STRTUNE: Data Dependence-Based Code Slicing for Binary Similarity Detection With Fine-Tuned Representation

Kaiyan He, Yikun Hu, Xuehui Li, Yunhao Song, Yubo Zhao, and Dawu Gu, *Member, IEEE*

***Abstract*— Binary Code Similarity Detection (BCSD) is significant for software security as it can address binary tasks such as malicious code snippets identification and binary patch analysis by comparing code patterns. Recently, there has been a growing focus on artificial intelligence-based approaches in BCSD due to their scalability and generalization. Because binaries are compiled with different compilation configurations, existing approaches still face notable limitations when comparing binary similarity. First, BCSD requires analysis on code behavior, and existing work claims to extract semantic, but actually still makes analysis in terms of syntax. Second, directly extracting features from assembly sequences, existing work cannot address the issues of instruction reordering and different syntax expressions caused by various compilation configurations. In this paper, we propose STRTUNE, which slices binary code based on data dependence and perform slice-level fine-tuning. To address the first limitation, STRTUNE performs backward slicing based on data dependence to capture how a value is computed along the execution. Each slice reflects the collecting semantics of the code, which is stable across different compilation configurations. STRTUNE introduces flow types to emphasize the independence of computations between slices, forming a graph representation. To overcome the second limitation, based on slices corresponding to the same value computation but having different syntax representation, STRTUNE utilizes a Siamese Network to fine-tune such pairs, making their representations closer in the feature space. This allows the cross-graph attention to focus more on the matching of similar slices based on slice contents and flow types involved. Our evaluation results demonstrate the effectiveness and practicality of STRTUNE. We show that STRTUNE outperforms the state-of-the-art methods for BCSD, achieving a Recall@1 that is 25.3% and 22.2% higher than jTrans and GMN in the task of function retrieval cross optimization in x64.**

***Index Terms*— Binary code similarity, data dependence, code representation, graph neural network.**

I. INTRODUCTION

BINARY Code Similarity Detection (BCSD) is commonly applied in retrieving vulnerable functions in third-party libraries or firmware [1], [2], [3], [4], [5], [6], [7], [8], [9], conduct malware analysis [10], [11], [12], [13] and perform binary patch analysis [14], [15]. It is crucial for ensuring security as it identifies potential threats by analyzing and comparing binary code patterns. However, current work has not yielded satisfactory results in BCSD. Hence, it remains an issue to identify similar functions due to differences arising from cross-optimization, cross-architecture, and cross-compiler during binary compilation.

Recently, the field of artificial intelligence has witnessed significant advancements in BCSD. Existing AI-based research [1], [3], [4], [5], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25] involves transforming the basic blocks into numerical vectors and using deep neural networks to generate function-level features. These methods can automatically extract features, have better generalization capabilities, and perform well at scale. However, they still suffer from low precision because directly embedding assembly instructions using NLP or graph neural networks fails to capture the features at the code behavior level.

To identify similar functions across different compilation settings, current approaches typically have two limitations. The first limitation is that BCSD needs to capture features of functions in terms of similar code behavior while existing approaches are unable to make an analysis on function behavior and actually extract features based on syntax and structures. These methods directly embed based on instruction tokens. With a slight change at the instruction level that does not affect code behavior, the resulting embeddings may undergo significant alterations. This is actually a feature at the syntax level and cannot reflect the code behavior of functions. PELICAN [26] designs a trigger to insert some instruction into the binary code snippet, while preserving the semantics of programs. This also indicates that current methods tend to lean towards syntactic analysis rather than code behavior.

The second limitation is that existing work has been insufficient in addressing the issues posed by instruction disorder and various syntax expressions resulting from different compilation configurations. ASM2VEC [16], INNEREYE [17] and SAFE [4] all treat instructions within a single basic block or all sequential instructions as fixed sequences to extract features. JTRANS [20] additionally considers the positional relationship of jump instructions. Different compilation configurations may lead to the disorder of instructions or even different syntactic forms without affecting the computational content of function execution. However, these methods overly emphasize the positional association between instructions. Instructions

Received 28 April 2024; revised 28 September 2024; accepted 29 September 2024. Date of current version 13 November 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3101402; in part by Shanghai Pujiang Program under Grant 22PJ1405700; and in part by Shanghai Committee of Science and Technology, China, under Grant 23511101000. The associate editor coordinating the review of this article and approving it for publication was Dr. Xiaojing Liao. *(Corresponding authors: Yikun Hu; Dawu Gu.)*

The authors are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200025, China (e-mail: pppgggyy@sjtu.edu.cn; yikunh@sjtu.edu.cn; tu.ana.ut@sjtu.edu.cn; songyunhao@sjtu.edu.cn; yubozhao@sjtu.edu.cn; dwgu@sjtu.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3484944

without data dependence can be interchanged forward or backward. These features of instructions make current methods of directly learning from instruction syntax not robust. Also, instructions have numerous combination forms to accomplish the same value calculation, which cannot be covered by these methods.

For the aforementioned limitations, we propose a novel graph representation model and implement a prototype STRTUNE for BCSD. To address the first limitation, since the computed values represent the stable behavior of functions, instructions can be sliced based on data dependence, where each slice corresponds to the computation of a value in the function. STRTUNE performs backward instruction slicing based on data dependence and reorganizes them to generate slices as nodes. Each slice represents the computation of one value, and such slicing ensures fixed instruction sequences within slices and data independence between slices. The computation of each value will invariably be manifested through different assembly instructions and sequences in similar functions and can be correspondingly captured by neural networks in the form of slices.

To tackle the second limitation, fine-tuning based on similar computational contents at the slice level can grasp identical slices under different syntactic expressions. This helps the model pay more attention to similar slice pairs between functions. Since the instructions within a slice obtained through data dependence-based slicing have fixed positions, STRTUNE adds a new type of flow to represent the computational independence between slices, forming a graph representation for functions. Utilizing the RoBERTa natural language model, STRTUNE conducts a two-step operation of pre-training and fine-tuning. In the fine-tuning phase, STRTUNE pairs slices from the same line of computational contents in source program. Then, STRTUNE uses a Siamese network to fine-tune slice pairs, allowing the cross-graph attention to focus more between similar slices based on slice contents and flow types involved.

In this paper, we propose a novel pipeline STRTUNE for BCSD. First, we put forward a novel graph representation for binary functions based on data dependence relationship. STRTUNE then employs a Siamese network to fine-tune pairwise slices obtained from the same code. For graph-level representation learning, STRTUNE leverages the graph matching network to assign higher attention to semantically equivalent slices. Also, STRTUNE allows for visualization of the matching cross graphs, facilitating an understanding of the similarity calculation. To evaluate STRTUNE, we conduct function recalls in tasks cross-architecture, cross-optimization, and cross-compiler. STRTUNE outperforms state-of-the-art (SOTA) baselines in both recall@1 and MRR. Furthermore, we conduct a real-world vulnerability search. For each CVE, STRTUNE demonstrates an impressive recall success, ranking first in the most cases, surpassing other baselines by a considerable margin.

In summary, our contributions are summarized as follows:

- We propose a novel graph representation model for binary code, slicing based on data dependence which aids in segmenting instructions for value computation, and thus remains a relatively resilient feature across various compilation configurations.

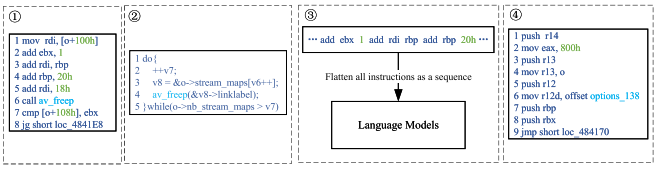


Fig. 1. ① Shows instructions of the function *uninit_options* with its corresponding computational contents in ②. ③ shows the general method that current approaches use to process the instruction token. ④ involves the prologue of the function *uninit_options*.

- We conduct a two-step of pre-training and pairwise fine-tuning for code slices with the same computational content but different syntax, utilizing Siamese network to make their representations closer.
- We employ the Graph Matching Network for function-level representation, with attention coefficient focusing on the matching of similar slices.
- We implement the above ideas and propose a model named STRTUNE. We validate the effectiveness and practicality of STRTUNE by comparing it with SOTA methods. Specifically, in the task of BCSD over different optimizations in x64, STRTUNE achieves a Recall@1 that is 25.3%, 154.9%, and 22.2% higher than jTrans, VulHawk and GMN.

II. BACKGROUND AND MOTIVATION

A. Problem Description

Binary code similarity detection (BCSD) is defined as the identification of similarities within assembly code when the source code is not available. We define two binary functions as semantically similar if they are compiled from the same or logically similar source code. This detection task is also viewed as a code search problem, where the aim is to locate the truly similar functions within a given pool of known functions for a query function. The challenge of binary code similarity highlights the necessity to abstract variations introduced by different compilers, compiler versions, optimization levels, architectures, and obfuscations. This is crucial for facilitating reverse engineering tasks, patch analysis, and the identification and remediation of binary vulnerabilities.

B. Motivation

This section explains the common problems in current work and our motivation for constructing a novel graph representation for binary functions.

Figure 1 captures a part of the instructions from the *uninit_options* function in the *ffmpeg* binary, compiled with the architecture x64, compiler gcc and optimization O3. The instructions in ① forms a complete basic block, corresponding to the pseudocode in ②, with instructions in ④ showing the prologue of this function. The current NLP-based approach, as depicted in Figure 1 ③, involves unfolding the instructions within a basic block sequentially. The entire sequence is considered as input to an NLP model for sentence-level learning. This type of approach has several unreasonable aspects:

- In Figure 1 ①, *'mov rdi, [o+100h]'* corresponds to the assignment of *v8*, and *'add ebx, 1'* corresponds to the

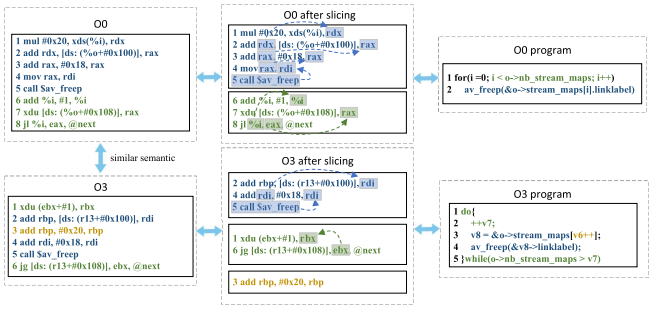


Fig. 2. IR of the *uninit_options* function compiled with O0 and O3, and our ways to slice the instructions, corresponding to the computational contents in the same color.

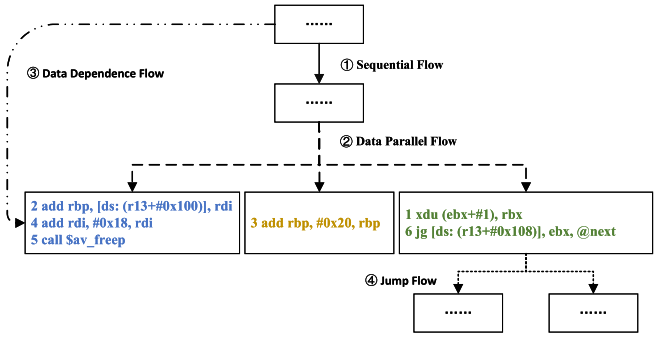


Fig. 3. Example of our graph representation, consisting of four types of flow: ① Sequential flow, ② Data parallel flow, ③ Data dependence flow and ④ Jump flow.

increment of *v7*. These two instructions can be interchanged without affecting the computation of two values. The instruction *'add rbp, 20h'* represents the increment of *v6* and must be placed after *'mov rdi, [o+100h]'*. *'add rdi, 18h'* is for parameter passing and must follow the *'add rdi, rbp'* due to the data dependence of the register *rdi*. Training based on NLP models would assume a fixed sequential relationship between these instructions. However, only instructions with data dependence have certain positional constraints, while instructions without data dependence can be arranged in any order without affecting the computational effect of the entire basic block. It is evident that the instructions in a basic block may contain several unrelated computation slices, whose sequences may be potentially changed by different compilation configurations.

- In Figure 1 ④, instructions related to stack frame preservation for function calls are necessary for the function's runtime execution. However, some of these instructions, like *'push rbp'*, may not be relevant to the computation of any values in this function. For instance, if *rbp* is immediately assigned a variable in subsequent instructions, *'push rbp'* becomes irrelevant to the overall semantic understanding of the function.

Computation refers to the value processing executed in a sequence of instructions, representing the concept of 'collecting semantics' [27] which captures and records the changes in values throughout the execution of the code. Assembly code is compiled from source code, and although assembly instructions may change under different compilation conditions, the underlying computation of the source code remain unchanged, making it a stable feature. Therefore, this work mainly capture the computational semantics reflected in the assembly instructions for BCSD. Read and write operations of registers in instructions are aimed at achieving these computations, which is reflected by data dependence. Hence, we perform backward slicing based on data dependence [28] to divide instructions into slices. Taking one basic block of O3 in Figure 2 as an example, during backward traversal, Instruction 6 uses *ebx*, and by tracing back to Instruction 1 where *rbx* is defined, 1 and 6 form a slice. Next, for Instruction 5, it uses *rdi* as the parameter of *av_freep*. Instruction 4 defines *rdi* and uses *rdi*, so the traversal continues until Instruction 2 defining *rdi*. Thus, 2, 4, and 5 form another slice. Finally, Instruction 3, which has no preceding data dependence, forms its own slice.

Slices obtained from the same basic block are data independent, and the contents within each slice correspond to computations of certain values in the program. We consider slices performing equivalent computations on the same values in the program as semantically similar. Therefore, slices that are semantically similar but syntactically different under various compilation configurations can be matched one-to-one after being split in this way.

As an example, we consider the similar IR slices of the *uninit_options* function compiled with O0 and O3 respectively, as shown in Figure 2. The slices based on data dependence correspond to the function calls and conditional statements respectively. We pair slices with the same color after pre-processing to fine-tune the model, aiming to learn instructions that have the same semantic but different syntactic forms. During slices matching, instructions in yellow will not be matched with any slices compiled with O0. Slices in yellow represent the increment operation of variable *v6* introduced in O3 optimization. The role of variable *v6* is equivalent to that of variable *v7*. This is because different compilation configurations may trigger optimizations, leading to the generation of additional intermediate computation variables in the assembly code. However, we do not match slice pairs for these intermediate variables based on slice matching. Therefore, during later-stage feature extraction, the neural network assigns lower attention to these slices, aiding in the accurate similarity calculation.

It's important to consider the connection relationships between slices, as the structure training takes into account the contents of slices and the types of edges involved. Therefore, We add four types of edges to the graph based on whether there is data dependence (② Data Parallel Flow, ③ Data Dependence Flow) or control flow dependence (① Sequential Flow, ④ Jump Flow), the model can focus on the flow process of variables in the graph, thereby highlighting the more similar parts between graphs.

As shown in Figure 3, after the previous step, we get our slices to serve as our graph nodes. Supposing the slices s_1 and s_2 respectively belong to the basic block bb_1 and bb_2, we add flow type between slices according to the following rules:

1) Sequential flow: To preserve the original unconditional jump information. If s_1 is equal to bb_1, and s_2 is equal to bb_2, meaning both basic blocks are split into only one slice each. Also, there is an unconditional jump between bb_1 and bb_2, we add a sequential flow between s_1 and s_2.

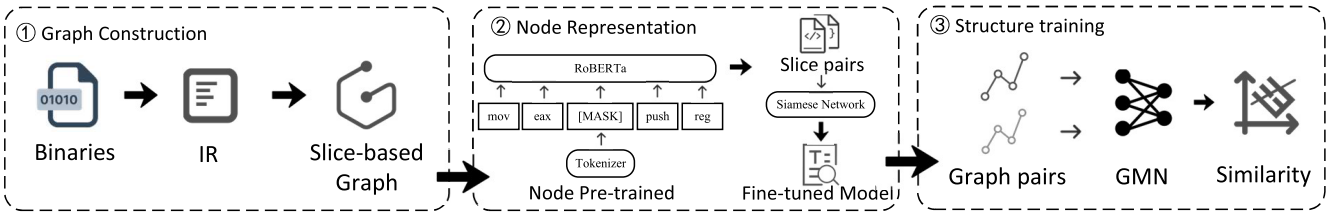


Fig. 4. The overview of STRTUNE consisting of three components: ① Graph Construction that builds novel graphs base on IR lifted from binaries. ② Node Representation that forms a two-step learning process for similar slices, including pre-training and fine-tuning of RoBERTa model. ③ Structure training that performs labeled learning for graph pairs based on the node representations given by RoBERTa, utilizing attention coefficients cross graphs. STRTUNE forms a binary similarity model which takes two functions as input and outputs a similarity score.

2) Jump flow: To represent the original conditional jump. If both basic blocks are split into only one slice each, and there is a conditional jump between bb_1 and bb_2, we add a jump flow between s_1 and s_2.
3) Data dependence flow: To capture the long-distance data dependence. If there exists data dependence between s_1 and s_2, meaning a variable defined in s_1 is used in s_2, we add a Data dependence flow between s_1 and s_2. It is worth noting that this type of flow only occurs between slices split from different basic blocks because our slicing already implies the data dependence within basic blocks.
4) Data parallel flow: Focusing on the unfixed positions between slices without data dependence. This type of flow can be further divided into jump parallel and sequential parallel. Taking sequential parallel as an example. If there is an unconditional jump between bb_1 and bb_2 and bb_1 or bb_2 is split into more than one slice, we add a sequential parallel flow between s_1 and s_2. The same is for the addition of jump parallel flow. Data parallel flow emphasizes the unfixed positions between slices, facilitating slice alignment between functions.

Therefore, slicing based on data dependence satisfies three features: 1. Instructions within a slice have data dependence and thus have fixed positions and cannot be swapped. 2. Slices split from a basic block do not contain data dependence, and their computations do not affect each other, hence we consider them as slices with data parallel. 3. A slice corresponds to the computation of a certain value in the source code shown in the middle column of Figure 2 with the same color. According to this pattern, we can obtain pairs of slices with different syntax but the same computational content, enabling the natural language model to generate similar embeddings for slices with similar semantic but different syntax. Additionally, in structural perception, the network can identify similar slices (comprising node contents and related edges). The greater the presence of such similar features, the higher the similarity of the corresponding source code computations. Consequently, the model tends to regard such pairs of functions as more similar.

III. DESIGN

A. Overview

Figure 4 illustrates the entire workflow of our approach, which primarily consists of three components.

1) Graph Construction (①): We lift the binary code into Intermediate Representation (IR) Microcode and perform instruction deletion and preservation due to specific rules as the pre-processing. Based on data dependence, we apply backward slicing to decompose the instructions into slices at the basic block level, as nodes of our graph. Each node can independently correspond to the computation of some values. Also, we categorize the edges into four types and thus form a representation graph for functions.

2) Node Representation (②): Based on the obtained graph, we normalize the instructions within nodes and perform a two-step operation involving pre-training and fine-tuning using a natural language model RoBERTa to obtain slice representations. We form pairs of code slices corresponding to the same computation content and use a Siamese Network for contrastive learning. This approach brings the embeddings of slices with different syntax but same semantic closer, facilitating the learning of code slices across different compilation configurations.

3) Structure Training (③): Based on slice embeddings and different flow types, we employ Graph Matching Network (GMN) between a pair of graphs, which can assign higher attention coefficients to similar slices across graphs. The network obtains two graph-level embeddings and calculates similarity scores. Finally, STRTUNE takes pairs of functions as inputs and outputs similarity scores.

B. Graph Construction

We first lift the binary to Microcode for analysis. Figure 5 ① shows a segment of Microcode corresponding to a real-world program. Each Microcode instruction consists of one opcode and 3 operands: left, right, and destination, although some operands may be missing for certain types of instructions. We first perform pre-processing of Microcode based on following rules.

1) Removal: Since Microcode includes EFLAGs as implicit operands, considering all EFLAGs introduces extra overhead and obscures the main feature of the binary functions. Therefore, we retain only those EFLAGs assignment instructions where the defined EFLAGs are used in subsequent blocks. As shown in Figure 5 ①, *cfadd* generates the carry bit. *setz* and *setp* compare the left and right operands and store the results in the *zf* and *pf* respectively. These EFLAGs are either redefined or not used in subsequent instructions. Additionally, *rax* defined in the first instruction is also redefined before used, so the assignments to EFLAGs and registers in these situations can be removed without influencing the computation of values in the functions.

2) Preservation: We also preserve unused registers in two cases. The first case is for assignments to global variables, identified as mop_v in Microcode. The second case is for

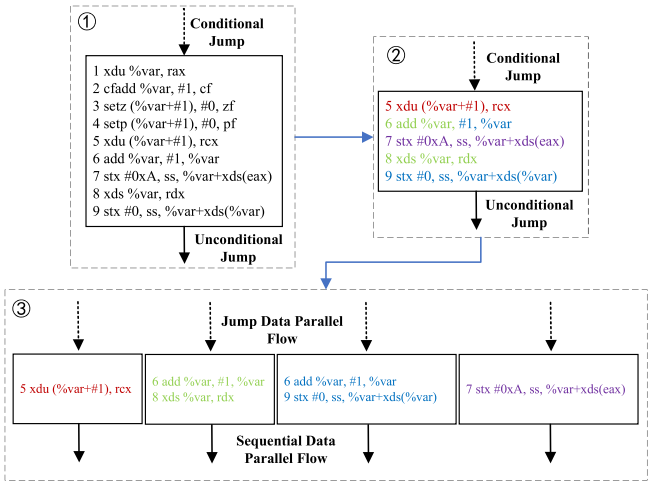


Fig. 5. Graph Construction of our model StrTune: ① shows unprocessed Microcode in a basic block. After removal and preservation, we obtain the remaining instructions in ②, where instructions of the same color exhibiting data dependence, are considered as one code slice. Then we add specific flow type between slices based on the rules mentioned before, as shown in ③.

passing arguments to a called function. Since the registers used for passing arguments differ for each architecture, and because the default function call is immediately followed by the argument passing instruction within the same basic block, we preserve instructions that assign values to unused registers before a call instruction. If the register is an argument register for the corresponding architecture, we also retain that instruction. Therefore, we pre-process the instructions in Figure 5 ① and obtain Figure 5 ②.

3) Slicing and Flow Adding: Based on the data dependence, we perform backward slicing and grouping of the instructions as mentioned in Section II. Starting from Instruction 9, it uses *var*, which is defined by Instruction 6, so 6 and 9 form a slice. Instruction 8 also uses *var*, and since one instruction can be associated with multiple slices, Instructions 6 and 8 form another slice. The remaining Instructions 7 and 5 each form their own slice. These obtained slices are considered data-parallel segments, having unfixed position with each other but fixed sequence of instructions within themselves. Therefore, even if a slice is produced by different compilation options, as long as it corresponds to the same computational logic in the source code, it can be captured and aligned by the subsequent network. Additionally, since the basic block is connected to the previous one by a conditional jump edge, all the slices are linked with jump data parallel flow coming from the previous slices. When encountering a conditional jump as an outgoing edge in the basic block, we only connect the slice containing the jump instruction with the subsequent slices. When encountering an unconditional jump, as in this example, we add sequential data parallel flow from each slice to the subsequent slices, forming the graph as shown in Figure 5 ③. At the same time, based on the data definition and use across basic blocks, these slices are also connected from previous slices with data dependence flow to handle remote data dependence. On the basis of slice alignment, by matching the flow types connecting the slices and the content of the connected slices, the network can enhance the confidence of similar slices using attention coefficient, thereby increasing the similarity score of similar functions.

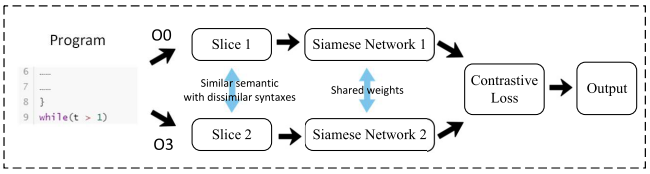


Fig. 6. Siamese neural network for fine-tuning RoBERTa model on similar slices with identical computation contents but different syntax.

Based on our observations, different optimization levels can lead to duplication of the same slice, resulting in repeated slice with the same semantic in the graph. We merge nodes that are completely identical in content and have the same preceding and following nodes, as well as the same types of connecting edges, achieving a partial effect of manual de-optimization. Additionally, we add data dependence flows between code slices. Since the splitting of slices already reflects the intra-block data dependence chain, data dependence flows only occur between slices split from different basic blocks.

C. Node Representation

In this section, we demonstrate how to learn the content of nodes in the constructed graph. Figure 6 shows the pipeline we use for slice content learning.

1) Tokenization and Normalization: As previously mentioned, a Microcode instruction is divided into one opcode and three operands, providing a natural condition for our tokenization. Since the opcode determines the basic type of an instruction, such as movement, shift, comparison, etc., it provides fundamental semantic features. Given the limited number of opcodes in Microcode, we construct a separate vocabulary for opcodes to alleviate out-of-vocabulary (OOV) issues during training and avoid fundamental semantic loss. Due to the chosen maturity level of Microcode parsing, operands might appear in compound expressions after some transformations, such as *var_160+#1*, which increases the complexity of operands. Forms with an underscore suffix typically involve the definition and use of a temporary variable. Since our instruction slices already consider such local information, and this global information is also connected through data dependence flows between slices, we directly remove the suffix such as '_160'. Also, terms starting with '#' are usually numeric constants, which we replace with the *num* character. String constants, usually carrying richer semantic, are retained in their entirety. Fortunately, microcode categorizes operands, such as mop_S for local stack variables and mop_l for local variables. Therefore, we can replace OOV operands with their types, minimizing the semantic loss in instruction representation learning.

2) Pre-training: We use RoBERTa, a natural language processing model based on the Transformer architecture, to perform pre-training on our normalized code slices. RoBERTa is designed for unsupervised training. Given a slice, we randomly select 15% of the opcode or operand tokens and replace the chosen words with a special [MASK] token. The model's objective is to predict the word marked by [MASK] based on the context.

We collect all slices from the training dataset to form our pre-training model and optimize the model parameters

using cross-entropy:

$$L(\theta) = -\sum_{i \in M} \log P(x_m | x_{<m}; \theta) \tag{1}$$

where M represents the mask set, x_m represents the original token marked as [MASK], $x_{<m}$ represents the context tokens of that mask token, and θ represents the model parameters. The model, after pre-training, can transform a given slice into an embedding vector in a high-dimensional vector space, where each dimension corresponds to different aspects of semantic information. We believe this embedding initially extracts the information from the slices but lacks targeted learning for instruction slices with the same semantic but different syntax.

3) Fine-tuning: To enhance the RoBERTa model's understanding of slices with the same computation content, we constructed a Siamese network. This network improves the quality of embeddings by matching slices from the same source code but compiled under different configurations. For example, given the significant optimization differences between O0 and O3, we primarily select slices from these two optimization levels for pairing, keeping the compiler, bit, and architecture unchanged. We can obtain two binaries compiled at O0 and O3 optimization levels as the objects to be matched.

Specifically, for a given C source code file, we consider that code involving conditional judgments (if, while, for loops) often contains more semantic information, which tends to be stable across different compilation configurations. Branch statements execute different code blocks based on input conditions, determining the behavior of the program. Instructions involving branch statements from different compilations are guaranteed to exist and be semantically equivalent. Therefore, we select statements involving conditional judgments and randomly pick 10% of statements without conditional jump to form our input set from this source file as the appropriate size of training dataset. For each program statement, we perform address mapping in the binary to obtain pairs of slices (s_1, s_2) under O0 and O3, representing slices with the same computation content in respect of different optimizations. We also construct slice pairs for different architectures, compilers, and bitness to enable the model to learn slice representations more comprehensively.

We utilize contrastive loss as the loss function:

$$L = \sum yd^2 + (1-y)\max(m-d, 0)^2 \tag{2}$$

where $y = 1$ indicates that the selected pair of slices are semantically similar, while $y = 0$ indicates dissimilarity. m represents a threshold, where if the distance exceeds this threshold, dissimilar sample pairs will no longer incur additional loss. This hyper-parameter is set to 1 in our experiments. d denotes the cosine distance of two vectors $\frac{v_1 \cdot v_2}{|v_1| \times |v_2|}$ generated by our pre-trained Roberta model. The model utilizes the cosine similarity of vectors of slice pairs to ensure that the pre-trained Roberta model maps slices corresponding to the same statement as close as possible, while pushing semantically dissimilar slices farther apart. Through this fine-tuning step, the Roberta model is capable of generating robust embeddings when faced with semantically similar slices under different compilation conditions.

D. Structure Training

We use Graph Matching Networks (GMN) as the structure-level feature extraction model. With respect to graph neural networks, GMNs not only consider aggregated messages on individual graphs but also incorporate a cross-graph attention mechanism. This attention coefficient measures the degree of alignment between cross-graph nodes. Cross-graph attention matches well with our fine-tuning based on code slices.

Given a pair of functions to compare, the node features are embeddings of slices generated by the RoBERTa model. Considering that our constructed function representation graph includes five types of edges, we adopt the commonly used encoding technique in machine learning and data processing, one-hot encoding, to distinguish the types of flow in the graph. The GMN model consists of an encoder, propagation layers, and an aggregator. The encoder maps the node and edge features to initial node and edge vectors through separate MLPs (Multi-Layer Perceptrons). Then, through multiple layers of propagation, the representation for each node will accumulate information in its local neighborhood and edges. Specifically, the propagation layers consider a cross-graph attention matching which measures how well a node in one graph can be matched to one or more nodes in another graph:

$$a_{j \to i} = \frac{\exp\left(s_h\left(h_i^{(t)}, h_j^{(t)}\right)\right)}{\sum_{j' \in G_2} \exp\left(s_h\left(h_i^{(t)}, h_{j'}^{(t)}\right)\right)} \tag{3}$$

$$\mu_{j \to i} = a_{j \to i}\left(h_i^{(t)} - h_j^{(t)}\right) \tag{4}$$

In this context, s_h represents the cosine similarity, with $h_i^{(t)}, h_j^{(t)}$ being the representations of node v_i in graph G_1 and node v_j in graph G_2 after the t-th round of propagation, respectively. Intuitively, the attention parameter $a_{j \to i}$ enables the instruction nodes in graph G_1 to pay more attention to semantically similar instruction nodes in graph G_2. Therefore, the attention based on the RoBERTa model's fine-tuning can more straightforwardly focus on nodes with similar semantic.

Based on the node representations $h_i^{(t)}$ from the previous round of propagation, the information from the neighboring nodes in the current round $m_{i' \to i}$ and the cross-graph node matching information $u_{j \to i}$, the model updates the node representations through a GRU (Gated Recurrent Unit):

$$h_i^{(t+1)} = GRU\left(h_i^{(t)}, \sum_{(i,i') \in G_1} m_{i' \to i}, \sum_{j \in G_2} \mu_{j \to i}\right) \tag{5}$$

Then, an aggregator takes the set of node representations $h_i^{(T)}, i \in G_1$ as input to generate a graph-level representation:

$$h_{G_1} = MLP\left(\sum_i \sigma(MLP_{gate}(h_i^{(T)})) \otimes MLP(h_i^{(T)})\right) \tag{6}$$

For each node, its representation transformed by an MLP is aggregated with the representations of other nodes through a weighted sum. This weighted sum uses weights or attention scores to determine the contribution of different nodes. Additionally, gating vectors can be used to modulate the importance of each node. This process helps to filter out irrelevant information, extract important node features, and

better capture the structural information in the graph. The computation of the graph representation of G_2 is identical. The similarity score of the graphs can be calculated using standard vector space similarity, $s = f_s\left(h_{G_1}, h_{G_2}\right)$.

We employ pairwise training and optimize using gradient descent algorithms. The pairwise loss function we used is:

$$L_{pair} = \max\left\{0, \gamma - t\left(1 - d\left(G_1, G_2\right)\right)\right\} \tag{7}$$

where $t = 1$ if the pair is a similar function, otherwise $t = -1$. $\gamma > 0$ is a margin parameter, and $d\left(G_1, G_2\right) = \left\|h_{G_1} - h_{G_2}\right\|^2$ represents the Euclidean distance. Our focus is not on designing new networks for feature extraction but rather on how to better utilize suitable networks for BCSD, specifically for the binary graph representation we have built.

IV. IMPLEMENTATION

We leverage IDA Pro 7.7 [29] into disassembling binaries and write scripts with IDAPython [30] that can extract Microcode with the information needed such as def-use lists and operand types. We lift the binary opting for the maturity level 'MMAT_LOCOPT' because the instructions at this optimization level are relatively concise, and the lower two maturity levels could not be successfully exported via scripts. Additionally, we do not choose a higher optimization level to avoid increased difficulty in node representation learning due to overly optimized instructions. This disassembling part is finished on a server running Windows 10 with Intel Xeon Silver 4210 CPU @ 2.20GHz and 32GB RAM. We implement RoBERTa [31] and graph matching network (GMN) [23] using Transformers [32], NetworkX [33], Tensorflow [34] based on Python 3.7.15. We run these parts of the experiments on a Linux server running Ubuntu 22.04.2, with an Intel Xeon Platinum 8362 CPU @ 2.80GHz, 251GB RAM and one NVIDIA RTX3090 GPU.

A. Hyper-Parameters

For the vocabulary generation, we consider the tokens which occur more than 10 times in our training dataset. The tokens not in our vocabulary will be seen as its operand type. In the RoBERTa model, the node embedding dimension is 768. In GMN, the number of propagation layers is 10, the aggregation type is 'sum' and the dimension of node hidden embedding is 128. In the structure training, the learning rate is 0.001 and the batch size is 20.

V. EVALUATION

A. Experiment Setup

1) Datasets: We evaluate our model on the dataset based on that from [35] due to the disassembler version. The dataset consists of 24 libraries compiled from seven open-source projects: ClamAV, Curl, Nmap, OpenSSL, Unrar, Z3 and Zlib. Each library is compiled using two compilers (GCC, Clang) with four versions each, for 5 architecture combinations (x64, x86, ARM-32bit, ARM-64bit and MIPS-32bit) and 5 optimization levels (O0, O1, O2, O3, Os).

2) Metrics: We select two commonly used metrics to measure ranking accuracy: MRR (Mean Reciprocal Rank) and Recall@K [35]. Recall@K calculates the proportion of truly similar items retrieved within the top K items. MRR measures how often a similar item appears at the top of a ranking list, with a value closer to 1 indicating better performance of the model in calculating similarity. MRR is computed by:

$$MRR = \frac{1}{\|Q\|}\sum_{q=1}^{Q}\frac{1}{rank_q} \tag{8}$$

where Q represents the set of all query functions, and $rank_q$ represents the rank of the ground truth function for the qth query function.

3) Baselines: We choose the following state-of-the-art (SOTA) methods for comparison:

- Zeek [22]: Computes hash values for instructions with data dependence as feature vectors of basic blocks, and then applies a two-layer fully connected neural network to learn cross-architecture function similarity.
- SAFE [4]: Utilizes a seq2seq model-based NLP encoder, using a self-attention sentence encoder to embed assembly code.
- GMN [23]: Uses a bag-of-words model based on opcodes as the feature of basic blocks, and then utilizing GMN to extract function-level semantic. It is proven to outperform concurrent work in [35].
- jTrans [20]: Based on the Transformer model, it extracts token embeddings from normalized Instructions and adds position embeddings to better learn jump-related information in instructions.
- Trex [21]: Besides the assembly instructions, it tracks the corresponding register values during the execution process, combining with transfer learning to extract the semantic of function execution.
- VulHawk [3]: In addition to using the RoBERTa model for embedding node information, it also determines binary optimization levels and compilers from an entropy perspective.

4) Evaluation Setup: Our evaluation setup is presented as follows: Section V-B showcases the effectiveness of STRTUNE on tasks related to similar function retrieval. Section V-C illustrates the computational time of each part of STRTUNE and efficiency of it. Section V-D delineates the contribution of each component of STRTUNE, accompanied by visual representations. Finally, Section V-E demonstrates the practicality of STRTUNE in real-world vulnerability search.

B. Effectiveness

We use four different tasks to evaluate our work: (1) XO (Cross-Optimization): Function pairs have different optimizations but use the same compiler, compiler version, and architecture. (2) XC (Cross-Compiler): Function pairs have different compilers, compiler versions, and optimizations, but use the same architecture and bitness. (3) XA (Cross-Architecture): Function pairs have different architectures and bitness, but use the same compiler, compiler version, and optimizations. (4) XM (Mixed): Function pairs come from any combination of architecture, bitness, compiler, compiler version, and optimization.

TABLE I

RESULTS OF OUR EXPERIMENTS ON THE TASKS FOR XA, XO, AND XM CONTAINING DATASETS OF ALL COMPILATION CONFIGURATIONS FOR POOLSIZE = 100 AND POOLSIZE = 1,000 RESPECTIVELY. TASKS FOR XO, XC ON THE X64 DATASETS ARE DENOTED AS X64-XO AND X64-XC.WE SET POOLSIZE = 100 FOR X64-XO AND POOLSIZE = 100/1,000 FOR X64-XC. THE METRICS ARE RECALL@1/MRR10

	poolsize=100					poolsize=1,000			
	XA	XO	XM	x64-XO	x64-XC	XA	XO	XM	x64-XC
SAFE	0.098/0.219	0.160/0.300	0.110/0.225	0.180/0.327	0.124/0.253	0.019/0.045	0.030/0.074	0.015/0.034	0.013/0.040
Zeek	0.257/0.416	0.281/0.428	0.212/0.346	0.258/0.419	0.193/0.338	0.052/0.111	0.087/0.164	0.045/0.091	0.045/0.090
GMN	0.658/0.766	0.589/0.708	0.465/0.601	0.684/0.788	0.422/0.568	0.450/0.580	0.472/0.561	0.276/0.383	0.169/0.258
VulHawk	0.113/0.381	0.094/0.360	0.103/0.348	0.328/0.579	0.268/0.539	-	-	-	-
Trex	0.013/0.046	0.675/0.744	0.160/0.206	0.792/0.842	0.623/0.730	0.002/0.004	0.527/0.579	0.110/0.131	0.299/0.409
jTrans	-	-	-	0.677/0.749	0.562/0.659	-	-	-	0.314/0.396
STRTUNE	**0.836/0.897**	**0.766/0.837**	**0.625/0.729**	**0.836/0.882**	**0.648/0.754**	**0.665/0.730**	**0.537/0.620**	**0.409/0.502**	**0.363/0.473**

Note: The "-" for jTrans is due to the fact that the model itself only analyzes x64 binaries. The "-" for VulHawk is due to the 24-hour time limit, within which it still cannot output all results.

For XO, XA and XM, we select 1,000 functions as query functions and randomly pick 100 and 1,000 functions that match the respective compilation setting variances as the function pool. Since jTrans is Binary Code Similarity Detection (BCSD) approaches set for x64, we set additional tasks of XO and XC function retrieval tasks on x64. For x64-XO task, we set the pool size to 100, and for x64-XC task, we set the pool size to 100 and 1,000.

Table I displays the results of the models tested in various tasks. The results indicate that STRTUNE outperforms other baselines in all tasks, achieving the highest Recall@1 and MRR@10. Specifically, in the x64-XO, x64-XC (poolsize = 100), and x64-XC (poolsize = 1,000) tasks, STRTUNE demonstrates Recall@1 improvements of 23.5%, 15.3%, and 15.6% compared to one of the latest models, jTrans. Compared to the earlier work SAFE and Zeek, STRTUNE shows approximately two and three times improvements respectively in Recall@1. Compared to GMN, STRTUNE showcases improvements of 47.8%, 13.8%, 48.2%, and 114.8% in the XA, XO, XM, and x64-XC (poolsize = 1,000) tasks, suggesting that directly using CFG as a function graph representation might not be appropriate, and our proposed graph representation can better reflect the features of functions.

Compared to VulHawk, our model shows a recall improvement of 2.54 times in Recall@1 and a 2.52 times increase in MRR@10. As shown in Figure 7a and Figure 7b, VulHawk's Recall@5 is slightly higher than STRTUNE. However, as the value of K increases, its performance is not as good as ours. Its accuracy depends on a comprehensive training dataset. VulHawk mainly uses the dataset in respect of vulnerability, while the distribution of our dataset may differ from theirs. This could cause VulHawk to lack in robustness when the predicted compilation configuration is incorrect, which directly impacts the results.

Figure 7 illustrates the Recall@K curves of models concerning different values of K. STRTUNE ranks high in both Recall@1 and convergence speed. In the tasks of poolsize = 100, STRTUNE essentially converges to nearly 1 at Recall@15, followed by GMN and jTrans. However, when poolsize is set to 1,000, as shown in Figure 7c, jTrans surpasses GMN when K is small. This might be due to the increase of poolsize, GNN-based GMN, compared to NLP-based jTrans, is more prone to erroneously matching basic blocks that are semantically different but syntactically similar. In other words, basic blocks with different computational content may exhibit the same combinations of opcodes, leading GMN to mistakenly identify them as similar nodes. Figure 7d, 7e and 7f show the exact advantages of STRTUNE over baselines.

Summary: STRTUNE ranks the first in all BCSD tasks, affirming its effectiveness in addressing cross-architecture, cross-optimization level, and cross-compiler challenges.

C. Runtime Efficiency

We present the efficiency of STRTUNE at each stage in Figure 8. 'Node Pre-training' represents the pre-training time of the RoBERTa model on given slices. 'Node Fine-tuning' signifies the fine-tuning time of our RoBERTa model. These two durations related to node representation are one-time efforts, hence slightly longer time can be accepted. 'Structure Training' denotes the time for training the model for 10 epochs. As for lifting and inferring time, we consider the average time tested for x64 dataset, totaling 100,000 pairs of functions under three rounds of execution. It is worth noting that due to RoBERTa training process where embeddings for slices are already stored in the database, the model's time consumption during training and inferring stages is less than expected.

Figure 9 shows average inferring time compared with baselines for 100,000 pairs of functions under three rounds of execution. It can be observed that while Zeek consumes the least time, its detection accuracy is comparatively lower, which is not to our expectation. VulHawk consumes the most time because its provided interface can only compute the similarity between all function pairs given two binaries. In our one-to-many testing scenario, part of computation in VulHawk are redundant. An appropriate interface might reduce its inferring time. STRTUNE's time falls within the medium

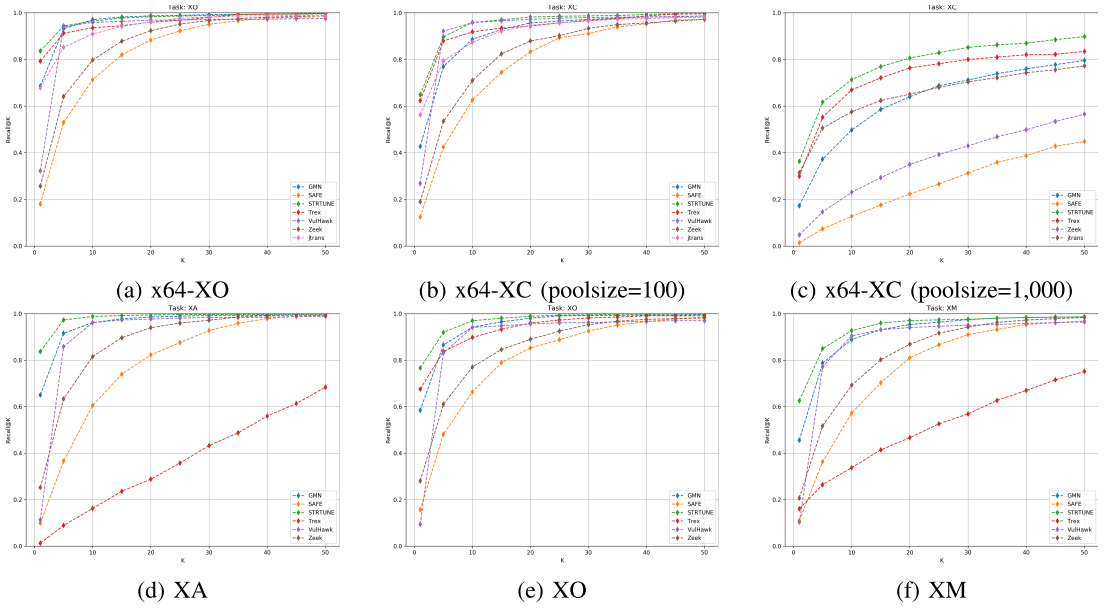


Fig. 7. Recall on different K for tasks.

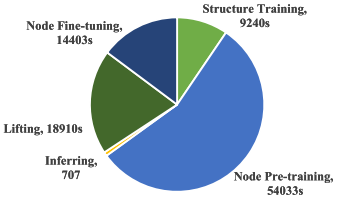


Fig. 8. Efficiency of STRTUNE for execution of each part. Lifting and Inferring denote the average time taken for 100,000 pairs of functions under three rounds of execution. The time unit is seconds.

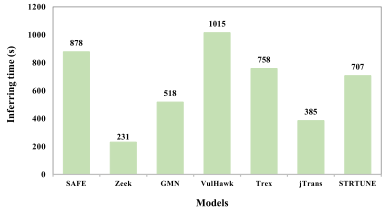


Fig. 9. Average inferring time compared with baselines for 100,000 pairs of functions under three rounds of execution.

range, being 36% more time-consuming than GMN. This is mainly attributed to the time taken for RoBERTa embedding of untreated slices. Due to the improved performance of STRTUNE compared to GMN, the additional time required is acceptable.

Summary: STRTUNE maintains high precision while incurring comparable time overhead contrast to baselines.

D. Ablation Study

We analyze the contributions of each part of our model through ablation analysis. We mainly test the models on the XO/XA/XC/XM tasks. We select the following input representation and embedding models as comparison:

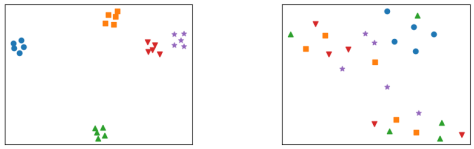
- w/o Data Parallel/Data Dependence/Jump/Sequential: We individually removed each flow type to observe its contribution to the overall performance, and the initial features of the removed flow type were set to 1. Nodes remain as code slices in the graph, and the node embedding method remains unchanged.
- w/o Fine-tuning: We change the node embedding method to an unfine-tuned RoBERTa model.
- w/o Slicing: The input graph is obtained directly from Microcode and nodes are basic blocks. The method of node embedding remains the same, while edges retain only two types: control flow and data dependence. We set this input to verify the effectiveness of our proposed graph presentation.
- w/o Attention Coefficient: We transform the structural learning GMN into a regular Graph Neural Network (GNN) model.

As shown in Table II, each component of STRTUNE contributes to the improvement of BCSD accuracy. The results of w/o Slicing are the poorest, indicating that regarding basic blocks as graph nodes are inadequate. In contrast, our proposed graph representation enhances Recall@1 by 69%, validating the effectiveness of node slicing based on data dependence. Model w/o Fine-tuning has a reduction of around 0.2 in Recall@1 when compared. This demonstrates that pre-training RoBERTa alone cannot effectively learn slice representations with similar semantic but different syntax, highlighting the importance of fine-tuning step. With regard to structure learning, employing GMN results in a 7.3% improvement compared to GNN, indicating that attention coefficients are beneficial for graph representation learning. In fact, STRTUNE focuses on the cross-graph node matching, which can be amplified by the attention coefficients through several rounds of node propagations, promoting efficiency in BCSD. With respect to flow type, every type of flow contributes to our overall performance, with data parallel contributing the most to the effectiveness, followed by data dependence. Data parallel and data dependence enrich the data-level relationships between slices, thereby enhancing the robustness of STRTUNE.

We can show the effectiveness of the fine-tuned RoBERTa model on representing code slices through visualization. We utilized Uniform Manifold Approximation and Projection

TABLE II
RESULTS OF THE ABLATION STUDY ON THE TASKS FOR XA, XO, XC AND XM WITH POOLSIZE SET TO 100. THE METRICS ARE RECALL@1/MRR10

	XA	XO	XC	XM
w/o Data Parallel	0.828/0.876	0.757/0.813	0.577/0.694	0.616/0.706
w/o Data Dependence	0.829/0.881	0.759/0.818	0.576/0.689	0.617/0.711
w/o Jump	0.832/0.886	0.761/0.823	0.583/0.704	0.620/0.716
w/o Sequential	0.833/0.891	0.764/0.828	0.580/0.699	0.621/0.721
w/o Fine-tuning	0.644/0.725	0.619/0.698	0.416/0.526	0.485/0.593
w/o Slicing	0.493/0.642	0.473/0.610	0.336/0.495	0.371/0.527
w/o Attention Coefficient	0.779/0.849	0.728/0.806	0.569/0.679	0.621/0.725
STRTUNE	**0.836/0.897**	**0.766/0.837**	**0.585/0.709**	**0.625/0.726**



(a) Code slices embedded by the fine-tuned RoBERTa of STRTUNE.

(b) Code slices embedded by the 'roberta-base' model used for NLP embedding.

Fig. 10. Visualization of clustering of code slices with different semantic. We select five syntactically different slices from different compilation configurations for five diverse computational contents and use Uniform Manifold Approximation and Projection (UMAP) for dimensionality reduction.

(UMAP) to perform non-linear dimensionality reduction on code slice embeddings. Figure 10 presents clustering of several code slices. Points with the same color represent different slices with the same semantic (from the same source code, for example, $while(o \rightarrow nb\_stream\_maps > v7)$ shown before). It is evident that our fine-tuned RoBERTa model can generate embeddings that bring semantically similar slices closer together. In contrast, the SOTA NLP model 'roberta-base' when used to embed code slices, cannot cluster them very effectively. This improvement provides a strong foundation for structure-level learning for functions.

We also visualize the attention coefficients between a pair of real-world function graphs under different rounds of propagation layers in GMN in Figure 11. To observe the changes in attention weights more clearly, we ignore the distinction between different flow types and primarily focus on how the attention mechanism can concentrate on similar slices between the constructed graphs. Specifically, Slice 2 in blue and Slice 6 in gray both refer to the source code void $*$ dst $=$ (uint8_t$*$)o $+$ po $\rightarrow$ u.off. The network focuses on the edges connecting them and the similarity between the corresponding slices. Specifically, Slice 2 in blue is connected to its preceding node Slice 16, along with the jump flow between them, and Slice 6 in gray is connected to its preceding node Slice 1, and the corresponding jump flow. Since Slice 16 and Slice 1 are semantically equivalent, in the following rounds, the network will assign higher attention coefficients to Slice 2 and Slice 6, as shown in Figure 11b. In this case, when the similarity score is calculated at the end, this pair of functions will receive a higher similarity score.

Summary: Each component of STRTUNE contributes positively to the overall performance and slicing based on data dependence contributes the most to the precision.

E. Real-World Vulnerability Search

In order to test the STRTUNE's generalizability in real-world scenarios, we conduct a real-world vulnerability search on STRTUNE and some baselines. We select six CVEs (Common Vulnerabilities and Exposures) from OpenSSL 1.0.2d, which include eight functions. The functions are compiled for four architectures and serving as query functions. We choose NetGear R7000 (ARM-32) from NetGear and TP-Link Deco M4 (MIPS-32) from TP-Link as the function repositories. Three query functions are present in the NetGear R7000, and seven query functions in the TP-Link Deco M4. For each firmware, we evaluate the ranking of the corresponding functions in four architectures.

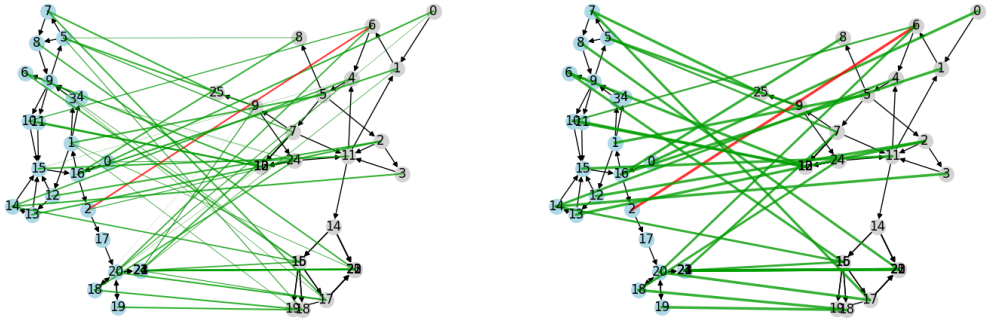
Table III and Table IV respectively show the search results of the CVE functions in the two firmwares. Each score represents the ranking of the corresponding ground truth function in the function repository, compiled for x86, x64, ARM-32bit, and MIPS-32bit.

SAFE and Zeek exhibit relatively low accuracy in both firmware detection tasks, even ranking the ground truth function beyond 100 for more than half CVE instances. In contrast, GMN achieves higher accuracy, almost performing as well as STRTUNE in searching for CVE-2016-0797 and CVE-2016-0797, but performing poorly for other CVEs. STRTUNE enables highly accurate matching, achieving a top ranking for five CVEs across four architectures, with the lowest search result ranking being 9. STRTUNE surpasses the searching accuracy over other baselines, which illustrates the high robustness of slice-based representation. We interpret that STRTUNE can effectively improve vulnerability search tasks in real-world scenarios, significantly reducing the cost of manual function comparison for users.

We analyze false positives in the aforementioned experiments, which means that functions compiled from different source code are ranked at 1 during function searching. For instance, in the case of CVE-2016-2182 compiled for ARM-32bit and searched in TP-Link Deco M4, *OBJ_dup* ranks at the top. We check the node matching pattern of two functions during the inferring phase. Due to our retention of all call function names, the code slices involving calls to *OPENSSL_malloc* and *OPENSSL_free*, along with their associated flow types, are nearly identical. This prompts STRTUNE to focus on these specific local details. GMN tends to assign high attention coefficients to these two functions, resulting in a higher similarity score. In contrast, the ground truth function ranks at 4 due to the failure to match slices with similar computational contents, thus leading to a false negative.

Since such cases account for a significant portion of false positives and false negatives, We can incorporate dissimilar slices from false positives into the negative samples, with similar slice pairs from false negatives adding as positive samples for fine-tuning and then re-fine-tune RoBERTa model. The GMN could assign lower attention to those dissimilar slices, thus giving a low similarity score to these functions.

In another case, we identify *EVP_DigestSignInit* as a false positive for *EVP_DigestVerifyInit*. These two functions are



(a) Attention after 1 round of propagation. (b) Attention after 10 rounds of propagation.

Fig. 11. Visualization of attention coefficients between a pair of similar functions during inference. The intensity and thickness of the line correspond to the attention coefficient, with darker and thicker lines indicating a higher degree of similarity between the connected slices.

TABLE III

RESULTS OF REAL-WORLD VULNERABILITY SEARCHING EXPERIMENTS IN TP-LINK. THE FOUR NUMBERS MEAN THE RANKSOF GROUND TRUTH FUNCTIONS WHEN THE QUERY FUNCTION IS COMPILED FOR THE ARCHITECTURES OF X86, X64, ARM-32BIT AND MIPS-32BIT RESPECTIVELY

CVE	CVE-2016-2182 (BN_bn2dec)	CVE-2016-0797 (BN_dec2bn)	CVE-2016-0797 (BN_hex2bn)	CVE-2016-2105 (EVP_EncodeUpdate)	CVE-2019-1563 (PKCS7_dataDecode)	CVE-2016-0798 (SRP_VBASE_get_by_user)	CVE-2016-2176 (X509_NAME_oneline)
SAFE	118;92;62;57	68;77;79;44	88;172;63;70	61;333;29;93	243;52;153;58	13;27;152;5	63;16;18;29
Zeek	86;71;51;102	5;1;5;51	434;2;319;281	10;4;23;**1**	**3;1**;1;3	11;94;14;19	3;1;1;4
GMN	9;35;104;39	1;1;1;2	1;1;1;1	79;6;2;**1**	17;92;36;1	1;1;2;6	29;24;16;61
STRTUNE	**1;1;4;1**	**1;1;1;1**	**1;1;1;1**	**1;1;1**;4	9;3;**1;1**	**1;1;1;1**	**1;1;1;1**

TABLE IV

RESULTS OF REAL-WORLD VULNERABILITY SEARCHING EXPERIMENTS IN NETGEAR. THE FOUR NUMBERS MEAN THE RANKSOF GROUND TRUTH FUNCTIONS WHEN THE QUERY FUNCTION IS COMPILED FOR THE ARCHITECTURES OF X86, X64, ARM-32BIT AND MIPS-32BIT RESPECTIVELY

CVE	CVE-2016-2182 (BN_bn2dec)	CVE-2016-6303 (MDC2_Update)	CVE-2019-1563 (PKCS7_dataDecode)
SAFE	42;42;128;17	183;110;139;384	99;3;28;26
Zeek	35;59;35;22	252;62;5;345	3;7;15;10
GMN	16;17;10;53	2;3;1;7	108;1;**1**;2
VulHawk	1;11;5;3	1;2;1;**1**	1;1;**1**;123
STRTUNE	**1;1;1;1**	**1;1;1**;2	**1;1**;6;**1**

used for digital signatures and are almost identical in terms of their functionality, due to which they are considered similar functions by STRTUNE.

Summary: STRTUNE demonstrates high performance in distinguishing vulnerable functions compared to SOTA methods in real-world scenarios.

VI. DISCUSSION

This section discusses the limitations of our work and future work.

A. Choice of IR

STRTUNE captures the semantic of functions based on Microcode. During the acquisition phase of Microcode, we utilize IDA Pro for decompilation, and the results of our work partly depend on the accuracy of this process. Additionally, due to version limitations of IDA Pro, STRTUNE is unable to obtain Microcode of binaries complied from MIPS64, leading to certain analytical limitations. Ghidra could also be used as a tool for obtaining IR for analysis.

B. Model Efficiency and Enhancements

STRTUNE is based on the pre-training and fine-tuning of models. How to effectively use NLP models for encoding or adopting more efficient training methods is one of our future directions. Also, in the structure training phase, it is also possible to represent the relationship between nodes using the attention mechanism of Query, Key, Value. Although this approach essentially involves increasing a certain amount of trainable parameters, we still consider it as one of our future work.

C. Hierarchical Fusion of Learning Representations

Since our work trains nodes and structures separately, we could also use a Hierarchical Attention Network for learning. It mainly includes two levels of attention mechanisms, which we can apply to instruction-level and structure-level attention, respectively, to enhance the model's understanding and representation capabilities of the entire function composition.

D. Problems of Same Code Behavior

In the real-world vulnerability search, we discovered that functions with different names but the same code logic introduce some noise into the analysis. How to effectively remove this part of the noise is also a potential future work.

VII. RELATED WORK

The related work in the domain of Binary Code Similarity Detection (BCSD) is various. In Asm2Vec [16], the

authors base their approach on the word2vec model, splitting assembly instructions into character-level units and training character-level embeddings using unsupervised methods. Like the word2vec-based model, SAFE [4] employs a seq2seq model-based NLP encoder. Ahn et al. [19] construct a pre-trained BERT model based on MLM and NSP tasks, along with fine-tuning using additional functions. In jTrans [20], the authors extract token embeddings from the normalized instruction set and add position embeddings, considering the positional relationships of source and target tokens for jump instructions. Trex [21] focuses on the concept of Micro-trace and utilizes transfer learning to learn the semantic of binary functions. In recent research [36], Xu et al. point out that compilation introduces instruction distribution bias, proposing classification importance and semantic importance for instructions to improve the accuracy of jTrans and Trex.

Feng et al. [1] introduce ACFG, where attributes mainly fall into statistical and structural categories. Gemini [5] employs a variant of the Structure2vec algorithm for each node in ACFG and uses Siamese networks for supervised learning. Zeek [22], based on data dependence, computes hash values for sets of instructions with data dependence, serving as basic block feature vectors. Li et al. [23] propose GMN, which uses attention mechanisms to reduce the impact of dissimilar nodes on the results. Yu et al. [24] combine the BERT algorithm to extract features for each basic block and proposed a framework with three components. In Sem2vec [25], the authors use symbolic constraints as node information and aggregate them using RoBERTa models and some structural network techniques. VulHawk [3], on the other hand, determines the optimization level and compiler of binaries from an entropy perspective and proposes a binary search framework that combines coarse-grained search with fine-grained filtering. Recently, CLAP [37] leverages contrastive language-assembly pre-training to improve the transferability of binary code representation learning by aligning assembly code with natural language explanations. BinCola [38] also leverages diversity-sensitive contrastive learning to improve performance across varying compilations. CEBin [39] combines embedding-based and comparison-based approaches to enhance accuracy, particularly for large-scale vulnerability detection. He et al. [40] presents a semantics-oriented graph representation to better capture binary semantic.

VIII. Conclusion

In this paper, we propose StrTune, a novel approach that fine-tunes the representation of code slices segmented based on data dependence. StrTune emphasizes non-interference in computation by introducing flow types to depict relationships among slices, presenting a novel graph representation for binary functions. We employ a Siamese network to fine-tune pairwise slices with the same computational contents. We use GMN for function-level features. The attention coefficient focuses specifically on similar nodes across graphs. Our experimental results demonstrate that the performance of StrTune surpasses state-of-the-art models for BCSD, proving its effectiveness and practicality.

Acknowledgment

The authors would like to thank the editor and anonymous reviewers for their valuable feedback to improve the manuscript.

References

[1] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, pp. 480–491, 2016.

[2] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2018, pp. 896–899.

[3] Z. Luo et al., "VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–30.

[4] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 309–329.

[5] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.

[6] B. Liu et al., "α diff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2018, pp. 667–678.

[7] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.

[8] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 406–415.

[9] P. Shirani et al., "B in a RM: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. 15th Int. Conf.*, 2018, pp. 114–138.

[10] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware VariantDetection," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 307–317, Jul. 2014.

[11] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. 8th Int. Conf. Softw. Secur. Rel. (SERE)*, Jun. 2014, pp. 78–87.

[12] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 611–620.

[13] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 81–96.

[14] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 342–352.

[15] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security patch analysis for binaries towards understanding the pain and pills," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 462–472.

[16] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

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

[18] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, "Exploring GNN based program embedding technologies for binary related tasks," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, May 2022, pp. 366–377.

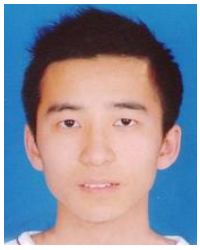
[19] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with BERT-based transferable similarity learning," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Dec. 2022, pp. 361–374.

[20] H. Wang et al., "Jtrans: Jump-aware transformer for binary code similarity," 2022, *arXiv:2205.12713*.

[21] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," 2020, *arXiv:2012.08680*.

[22] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proc. 13th Workshop Program. Lang. Anal. Secur.*, Jan. 2018, pp. 42–47.
[23] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3835–3845.
[24] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, 2020, pp. 1145–1152.
[25] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu, "Sem2vec : Semantics-aware assembly tracelet embedding," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 1–34, Oct. 2023.
[26] Z. Zhang et al., "Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 2365–2382.
[27] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *J. Log. Comput.*, vol. 2, no. 4, pp. 511–547, 1992.
[28] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. Softw. Eng.*, vol. 16, no. 9, pp. 965–979, Sep. 1990.
[29] (2008). *Ida Pro*. [Online]. Available: https://hex-rays.com/ida-pro/
[30] (2008). *Ida Python Documentation*. [Online]. Available: https://hex-rays.com/products/ida/support/idapython_docs/
[31] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.
[32] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Syst. Demonstrations*, 2020, pp. 38–45.
[33] A. Hagberg, P. Swart, and D. S. Chult, "Exploring network structure, dynamics, and function using network," Tech. Rep., 2008.
[34] M. Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*.
[35] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2099–2116.
[36] X. Xu et al., "Improving binary code similarity transformer models by semantics-driven instruction deemphasis," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw.*, 2023, pp. 1106–1118.
[37] H. Wang et al., "CLAP: Learning transferable binary code representations with natural language supervision," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, vol. 35, Sep. 2024, pp. 503–515.
[38] S. Jiang, C. Fu, S. He, J. Lv, L. Han, and H. Hu, "BinCola: Diversity-sensitive contrastive learning for binary code similarity detection," *IEEE Trans. Softw. Eng.*, vol. 50, no. 10, pp. 2485–2497, Oct. 2024.
[39] H. Wang et al., "CEBin: A cost-effective framework for large-scale binary code similarity detection," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, vol. 5, Sep. 2024, pp. 149–161.
[40] H. He et al., "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *Proc. 33rd USENIX Secur. Symp.*, 2024, pp. 1–24.

Kaiyan He received the B.S. degree in computer science and technology from Xidian University in 2022. She is currently pursuing the M.S. degree with Shanghai Jiao Tong University. Her research interests include software security and artificial intelligence.



Yikun Hu received the B.S. degree from South China University of Technology and the Ph.D. degree from Shanghai Jiao Tong University. He is currently an Assistant Research Fellow with the School of Cyber Science and Engineering, Shanghai Jiao Tong University. His research interests include binary program analysis and software engineering.

Xuehui Li is currently pursuing the B.S. degree and plans to pursue the master's degree with Shanghai Jiao Tong University (SJTU). She is a senior in the IEEE Pilot Program in Information Security, SJTU. Her research interests include artificial intelligence, AI applications, and AI security.

Yunhao Song is currently pursuing the B.S. degree in information security with Shanghai Jiao Tong University. His research interests include cybersecurity and software security.

Yubo Zhao received the B.S. degree from Harbin Institute of Technology. He is currently pursuing the Ph.D. degree with the School of Cyber Science and Engineering, Shanghai Jiao Tong University. His research interests include program analysis and machine learning.

Dawu Gu (Member, IEEE) received the B.S. degree in applied mathematics from Xidian University, Xi'an, China, in 1992, and the M.S. and Ph.D. degrees in cryptography in 1995 and 1998, respectively. He is currently a Chair Professor with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University (SJTU), Shanghai, where he leads the Laboratory of Cryptology and Computer Security (LoCCS). He has over 150 scientific papers in academic journals and conferences and owns 28 innovation patents. His research interests include crypto algorithms, crypto engineering, and system security. He was the Winner of the Chang Jiang Scholars Distinguished Professors Program made by the Ministry of Education of China in 2014. He won the National Award for Science and Technology Progress in 2017.