electronics

MDPI

Article

Codeformer: A GNN-Nested Transformer Model for Binary Code Similarity Detection

Guangming Liu [1,2], Xin Zhou [2,*], Jianmin Pang [2,*], Feng Yue [2], Wenfu Liu [2,3] and Junchao Wang [2]

[1] School of Cyber Science and Engineering, Zhengzhou University, Zhengzhou 450002, China; liu.gm@163.com
[2] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China
[3] State Key Laboratory of Complex Electromagnetic Environment Effects on Electronics and Information System, Luoyang 471000, China
* Correspondence: qf_zhouxin@126.com (X.Z.); jianmin_pang@126.com (J.P.)

Abstract: Binary code similarity detection is used to calculate the code similarity of a pair of binary functions or files, through a certain calculation method and judgment method. It is a fundamental task in the field of computer binary security. Traditional methods of similarity detection usually use graph matching algorithms, but these methods have poor performance and unsatisfactory effects. Recently, graph neural networks have become an effective method for analyzing graph embeddings in natural language processing. Although these methods are effective, the existing methods still do not sufficiently learn the information of the binary code. To solve this problem, we propose Codeformer, an iterative model of a graph neural network (GNN)-nested Transformer. The model uses a Transformer to obtain an embedding vector of the basic block and uses the GNN to update the embedding vector of each basic block of the control flow graph (CFG). Codeformer iteratively executes basic block embedding to learn abundant global information and finally uses the GNN to aggregate all the basic blocks of a function. We conducted experiments on the OpenSSL, Clamav and Curl datasets. The evaluation results show that our method outperforms the state-of-the-art models.

Keywords: software analysis; binary security; code similarity detection; neural networks

check for updates

Citation: Liu, G.; Zhou, X.; Pang, J.; Yue, F.; Liu, W.; Wang, J. Codeformer: A GNN-Nested Transformer Model for Binary Code Similarity Detection. *Electronics* **2023**, *12*, 1722. https://doi.org/10.3390/electronics12071722

Academic Editor: Aryya Gangopadhyay

Received: 2 March 2023
Revised: 25 March 2023
Accepted: 31 March 2023
Published: 4 April 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The purpose of binary code similarity detection is to detect the similarity of two code gadgets using only binary executable files. Binary code similarity detection has a wide range of applications, such as bug searching [1,2], clone detection [3–5], malware clustering [6–8], malware genealogy tracking [9], patch generation [10,11] and software plagiarism detection [12–14], among other application scenarios. However, it is challenging to determine the similarity of binary code because a large amount of program semantics will be lost during the compilation, such as function names, variable names and data structures. In addition, there is a large amount of code coexisting during the compilation process, from different optimization levels and different architectures (e.g., ARM, MIPS, x86-64). The generated binary code can be changed significantly when using different compilers, changing compiler optimization options or selecting different CPU architectures. Although there are differences between executables generated from the same source code, binary vulnerabilities found on one architecture may also exist on other architectures.

In fact, there is a large amount of reused code in the industry. This phenomenon may occur due to the unskilled business of the programmer and other reasons, such as downloading the source code directly from github and other open-source platforms for modifications. In this way, the vulnerabilities that existed within the original code will always be there. When a vulnerability is discovered, computer security staff need to determine whether the vulnerability exists in historical versions of the program and whether the vulnerability exists on other code. Workers need a tool to quickly determine

code similarity in binary files to reduce the manual review effort [15]. It was only a short time ago that researchers first started to work on binary file similarity across architectures.

The work on binary code similarity detection was first proposed in 1999. A tool named EXEDIFF [16] generates patches for DEC Alpha, comparing instruction sequences. A tool named BMAT [17] enables the propagation of profile information from an older build to a newer build, thus greatly reducing the need for re-profiling. Over the next decade, there were many binary code similarity comparison methods proposed, and these efforts progressed from syntactic similarity analysis to semantic similarity and extended applications to malware analysis. Among them are the heuristic-based method for determining constructor call graphs, first proposed by H Flake [18] in 2004, and later his work introducing the Some Primes Product (SPP) hash to identify similar basic blocks, both of which are also the basis for the bindiff plugin in the IDA decompiler [19]. In 2005, Kruegel et al. [20] proposed a graph coloring technique for detecting malware variants. They grouped instructions with similar functions into 14 semantic types and used these 14 semantic types to color the inter-process control flow graph. The method is robust to partial obfuscation techniques. In 2008, Gao et al. [21] proposed BinHunt to identify the gap between two versions of the same program. The technique uses symbolic execution and a constraint solver to analyze the similarity of two basic blocks. In the last decade, a large number of binary code analysis methods have emerged and work has focused on binary code search methods [1–3,22–27]. In fact, most of the binary code search methods have been proposed to search for vulnerabilities. Since 2015, work has been focused on cross-architectural similarity analysis. Feng et al. [2] extracted the binary executable as a control flow graph with feature vectors, called the attribute control flow graph (ACFG). ACFG is based on the original features of the basic blocks in the binary code. They used the control flow graph with basic block features in a graph matching algorithm and implemented Genius, a cross-architectural bug search engine.

With the re-emergence of neural networks, there has been a neural network boom in the field of binary code similarity detection. Since 2016, a large number of neural-network-based detection methods have proliferated. For example, in 2016, Lageman et al. [28] trained a neural network to determine whether two binary codes were compiled from the same source code. In 2017, Xu et al. [29] proposed Gemini, a neural network model using ACFG as input features, to detect the similarity between two functions by calculating the distance between the embedding vectors. In 2020, Yu et al. [30] from Tencent's Cohen Lab proposed OrderMatter, a similarity analysis model based on graph neural networks. This work directed the research to the overall structure of CFG and the order of nodes, providing new ideas for future research. There were also αDiff [31] in 2018, Asm2Vec [3] in 2019, and DeepBinDiff [32] in 2020, all of which have made significant contributions in the field of binary code similarity detection based on neural networks [30,33–36].

Despite the great achievements of the current work based on neural networks, some important issues have still not been considered. Most previous work used pre-trained models to extract basic block features and then aggregate them using some kind of neural network. In this way, in the feature extraction phase, the pre-trained semantic model cannot learn the global position of the basic blocks and the information of other blocks in the graph. Moreover, in the feature aggregation phase, the neural network only learns the features of the embedded basic blocks. In addition, the same register in a control flow may be operated in multiple basic blocks. In the form of embedding before aggregation, it is difficult for a sequentially structured model to learn this connection, even when setting up some tasks such as neighbor prediction in pretraining. This is because the corpus training and the overall embedding of the basic blocks are severed. A sequential execution model would result in the loss of a large amount of global information and information about block-to-block connections.

To solve the above problem, in this paper, we propose Codeformer, a GNN-nested Transformer iterative neural network. In this work, we try to extract the basic block features and structural features of CFG iteratively. With each iteration, the node can obtain

information about its more distant neighbors. As the model runs iteratively, it is possible to obtain a more accurate embedding of each basic block in the function. As a result, Codeformer can obtain a more accurate embedding feature.

The contributions of this paper are as follows:

- We propose a generic framework called Codeformer to learn the graph embeddings of CFGs, which learns the node information of basic blocks in functions as well as global information.
- We propose to use an iterative network model for code similarity detection for the first time. Experimentally, iterative models are shown to learn the features of binary codes in greater depth.
- We analyze the learning ability of Codeformer for the structure features of function control flow graphs. Experiments show that updating graph nodes in an iteration enables the basic block to learn the structural features of its nearby basic blocks and thus better represent the information of the whole graph.
- We have evaluated the model using several datasets and the results show that our proposed model has better performance than previous methods and achieves state-of-the-art results.

2. Background

2.1. Graph Neural Network

Graph neural network (GNN) is a general term for algorithms that use neural networks to learn graph structured data, and extract and discover features and patterns in graph structured data, which can meet the needs of graph learning tasks such as clustering, classification, prediction, segmentation and generation.

In the field of binary code similarity detection, CFG and its derivatives (ICFG, ACFG) can be extracted from binary codes via decompiling techniques. It has been shown that, given the same source code, there is a strong similarity in the graph structure extracted into the CFG by decompiling programs compiled at different optimization levels. The control flow graphs obtained by decompiling programs of different architectures also have certain similarities, and these similar features are sufficient for the neural network to learn. Therefore, we can study CFG using graph neural networks to analyze the similarity of functions and programs across optimization levels as well as across architectures.

2.2. Attributed Control Flow Graph

Feng et al. [2] proposed a workflow for graph embedding called Genius. Given a binary function, Genius extracts the original features of the function in the form of an attribute control flow graph (ACFG). The ACFG is defined as a directed graph $G = \langle V, E, \phi \rangle$, where V is the set of basic blocks, $E \subseteq V * V$ is the set of edges between basic blocks and $\phi : V \rightarrow \sum$ is the labeling function, which maps a basic block in V to a set of attributes in $\sum$.

The attribute set $\sum$ contains two types of features: statistical features and structural features. The statistical features describe the local features within the basic block, and the structural features capture the location features of the basic block in the CFG, as shown in Table 1.

Table 1. Basic-block attributes of ACFG.

Type	Attribute Name
Block-level attributes	String Constants Numeric constants No. of transfer instructions No. of calls No. of instructions No. of arithmetic instructions
Inter-block attributes	No. of offspring Betweenness

2.3. Transformer

In 2017, Google proposed in a published paper a network based on the attention mechanism to deal with sequential model problems [37], such as machine translation. The model can work in a highly parallel way, thus improving the training speed as well as the performance. The structure of Transformer is not complicated; the model presented by Vaswani et al. [37] has a seq2seq encoder-decoder structure, as shown in Figure 1. The encoder consists of multiple network layers, each with two sub-layers. The first sublayer is a multi-headed self-attention mechanism and the second is a simple location fully connected feedforward network. The decoder is also composed of multiple network layers. In addition to the two sublayers in each encoder layer, the decoder inserts a third sublayer that performs multi-headed attention on the output of the encoder stack.

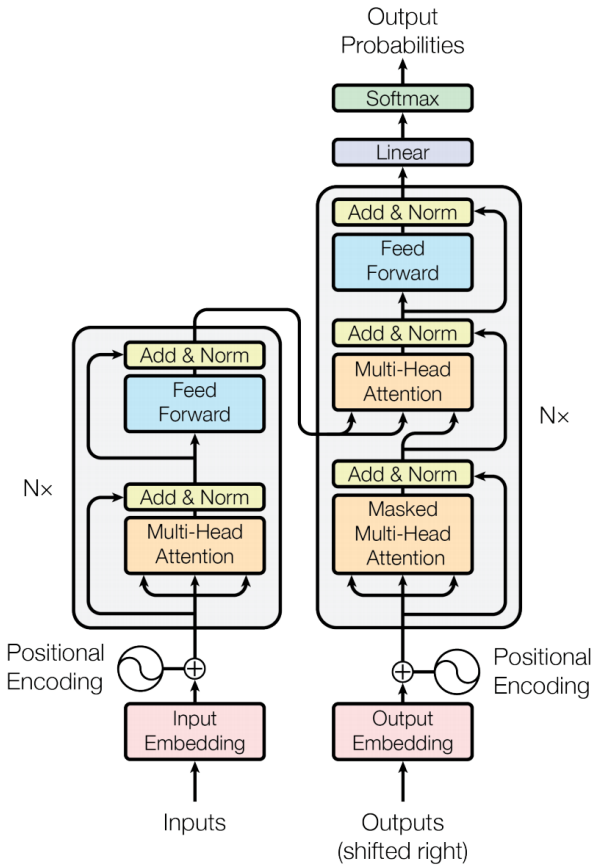


Figure 1. Transformer model architecture.

2.3.1. Attention

The attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, key, value and output are all vectors. The output is a weighted sum of values, where the weight assigned to each value is calculated by the compatibility function of the query with the corresponding key.

2.3.2. Multi-Attention

Multi-attention is actually the use of the same query of multiple independent attention calculations, which works as an integration and can effectively prevent overfitting. The paper [37] mentions that the input sequence of the model is identical. This means that each attention layer uses the same initial values to linear transform, including Q, K and V. Therefore, each attention mechanism function is responsible for only one subspace in the final output sequence, allowing the model to learn more information.

2.4. Graphformers

In 2021, Yang et al. [38] proposed the GNN-nested Transformer model named graphformers. In this project, the target object to deal with is text graph data, where each node x in the graph G(x) is a sentence. The model plays an important role in combining a GNN with text and makes an active contribution in the field of neighborhood prediction. Since the network is a nested structure, its text encoding and full-graph information transfer can be performed iteratively. Therefore, compared with the cascading structure, the information transmission across nodes is more sufficient, obtaining better representation characteristics.

However, the graph defined by the model is one central node with five edge nodes, which does not match the complexity of the actual graph. Therefore, the model can only be applied to simple star graphs. Depending on whether it is a directed or undirected graph, the structure of the graph will be more complex. To put it simply, if the graphformers model is applied to binary similarity detection, it can only handle the basic block embedding, but cannot handle the entire control flow graph. These raw features extracted by feature engineering exhibit the basic information of a function. However, due to the low dimensionality, ACFG does not adequately represent the information of the function, and the model is not effective.

3. Codeformer

As mentioned in previous work [39], a normalized analysis of binary assembly code can determine that the curves of the instruction distribution conform to Zipf's law, similar to natural language. Zipf's law is the law of word frequency distribution, where the number of occurrences of a word in the corpus of a natural language is inversely proportional to its ranking in the frequency table. Therefore, we can embed assembly instructions with Transformer as we do with natural languages.

3.1. Workflow

In this section, we describe the overall workflow of the model to detect binary code similarity. Our work has two components: the CFG extraction script, and the neural network for binary code similarity computation based on Transformer and a GNN. The CFG extraction script uses the decompiler tool IDA to extract the CFG of all functions. The neural network model can handle the control flow graphs of binary files, each of which is a disassembly message of a function. The information within each node in the graph is an assembly instruction for a basic block, and all nodes and edges within the function form the graph $G = \langle V, E \rangle$. The model iterates instruction-embedding within blocks and graph-embedding within functions to learn the embedding features of graph G. We expect that the generated embedding information can sufficiently learn the intra-block instruction information and inter-block relationship, and predict whether two functions are similar based on the graph embedding information similarity.

The input to the work is a binary file run through the IDA script to obtain a control flow graph. The granularity of the study is at the basic block level. The overall workflow is shown in Figure 2. Transformer treats all instructions within a basic block as a paragraph and embeds this paragraph to a vector. After embedding all the basic blocks of the decompiled control flow graph pairs as vector features, the GNN learns and updates the features of these basic blocks with a full graph view. Note that such nested learning will be performed several times. Then, the aggregation function of the GNN will aggregate all nodes of the whole graph to obtain the embedding vector of the graph. Finally, the similarity of this function pair is calculated by the similarity measurement algorithm. We summarize the workflow of Codeformer as Algorithm 1.

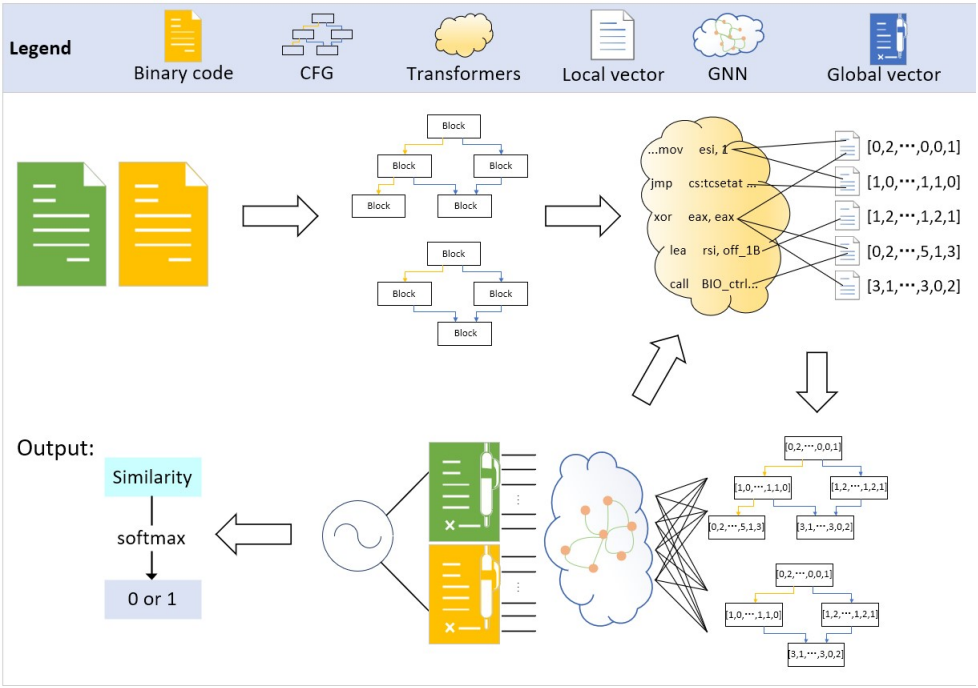


Figure 2. Workflow of Codeformer.

Algorithm 1 Calculation of CFG feature vectors.

Require: Control flow graph of the function $G = \langle V, E \rangle$
Ensure: Hidden status of the control flow graph

for line in iterates **do**
  for $v \in V$ **do**
    for $s \in v$ **do**
      $h_s = TRM(s)$
    end for
    $h_v = MHA(\{\sum s | s \in v\})$
  end for
  $\widehat{h_V} = BMP(h_V)$
end for
$h_G = BMA(\widehat{h_V})$
return h_G;

3.2. Calculation of CFG Feature Vectors and Similarity

Algorithm 1 describes the specific process that Codeformer uses to calculate the feature vector of a function, and the input of the model is a control flow graph. For each iteration of the node hidden state update, for each basic block within the graph, the instructions inside the block are embedded one by one using Transformer (TRM), and then the multi-headed attention network (MHA) aggregates the basic blocks into a hidden state (h_v). After obtaining the hidden status of all basic blocks, the block message passing (BMP) module updates the hidden status of each basic block based on the neighboring nodes. After the last encoding layer, the block message aggregation (BMA) module aggregates all the basic blocks into an embedded feature of the graph (h_G). In Codeformer, the BMP module is implemented by a gated recurrent neural network (GRU) and the BMA module is implemented by an MLP.

Codeformer calculates the similarity of a pair of basic block embedding features h_G by the cosine distance of the function pair, as shown in Equation (10). Codeformer determines whether the two functions are similar by calculating whether the similarity of the function pair exceeds a threshold value. The specific threshold selection is discussed in Section 5.3.1.

3.3. GNN-Nested Transformer Neural Network

The neural network proposed in this paper improves and enhances Microsoft's graphformers [38], a neural network that processes text graphs. We take as input the control flow graph $CFG = \langle V, E \rangle$ of the binary code, where V is the set of vertices and inside each $v \in V$ node is the intra-block information of one basic block of the control flow graph; E is the set of edges between nodes. The neural network considers the information within each block as a paragraph and embeds each paragraph into the $n_b = 144$-dimensional vector. The update function of the GNN does not change the dimension of the vector when updating. Based on the word-embedding vector and the adjacency matrix of the CFG, the GNN aggregates the CFG of the input function into an $n_g = 100$-dimensional vector. The entire network model is shown in Figure 3, with multiple layers of nested Transformer (for embedding text) and GNN (for messaging and message aggregation) iterative execution. Among them, we use MPNN [40] to implement the GNN functionality.

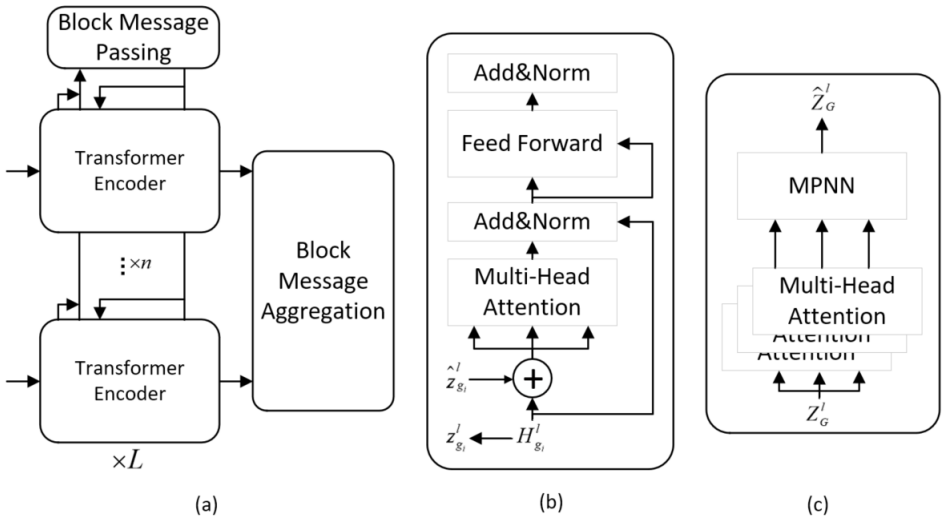


Figure 3. Structure of Codeformer. (**a**) shows the overall structure of Codeformer, including Transformer Encoder, Block Message Passing and Block Message Aggregation. The model has n Transformer Encoder per layer iteration, where n is the number of basic blocks in the CFG. (**b**) shows the internal structure of the Transformer Encoder. (**c**) shows the internal structure of the Block Message Passing module, including n multi-headed attention and the message passing method of MPNN. The Block Message Aggregation module is implemented by MLP.

3.3.1. Model Structure

First, the function is disassembled to obtain control flow graph. We consider each basic block as a paragraph, each assembly instruction as a sentence, and the operators and operands within the block as tokens. For layer l, in the intra-block instruction embedding, we consider the first instruction of each block as the aggregation center and use the multi-headed attention network to embed $Z_G^l = \sum_{z \in Z, g \in G} z_g^l$ to obtain the feature vector of each basic block. The nodes of the CFG are then updated and aggregated using a message-passing network to generate function embedding features.

3.3.2. Text Embedding in Transformer

We use the Transformer component to calculate the instruction embedding $\widehat{H}_g^l$ for graph augmentation, which is calculated as follows:

$$\widehat{H}_g^l = LN(H_g^l + BMP(\widehat{H}_g^l)) \tag{1}$$

$$H_g^{l+1} = LN(\widehat{H}_g^l + MLP(\widehat{H}_g^l)) \tag{2}$$

$$BMP(Z) = MPNN(Z) \tag{3}$$

In the equation above, LN is the layer-norm function, BMP is the block message-passing function for updating the embedding of basic blocks, MLP is the multi-layer projection network, MPNN is the message passing and updating part of the message passing network, and MHA is the multi-head attention. Each MHA calculates the embedding vector of each basic block and then updates the embedding of the basic block with MPNN; the updated result is used as the input of the next layer of embedding.

3.3.3. Graph Neural Network

We use a message-passing network (MPNN) for message passing and message aggregation for each CFG. In the block message-passing module, for each node, the features of its neighboring nodes and the corresponding edges are aggregated with the node itself, and then the node is updated with the obtained message and the previous features of the node to obtain the new features, called message passing. In the block message aggregation module, all the node features are aggregated into a single graph feature using the aggregation method. The formula for the MPNN is as follows:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \tag{4}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \tag{5}$$

$$g = R(h_v^T | v \in G) \tag{6}$$

where m_v^t denotes the message obtained by node v for the tth time, $w \in N(v)$ is the neighbor node of node v, h_ε^t is the hidden state of ε for the tth time and e_{vw} is the feature of the edge between v and w. M_t is an arbitrary messaging method, U_t is an arbitrary update function and R is an arbitrary aggregation function. G indicates the full graph and g is the embedding feature of the full graph.

In Codeformer, we use MLP as the messaging method M_t, GRU as the update function U_t and MLP as the aggregation function R. In addition, there is no parameter e_{vw} in Equation (7), since there is only a simple jumping relationship between the basic blocks and the edges have no feature vectors. The formula is as follows:

$$m_v^{t+1} = \sum_{w \in N(v)} MLP(h_v^t, h_w^t) \tag{7}$$

$$h_v^{t+1} = GRU(h_v^t, m_v^{t+1}) \tag{8}$$

$$g = MLP(h_v^T | v \in G) \tag{9}$$

3.4. Model Training

Training Objectives: The goal of this work was to improve the accuracy and efficiency of binary code similarity detection with the lowest false-positive rate. Given a pair of functions $\langle f_1, f_2 \rangle$, CoderMatters learns their embeddings and calculates the similarity to determine whether the two functions are homologous.

3.4.1. Siamese Network

A Siamese network uses two structurally consistent sub-networks, which are trained simultaneously during model training and share network parameter weights. The network receives function pairs $\langle f_1, f_2 \rangle$, embeds them as vectors $\langle G_w(f_1), G_w(f_2) \rangle$, and then calculates the distance E_w of the two output vectors via the cosine distance metric. By learning positive and negative samples, the network can make similar inputs become closer after model embedding, while dissimilar inputs move further away.

$$\begin{aligned} E_w(F_1, F_2) &= Dist(G_W(F_1), G_W(F_2)) \\ &= cos(G_W(F_1), G_W(F_2)) \\ &= \frac{G_W(F_1) \cdot G_W(F_2)}{\|G_W(F_1)\| \|G_W(F_2)\|} \end{aligned} \tag{10}$$

3.4.2. Loss Function

The goal of Codeformer's work is to detect whether the function pairs $\langle f_1, f_2 \rangle$ are similar, and its loss function is the cosine similarity loss, which can determine whether the two input vectors are similar. We define the control flow graph embedding g after BMA aggregation as the embedding of a function. Specifically, $\langle g1, g2 \rangle$ are two vectors of the final embedding of the function pair, and y represents the real similarity label, which belongs to $\{1, -1\}$, representing two functions that are similar or dissimilar, respectively. The loss of the i-th sample is shown in the following equation.

$$l = \begin{cases} 1 - cos(g_{i1}, g_{i2}), & if\ y_i = 1 \\ max(0, cos(g_{i1}, g_{i2}) - margin) & if\ y_i = -1 \end{cases} \tag{11}$$

4. Experiment

In this section, we evaluate the functionality and performance of Codeformer.

4.1. Research Questions

The basic idea of our study is to learn the semantic and structural information of the control flow graph, and then determine the similarity of the function pairs by comparing their cosine distances. In short, we set up experiments to answer the following questions.

- RQ1: Is our proposed Codeformer effective in detecting binary code similarity?
- RQ2: Does our proposed Codeformer learn the structural features of control flow graphs?
- RQ3: What are the main factors that affect Codeformer?

4.2. Experimental Setup

Our experiments were deployed on a server with an Intel i7-10700 CPU running at 2.90 GHz, 64 GB of RAM on board and an Nvidia RTX3080 GPU with 10 GB of video memory. The operating system is Ubuntu 20.04, the entire network uses the PyTorch architecture and the Python version is 3.8.13.

4.3. Datasets and Data Pre-Processing

The datasets compiled by IDA are OpenSSL, Clamav and Curl, where Clamav does not contain dynamic link libraries. We used all datasets for training and evaluation. The datasets include two compiled architectures, X86-64 and ARM, with four levels of optimization from O0-O3. As shown in Table 2, the third column shows the number of binaries in a particular architecture in the dataset, and the fourth column shows the number of functions. We cleaned the original data and removed all orphan nodes (graphs with only one node). Since there is no information about streams, graph neural networks learn orphan nodes with only intra-node information, and such learning is actually no different from using Transformer alone. Therefore, orphan nodes are out of the learning scope of our model. In addition, we do not limit the data size to a fixed dimension in order to allow the model to fully learn the information within the block, a setting that is consistent with the diversity of binary codes. However, such a design gives us an uneven dataset, where the function may consist of an amount of basic blocks, and the longest statement in the basic block is too long. The problem lies in the high O(L^2) time complexity and space complexity in Transformer when performing the semantic embedding of code in long basic blocks, resulting in excessive consumption of memory (L is the length of the basic block). For example, a basic block with 300 instructions would require approximately 5.4 GB of display

memory to run Transformer only when the batch size is 1. When training two basic blocks of such size at the same time, the space required would already exceed the capacity of a normal GPU (10 GB for the RTX3080).

Such a data input into the model can lead to GPU memory overflow. We list the distribution of block numbers in the functions in Table 3. Moreover, Figure 4 shows the cumulative distribution of CFG size for 49,056 CFGs. Functions with fewer than 50 blocks account for 91.7% of all functions. Therefore, in our work, we exclude functions with oversized basic blocks from our datasets.

Table 2. Number of binaries and functions in the dataset.

Name	Platform	# of Binaries	# of Functions
OpenSSL	X64	4	13,598
	ARM	4	13,599
Clamav	X64	16	2497
	ARM	16	2550
Curl	X64	4	8512
	ARM	4	8300
Total		48	49,056

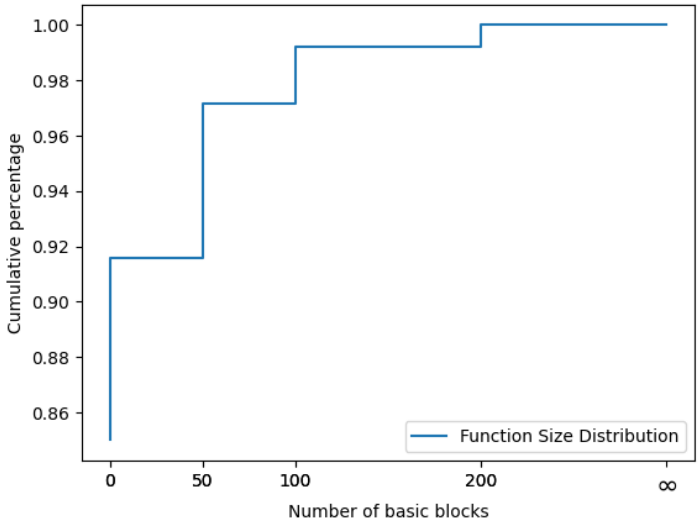


Figure 4. Cumulative distribution of CFG size for 49,056 CFGs.

Table 3. Distribution of block numbers in the functions.

The Region of Block Numbers	Amounts
$[0, 50]$	44,963
$[50, 100]$	2745
$[100, 200]$	1014
$[200, \infty]$	334

We decompile all the binaries to obtain several functions, among which there are a large number of renamed functions, noting that they are compiled from the same source code under different architectures and different optimization levels. In model learning, we assign the training set into several function pairs $\langle f_1, f_2 \rangle$ and label them according to the function names.

We first take 50% pairs of homonymous functions as positive samples. Since the homonymous functions are obtained by compiling the same source code under different optimization levels and architectures, we determine that the homonymous functions are similar and set the label as 1. Similarly, we randomly pair the remaining functions as negative samples, and label them as -1. In the random pairing, if there is a pairing of functions with the same name, it will be re-paired. Therefore, there will not be a situation in which some functions with the same name are paired together. Finally, we randomly divide these function pairs into a training set, a test set and a validation set in the ratio 8:1:1. These function pairs are actually the inputs to the Siamese network.

4.4. Evaluated Tasks

4.4.1. Binary Code Similarity Detection

In this task, we compared the performance of benchmark models and evaluated the ability of the models in detecting code similarity.

4.4.2. Evaluation of the Model's Ability to Learn Graph Structure Information

This task aimed to evaluate the ability of our proposed Codeformer to learn the graph structure information of CFGs. We conducted comparison experiments with graphformers, a semantic learning model that aggregates only one paragraph.

4.4.3. Parameter Experiments

The task aimed to evaluate the main factors influencing the model, which we achieved by control variates. The variables being controlled were the similarity threshold, learning rate and whether iteration.

4.5. Benchmark Method

4.5.1. Gemini

Gemini uses a neural-network-based approach to compute the embedding vector of ACFG. ACFG is a graph structure with primitive features of basic blocks proposed by Xu et al. [29]. It manually extracts the primitive features of the basic blocks as the node information of the graph. For Gemini evaluation, we obtained the ACFG, and fed it into a graph neural network for testing.

4.5.2. Asm2Vec

Asm2Vec proposes an assembly code representation learning model. It requires assembly code as input to find and merge semantic relationships between tokens that appear in the assembly code.

4.5.3. ASTERIA

The baseline extracts the binary code as an abstract syntax tree (AST) and uses a neural network to learn the semantic information of the function from the AST [41]. We implemented the method according to the official Asteria code and tested it.

4.5.4. OrderMatter

This baseline proposes semantic-aware neural networks to extract the semantic information of the binary code and adopts a convolutional neural network (CNN) on adjacency matrices to extract the order information. We replicated the method and tested it.

4.5.5. BinShot

This baseline tackles the problem of detecting code similarity with one-shot learning (a special case of few-shot learning) and adopts a weighted distance vector with binary cross-entropy as a loss function on top of BERT. We tested the model using BinShot's official code.

4.6. Evaluation Metrics

We would like to train Codeformer to be a conservative model. A good similarity detection method should be as free of misrepresentation as possible and reduce the percentage of false positives as much as possible. Therefore, our assessment methods were AUC (ROC) and ACC.

The receiver operating characteristic curve (ROC) and the area under the curve (AUC) are used to measure the performance of the classifier. The ROC curve can easily identify the effect of the threshold on the generalizability function of the model, which helps to choose the best threshold. Assuming that the positive sample is P, the negative sample is N, the correct judgment sample attribute is T and the wrong judgment sample attribute is F, the vertical coordinate of the ROC curve is the true positive rate TPR and the horizontal coordinate is the false positive rate FPR. The calculation formula is as follows.

$$TPR = \frac{TP}{TP + FN} \tag{12}$$

$$FPR = \frac{FP}{FP + TN} \tag{13}$$

AUC (ROC) is the area under the ROC curve, and in a 1×1 coordinate system, the AUC (ROC) takes values between 0.5 and 1. The closer the AUC (ROC) is to 1, the better the prediction.

The accuracy rate shows how accurately Codeformer classifies. In calculating the accuracy, we determined that the threshold of similarity is 0.8. This means that when the similarity of function pairs is greater than 80%, we determine that they are similar, and when it is less than 80%, they are non-similar. The ACC calculation formula is as follows:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{14}$$

5. Evaluation Results and Analysis

5.1. RQ1: Effectiveness of Binary Code Similarity Detection

To validate the performance of our proposed Codeformer, we compared it with a benchmark model and evaluated the effectiveness of Codeformer. Table 4 shows the performance comparison between models. The results show that Codeformer has the best performance compared with other models, reaching an AUC (ROC) of 97.50 and an ACC of 93.38.

Table 4. Performance comparison between models.

Model	ACC	AUC (ROC)
Gemini	85.63	89.28
Asm2Vec	46.28	76.53
Asteria	92.34	97.17
OrderMatter	89.16	91.67
BinShot	92.73	96.45
Codeformer	93.38	97.50

Since Asteria works at the AST level, it can largely reduce cross-architectural differences and therefore has a suitable AUC (ROC). Since the input to Gemini is ACFG, a hand-extracted function feature, the model does not learn sufficient function features and is therefore less effective. Asm2Vec is the worst of all models because Asm2Vec is not applicable to cross-architecture similarity detection. Both Ordermatters and BinShot use deep learning methods for binary code similarity detection, and in comparison to these two benchmark methods, we can see that Codeformer has the best results of the three. These two works focus on learning the semantic features of functions using a bert-based

approach, achieving highly desirable results. Ordermatters has attempted to improve the performance of its model by using the adjacency matrix of CFG to learn the structural features of the function. Nevertheless, Codeformer still outperforms them, which shows the importance of the structural information in CFGs. Codeformer learns the structural features of the control flow diagram and thus has a better performance.

This indicates that Codeformer is effective in binary code similarity detection work and the iterative structure of Codeformer can learn the semantic and structural features of binary codes well.

5.2. RQ2: Ability of Learning Graph Structure Information

In order to evaluate the graph learning capability of Codeformer, we simplified the structure of Codeformer so that the neural network did not take into account the graph structure of CFG. We implemented Codeformer without graph information, named Codeformer-withoutGraph. Then, we removed the graph neural network part of Codeformer so that the model only performed aggregation operations on the CFG without updating the nodes after embedding the basic blocks of the function.

The effects of Codeformer-withoutGraph and Codeformer are shown in Table 5. From Table 5, it can be seen that the ACC, AUC (ROC) and F1 scores of Codeformer are all higher than Codeformer-withoutGraph.

This indicates that the structural feature of CFG is useful in binary code similarity detection, and Codeformer can learn this feature to improve the similarity detection.

Table 5. Performance comparison between model variants.

Model	ACC	AUC (ROC)	F1
Codeformer-withoutGraph	89.44	94.79	91.65
Codeformer-withoutIteration	83.88	91.27	83.18
Codeformer	93.38	97.50	95.93

5.3. RQ3: Effect of Parameters on Model

5.3.1. Threshold

We evaluated the effect of similarity thresholds on the effect of similarity detection. In particular, we only adjusted the similarity thresholds in $\{0.5, 0.6, 0.7, 0.8, 0.9\}$, and other parameters remained unchanged.

Figure 5 shows the ACC and AUC (ROC) scores when different thresholds were chosen. We can observe that when the threshold was 0.5, the AUC (ROC) reached the highest value of 0.975. When the threshold increased, the AUC (ROC) floated in a small range, while the ACC gradually increased. According to the results, Codeformer can achieve the best performance at the threshold value of 0.8; AUC (ROC) and ACC reach the highest values at the same time and start to decrease as the threshold value increases.

This means that Codeformer can be trained effectively at any threshold setting. This feature allows Codeformer to adapt to diverse detection requirements; that is, it can efficiently and accurately perform similarity detection under different threshold requirements.

5.3.2. Learning Rate

In exploring the effect of learning rate on the model, we set three different learning rates of $\{10^{-3}, 10^{-4}, 10^{-5}\}$.

The loss curves for different learning rates are shown in Figure 6. The results show that the best learning rate for Codeformer is 10^{-4}. When the learning rate is 10^{-3}, the model oscillates and does not converge because the learning rate is too large. When the learning rate is 10^{-5}, the learning rate is too small, resulting in slow convergence and almost no convergence in the first 6000 steps of training, which seriously affects the efficiency of the model.

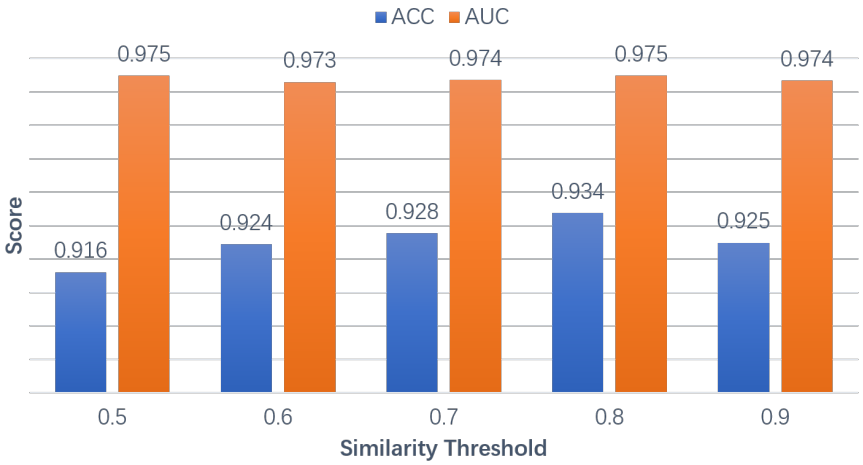


Figure 5. Similarity threshold impact on performance.

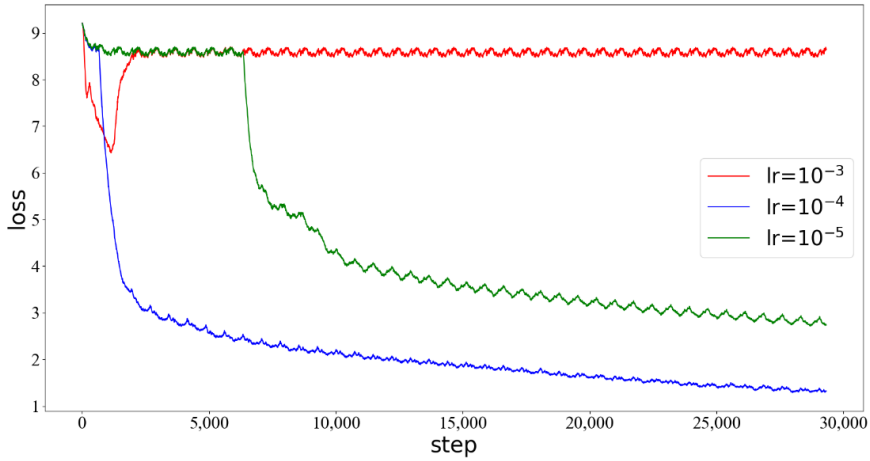


Figure 6. Learning rate impact on performance.

5.3.3. Whether Iteration

We evaluated the effectiveness of iterations of Codeformer. Specifically, we implemented Codeformer-withoutIteration, a non-iterative Codeformer. This model simply uses Transformer to extract the basic block features and then aggregates all the blocks in the CFG by MPNN.

The performance of whether-iterating Codeformer is shown in Table 5. We observe that Codeformer has a higher AUC (ROC) than Codeformer-withoutIteration by 0.06, a higher ACC by 9.5 and a higher F1 score by 12.75.

This indicates that the iterative structure of Codeformer is working satisfactorily. Codeformer can learn more comprehensive information about the CFG, including instruction features and graph structure features, through iteration.

6. Discussion

6.1. Why Embed and Aggregate Nested Iterations

With the same basic block, Transformer can learn an important feature of the basic block, i.e., the number of instructions within the block. However, in the traditional Transformer+GNN model, the basic block-embedding generates a feature vector of fixed dimensionality, which leads to the inability to observe the basic block volume in the graph embedding phase. In the traditional Transformer+GNN approach, the model cannot fully learn enough structural information because the basic block embedding and graph embedding are split, even though the information of node neighbors may be considered when using Transformer for basic block embedding. As shown in Figure 7 method b, after each

basic embedding, the embedding dimension is fixed regardless of the length of statements within the block. Such processing does not allow the graph neural network to extract the actual dimensionality of the basic blocks in the subsequent graph embedding. Therefore, based on the consideration of this problem, Codeformer is a Transformer-nested GNN neural network.

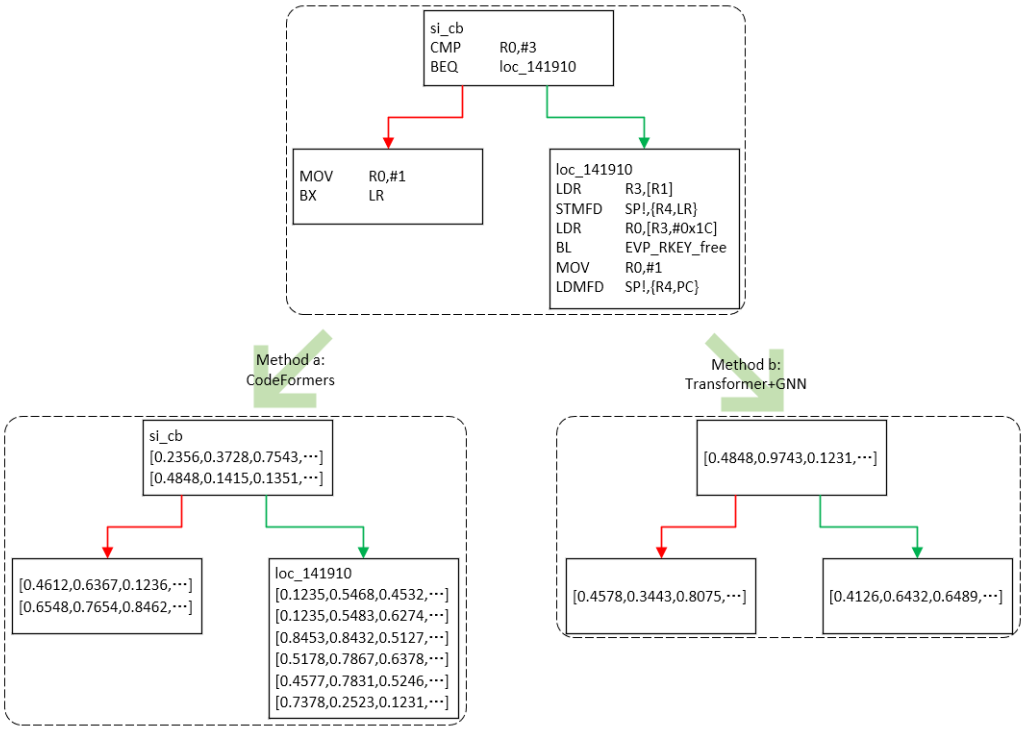


Figure 7. Different embedding vectors.

6.2. Limitations

Codeformer does not fix the embedding dimension in instruction embedding and basic block-embedding, which may cause the whole batch to be too large when embedding large basic blocks, taking up a lot of GPU memory. For the moment, we cannot help but make a trade-off between the exception data and the memory allocation. Specifically, we clean the data containing the large basic blocks and retain the data that will not cause memory overflow. Solving the memory consumption problem due to the length of the basic blocks will be a focus of our work in the future.

7. Conclusions

In this paper, we proposed a new framework for binary code similarity detection named Codeformer, which is an iterative model including two components: Transformer and a GNN. We iteratively used Transformer to extract basic block semantic features, and used the GNN to update and aggregate the extracted basic block features. We conducted experiments on the OpenSSL Clamav and Curl datasets, and the evaluation results show that Codeformer is a state-of-art method in binary code similarity detection, reaching an accuracy of 93.38%.

Author Contributions: Conceptualization, J.P. and G.L.; methodology, G.L.; software, X.Z.; validation, G.L., X.Z. and W.L.; formal analysis, F.Y.; investigation, W.L.; resources, J.W.; data curation, X.Z.; writing—original draft preparation, G.L.; writing—review and editing, X.Z.; visualization, F.Y.; supervision, J.W.; project administration, J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author upon reasonable request.

Acknowledgments: The authors thank the editor and the anonymous reviewers for their constructive feedback and valuable advice.

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

Abbreviations

The following abbreviations are used in this manuscript:

GNN	Graph neural network
TRM	Transformer
MPNN	Message-passing neural network
CFG	Control flow graph
MHA	Multi-headed attention network
BMP	Block message passing
BMA	Block message aggregation

References

1. Feng, Q.; Wang, M.; Zhang, M.; Zhou, R.; Henderson, A.; Yin, H. Extracting conditional formulas for cross-platform bug search. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 346–359.
2. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
3. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 472–489.
4. Bowman, B.; Huang, H.H. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In Proceedings of the 2020 IEEE European Symposium on Security and Privacy (EuroS&P), Genoa, Italy, 7–11 September 2020; pp. 53–69.
5. Golubev, Y.; Poletansky, V.; Povarov, N.; Bryksin, T. Multi-threshold token-based code clone detection. In Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; pp. 496–500.
6. Wicherski, G. peHash: A Novel Approach to Fast Malware Clustering. *LEET* **2009**, *9*, 8.
7. Rafique, M.Z.; Caballero, J. Firma: Malware clustering and network signature generation with mixed network behaviors. In *Research in Attacks, Intrusions, and Defenses. RAID 2013. Lecture Notes in Computer Science* ; Springer: Berlin/Heidelberg, Germany, 2013; pp. 144–163.
8. Hu, X.; Shin, K.G.; Bhatkar, S.; Griffin, K. MutantX-S: Scalable Malware Clustering Based on Static Features. In Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), San Jose, CA, USA, 26–28 June 2013; pp. 187–198.
9. Saha, R.K.; Asaduzzaman, M.; Zibran, M.F.; Roy, C.K.; Schneider, K.A. Evaluating code clone genealogies at release level: An empirical study. In Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, Timisoara, Romania, 12–13 September 2010; pp. 87–96.
10. Jang, J.; Agrawal, A.; Brumley, D. ReDeBug: Finding unpatched code clones in entire os distributions. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 48–62.
11. Xu, Z.; Chen, B.; Chandramohan, M.; Liu, Y.; Song, F. Spain: Security patch analysis for binaries towards understanding the pain and pills. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 462–472.
12. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 389–400.
13. Jhi, Y.C.; Wang, X.; Jia, X.; Zhu, S.; Liu, P.; Wu, D. Value-based program characterization and its application to software plagiarism detection. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011; pp. 756–765.

14. Liu, C.; Chen, C.; Han, J.; Yu, P.S. GPLAG: Detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; pp. 872–881.
15. David, Y.; Partush, N.; Yahav, E. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Not.* **2018**, *53*, 392–404. [CrossRef]
16. Baker, B.S.; Manber, U.; Muth, R. Compressing differences of executable code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*; Citeseer: Princeton, NJ, USA, 1999; pp. 1–10.
17. Wang, Z.; Pierce, K.; McFarling, S. Bmat-a binary matching tool. *Feedback-Dir. Optim.* **1999**.
18. Flake, H. Structural comparison of executable objects. In *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004*; Gesellschaft für Informatik eV: Bonn, Germany, 2004.
19. Hex Rays. Ida Pro. 2022. Available online: https://hex-rays.com/products/ida/ (accessed on 1 March 2023).
20. Kruegel, C.; Kirda, E.; Mutz, D.; Robertson, W.; Vigna, G. Polymorphic worm detection using structural information of executables. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Seattle, WA, USA, 7–9 September 2005; Springer: Berlin/Heidelberg, Germany, 2005; pp. 207–226.
21. Gao, D.; Reiter, M.K.; Song, D. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security. ICICS 2008. Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 238–255.
22. Yang, J.; Fu, C.; Liu, X.Y.; Yin, H.; Zhou, P. Codee: A tensor embedding scheme for binary code search. *IEEE Trans. Softw. Eng.* **2021**, *48*, 2224–2244. [CrossRef]
23. Zhao, D.; Lin, H.; Ran, L.; Han, M.; Tian, J.; Lu, L.; Xiong, S.; Xiang, J. CVSkSA: Cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. *Softw. Qual. J.* **2019**, *27*, 1045–1068. [CrossRef]
24. Lin, H.; Zhao, D.; Ran, L.; Han, M.; Tian, J.; Xiang, J.; Ma, X.; Zhong, Y. Cvssa: Cross-architecture vulnerability search in firmware based on support vector machine and attributed control flow graph. In Proceedings of the 2017 International Conference on Dependable Systems and Their Applications (DSA), Beijing, China, 31 October–2 November 2017; pp. 35–41.
25. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Tan, H.B.K. Bingo: Cross-architecture cross-os binary search. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 678–689.
26. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724.
27. David, Y.; Yahav, E. Tracelet-based code search in executables. *ACM SIGPLAN Not.* **2014**, *49*, 349–360. [CrossRef]
28. Lageman, N.; Kilmer, E.D.; Walls, R.J.; McDaniel, P.D. B in DNN: Resilient Function Matching Using Deep Learning. In *Security and Privacy in Communication Networks. SecureComm 2016. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*; Springer: Cham, Switzerland, 2016; pp. 517–537.
29. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376.
30. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1145–1152.
31. Liu, B.; Huo, W.; Zhang, C.; Li, W.; Li, F.; Piao, A.; Zou, W. αdiff: Cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 667–678.
32. Duan, Y.; Li, X.; Wang, J.; Yin, H. Deepbindiff: Learning program-wide code representations for binary diffing. In Proceedings of the Network and Distributed System Security Symposium (NDSS) 2020, NDSS, San Diego, CA, USA, 23–26 February 2020.
33. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jTrans: Jump-Aware Transformer for Binary Code Similarity. *arXiv* **2022**, arXiv:2205.12713.
34. Marcelli, A.; Graziano, M.; Ugarte-Pedrero, X.; Fratantonio, Y.; Mansouri, M.; Balzarotti, D. How Machine Learning Is Solving the Binary Function Similarity Problem. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*; USENIX Association: Boston, MA, USA, 2022; pp. 2099–2116.
35. Tian, D.; Jia, X.; Ma, R.; Liu, S.; Liu, W.; Hu, C. BinDeep: A deep learning approach to binary code similarity detection. *Expert Syst. Appl.* **2021**, *168*, 114348. [CrossRef]
36. Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 261–271.
37. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1–11.
38. Yang, J.; Liu, Z.; Xiao, S.; Li, C.; Lian, D.; Agrawal, S.; Singh, A.; Sun, G.; Xie, X. GraphFormers: GNN-nested transformers for representation learning on textual graph. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 28798–28810.
39. Koo, H.; Park, S.; Choi, D.; Kim, T. Semantic-aware Binary Code Representation with BERT. *arXiv* **2021**, arXiv:2106.05478.

40. Gilmer, J.; Schoenholz, S.S.; Riley, P.F.; Vinyals, O.; Dahl, G.E. Neural message passing for quantum chemistry. In Proceedings of the International Conference on Machine Learning, PMLR, Sydney, Australia, 6–11 August 2017; pp. 1263–1272.
41. Yang, S.; Cheng, L.; Zeng, Y.; Lang, Z.; Zhu, H.; Shi, Z. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, 21–24 June 2021; pp. 224–236.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.