GTrans: Graph Transformer-Based Obfuscation-resilient Binary Code Similarity Detection

Yun Zhang*, Yuling Liu*, Ge Cheng† Bo Ou*
*College of Computer Science and Electronic Engineering, Hunan University
†School of Computer Science & School of Cyberspace Science, Xiangtan University
yunzhangcn@outlook.com, yuling_liu@hnu.edu.cn, chengge@xtu.edu.cn and oubo@hnu.edu.cn

***Abstract*—In the field of computer security, binary code similarity detection is a crucial for identifying malicious software, copyright infringement, and software vulnerabilities. However, obfuscation techniques not only changes the structure and features of the code but also effectively conceal its potential malicious behaviors or infringing nature, thereby increasing the complexity of detection. Although methods based on graph neural networks have become the forefront technology for solving code similarity detection due to their effective processing and representation of code structures, they have limitations in dealing with obfuscated function matching, especially in scenarios involving control flow obfuscation. This paper proposes a method based on Graph Transformers aimed at improving the accuracy and efficiency of obfuscation-resilient binary code similarity detection. Our method utilizes Transformers to extract global information and employs three different encodings to determine the relative importance or influence of nodes in the CFG, the relative position between nodes, and the hierarchical relationships within the CFG. This method demonstrates significant adaptability to various obfuscation techniques and exhibits enhanced robustness and scalability when processing large datasets.**

I. Introduction

In the field of computer security, binary code similarity detection is widely used in various scenarios such as malware identification, software plagiarism detection, and vulnerability discovery. According to the 2023 Synopsys report [1], 84% of the audited codebases contained at least one known open-source vulnerability, an increase of nearly 4% compared to the previous year. This problem is particularly severe in the field of the Internet of Things (IoT), where high-risk vulnerabilities have increased by 130% since 2018, and in 2023, 53% of audited applications contained high-risk vulnerabilities. Given that IoT software is deployed on thousands or even millions of devices and is always on, vulnerabilities are amplified across a wider attack surface. However, code obfuscation techniques can not only obscure the logic and structure of the code, making it difficult to understand its true purpose but can also hide malicious behavior within the code, significantly increasing the difficulty of detection.

[1] https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23006
www.ndss-symposium.org

In recent years, methods based on Graph Neural Networks (GNNs) have achieved significant results in binary code similarity detection across optimization levels and architectures. For example, Focus [9] designed a novel GNN, using a customized learning model and multi-head attention mechanism to extract semantic and key features; GMN [18] adds cross-graph attention in GNN to compute graph matching information. GNNs typically consider only the direct neighbors of a node in each iteration, which leads them to capture local structural features better than the global features of the entire graph. However, obfuscation techniques (such as fake control flow, control flow flattening) can significantly change the program's control flow graph (CFG), potentially misleading GNNs in interpreting local information. This misleading design could cause GNN-based methods to make incorrect inferences when conducting obfuscation-resilient similarity detection.

Furthermore, current research methods have not considered the differences between the graph structure of binary code and the topological structure of traditional graph data. The CFG of binary code contains not only the logical paths of program execution but also a unique hierarchical structure composed of dominance relations. This hierarchical structure plays a key role in the analysis of binary code. It not only helps in understanding the execution paths of the program but also facilitates identifying critical decision points within the program.

For this purpose, we proposed an obfuscation-resilient binary code similarity detection method based on the graph Transformer. This method leverages the Transformer model's ability to capture global information for graph embedding and uses spatial and hierarchical encoding to capture invariant information after control flow obfuscation. Specifically, first, the method employs the self-attention mechanism of the Transformer to establish connections between any nodes in the graph, capturing dependencies over longer node distances without being constrained by the adjacent node aggregation method of graph neural networks.

Secondly, we added centrality encoding, spatial encoding, and hierarchical encoding to the Transformer to capture the relatively stable logical and semantic information in binary code during control flow obfuscation and code structure transformation. The centrality encoding is used to capture the importance of nodes in the CFG. Some nodes in the graph may play more important roles in the program's operation and logic, such as loops, conditional judgments, or critical computations. For this, we use the number of nodes dominated by a node as a measure of its importance, and encode this information to add to the node's feature vector.

Spatial encoding and hierarchical encoding are used to capture the relationships between nodes in the CFG. Unlike structured data, the nodes in graph data are connected by edges. We model this topological structure, using the spatial relationships between nodes as their relative positions, and assign a learnable embedding as a bias term for attention. Hierarchical encoding focuses on the dominance relationships between nodes in the CFG. We use the dominator tree, constructed from dominance relationships, as a concrete representation of these relationships, and compute the path lengths between nodes in the dominator tree as a representation of the hierarchical relationship, likewise assigning an embedding vector to add to the attention.

We studied the performance and accuracy of our method under five different obfuscation strategies. The experimental results show that our method's average AUC score exceeded 0.91, reaching as high as 0.95, and outperformed existing methods.

The main contributions of this paper are:

1) We propose an obfuscation-resilient binary code similarity detection method based on the graph Transformer, which is used to capture global information of the graph. This method utilizes three graph encoding strategies to obtain invariant information of binary code after obfuscation, especially control flow obfuscation.
2) In addition to adding relative position encoding of nodes in the CFG, our method also introduces a hierarchical encoding unique to binary code.
3) Extensive evaluations show that the performance of GTrans in detecting function similarity under different obfuscation strategies surpasses the latest binary similarity detection methods. The experiments cover various aspects of obfuscation techniques, including substituting instructions, splitting basic blocks, adding pseudo-logic, and completely disrupting the original CFG.

The rest of this paper is organized as follows: In Section 2, we describe GTrans in detail, then in Section 3, we provide specific details of the experiments and report the experimental results. Section 4 discusses related work.

II. Design of GTrans

We first preprocess the binary functions, which includes using disassembly tools such as IDA Pro [2] and ANGR [28] to obtain the CFG, constructing the dominator tree based on the control flow graph, and treating the instructions within basic blocks as an instruction sequence. We adopt natural language processing methods to extract semantic information, generating initial feature vectors for the nodes.

[2]https://hex-rays.com/ida-pro/

Next, we feed a pair of CFG and dominator tree into a Siamese network, which uses graph Transformer technology to extract features from the input data, employing cosine similarity for the measurement of similarity, ultimately outputting a similarity score between two functions. In order to accurately capture the logical and semantic information of the program amidst control flow obfuscation and code structure changes, we have designed three types of encoding in the graph Transformer:

Centrality encoding is used to obtain the relative importance or influence of nodes in the CFG. In the CFG, when a node dominates a significant number of other nodes, it usually means that the code segment of that node has a critical control effect on multiple parts of the entire program and may be considered a key segment or decision node of the program.

Spatial encoding is used to capture the relative positional relationships between nodes in the graph. Graph data differs fundamentally from other types of structured data, such as text or images. In graph data, nodes exist in a non-Euclidean space and are defined by edges that connect them. Based on this characteristic, spatial encoding is used to capture the topological relationships of nodes in this non-Euclidean space.

Hierarchical encoding is used to capture the hierarchical relationships in the CFG. Unlike the topology depicted by traditional graph data, CFGs also contain a hierarchical structure built from dominance relationships. To capture this hierarchical information, we represent the dominance relationships between basic blocks by constructing a dominator tree of the CFG.

A. Dominator Tree

The dominator tree is a tree-like structure that represents the dominance relationships between nodes in a CFG. It is used to extract structural features of binary code, such as centrality and hierarchy. As shown in Figure 1, if every path from the entry node of the CFG to node n passes through node d, then node d dominates node n. The dominator tree more vividly describes this domination information. In this tree structure, the entry node serves as the root node. Each node d only dominates its descendant nodes in the tree, and this dominance relationship is entirely based on the corresponding relationships in the CFG.

The dominator tree can help understand the control flow structure within a program and the program's execution paths. In the dominator tree, each successor node of a node is dominated by that node, meaning to reach these successor nodes, the program must pass through that node. Simultaneously, the dominator tree is a foundation for static program analysis, which can be used to determine critical decision points in a program, such as which lines of code influence a particular line of the program, or which lines must be executed before executing a specific line of code.

B. Graph Transformer

Figure 2 depicts the architecture of the graph Transformer. In addition to the three types of graph encoding added, we also choose to apply layer normalization (LN) [33] before the multi-head self-attention (MHA) and feed-forward network (FFN) blocks. Specifically:

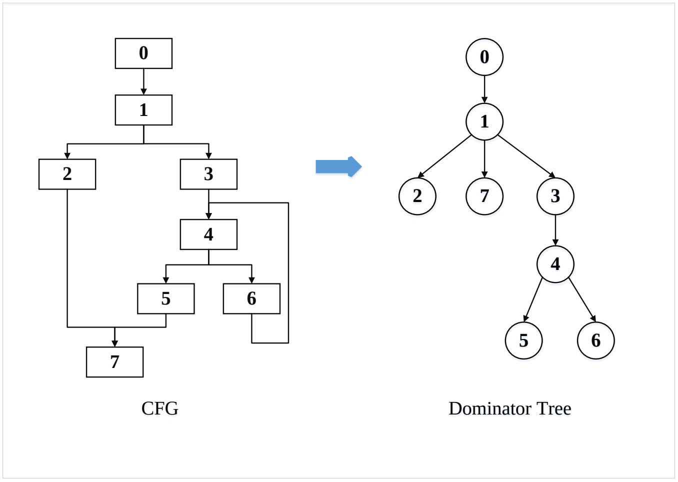

Fig. 1: CFG and Dominant Node Tree

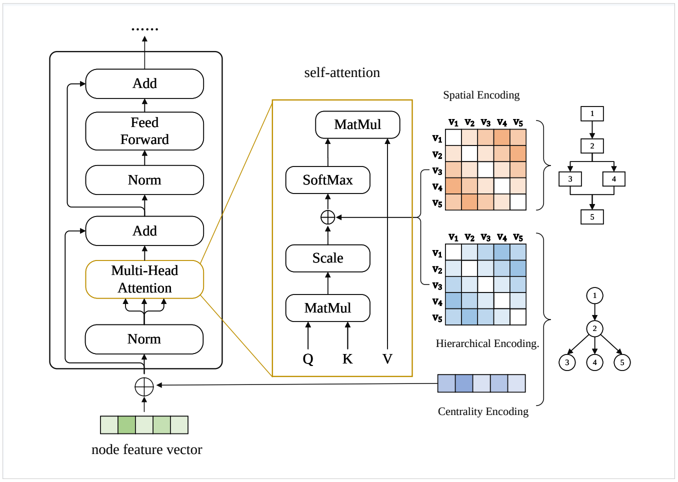

Fig. 2: Graph Transformer Architecture

$$h^{'(l)} = MHA(LN(h^{(l-1)})) + h^{(l-1)} \quad (1)$$

$$h^{(l)} = FNN(LN(h^{'(l)})) + h^{'(l)} \quad (2)$$

In the formula, $h^{(l-1)}$ represents the feature vector generated by the $(l-1)$th layer encoder, and $h^{'(l)}$ is the intermediate result of the lth layer.

Centrality Encoding. In graph theory, the centrality of a node is often considered a key factor in understanding the graph, but this can be easily overlooked in the traditional Transformer's self-attention mechanism. We use the number of dominated nodes as a measure of centrality. If a node dominates many other nodes, it may imply that the node plays a key role in the control flow, such as being the entry point of a loop or the start of a conditional branch. Considering that this centrality encoding is applied to each individual node, we choose to incorporate it into the node's features as input, as follows:

$$\hat{x}_i = x_i + z_{\varphi_d(v_i)} \quad (3)$$

In the diagram, x_i represents the feature vector of node v_i , and $z \in R^d$ is a learnable embedding vector based on the number of other nodes dominated by v_i , denoted as $\varphi_d(v_i)$. By integrating centrality encoding into the input, the self-attention mechanism can more effectively capture the node importance signals between the query and the key. Thus, this method can not only capture semantic relevance in the attention mechanism but also identify the importance of nodes.

Spatial Encoding. In a CFG, nodes exist in a non-Euclidean space, and their connections are defined by edges. However, this mechanism raises a problem: the model must provide explicit indications of different positions, or appropriately encode the dependency on positions (such as locality) at each layer. When handling sequential data, adding embeddings for each position is a common practice. However, in a graph structure, nodes are not arranged in a sequence; they may be scattered across a multi-dimensional space and connected by edges. To better capture the graph structure information in the model, we introduce a specific spatial encoding method to describe the position of nodes.

Specifically, the function $\varphi_s(i,j)$ is used to measure the relative spatial position between nodes v_i and v_j in the CFG. If two nodes are connected, $\varphi_s(i,j)$ corresponds to the distance of the shortest path between v_i and v_j. For disconnected nodes, the output of φ_s is defined as a special value, -1. For each output value of φ_s , we assign a learnable scalar $b_{\varphi_s(i,j)}$ and choose to consider this scalar as a bias term in the self-attention mechanism. Let A_{ij} represent the element at position (i,j) in the Query-Key product matrix A, then:

$$A_{ij} = \frac{(x_i W_Q)(x_j W_K)^T}{\sqrt{d}} + b_{\varphi_s(i,j)} \quad (4)$$

In this context, $b_{\varphi_s(i,j)}$ is a learnable scalar indexed by $\varphi_s(i,j)$ and is shared across all layers. With spatial encoding, each node within every layer of the Transformer can adaptively focus on all other nodes based on the graph structure information.

Hierarchical Encoding. Unlike conventional graph data, CFGs not only have the topological features of graphs but also integrate the hierarchical structure of trees, specifically the dominator tree. To capture this unique feature, we employ hierarchical encoding to describe the relative positioning of nodes within the tree structure. This method assigns a real-valued embedding vector to each pair of nodes based on their relative hierarchical positions. Specifically, for each pair of nodes (v_i, v_j), we use $\varphi_h(i,j)$ to describe the hierarchical position of node v_i relative to node v_j in the dominator tree. If there is a dominance relationship between two nodes, $\varphi_h(i,j)$ represents the path length between them. For pairs of nodes without a dominance relationship, the output of φ_h is set to a specific value, -1. We assign a learnable scalar $b_{\varphi_h(i,j)}$ to each output of φ_h and add it to the bias term of the attention module. That is, we incorporate hierarchical encoding in formula 4:

$$A_{ij} = \frac{(x_i W_Q)(x_j W_K)^T}{\sqrt{d}} + b_{\varphi_s(i,j)} + b_{\varphi_h(i,j)} \tag{5}$$

In the graph Transformer, spatial encoding and hierarchical encoding are implemented as two bias terms in the calculation of attention weights. This means that in estimating attention weights, in addition to considering the traditional contributions of queries, keys, and values, the spatial and hierarchical structure information of the CFG is also integrated.

Graph-Level Representation. To optimize the representation of the entire graph, we add a special graph-level representation node. In the CFG, this special node forms a unique connection with every node within the graph. In the dominator tree, this node has dominance over all other nodes, making the representation of the entire graph equivalent to the feature vector of this special node in the final hierarchical level. The graph-level representation node integrates the overall global information of the graph, avoiding the problem of excessive smoothing. At the same time, it enables global propagation of information for each node without the need for additional encoding. This optimization not only simplifies the representation of the graph but also enhances the representation and processing capabilities of graph information, thereby making the performance of binary function similarity detection more stable across multiple scenarios.

C. *Similarity Measurement*

Given a pair of inputs x_1 and x_2, the Siamese network extracts their feature representations $f(x_1)$ and $f(x_2)$, through a graph Transformer respectively. Then, the similarity between the two feature vectors is calculated using cosine similarity:

$$similarity = cos(f(x_1), f(x_2)) = \frac{f(x_1) \cdot f(x_2)}{\|f(x_1\| \, \|f(x_2\|} \tag{6}$$

The loss function uses binary cross-entropy, utilizing the difference between the cosine similarity scores and the true labels, with the loss for a batch of size N being:

$$L = -\frac{1}{N} \textstyle\sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right] \tag{7}$$

$$\hat{y}_i = \frac{1}{2}(similarity_i + 1) \tag{8}$$

III. Evaluation

To verify the advantages of our method in binary obfuscation scenarios, we selected the dataset from Trex [25] and conducted empirical analysis to compare the performance and accuracy of our method in detecting binary code similarity under various obfuscation and transformation techniques. Additionally, we conducted a series of ablation experiments to validate the significant role of structural encoding in enhancing model performance. These experimental results will further demonstrate the advantages of our method.

A. *Implementation*

GTrans is implemented using Python 3 and the PyTorch framework, with the preprocessing phase employing the ANGR [28] framework. The experimental environment is set up on the Ubuntu 18.04 operating system, with the server hardware configuration including a 3.20GHz Intel Core i7-8700 processor, 128GB of RAM, and 4 NVIDIA GeForce GTX 1060 graphics cards.

During the training phase of the model, the batch size was set to 128, with a learning rate of 2e-4, using the AdamW optimizer. The model had 32 attention heads, 6 encoders, and the dimension of the node feature vectors was 512. The model was trained for approximately 19 hours.

Datasets. To comprehensively and deeply evaluate the effectiveness and practicality of GTrans, we selected a dataset provided by Trex [25] for our study. This dataset includes 5,384 binary files from several different software projects on the X64 platform, where various code obfuscation techniques by Hikari [3] were applied.

During the training process, we divided the dataset into a training set (comprising 80%), a validation set (10%), and a test set (10%). Simultaneously, the original and obfuscated versions of the same function were considered as similar samples, while combinations of different functions were regarded as dissimilar samples.

Baseline Comparisons. GTrans is compared in performance with Gemini [34], Asm2Vec [6], Asteria [36], Asteria-Pro [38] and GMN [18] to comprehensively assess the effectiveness and superiority of the method in the analysis of obfuscated function similarity.

B. *Effectiveness*

Obfuscated Function Similarity Detection. We conducted a series of detailed experiments on the O-LLVM [14] obfuscator to comprehensively evaluate the efficiency and robustness of our method under various obfuscation strategies and configurations. These experiments are based on multiple projects covered in the dataset and specifically focus on various obfuscation strategies. These include Bogus Control Flow (BCF), Control Flow Flattening (CFF), Instruction Substitution (SUB), Basic Block Splitting (SPL), and Sequentially Enabling All the Above Obfuscations (ALL), for a comprehensive performance evaluation.

Table I reports the average test AUC scores of GTrans running five times on each project. Overall, GTrans achieved an average AUC score of over 0.9 across all test projects, with the highest score reaching 0.975. These results indicate that GTrans can effectively detect similarities between functions even in different obfuscation strategy environments. We note that GTrans performs best under SUB, which is intuitive, as this type of obfuscation mainly targets the code and has less impact on its basic structure (such as the core logic of functions).

When considering different software projects, GTrans particularly stands out in projects such as Libmicrohttpd and Findutils, with high AUC scores. Conversely, in some projects

[3]https://github.com/HikariObfuscator/Hikari

TABLE I: Average Test AUC Scores of GTrans Running Five Times on Each Project

	BCF	CFF	SUB	SPL	ALL
Binutils	0.906	0.923	0.951	0.921	0.943
Busybox	0.938	0.917	0.928	0.920	0.937
Coreutils	0.929	0.920	0.937	0.932	0.927
Curl	0.917	0.908	0.943	0.923	0.926
Diffutils	0.900	**0.925**	0.968	0.929	0.918
Findutils	**0.942**	0.915	0.953	0.918	0.945
GMP	0.919	0.915	0.933	0.908	0.932
ImageMagick	0.904	0.908	0.968	0.911	0.932
Libmicrohttpd	0.938	0.918	**0.975**	**0.939**	**0.949**
LibtomCrypt	0.917	0.914	0.967	0.915	0.944
OpenSSL	0.908	0.907	0.943	0.906	0.930
Putty	0.901	0.900	0.927	0.932	0.934
Sqlite	0.905	0.916	0.951	0.928	0.934
Zlib	0.914	0.907	0.932	0.921	0.937
Average	0.917	0.914	0.948	0.922	0.935

TABLE II: AUC Scores of GTrans, Gemini, GMN, Asm2Vec, Asteria and Asteria-Pro under Different Obfuscation Strategies

	BCF	CFF	SUB	SPL	ALL
Gemini	0.804	0.741	0.902	0.811	0.846
GMN	0.802	0.705	0.914	0.812	0.852
Asm2Vec	0.857	0.834	0.892	0.847	0.810
Asteria	0.818	0.715	0.908	0.835	0.821
Asteria-Pro	0.824	0.708	0.915	0.821	0.832
GTrans	**0.908**	**0.911**	**0.955**	**0.921**	**0.933**

like Binutils and LibtomCrypt, the performance under all obfuscation strategies is relatively lower, which may be related to the specific code structures and obfuscation methods of these projects. Overall, GTrans demonstrates good performance in detecting code similarity under different obfuscation strategies, especially under the SUB strategy. Below, we will detail GTrans's comprehensive performance under each obfuscation strategy and compare it with other methods.

Table II shows the AUC scores of GTrans and other methods under different obfuscation strategies. Through analysis, it is known that GTrans leads in AUC scores across all tested obfuscation strategies compared to the other three methods. Specifically, GTrans has AUC scores of 0.908 and 0.911 in BCF and CFF, respectively, significantly higher than the other methods. It was also found that GMN performs the worst under CFF. CFF reorganizes all the basic blocks in the CFG into a large loop structure, substantially altering the structure of the CFG. Therefore, the GMN method, which relies on graph neural networks to capture the local structure of graphs, is most affected, and GTrans improves by 29.22% compared to GMN. BCF achieves obfuscation by introducing additional control flow paths in the program, greatly affecting the structure. Compared to other methods, GTrans has higher AUC scores by 12.93%, 13.21%, 5.95%, 11%, and 10.2%, respectively.

This indicates that GTrans has stronger robustness and effectiveness in dealing with these structural obfuscations. The self-attention mechanism of GTrans's Transformer can better understand the relationships between distant nodes in the graph and consider the information of the entire graph at each step, helping to understand the graph structure more comprehensively.

TABLE III: Precision@1 of GTrans, Gemini, GMN, Asm2Vec, Asteria and Asteria-Pro under Different Obfuscation Strategies

	BCF	CFF	SUB	SPL	ALL
Gemini	0.758	0.702	0.954	0.758	0.714
GMN	0.707	0.668	0.961	0.751	0.707
Asm2Vec	0.881	0.836	0.955	0.891	0.815
Asteria	0.719	0.649	0.969	0.757	0.717
Asteria-Pro	0.729	0.684	0.968	0.758	0.721
GTrans	**0.943**	**0.937**	**0.987**	**0.951**	**0.947**

SUB replaces simple instructions in the original code with functionally equivalent but more complex instruction sequences, having a significant impact on assembly instructions. Under this obfuscation strategy, GTrans's AUC score reached 0.955, 4.48% higher than GMN and 4.37% higher than Asteria-Pro. Under SPL, GTrans improved by 8.73%, 10.30%, and 13.56% compared to Asm2Vec, Asteria, and Gemini, respectively. SPL splits sequentially executed code blocks into smaller blocks and transforms the direct and clear execution paths into more complex control flows. This obfuscation strategy affects both assembly instructions and the CFG structure. This shows that GTrans maintains its robustness and efficiency whether under instruction obfuscation strategies or strategies that obfuscate both instructions and structure.

Table III details the Precision@1 performance of each method under different obfuscations. The experimental results show that GTrans exhibits significant advantages under all obfuscation strategies. Specifically, GTrans achieved a Precision@1 of 0.943 under BCF, which is an improvement of 7.04%, 33.38%, and 31.15% compared to Asm2Vec, GMN, and Asteria, respectively. In instruction substitution, which greatly affects assembly instructions, GTrans performed the best with a Precision@1 of 0.987. Under other obfuscations, it also exceeded 0.93, further confirming its stability and efficiency in different obfuscation environments.

In summary, GMN, which is based on local structure, exhibits significant limitations in similarity detection under obfuscation strategies that greatly alter structural information, such as fake control flow and control flow flattening. This may be due to the fact that in structural obfuscation, increasing local structural information introduces noise into the function embedding, reducing the efficiency of the model. In contrast, GTrans extracts global information and uses an attention mechanism to focus on invariant information in obfuscation. Therefore, GTrans remains stable in performance even in scenarios with significant structural changes.

Ablation Experiment. We conducted ablation experiments on the dataset to separately assess the impact of centrality encoding and hierarchical encoding on the method. Since Graphormer [39] utilizes degree centrality encoding as a measure of node importance, GTrans was compared and analyzed with methods using degree centrality as the centrality encoding strategy and with methods that disabled hierarchical encoding. The experimental results are shown in Table IV.

TABLE IV: AUC Scores of GTrans, Degree Centrality Encoding Methods and Disabled Hierarchical Encoding Methods

	BCF	CFF	SUB	SPL	ALL
Degree Centrality Encoding	0.812	0.835	0.871	0.851	0.863
Disabled Hierarchical Encoding	0.801	0.816	0.859	0.839	0.846
GTrans	0.897	0.911	0.958	0.922	0.934

From the experimental results, it can be seen that among all obfuscation strategies, GTrans' AUC score is significantly better than the other two methods. Specifically, compared to GTrans, the degree centrality encoding method experienced a 9.48% drop in AUC score in BCF and a 9.08% drop in SUB. This is because, unlike traditional social networks, in CFGs, the degree of each node varies little, mostly ranging between 1 and 4. The centrality encoding that considers the number of dominant nodes takes into account the position and role of a node in the entire CFG. It can effectively measure a node's role as a bridge or key player in information flow within the network. Therefore, GTrans' centrality encoding is a more reasonable choice for assessing the importance of nodes in CFGs. Compared to methods that disable hierarchical encoding, GTrans showed an average improvement of 11.1% across different obfuscation strategies. This indicates that hierarchical encoding can better cope with structural changes brought about by obfuscation, enhancing the accuracy and reliability of detecting similarities in obfuscation functions.

IV. Related Work

A. Graph Transformers

Currently, achievements have been made in processing graph data using Transformers by modifying the Transformer architecture to suit graph representation tasks [24]. These studies mainly involve modifying the layers of the Transformer or changing the encoding method for positional information in graphs.

Customized Transformer. By customizing the Transformer, such as adding GNN [26] to the self-attention of the Transformer, incorporating substructures of the graph [43], or expanding multi-head attention [4], it is adapted to capture the structural features of graphs, enabling it to process graph data. Dwivedi et al. [8] proposed an attention mechanism based on neighborhood connectivity. Exphormer [27] introduced sparse attention mechanisms and global attention with virtual nodes. Simultaneously, EGT [13] introduced an edge channel, allowing the model to process not only node information but also the relational information between nodes.

Graph Positional Encoding. Early researchers introduced absolute positional encoding to explicitly encode the positional information of graphs in Transformers. For example, Dwivedi [43] used the graph Laplacian [3] as positional encoding for input features. SAN [16] uses learned positional encodings to capture the complete Laplacian spectrum of the graph. Later researchers began to explore encoding relative positional information, for instance, Graphormer [39] assigns a learnable embedding to each pair of nodes based on their spatial relationship, adding it as a bias term in the attention calculation.

B. Binary Code Similarity Detection

In the field of binary code similarity detection, significant achievements have been made using deep learning methods [21]. Given the high similarity between binary instructions and text, researchers have begun to introduce natural language models for binary code similarity detection [42]. Initially, research primarily relied on word embedding models to generate instruction embeddings and further revealed the dependencies between instructions through neural networks [30], [22], such as Inter-BIN [29], which proposed a multi-feature fusion lightweight instruction embedding method, and Deepbindiff [7] that adopts TADW [35] to incorporate node features into graph representations. With the development of Transformer [31], researchers began to use its self-attention mechanism to generate embeddings of control flows [32], optimizing them specifically to produce high-quality embeddings [11], [17]. For example, Order Matters [41] uses the BERT [15] model to extract semantic information from CFGs, while Ahn and others [1] use binary cross-entropy as an optimization loss function.

In addition to semantic information, researchers are also highly focused on the structural information of binary code, often using graph neural networks to mine the structural features of CFGs [9] or convolutional neural networks to learn the feature representations of code [12]. Furthermore, constructing more complex graphs [10], adopting hierarchical architectures [2], or using the program's AST to introduce hierarchical structures [23], [37]. For instance, Gemini [34] uses Structure2Vec [5] for graph embedding. Asteria-Pro [38] adds pre-filtering and reordering modules on the basis of Asteria [37]. αDiff [19] combines three different levels of semantic features for a more comprehensive function description. VulHawk [20] elevates binary code to microcode and then embeds it using graph convolutional networks. BEDetector [40] adopts a graph attention encoding model to extract multi-granularity features of functions

The existing methods, whether based on hierarchical structures or relying on graph neural networks, strongly depend on the connectivity among nodes in the CFG. This dependency means that when control flow obfuscation alters the program structure, their performance and accuracy often suffer.

However, GTrans, when processing CFGs with Transformers, does not simply rely on the statically defined connections between nodes in the CFG. Instead, it learns the dynamic relationships between nodes through the training process. This is particularly important because it means that even when the control flow is obfuscated and the CFG structure changes, GTrans can maintain its performance unaffected.

V. Conclusion

GTrans incorporates three types of encodings within the Transformer to address the issue of binary code similarity detection under various obfuscation strategies. We believe this to be sufficient, as these encodings capture both types of structures in the CFG and reflect the differing importance of nodes. In addition, it is worth exploring whether there are better ways to represent the relative positions of nodes in the CFG and the dominator tree through encodings.

References

[1] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 361–374.

[2] S. Alrabaee, "A stratified approach to function fingerprinting in program binaries using diverse features," *Expert Systems with Applications*, vol. 193, p. 116384, 2022.

[3] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," *Neural computation*, vol. 15, no. 6, pp. 1373–1396, 2003.

[4] D. Cai and W. Lam, "Graph transformer for graph-to-sequence learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 7464–7471.

[5] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*. PMLR, 2016, pp. 2702–2711.

[6] 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.

[7] 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.

[8] V. P. Dwivedi and X. Bresson, "A generalization of transformer networks to graphs," 2021.

[9] L. Fu, S. Ji, C. Liu, P. Liu, F. Duan, Z. Wang, W. Chen, and T. Wang, "Focus: Function clone identification on cross-platform," *International Journal of Intelligent Systems*, vol. 37, no. 8, pp. 5082–5112, 2022.

[10] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 896–899.

[11] Y. Gu, H. Shu, and F. Hu, "Uniasm: Binary code similarity detection without fine-tuning," 2023.

[12] X. Huo, M. Li, Z.-H. Zhou *et al.*, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, vol. 16, no. 2016, 2016, pp. 1606–1612.

[13] M. S. Hussain, M. J. Zaki, and D. Subramanian, "Edge-augmented graph transformers: Global self-attention is enough for graphs," *arXiv preprint arXiv:2108.03348*, 2021.

[14] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 ieee/acm 1st international workshop on software protection*. IEEE, 2015, pp. 3–9.

[15] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, 2019, p. 2.

[16] D. Kreuzer, D. Beaini, W. Hamilton, V. Létourneau, and P. Tossou, "Rethinking graph transformers with spectral attention," *Advances in Neural Information Processing Systems*, vol. 34, pp. 21 618–21 629, 2021.

[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] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[19] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

[20] 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.

[21] 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 *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[22] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 309–329.

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[24] L. Müller, M. Galkin, C. Morris, and L. Rampášek, "Attending to graph transformers," 2023.

[25] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, 2022.

[26] Y. Rong, Y. Bian, T. Xu, W. Xie, Y. Wei, W. Huang, and J. Huang, "Self-supervised graph transformer on large-scale molecular data," *Advances in Neural Information Processing Systems*, vol. 33, pp. 12 559–12 571, 2020.

[27] H. Shirzad, A. Velingker, B. Venkatachalam, D. J. Sutherland, and A. K. Sinop, "Exphormer: Sparse transformers for graphs," 2023.

[28] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[29] Q. Song, Y. Zhang, B. Wang, and Y. Chen, "Inter-bin: Interaction-based cross-architecture iot binary similarity comparison," *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 018–20 033, 2022.

[30] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, "Bindeep: A deep learning approach to binary code similarity detection," *Expert Systems with Applications*, vol. 168, p. 114348, 2021.

[31] 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.

[32] 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.

[33] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu, "On layer normalization in the transformer architecture," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 524–10 533.

[34] 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.

[35] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, vol. 2015, 2015, pp. 2111–2117.

[36] S. Yang, L. Cheng, and Zeng, "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.

[37] 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.

[38] S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun, "Asteria-pro: Enhancing deep-learning based binary code similarity detection by incorporating domain knowledge," *ACM Transactions on Software Engineering and Methodology*, 2023.

[39] C. Ying, T. Cai, S. Luo, S. Zheng, G. Ke, D. He, Y. Shen, and T.-Y. Liu, "Do transformers really perform badly for graph representation?" *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 877–28 888, 2021.

[40] L. Yu, Y. Lu, Y. Shen, H. Huang, and K. Zhu, "Bedetector: A two-

channel encoding method to detect vulnerabilities based on binary similarity," *IEEE Access*, vol. 9, pp. 51 631–51 645, 2021.

[41] 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.

[42] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and Distributed Systems Security (NDSS) Symposium 2019*, 2019.

[43] Łukasz Maziarka, T. Danel, S. Mucha, K. Rataj, J. Tabor, and S. Jastrzebski, "Molecule attention transformer," 2020.