FlowEmbed: Binary function embedding model based on relational control flow graph and byte sequence

Yongpan Wang[†‡], Chaopeng Dong[†‡], Siyuan Li[†‡], Fucai Luo[§], Renjie Su[§], Zhanwei Song[†], Hong Li[†‡*]
[†]Institute of Information Engineering, Chinese Academy of Sciences, China
[‡]School of Cyber Security, University of Chinese Academy of Sciences, China
[§]State Grid Fujian Electric Power Company, China
{wangyongpan, dongchaopeng, lisiyuan, songzhanwei, lihong}@iie.ac.cn, 13651656090@163.com, tuke@sgitg.sgcc.com.cn

***Abstract*—Binary function embedding models are applicable to various downstream tasks within IoT device software systems and have demonstrated advantages in numerous binary analysis tasks, such as vulnerability (homologous) function search and compilation optimization option identification. However, current binary function embedding methods either learn embedding based on code sequence, which lack the program semantics of functions (e.g., control flow, etc.) or based on program structure graphs, which omit global sequential information. As a result, these methods fall short in enabling models to learn the complete semantic of function. In this paper, we introduce FlowEmbed, a novel approach that synergistically integrates control flow and global semantic learning to facilitate exhaustive code comprehension. Initially, FlowEmbed harnesses a distinct relational control flow graph combined with the power of BERT and RGCN models to aptly capture the nuances of control flow semantics. Moreover, by deploying the DPCNN model on a byte sequence constructed from function machine code, FlowEmbed adeptly discerns the inherent global sequential semantics of binary functions. Through rigorous evaluations spanning three IoT-related tasks, FlowEmbed's efficacy becomes evident, showcasing notable improvements: a 20.6% improvement in compilation optimization option identification, a 1.8% improvement in binary function similarity analysis, and an 11.9% improvement in homologous function search. Collectively, these results underscore FlowEmbed's superior capability, positioning it as a invaluable asset in a binary analysis application.**

***Index Terms*—deep learning, binary function embedding, static analysis, binary code search, binary code similarity detection**

I. Introduction

As the number of IoT (Internet of Things) devices continues to grow, ensuring the security of software running on these devices becomes paramount. However, many IoT device suppliers only provide binary files of software systems, making source code unavailable for security analysis. These binaries, often compiled with varying compiler settings, complicate binary security analysis. Furthermore, vendors strip symbolic information from these files, rendering text-based security analysis nearly unfeasible. Therefore, as deep learning gains momentum, the focus of research has shifted towards leveraging deep learning to learn binary code embedding and apply them to downstream security analysis tasks.

Binary function embedding models employ deep learning techniques to transform binary function into embedding. They are utilized for various downstream tasks in IoT scenarios, such as compilation optimization option identification [1], binary function similarity analysis [2]–[4], and homologous function search [5], [6]. Despite encouraging developments in binary function embedding, challenges persist that can compromise embedding quality and downstream task performance.

G1: Code Sequence-Based Methods. Techniques in this category utilize natural language processing to learn binary code sequences, such as assembly code and intermediate representations. InnerEye [7] and DEEPBindiff [8] encode binary files with word2vec models after normalizing assembly code using normalization strategies. Asm2vec [9] uses the PV-DM model to learn both instruction and function embeddings. XLIR [10] learns the embedding of IR using the BERT model after converting the assembly code to an intermediate representation. A notable limitation (**P1**) is these methods' inability to explicitly learn semantic features of the code, such as control flow semantics or data flow semantics.

G2: Program Structure-Based Methods. These techniques, relying on tools like graph neural networks, study program structure graphs derived from binary code, such as control flow graphs (CFGs) and abstract syntax trees, to grasp code semantics. For instance, Gemini [2] employs Structure2vec to study CFG. Guo et al. [3] uses graph neural networks to learn various types of program graphs, including CFG, data flow graph and call graph. Asteria [4] applies the Tree-LSTM to understand a function's abstract syntax tree. However, these approaches often neglect the unique information inherent in different program structures (**P2**). For example, each edge in a CFG represents a different jump relationship. Moreover, Since this type of approach relies on program structure graphs, they also miss the global sequential information (**P3**), making it difficult for the model to capture

[*]Corresponding author

comprehensive code semantics.

To address the aforementioned problems, we introduce FlowEmbed, a novel method rooted in control flow semantic and global semantic, ensuring a holistic understanding of the code. Distinct from previous approaches, we propose the relational control flow graph (**for P1 and P2**) based on different control flow jump relationships. We then employ BERT and RGCN models to capture control flow semantics. To acquire the global sequential information of the code, we construct a byte sequence from the function's machine code and leverage the DPCNN model, known for grasping long-range dependencies in sequences, to study the global semantics within these byte sequences (**for P3**).

We extensively evaluated FlowEmbed on three downstream tasks closely aligned with IoT scenarios: **compilation optimization option identification**, **binary function similarity analysis**, and **homologous function search**. Our findings underline FlowEmbed's superior performance and generalizability over baseline models. Ablation studies further substantiate the enhancements brought about by FlowEmbed's control flow semantic learning and global semantic learning, with performance gains of 6.8% and 20.6% respectively in the homologous function search task. In conclusion, FlowEmbed proves adept at generating high-quality function embedding, invaluable for various binary analysis tasks.

In summary, we made the following contributions:

- We propose and implement a novel method, FlowEmbed, which primarily combines control flow semantic and global semantic to obtain embedding for binary function.
- We conducted a comprehensive evaluation across three pertinent downstream tasks, affirming its efficacy. Specifically, FlowEmbed showcased improvements: compilation optimization option identification (**20.6%**), binary function similarity analysis (**1.8%**), and homologous function search (**11.9%**).
- We designed a relational control flow graph that aptly represents the control flow semantics of functions. This graph distinguishes subtle control flow differences by constructing diverse jump relationships between basic blocks.

II. Related work

Language Embedding Model. In recent years, Language Embedding Models are pivotal in the realm of natural language processing. They utilize neural networks to transform natural language into high-dimensional embedding vectors. Word2vec [11] utilizes Skip-gram and CBOW models for contextual word semantics, but lacks context-specific word embeddings. ELMo [12] uses RNNs to generate dynamic context-based embeddings. The Transformer [13] in 2017 introduced self-attention for parallel embedding training. BERT [14], a bidirectional Transformer, added two pre-training tasks for comprehensive semantic understanding. ELECTRA [15] optimizes BERT by emphasizing token replacement detection during pretraining.

Binary Code Embedding Model. As the field of deep learning continues to advance, the conversion of binary code into assembly code, coupled with the integration of language embedding models from the domain of natural language processing for learning assembly code embeddings, has emerged as a prominent area of research. Gemini [2] employs seven statistical features of basic blocks and the Structure2vec model to obtain embedded representations of CFGs for binary functions. InnerEye [7] views each instruction as a word, using the Skip-gram model from word2vec to encode binaries. Asm2vec [9] trains on assembly code sequences, inspired by the PV-DM model, and samples based on CFGs. OrderMatters [16] develops four pre-training tasks for assembly code, uses the BERT model to refine basic block embeddings, and integrates graph neural networks with ResNet for CFGs embeddings. Asteria [4], rooted in the Tree-LSTM model, trains on ASTs from decompiled pseudocode, updating embeddings for binary functions. Palmtree [17], a Transformer-based

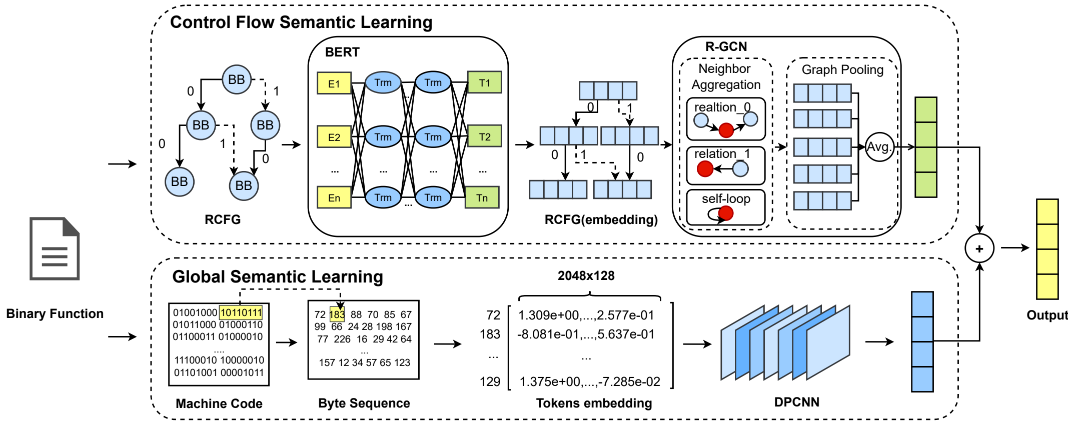

Fig. 1. Detailed design of FlowEmbed. Trm is the transformer encoder unit, BB is a basic block, $E_n(n = 1...N)$ are tokens of a basic block, $T_n(n = 1...N)$ are hidden states of other tokens of the sequence.

model, presents three pre-training tasks to secure contextual semantics and produce instruction embeddings.

III. Detailed design

As illustrated in fig. 1, FlowEmbed consists of two components: Control Flow Semantic Learning (III-A) and Global Semantic Learning (III-B). The former employs a basic block embedding model and a relational control flow graph to understand the intricate control flow information of a function. Futhermore, the latter leverages DPCNN and a byte sequence to capture the global sequence information of the function. Ultimately, these two components are integrated to form a comprehensive semantic representation of the function's code.

A. *Control Flow Semantic Learning*

In this section, our target is to learn the control flow semantic of function. Initially, we employ the basic block embedding model (III-A1) to capture the embeddings of basic block nodes. Subsequently, we derive the control flow semantic of function using relational control flow graph (III-A2).

1) Basic Block Embedding Model: In line with prior work [16], we employ the BERT model to obtain embedding for assembly language. One challenge arising from the assembly code is the presence of constants and strings, leading to Out-of-Vocabulary (OOV) issues. To alleviate this, we adopt normalization strategies. Specifically, We replace strings with a special token $[STR]$, and constants exceeding a length of five(from [17]) are replaced with $[ADDR]$. A basic block, denoted as B_i where i ranges from 1 to k, is a sequence of consecutive instructions without any branching. It comprises multiple instructions but lacks control flow jump instructions such as JMP, JZ, and so on. Semantic embeddings of basic block are crucial for learning control flow semantics.

To enhance the BERT model's capability in accurately capturing basic block embeddings, we construct four pre-training data related to basic blocks based on the function's CFG. As depicted in fig. 2, four pre-training data correspond to four distinct pre-training tasks. **Masked Language Model Task**: Analogous to the original BERT training task, we randomly replace some tokens in the code sequence with a special token $[MASK]$. The BERT model is then tasked with predicting the original tokens that were replaced. The objective of this task is to make the model discern the relationships between individual tokens within a basic block. **Adjacent Block Prediction Task**: To facilitate the model in understanding the relationships between basic blocks and recognizing the execution context of a basic block, we create inputs consisting of basic block pairs for the BERT model. The aim is to predict if two basic blocks are adjacent nodes in the CFG. We randomly sample an equal number of block pairs from same CFG (one set where the blocks are adjacent and another where they are not). **Block-Graph Relationship Task**: Similarly, for the third task, we randomly sample equal numbers of basic block pairs from different CFGs. The model predicts if two basic blocks belong to same graph, which aids model in learning the relationship between basic blocks and their corresponding graphs, benefiting graph embedding generation. **Compilation Configuration Discrimination Task**: The fourth task hones model's ability to discern different compilation configurations. During training, we categorize basic blocks based on their compilation configurations and task the model with predicting these categories. By engaging in these tasks, our methodology aims to provide a holistic understanding of assembly language semantics through BERT model.

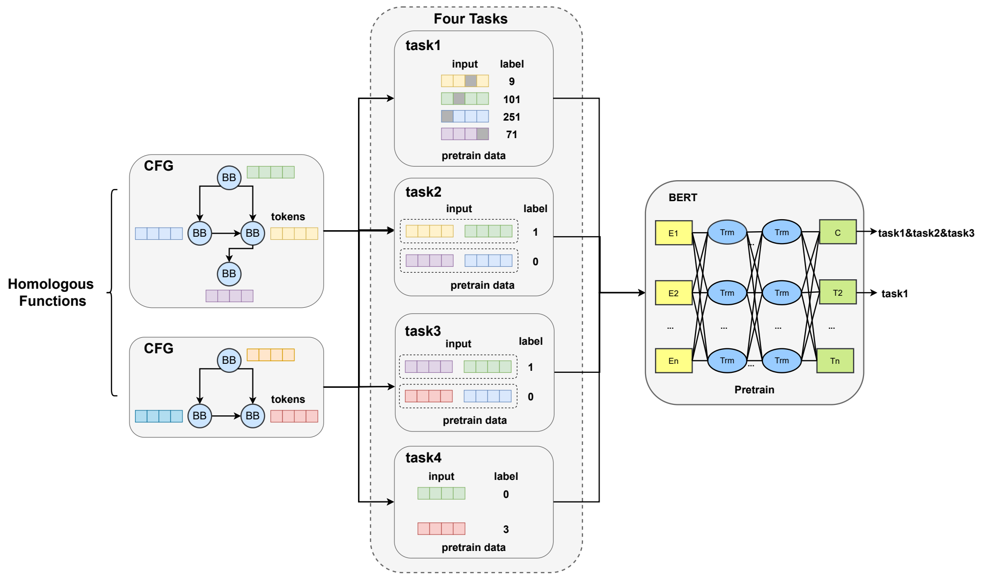

Fig. 2. Pretrain process of BERT model. Homologous functions are obtained from the same source code according to different compilation configurations.

2) Relational Control Flow Graph: Control Flow Graph (CFG) have frequently been employed to depict the control flow within function [16]. Subsequently, Graph Neural Networks (GNNs) were utilized to learn graph embedding from CFG, aiming to represent the control flow semantic embeddings of functions. However, conventional CFG representations often lack a comprehensive portrayal of control flow semantics, overlooking diverse types of control flow edges. This oversight may lead to the omission of certain pertinent control flow semantics.To address this limitation, we introduce the Relational Control Flow Graph (RCFG), which models various control flow edge types to encapsulate a more comprehensive control flow semantics. Consistent with the approach of CFG, we first partition all basic blocks within functions. We then categorize the jump relations between basic blocks into sequential jumps (represented by a type 0 edge, indicating truthful conditions and sequential execution flow) and conditional jumps (represented by a type 1 edge, indicating false conditions execution flow). After constructing the RCFG, We utilize the basic block embedding model as previously mentioned, transforming the assembly codes of all basic block nodes into their corresponding embeddings.

Subsequently, we apply RGCN [18] to derive the semantic and structural embeddings for each RCFG. The graph embedding process of RGCN can be divided into two primary stages: neighbor node aggregation and graph pooling. Distinct from traditional GNNs, RGCN contemplates multiple internode relationships and leverages graph matrix neural networks to address the impact of different edge relations on nodes, harmonizing impeccably with the RCFG structure. For each basic block node B_i in RCFG, the neighbor node aggregation formula is delineated in eq. (1). Here, N_i^r signifies the set of neighbor nodes related to B_i by relation r, while $c_{i,r}$ is a normalization constant with a value of $|N_i^r|$. $W_r^{(l)}$ is a linear transformation function, translating neighbor nodes of the same edge type using a shared parameter matrix $W_r^{(l)}$. The quantity of $W_r^{(l)}$ equals the number of edge types in RCFG, and l represents the depth of the network.

The graph pooling procedure, presented in eq. (2), involves summing and then averaging the embeddings of all updated basic block nodes, thereby yielding the overall RCFG graph embedding, where k is the total number of basic block nodes in RCFG. Through these comprehensive procedures, we have effectively obtained an embedding saturated with complete control flow semantics.

$$B_i^{(l+1)} = LeakyRelu(\sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} B_j^{(l)}) \tag{1}$$

$$G_{embed} = \frac{1}{k} \sum_{i=1}^{k} B_i \tag{2}$$

B. Global Semantic Learning

While the graph embeddings from RCFG provide comprehensive control flow semantic information, their dependence on graph structures inadvertently results in a loss of global sequential semantic information of the function. Addressing this shortcoming, we introduced a module specifically designed to learn the global semantics of functions. Initially, upon acquiring the machine code of a function, we segment the machine code sequence into multiple bytes, taking one byte as a unit. Each byte of machine code is then converted into its corresponding decimal value, ranging from 0 to 255, culminating in the formation of a byte sequence. Once the byte sequence of the function is obtained, our goal is to ensure a comprehensive coverage of the sequence; hence, we standardize the length of the byte sequence to 2048. In instances where the length of the function's byte sequence is insufficient, we employ the numeral 256 as a special padding token to achieve completeness. Consequently, the vocabulary size of the byte sequence is 257. Leveraging the PyTorch framework, we initialize an embedding layer for embedding each token, resulting in 128-dimensional embedding. This transformation converts the machine code into a 2048x128-dimensional tokens embedding matrix.

$$E = G_{embed} + M_{embed} \tag{3}$$

Recognizing the capability of DPCNN [19] in extracting long-range dependencies by progressively deepening the network, we utilize DPCNN to learn the global sequential semantic information present within the tokens embeddings. The resulting embedding from this process is denoted as M_{embed}. Upon completion of both the control flow semantic learning and the global semantic learning phases, we obtain two distinct embeddings: control flow semantic embedding (G_{embed}) and global sequential semantic embedding (M_{embed}). To consolidate the semantic information from both embeddings, we adopt a straightforward summation approach. The final comprehensive semantic embedding of the function, E, is thus obtained, as illustrated in eq. (3).

IV. Experiment

In this section, we focus on evaluating three downstream tasks commonly used in IoT scenarios [16] [1] [5]. Therefore, we aim to answer the following research questions:

RQ1: How well does FlowEmbed perform on compilation optimization option identification (COOI) task?

RQ2: How well does FlowEmbed perform on binary function similarity analysis (BFSA) task?

RQ3: How well does FlowEmbed perform on homologous function search (HFS) task?

RQ4: How is the contribution of each component?

A. Experiment Setup

1) Experiment Environment: We use IDA Pro 7.5 to disassemble the binaries, and our code uses python 3.8, using DGL, and Pytorch to implement FlowEmbed. We run our evaluation on Ubuntu 22.04 with an Intel Xeon 128-core 3.0GHz CPU, including hyperthreading, 1TB memory, and 2 Nvidia V100 32GB GPUs.

2) Compared Methods: Given the cross-architecture scenario in IoT and the setting for function code embedding learning, we opted to replicate and train five relevant state-of-the-art (SOTA) models, based on official papers or their corresponding code implementations. These models are delineated as follows: **GCN** employs GCN [20] to learn embedding from Control Flow Graph (CFG). Notably, the node embeddings are initialized through a random generation process; **RGCN** utilizes RGCN to obtain embedding from the RCFG. The node embeddings initialization is similar to GCN; **Gemini** utilizes the Structure2vec approach to derive embeddings from CFG (nodes are composed of seven statistical features); **OrderMatters** collects information from three diverse perspectives (semantic, structural, and order) to produce a robust function embedding; **Asteria** pivotes on the power of abstract syntax trees and employs a Tree-LSTM model to distill embeddings for functions.

3) Datasets: As illustrated in table I, we have meticulously constructed four distinct datasets to facilitate both training and evaluation. These datasets encompass two architectures (ARM, X86) and four optimization levels (O0, O1, O2, and O3). The constructed datasets serve multiple objectives: **BERT** is designed explicitly for pre-training the BERT model; **COOI** is used to evaluate compilation optimization option identification task; **BFSA** is used to evaluate binary function similarity analysis task; **HFS** is used to evaluate homologous function search task. For BERT pre-training, we sourced 79 open-source projects from GitHub and SourceForge, including libsndfile, busybox, and binutils. After manual compilation, this yielded 318,442 functions. Since BERT processes basic blocks, this provides extensive data. For evaluation, we chose projects not in the pre-training set, like openssl (versions 1.0.1f and 1.1.1m), libcurl, and ncurses, totaling 104,350 functions. These inform our evaluation datasets for the specified tasks.

TABLE I
BASIC STATISTICS OF THE DATASETS.

Dataset	Training	Validation	Testing	All
BERT	254753	31844	31845	318442
COOI	83480	10437	10433	104350
BFSA	58436	12520	12524	83480
HFS	66784	8348	8339	83480

4) Evaluation Metrics: Given the unique objectives associated with each task, we utilize distinct evaluation metrics to evaluate their performances. For the COOI task, accuracy serves as the primary evaluation metric, capturing the proportion of correctly identified optimization options. The BFSA task employs the Area Under Curve (AUC) as its benchmark metric. AUC provides insights into the model's ability to differentiate between positive and negative samples. The formula for AUC is illustrated in eq. (4), where P denotes the count of positive samples, N denotes the count of negative samples, p_i is the predicted score for a positive sample, n_j is the predicted score for a negative sample, $I(p_i, n_j)$ denotes that if p_i is greater than n_j, the value is 1, which is equal to 0.5, otherwise 0. The evaluation criterion for the HFS task is the mean average precision (MAP, eq. (6)). Average precision (AP, eq. (5)) is the average accuracy of one prediction result, where: n is the total number of prediction functions; k is the ranking position of the prediction functions; $P(k)$ is the proportion of homologous functions among the first k prediction functions; $sim(k)$ indicates whether the kth prediction function is a homologous function, 1 for yes, and 0 for no;N_{sim} indicates the total number of homologous functions in the test dataset. MAP is the mean value of the average accuracy of the multiple prediction results, where: q is the serial number of the prediction, Q is the total number of predictions, and $AP(q)$ is the average accuracy of the q_th prediction result.

$$AUC = \frac{\sum I(p_i, n_j)}{P * N} \tag{4}$$

$$AP = \frac{\sum_{k=1}^{n}(P(k) \times sim(k))}{N_{sim}} \tag{5}$$

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{6}$$

B. Performance of COOI task

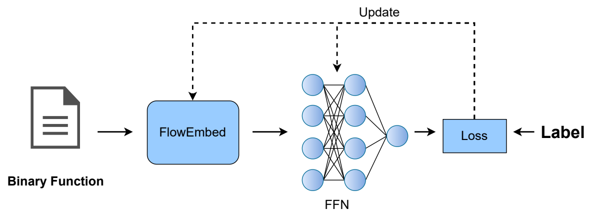

Fig. 3. The training framework of COOI task.

It seeks to infer which compiler settings are utilized during the compilation process of a function. This task has found widespread applications, notably in the realm of vulnerability search within the IoT. In this paper, we focus on the identification of four compilation optimization option (O0-O3) on two prominent architectures, namely ARM and X86. To evaluate the quality of function embeddings in this context, we employ a framework as illustrated in fig. 3. We input the function embeddings obtained from FlowEmbed into a feed-forward neural network. Upon processing through the network, these embeddings are transformed into a 4-dimensional vector. This vector is then passed through a softmax activation function. The resultant values are then compared with the true labels to compute cross-entropy loss. Subsequently, through back-propagation, the parameters of both the FlowEmbed and the feed-forward network are updated. From the results (table II) we can make the following observations:

1) The embedding produced by FlowEmbed outperform the quality of those generated by Order Matters and Gemini in the context of COOI task. It achieves the

TABLE II
ACCURACY OF COOI TASK.

Models	ARM	X86	Avg
GCN	0.298	0.290	0.294
RGCN	0.583	0.605	0.594
Gemini	0.523	0.594	0.556
Order Matters	0.568	0.659	0.614
FlowEmbed-CF	0.708	**0.873**	0.791
FlowEmbed-GS	0.628	0.694	0.661
FlowEmbed	**0.777**	0.863	**0.820**

highest accuracy, surpassing even 26.4%. This observation underscores that the function embedding generated by FlowEmbed exhibit superior generalization when employed in downstream tasks.

2) GCN achieves an average accuracy of only 29.4%, while Gemini, leveraging expert-designed node features, attains an average accuracy of merely 55.6%. Conversely, Order Matters, employing the basic block embedding model (similar to FlowEmbed), achieves an average accuracy of 61.4%, surpassing GCN by 32.0% and Gemini by 5.8%. These results substantiate the efficacy of the basic block embedding model and underscore the necessity of a well-structured basic block embedding model.
3) RGCN achieves an average accuracy of 59.4%, surpassing GCN, which also employs random node embeddings, by 30.0%. The key distinction between the two lies in the fact that RGCN utilizes a RCFG, while GCN employs a CFG. This finding underscores the effectiveness of utilizing a RCFG.

C. Performance of BFSA task

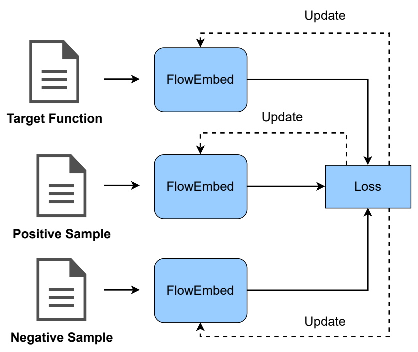

Fig. 4. The training framework of BFSA task.

It determines two functions were compiled from same source code (homologous function). It is extensively utilized in the realm of IoT for vulnerability searches and component analysis tasks. In this study, we have chosen two distinct scenarios to evaluate the performance of FlowEmbed: Cross-Architecture (XA), where two functions are compiled from the same compilation optimization option but hail from different architectures; and Cross-Optimization (XO), where both functions are compiled from the same architecture but are derived from different compilation optimization options.

$$Loss = \frac{1}{m}\sum_{m}(marigin - Sim_p + Sim_n) \qquad (7)$$

To evaluate the quality of function embedding for this task, we employ a framework as depicted in fig. 4. The framework accepts three inputs: the target function and its corresponding homologous function and non-homologous function. Once passed through FlowEmbed, embeddings for the respective functions are generated. We employ cosine distance to gauge the similarity between two embeddings. Lastly, the loss between positive and negative samples is computed using eq. (7), where $margin$ represents the distance between positive and negative samples, m stands for batch size, Sim_p symbolizes the cosine similarity between the positive sample and the target function, and Sim_n illustrates the cosine similarity between the negative sample and the target function. From the results (table III) we can make the following observations:

1) FlowEmbed's function embeddings excel over Gemini, Order Matters, and Asteria in binary function similarity analysis, achieving an AUC higher even by 3.3%. This advantage stems from FlowEmbed's comprehensive semantic capture.
2) Methods like Gemini and Asteria, without the basic block embedding, show significant AUC fluctuations for XA and XO tasks. In contrast, Order Matters and FlowEmbed, using the basic block embedding model, demonstrate more robustness, indicating the model's efficacy in capturing distinct semantics in XA and XO.
3) Both GCN and RGCN use the same node embedding initialization. Yet, RGCN achieves an AUC 4.9% higher than GCN, suggesting that RCFG outperforms CFG for the BFSA task.

D. Performance of HFS task

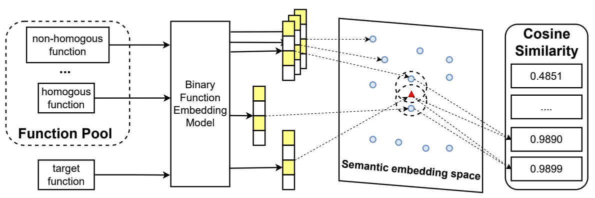

Fig. 5. The whole process of HFS task.

The BFSA task identifies function similarities, while the homologous function search (HFS) task requires a stricter distinction between positive and negative samples. As shown in fig. 5, after processing through the binary function embedding model, the target function and functions from the function pool (mostly non-homologous functions) are placed in a semantic embedding space. The HFS task's goal is to rank these functions based on their cosine similarity to the target, striving to prioritize the homologous function. We use

TABLE III
AUC OF BFSA TASK.

Models	XA				XO		Mix	Avg
	O0	**O1**	**O2**	**O3**	**ARM**	**X86**		
GCN	0.950	0.897	0.896	0.905	0.876	0.855	0.875	0.893
RGCN	0.973	0.958	0.951	0.962	0.924	0.900	0.928	0.942
Gemini	0.991	0.969	0.968	0.970	0.943	0.951	0.953	0.964
Order Matters	0.978	0.976	0.975	0.978	0.973	0.961	0.961	0.972
Asteria	0.990	0.988	0.989	0.987	0.962	0.968	0.966	0.979
FlowEmbed-CF	**0.999**	0.997	0.995	**0.998**	0.993	0.989	0.993	0.995
FlowEmbed-GS	0.996	0.991	0.992	0.992	0.982	0.972	0.984	0.987
FlowEmbed	**0.999**	**0.998**	**0.998**	**0.998**	**0.996**	**0.992**	**0.995**	**0.997**

the Mean Average Precision (MAP) metric for evaluation. We adopted the framework (fig. 4) from the BFSA task to train and also explored varying function pool sizes to determine their impact on method performance. From the results (table IV) we can make the following observations:

1) FlowEmbed's function embeddings outperform those of Asteria, Order Matters, and Gemini in the task of homologous function search. They exhibit superior capability in distinguishing between homologous and non-homologous functions, achieving the highest MAP score, even surpassing it by 29.0%.
2) As the function pool size increases, and despite a decrease in MAP scores for all existing methods, this trend is expected. FlowEmbed maintains a higher score than all other testing methods, demonstrating its superiority.
3) RGCN significantly outperforms GCN in three distinct tasks, evidencing that RCFG offers better generalization and control flow semantic representation than CFG.
4) Gemini creates embeddings from manually designed features but lags behind deep learning-based models like FlowEmbed. This underscores the advantage of deep neural networks in binary function embedding.

In short, FlowEmbed generates high-quality (better than existing work) function embedding, which are helpful for three IoT-related tasks.

E. Impact of each component in FlowEmbed

In this section, we analyze the contribution of two components, Control Flow Semantic Learning and Global Semantic Learning, to FLowEmbed.

Control Flow Semantic Learning. As demonstrated in table II, table III, table IV, the performance metrics after employing FlowEmbed-CF showed a decline when compared to the original FlowEmbed. This change is attributed to the enhancement in FlowEmbed's performance due to control flow semantic learning. By employing RCFG, control flow semantic learning presents the control flow relationships of functions more comprehensively. The combination of RGCN and basic block embedding models facilitates the extraction of control flow semantics from RCFG, thus enhancing the expressiveness of FlowEmbed embeddings. For instance, in the homologous function search task, the MAP score of FlowEmbed-CF stood at 65.0%. However, with the aid of control flow semantic learning, FlowEmbed enriched its semantic information, pushing the MAP score up to 71.8%.

Global Semantic Learning. In three related downstream tasks in table II, table III, table IV,, the performance metrics post-integration with FlowEmbed-GS notably decreased in comparison to the standalone FlowEmbed. This shift is indeed due to the significant contribution of global semantic learning to FlowEmbed's improved performance. With the assistance of the DPCNN model, global semantic learning captures the long-range dependencies in byte sequences, leading to embeddings with comprehensive sequential semantics. For example, in the compilation optimization option identification task, the accuracy rate of FlowEmbed-GS was only 66.1%. In contrast, with the support of global semantic learning, FlowEmbed's accuracy surged to 82.0%.

V. DISCUSSION

In this paper, our goal is to develop a high-quality binary function embedding model that can be applied to various IoT-related downstream tasks. When applying, only an output layer needs is added without changes to the model's structure.

However, in this paper, we only evaluated the role of control flow semantics in the model's understanding of binary function semantics, without involving other code semantic information (such as data flow, abstract syntax trees, etc.). In fact, utilizing richer semantic information allows the model to better model the semantics of functions, resulting in higher-quality embeddings, which is very beneficial for improving performance. We leave this for future work.

It's worth noting that for global semantic learning, we only used DPCNN and byte sequences. This is because the combination of the two has relatively low hardware requirements, preventing excessive hardware demands when combined with control flow semantic learning. In reality, by substituting more complex models, the learning of global sequential information is greatly beneficial. We leave this for future work.

VI. CONCLUSION

In this paper, we address unresolved challenges in the field of function embedding learning. We present FlowEmbed, a

TABLE IV
MAP SCORES OF HFS TASK.

Models	PoolSize=100	PoolSize=500	PoolSize=1000	PoolSize=2000	PoolSize=3000	Avg
GCN	0.379	0.235	0.194	0.160	0.141	0.222
RGCN	0.488	0.324	0.282	0.244	0.225	0.313
Gemini	0.640	0.461	0.396	0.337	0.306	0.428
Order Matters	0.725	0.611	0.529	0.462	0.398	0.545
Asteria	0.806	0.646	0.558	0.505	0.479	0.599
FlowEmbed-CF	0.863	0.714	0.635	0.543	0.495	0.650
FlowEmbed-GS	0.753	0.565	0.476	0.403	0.363	0.512
FlowEmbed	**0.902**	**0.777**	**0.703**	**0.627**	**0.579**	**0.718**

binary function embedding model that employs RCFG and byte sequences for a comprehensive semantic capture. RCFG, representing control flow semantics, is learned through the RGCN model. The model's basic block embedding undergoes pre-training with four distinct tasks. To ensure complete semantic understanding, FlowEmbed integrates the DPCNN model to process function byte sequences. We thoroughly evaluate FlowEmbed against three pivotal IoT security tasks: compilation optimization option identification, binary function similarity analysis, and homologous function search. Results confirm FlowEmbed's superior efficacy compared to existing models, reinforcing its potential as a go-to tool for diverse IoT binary security analysis tasks.

ACKNOWLEDGEMENT

This work is supported by the Science and Technology Project of State Grid Corporation of China (Research on the key technologies of the full stack domestic replacement for the company's specialized network security protection equipment, 5700-202258499A-3-0-ZZ)

REFERENCES

[1] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun, "Understand code style: Efficient cnn-based compiler optimization recognition system," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[2] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376.

[3] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, "Exploring gnn based program embedding technologies for binary related tasks," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 366–377.

[4] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 224–236.

[5] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.

[6] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.

[7] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

[8] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and distributed system security symposium*, 2020.

[9] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[10] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 601–612.

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[12] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237. [Online]. Available: https://aclanthology.org/N18-1202

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

[16] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[17] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[18] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 2018, pp. 593–607.

[19] R. Johnson and T. Zhang, "Deep pyramid convolutional neural networks for text categorization," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 562–570.

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.