn, and the convolutional layer does not learn the graph information of the basic blocks. Similarity comparison information comes from the inherent differences between samples. The parameters of the insufficiently pre-trained AAPFE-20 were not sufficiently updated.

4.4. Application Evaluation

In addition to the intrinsic comparative evaluation in similarity comparison and searching, the application of AAPFE to downstream tasks is necessary to demonstrate the effectiveness and robustness of the functional embedding model. Because binary analysis is essentially a classification task, which is related to the similarity comparison pre-training task, the embedding model was introduced into the binary malware analysis task. According to the general requirements of malware analysis, two main tasks were performed in the experiment: malware detection and malware classification.

4.4.1. Malware Detection

The malicious dataset employed in this study was derived from VX Heaven [36], a well-known public virus library widely used for malware analysis tasks. VX Heaven has a collection of 258,296 malicious binary samples in different formats, accounting for a total of 60 GB. These samples are well labeled with four fields: virus type, platform, family, and variant.

In the malware detection task, 55,600 malware binaries were classified and randomly extracted from VX Heaven. Then, 51,600 benign samples were randomly selected from the C/C++ open-source projects. After the 107,200 binaries to be detected were disassembled into assembly functions by IDA, they were connected to the embedded network initialized by the AAPFE parameters to generate function vectors.

To simplify the classifier as much as possible, the function vectors were added and averaged by dimension and the binary file representation vector was obtained. To compare the effects of the embedding part, we directly fed the binary vector to the fully connected layer and the SoftMax function for detection. The embedding methods of Gemini and SAFE were also used as comparison models. Because the AAPFE was pre-trained, simple fine-tuning training was performed on the classifier. The precision, accuracy, recall, and area under the curve (AUC) were selected as indicators to measure performance in the malware detection tasks. The results are listed in Table 6.

Table 6. Comparison of different models applied in malware detection.

Model	Accuracy	Precision	Recall	AUC
Gemini	88.21%	90.32%	92.71%	0.9141
SAFE	91.6%	94.09%	98.44%	0.93153
AAPFE	94.28%	96.36%	97.05%	0.9463

The results show that AAPFE achieved superior performance in the malware detection task. The accuracy, precision, and AUC indicators of the proposed method mostly exceeded those of the baselines, except for the recall indicator of SAFE, which was slightly superior.

This may be a result of the fact that the GCN overcomes the problem of parallelism in CFG graph calculation, and thus is more efficient than Structure2Vec, which is used by Gemini. SAFE also uses a deep network to learn instruction sequence information; however, it only converts the basic block jump relationship into a sequence structure and loses the behavioral function information. Nevertheless, SAFE uses a normal architecture fused with Sentence2Ve and the biRNN method, which is more compatible.

4.4.2. Malware Classification

The malware classification task was conducted using a subset of VX Heaven that was previously classified and labeled with malware functionality. In this study, 79,262 malwares samples were selected from the top five types of samples; their distribution is shown in Figure 12.

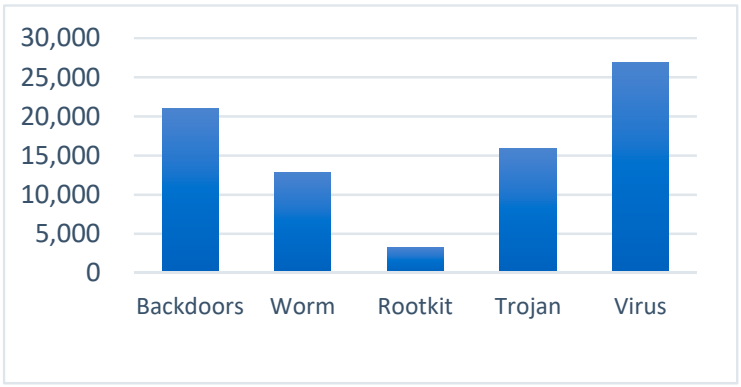


Figure 12. Type distribution of malware samples in a classification dataset.

For simplicity, the same approach employed to obtain the binary vector representing the sample was utilized here. These binary vectors were used to perform five classifications directly using the fully connected layer and the SoftMax function. The results of each model in terms of the evaluation indicators of accuracy, precision, and F1-score are described in Table 7 below.

Table 7. Comparison of different models applied in malware classification.

Model	Accuracy	Precision	Recall	F1-Score
Gemini	78.33%	79.81%	80.61%	0.802
SAFE	80.65%	82.5%	85.06%	0.8376
AAPFE	83.37%	84.22%	84.64%	0.8443

Through the analysis of the results, it is evident that the performances of all three models were limited. There are two reasons for this result. First, the distribution of the chosen sample types was not sufficiently uniform to affect the overall prediction accuracy,

especially the accuracy performance. Second, the function vector was simply averaged; therefore, some functional relationship information may have been lost. These two insufficiencies will be further improved in our future work; however, AAPFE performed better than the other two models in binary embedding for malware classification tasks.

5. Conclusions

This study proposed a pre-trained aligned assembly function embedding model, which takes an aligned assembly function as input data, and an embedding net that adopts a triplet structure with sharing parameters. Each sub-embedding network in triplet architecture uses the self-attention mechanism and the GCN layer to extract and fuse sequence and graph information. The model provides both instruction context information and a basic block association relationship; however, binary obfuscation and packing problems prevalent in the field of anti-analysis are not considered.

Compared to the baseline method, not only did the embedding model of this scheme achieve higher accuracy in the function similarity comparison task, but the embedded vector also performed well in the downstream task. This improvement shows that the embedded model based on pre-training is robust against various downstream tasks. A series of experimental comparisons proved that the model has an over-performance effect.

In the future, different compilers, compilation options, and instruction architectures can be used to generate equivalent homologous functions to build multiple aligned assemblies. More diverse ranked losses will be considered as the convergence target of model training. Another future direction is to use multitasking pretraining methods, such as BERT-based models, to extract more instruction-level and block-level information.

Author Contributions: Conceptualization, H.G. and K.T.; methodology, H.G. and F.L.; software, H.G. and K.T.; validation, F.L., H.G. and K.T.; formal analysis, H.G. and Z.S.; investigation, H.G. and M.Q.; resources, H.G. and K.T.; data curation, H.G. and Y.H.; writing—original draft preparation, H.G.; writing—review and editing, H.G.; visualization, K.T. and C.Z.; supervision, Z.S.; project administration, H.G. and K.T.; funding acquisition, F.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Science Foundation of China, grant number 61802435.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yakdan, K.; Dechand, S.; Gerhards-Padilla, E.; Smith, M. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 23–25 May 2016; pp. 158–177.
2. Xu, Z.; Ray, S.; Subramanyan, P.; Malik, S. Malware Detection Using Machine Learning Based Analysis of Virtual Memory Access Patterns. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 169–174.
3. Liu, L.; Wang, B.; Yu, B.; Zhong, Q. Automatic malware classification and new malware detection using machine learning. *Front. Inf. Technol. Electron. Eng.* **2017**, *18*, 1336–1347. [CrossRef]
4. Kong, D.; Yan, G. Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, IL, USA, 11–14 August 2013; pp. 1357–1365.
5. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2019; pp. 472–489.
6. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Baldoni, R.; Querzoni, L. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Gothenburg, Sweden, 19–20 June 2019; Springer: Berlin, Germany, 2019; pp. 309–329.
7. Zuo, F.; Li, X.; Zhang, Z.; Oung, P.Y.; Luo, L.; Zeng, Q. Neural Machine Translation Inspired Binary Code Similarity Comparison Beyond Function Pairs. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 24–27 February 2019; pp. 24–27.
8. Redmond, K.; Luo, L.; Zeng, Q. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv Prepr.* **2018**, arXiv:1812.09652.

9. Duan, Y.; Li, X.; Wang, J.; Yin, H. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020.
10. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas TX, USA, 30 October–3 November 2017; pp. 363–376.
11. Qiao, M.; Zhang, X.; Sun, H.; Shan, Z.; Liu, F.; Sun, W.; Li, X. Multi-level cross-architecture binary code similarity metric. *Arab. J. Sci. Eng.* **2021**, *46*, 8603–8615. [CrossRef]
12. Huang, H.; Youssef, A.M.; Debbabi, M. Binsequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 155–166.
13. Cauteruccio, F.; Cinelli, L.; Corradini, E.; Terracina, G.; Sa Va Glio, C. A framework for anomaly detection and classification in multiple IoT scenarios. *Future Gener. Comput. Syst.* **2021**, *114*, 322–335. [CrossRef]
14. Liu, J.; Chen, Y.; Liu, K.; Zhao, J. Event Detection Via Gated Multilingual Attention Mechanism. In Proceedings of the AAAI Conference on Artificial Intelligence, New Orleans, LA, USA; 2018; pp. 4865–4872.
15. Liu, J.; Chen, Y.; Liu, K.; Zhao, J. Neural Cross-Lingual Event Detection with Minimal Parallel Resources. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 3–7 November 2019; pp. 738–748.
16. Zhang, X.; Sun, W.; Pang, J.; Liu, F.; Ma, Z. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture. In Proceedings of the 2020 Workshop on Binary Analysis Research, San Diego, CA, USA, 23 February 2020.
17. Li, X.; Qu, Y.; Yin, H. Palmtree: Learning an Assembly Language Model for Instruction Embedding. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Korea, 15–19 November 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 3236–3251.
18. Li, W.; Jin, S. A Simple Function Embedding Approach for Binary Similarity Detection. In Proceedings of the 2020 IEEE Intl Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Exeter, UK, 17–19 December 2020; pp. 570–577.
19. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable Graph-Based Bug Search for Firmware Images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
20. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1145–1152.
21. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the NAACLHLT 2019: Annual Conference of the North American Chapter of the Association for Computational Linguistics, Minneapolis, MI, USA, 3–5 June 2019; pp. 4171–4186.
22. Hoffer, E.; Ailon, N. Deep Metric Learning Using Triplet Network. In Proceedings of the International Workshop on Similarity-Based Pattern Recognition, Berlin, Germany, 12–14 October 2015; pp. 84–92.
23. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*. Available online: https://arxiv.org/abs/1706.03762 (accessed on 13 March 2022).
24. Shaw, P.; Uszkoreit, J.; Vaswani, A. Self-attention with relative position representations. *arXiv Prepr.* **2018**, arXiv:1803.02155.
25. Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; Yu, P.S. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 4–24. [CrossRef]
26. Bronstein, M.M.; Bruna, J.; LeCun, Y.; Szlam, A.; Vandergheynst, P. Geometric deep learning: Going beyond Euclidean data. *IEEE Signal Process. Mag.* **2017**, *34*, 18–42. [CrossRef]
27. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv Prepr.* **2016**, arXiv:1609.02907.
28. Marcheggiani, D.; Titov, I. Encoding sentences with graph convolutional networks for semantic role labeling. *arXiv Prepr.* **2017**, arXiv:1703.04826.
29. Huang, Y.; Qiao, M.; Liu, F.; Li, X.; Gui, H.; Zhang, C. Binary code traceability of multigranularity information fusion from the perspective of software genes. *Comput. Sec.* **2022**, *114*, 102607. [CrossRef]
30. Henaff, M.; Bruna, J.; LeCun, Y. Deep convolutional networks on graph-structured data. *arXiv Prepr.* **2015**, arXiv:1506.05163.
31. Xu, K.; Hu, W.; Leskovec, J.; Jegelka, S. How powerful are graph neural networks. *arXiv Prepr.* **2018**, arXiv:1810.00826.
32. Qiu, X.; Sun, T.; Xu, Y.; Shao, Y.; Dai, N.; Huang, X. Pre-trained models for natural language processing: A survey. *Sci. China Technol. Sci.* **2020**, *63*, 1872–1897. [CrossRef]
33. Perozzi, B.; Al-Rfou, R.; Skiena, S. DeepWalk: Online Learning of Social Representations. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, New York, USA, 24–27 August 2014; pp. 701–710.

34. Spitzer, F. *Principles of Random Walk*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2001; Volume 34.
35. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv Prepr.* **2013**, arXiv:1301.3781.
36. Available online: https://academictorrents.com/details/34ebe49a48aa532deb9c0dd08a08a017aa04d810 (accessed on 10 January 2022).