

Article

Enhancing Binary Security Analysis Through Pre-Trained Semantic and Structural Feature Matching

Chen Yi ¹, Wei Dai ¹, Yiqi Deng ¹ , Liang Bao ² and Guoai Xu ^{1,*} 

¹ School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China; 24bg51902@stu.hit.edu.cn (C.Y.); 24b951054@stu.hit.edu.cn (W.D.); dengyiqi@hit.edu.cn (Y.D.)

² The Third Research Institute of Ministry of Public Security, Shanghai 201204, China; baoliang@stars.org.cn

* Correspondence: xga@hit.edu.cn

Abstract: Binary code similarity detection serves as a critical front-line defense mechanism in cybersecurity, playing an indispensable role in identifying known vulnerabilities, detecting emergent malware families, and preventing intellectual property theft via code plagiarism. However, existing methods based on Control Flow Graphs (CFGs) often suffer from two major limitations: the inadequate capture of deep semantic information within CFG nodes, and the neglect of structural relationships across different functions. To address these issues, we propose Breg, a novel framework that synergistically integrates pre-trained semantic features with cross-graph structural features. Breg employs a BERT model pre-trained on a large-scale binary corpus to capture nuanced semantic relationships, and introduces a Cross-Graph Neural Network (CGNN) to explicitly model topological correlations between two CFGs, thereby generating highly discriminative embeddings. Extensive experimental validation demonstrates that Breg achieves leading F1-scores of 0.8682 and 0.8970 on Dataset3. In real-world vulnerability search tasks on Dataset4, Breg achieves an MRR@10 of 0.9333 in the challenging MIPS32-to-x64 search task, a clear improvement over the 0.8533 scored by the strongest baseline. This underscores its superior effectiveness and robustness across diverse compilation environments and architectures. To the best of our knowledge, this is the first work to integrate a pre-trained language model with cross-graph structural learning for binary code similarity detection, offering enhanced effectiveness, generalization, and practical applicability in real-world security scenarios.



Academic Editor: George Drosatos

Received: 20 September 2025

Revised: 23 October 2025

Accepted: 28 October 2025

Published: 30 October 2025

Citation: Yi, C.; Dai, W.; Deng, Y.; Bao, L.; Xu, G. Enhancing Binary Security Analysis Through Pre-Trained Semantic and Structural Feature Matching. *Appl. Sci.* **2025**, *15*, 11610. <https://doi.org/10.3390/app152111610>

Copyright: © 2025 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/>).

Keywords: vulnerability detection; binary similarity; language model; graph neural network; deep learning

1. Introduction

In the modern cybersecurity landscape, where acquiring source code is often impossible for proprietary software and complex open-source dependencies, direct binary analysis has become a critical and indispensable discipline. Binary analysis forms the bedrock of crucial security operations, including proactive vulnerability discovery, malware reverse engineering, and digital forensics [1]. The pervasive use of open-source libraries creates a vast and often unmonitored attack surface; a single vulnerability at the source-code level can be propagated across thousands of diverse hardware architectures and platforms, silently endangering systems worldwide. Alarming studies show that 80.4% of open-source projects contain known vulnerabilities, with some vulnerabilities persisting for over eight years within the third-party libraries of actively maintained software [2]. In this high-risk environment, binary code similarity detection emerges as a vital triage and threat-hunting

tool. It empowers security analysts to rapidly pinpoint known vulnerable functions, track malware variants, and identify plagiarized code [3] within a massive corpus of binaries, determining whether two code fragments are functionally equivalent even without access to their source.

The success of deep learning has inspired many to reframe binary similarity detection as a learnable classification task [4,5]. However, applying deep learning to this security-critical domain is fraught with challenges, primarily stemming from the inherent diversity and ambiguity of binary code. Two major roadblocks must be overcome to design a robust binary function embedding scheme.

First, the structure of binary code is highly mutable, varying significantly with different compilers, optimization flags, and target architectures. This diversity acts as a form of natural obfuscation, where the same source function can produce binaries with radically different Control Flow Graphs (CFGs) as in Figure 1. While many schemes employ Graph Neural Networks (GNNs) and siamese networks [5–7] to grapple with this structural complexity, they suffer from a critical flaw: they almost exclusively focus on features within a single graph. By ignoring the comparative cross-graph structural features between the two input functions, they lose essential context, which diminishes the quality of the resulting embeddings and weakens the model’s ability to make a reliable security assessment.

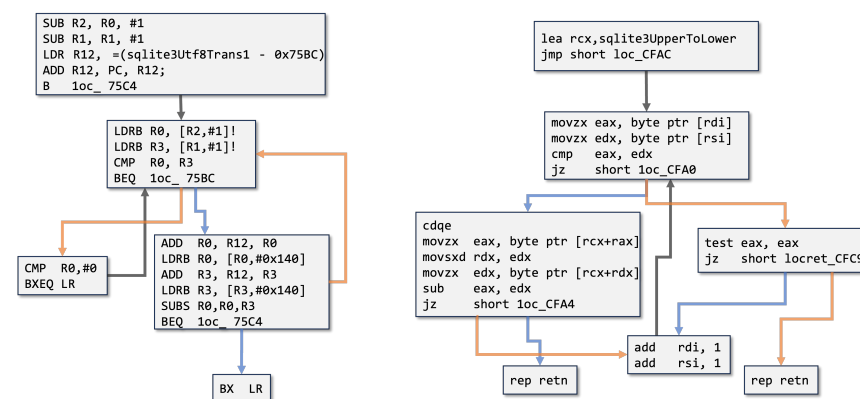


Figure 1. The function `sqlite3StrICmp` compiled under different platforms (x64 on the left, arm32 on the right) results in starkly different graph structures, posing a challenge for matching.

Second, learning the true semantic intent of binary instructions is a profound challenge. The functionality of a program is defined by the complex interplay and execution of its instructions. Yet, existing approaches are ill-equipped to holistically leverage both semantic and structural features, often prioritizing one at the expense of the other. For instance, well-known models like Gemini [5] and Genius [6] rely on an Attributed Control Flow Graph (ACFG) where node features are a hand-picked set of just eight statistical metrics. Such simplistic, manually engineered features are dangerously insufficient for capturing the nuanced logic that distinguishes a benign function from a cleverly disguised vulnerability or malicious payload. These approaches fail to grasp the deeper operational semantics of the code, limiting their defensive capabilities.

To counter these significant security challenges, the primary objective of this work is to develop a robust binary code similarity detection framework that effectively integrates deep semantic understanding with cross-graph structural analysis. To this end, we have engineered Breg, a pre-trained model designed from the ground up to integrate deep semantic understanding with robust structural analysis. To capture the rich semantic features inherent in binary code, we pre-train a BERT [8] model on a vast and diverse binary corpus. More importantly, we introduce a novel pre-training task specifically tailored to the unique characteristics of binary instructions, forcing the model to learn the

critical sequential and dependency relationships within basic blocks. To overcome the structural challenge, we employ a variant of GNN [9] that explicitly learns from cross-graph interactions, allowing it to “see” the similarities between two graphs rather than analyzing them in isolation. Our extensive evaluations demonstrate that Breg establishes a new state-of-the-art, outperforming existing methods in performance, generalization, and practical, real-world security scenarios.

The remainder of this paper is organized as follows: Section 2 introduces the background on BERT, binary function representation, and semantic complexity. Section 3 details the design of Breg, including semantic feature extraction and cross-graph structural feature extraction. Section 4 presents the experimental setup, results, and analysis across multiple datasets. Section 5 discusses the limitations of our work. Section 6 reviews the related work, and Section 7 concludes the paper.

To summarize, our key contributions to the field of binary security are as follows:

- We propose a novel approach that synergistically combines deep semantic features with cross-graph structural features, and to our knowledge, we are the first to integrate a pre-trained language model with a Cross-Graph Neural Network (CGNN) for binary security.
- We introduce a new pre-training task specifically designed for assembly language, enabling the model to learn the nuanced sequential nature of instructions, which is critical for understanding low-level code logic.
- We implement and validate a prototype system, Breg, and introduce four comprehensive datasets for training and evaluation. These include a real-world vulnerability dataset composed of CVEs, on which Breg demonstrates superior performance in vulnerability hunting tasks.
- Our experiments provide critical insights into the security-relevance of different feature types, quantifying the distinct impacts of semantic features, single-graph structural features, and cross-graph structural features on the task of binary code similarity detection.

2. Background

BERT. BERT [8] is a pre-trained model [10] based on the Transformer [11] architecture originating from the field of Natural Language Processing (NLP). The BERT architecture is derived from the Transformer’s encoder, which comprises multi-head self-attention. Initially, BERT’s pre-training aims to capture both token-level and sentence-level contextual information, constructing a universal model consisting of embedding vectors for each token. In this paper, we utilize the Masked Language Model (MLM) task proposed in BERT [8] to learn the relationships between binary instruction operands at the token level.

Representation of binary function. Some approaches directly represent binary functions using the raw binary byte stream, allowing the model to learn features autonomously [12–14]; some focus on structural features of binary code, including the structural characteristics of the binary function CFG [5–7] and the Abstract Syntax Tree (AST) [15,16]. During the transformation of functions into native binary bytes, it loses higher-level features such as semantics and call relationships, which can impact the model’s understanding of function functionality. As a structural feature, ASTs tend to emphasize semantic similarity and may not provide explicit control flow and data flow information. CFGs contain both function instruction information and can represent data dependency relationships between various blocks. Therefore, we believe that the CFG is the most suitable function representation for binary code similarity detection.

Semantic complexity. CFG nodes contain segments of binary instructions that exhibit a level of naturalness akin to programming languages [17,18]. These binary instructions

can be perceived as learnable semantic features. However, binary instructions differ from natural language in several aspects. Binary code exhibits diverse instruction formats and flexible execution orders. Listing 1 is a fragment of binary code from the x64 platform, containing two conditional jumps (jz, ja) and one unconditional jump (jmp), where the execution of the conditional jumps depends on their preceding instructions.

Listing 1. A fragment of binary code from the x64 platform.

test	r15, r15	# Test the value of register r15 (sets Zero Flag)
jz	loc_BEFC	# Jump to loc_BEFC if zero flag is set
cmp	rdx, 4	# Compare the value in register rdx with 4
ja	def_BE22	# Jump to def_BE22 if rdx > 4
lea	rax, jpt_BE22	# Load effective address of jpt_BE22 into rax
movsxd	rcx, ds:(jpt_BE22 - 25684Ch)[rax+rdx*4]	# A memory operand
add	rcx, rax	# Add rax to rcx (calculating a target address)
jmp	rcx	# Indirect jump to the address stored in rcx

For instance, the instruction “ja def_BE22” implies that, when the value in the “rdx” register is greater than 4, the program’s execution jumps to the address labeled as “def_BE22” instead of executing the immediately following instruction “lea rax, jpt_BE22”. Additionally, the instruction “movsxd rcx, ds:(jpt_BE22-25684Ch)[rax+rdx*4]” employs a segment override prefix (“DS”) to specify the data segment. Its complex memory operand combines a symbolic displacement (“jpt_BE22-25684Ch”), base and index registers (“rax” and “rdx”), and a scale factor (4). We devise an innovative pre-training task for BERT to learn the semantic features of binary instructions.

3. Design of Breg

3.1. Overview

In order to better address the challenges summarized in Section 1, we propose Breg, a pre-trained binary detection model. Breg is built upon BERT [8] and GNN [9], which correspond to semantic feature extraction and structural feature extraction, respectively, and incorporates the following important design considerations. As shown in Figure 2, our system takes two native binary functions as input, tokenizes their instructions, and extracts semantic features. To address the semantic learning challenges arising from the distinctive dependency relationships in binary instructions compared to natural language, we design a semantic feature extraction component with cross-platform generality. The semantic feature extraction component is based on BERT design; we select a corpus dataset comprising approximately 150 million instructions from various platforms for pre-training the BERT.

In response to the example regarding instruction semantic features raised in Section 2, we design a novel pre-training task called instruction sequence prediction (ISP). This task generates some shuffled sequences of strongly ordered instructions, aiding the BERT model in better understanding the complex relationships between instructions within functions. Furthermore, we introduce a cross-graph structure feature extraction component to learn the complex structural features of binary functions.

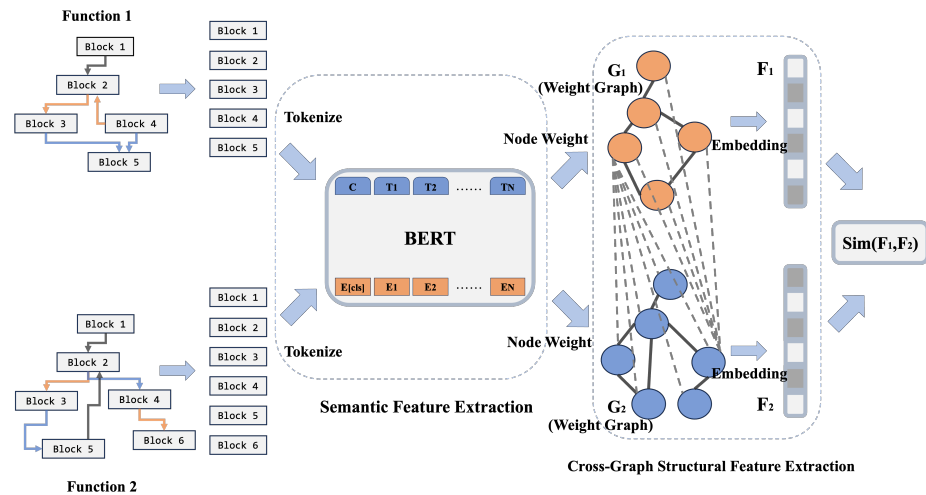


Figure 2. System design of Breg. Block represents instructions within CFG nodes, G_1 and G_2 represent CFGs after semantic feature extraction, and F_1 and F_2 are embedding vectors for functions.

3.2. Semantic Feature Extraction

We pre-train a BERT model for the semantic feature extraction part of our model, and this section introduces our training data generation strategy and pre-training task design.

3.2.1. Pre-Training Data Generation

The pretraining dataset is compiled from open-source projects, and instructions are extracted from their CFGs.

Initially, each block within the CFG is treated as a natural paragraph, and each instruction is considered a sentence. They are arranged in the order of execution within the block.

Subsequently, each instruction is tokenized using a fine-grained strategy, where each indivisible element in the instruction serves as a token. For example, given the instruction “mov dword ptr [esp + 8], 0”, it is broken down into tokens such as “mov”, “dword”, “ptr”, “[”, “esp”, “+”, “8”, “]”, “0”. To mitigate the impact of Out-of-Vocabulary (OOV) words, a special token “[str]” is used to replace strings. For large constants (consisting of at least five hexadecimal digits), another special token “[addr]” is used to normalize their exact values.

3.2.2. Pre-Training Task

Our pre-training task divides the semantic features of binary instructions into token-level and instruction-level. We employ the MLM introduced in Section 2 to learn the relationships between operation codes within binary instructions. In addition, we introduce the ISP task to learn the sequence and call relationships between instructions within functions, aiming to achieve higher-quality semantic feature extraction.

Initially, we track variables or register names that are repeatedly called within a limited instruction context. These instructions with repeatedly called variable names reflect the relationships between instructions. When a variable name appears repeatedly in three consecutive lines of instructions, we consider these three lines of code to exhibit strong sequential correlation, as shown in Listing 2. We shuffle the order of instructions with strong sequential correlation, enabling the model to learn the semantic relationships between instructions with strong sequential correlation.

Listing 2. This is a snippet of code with strong sequential correlation, used for the pretraining task ISP. The instructions I_1 , I_2 , and I_3 form a strongly ordered sequence due to shared register dependencies (rax, rcx).

```
lea    rax, jpt_BEf22          # I1: Load base address into rax
movsxd rcx, ds:(jpt_BEf22 - 25684Ch)[rax+rdx*4] # I2: Memory load using rax, to rcx
add    rcx, rax                # I3: Use both rax and rcx for address calculation
```

For example, for three instructions with the natural order $I_1||I_2||I_3$, when generating pretraining data for them, consider $(I_1||I_2, 0)$ as instruction pairs in their normal order and $(I_3||I_2, 1)$ as counterexamples where the natural order is swapped.

For each instruction pair i ,

$$p(\hat{i}|i) = \frac{1}{1 + \exp(i_{ISP})} \quad (1)$$

$$L_{ISP} = - \sum_{i \in P} \log p(\hat{i}|i) \quad (2)$$

where i_{ISP} is the ISP label for instruction pair i , and P represents all instruction pairs within the entire pretraining dataset.

The loss function for the pretraining model is the sum of two task's losses:

$$L = L_{MLM} + L_{ISP} \quad (3)$$

3.3. Cross-Graph Structural Feature Extraction

The structural features are equally important for representing binary functions. In Breg, a cross-graph neural network (CGNN) is employed to learn the structural features of the CFG. The input to the CGNN is the CFG after semantic feature extraction, and it outputs a 128-dimensional vector as the representation of the binary function. As shown in Figure 3, the CGNN consists of an embedding layer, a fusion layer, and a pooling layer.

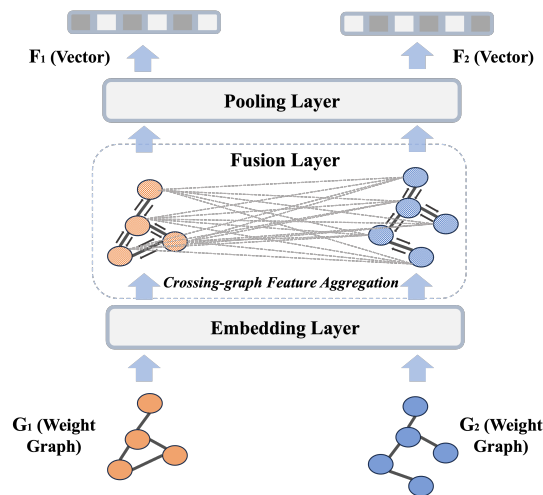


Figure 3. Cross-Graph Neural Network.

We first define two graphs, $G1 = (N_1, E_1)$ and $G2 = (N_2, E_2)$, where N and E represent the sets of nodes and edges in the graphs, respectively. We consider f_i as node features, where $i \in N$. Since there is no feature information on the CFG edges, edge features are not considered.

The CFG first undergoes node feature embedding in the embedding layer, which is composed of a Multilayer Perceptron (MLP), for each $i \in N$:

$$\gamma_i = MLP(f_i) \quad (4)$$

The fusion layer also consists of MLPs. MLP_{fusion} is used to aggregate node features within the current graph, thereby learning the topological relationships among nodes within the current graph. MLP_{cross} utilizes an attention mechanism to aggregate node features across graphs, combining information from both inter-graph and intra-graph topological relationships to update node features. Assuming the current state of node i is γ_i^t , the updated state of node i , γ_i^{t+1} , is as follows:

$$\gamma_i^{t+1} = MLP_{cross}(\gamma_i^t, \sum_j s_{ji}, \sum_{j'} Q_{j'i}) \quad (5)$$

where s_{ji} represents the current features of all nodes j connected to node i in the current graph (i.e., $(i, j) \in E$) after fusion through MLP_{fusion} . $Q_{j'i}$ represents the cross-graph coefficients between all nodes j' in another graph and node i :

$$s_{ji} = MLP_{fusion}(\gamma_i^t, \gamma_j^t) \quad (6)$$

$$Q_{ji} = \mu_{ji}(\gamma_i^t - \gamma_j^t) \quad (7)$$

$$\mu_{ji} = \frac{\exp(D_\gamma(\gamma_i^t, \gamma_j^t))}{\sum_{j'} \exp(D_\gamma(\gamma_i^t, \gamma_{j'}^t))} \quad (8)$$

In (8), D_γ is the dot product function used to compute the vector distance between γ_i^t and γ_j^t .

The pooling layer is used to compute the representation vectors for G_1 and G_2 . After a certain number of t fusion rounds, the pooling layer takes the current states of all nodes as input and computes the representation vectors. MLP_t computes the node states γ_i^t after t rounds of fusion, while $MLP_{pooling}$ aggregates the states of all nodes in graph G and outputs them as a vector representation F_G .

$$\alpha_i^t = MLP_t(\gamma_i^t) \quad (9)$$

$$F_G = MLP_{pooling}(\sum_{i \in N} \text{Sigmoid}(\gamma_i^t) \cdot \alpha_i^t) \quad (10)$$

Unlike other similarity functions, we use the $\tanh$ activation function to ensure that each element i of vectors F_{G1} and F_{G2} satisfies $i \in [-1, 1]$. Let F_{G1} and F_{G2} both be vectors of length m , and their element sets are denoted as M , where i corresponds to an element within the vector:

$$\text{Sim}(F_{G1}, F_{G2}) = \frac{1}{m} \sum_{i \in M} \tanh(F_{G1i}) \cdot \tanh(F_{G2i}) \quad (11)$$

To calculate the loss L for training the model, where y represents the true labels for CFG pairs $G1$ and $G2$:

$$L = 0.25 * (y - \text{Sim}(F_{G1}, F_{G2}))^2 \quad (12)$$

The coefficient 0.25 in the loss function scales the squared error term to the range $[0, 1]$. This is because the similarity score $\text{Sim}(F_{G1}, F_{G2}) \in [-1, 1]$ and the label $y \in [-1, 1]$, leading to $(y - \text{Sim}(F_{G1}, F_{G2}))^2 \in [0, 4]$. This scaling promotes stable gradient behavior and facilitates more efficient model convergence during training. Thereby, the final loss

is bounded within the range $[0, 1]$, which ultimately encourages positive pairs to have a similarity close to 1 and negative pairs close to -1 .

4. Evaluation

In this section, we first introduce the experimental datasets, experimental environment, and baselines. Then, we report and discuss the comparative experimental results of Breg with five other baselines.

4.1. Datasets

The experimental datasets are sourced from various mainstream open-source projects, encompassing six platforms: arm32, arm64, mips32, mips64, x86, and x64. Multiple different copies of the same binary function are generated by compiling it with different compilers, compiler versions (gcc, clang), and different compilation optimization levels. We generate four datasets for Breg:

- **Dataset1:** Dataset1 is compiled from three open-source projects: clamav, curl, and nmap, each compiled on six different platforms: arm32, arm64, mips32, mips64, x86, and x64. Compilation is performed using eight different compilers, including different versions of gcc and clang. The code is compiled using five optimization levels: (O0, O1, O2, O3, OS). Approximately 153.2 million assembly instructions from around 3,531,451 functions, serving as Breg's pretraining dataset.
- **Dataset2:** Dataset2 is compiled from three open-source projects: openssl, unrar, and zlib, following the same compilation rules as Dataset1. It comprises a total of 1,148,156 functions. Dataset2 is used to assess the effectiveness of Breg.
- **Dataset3:** Dataset3 is compiled from a set of lightweight but diverse open-source projects, including binutils, coreutils, gmp, sqlite, and others. It is expected to be smaller in scale compared to Dataset2. Compilation for this dataset is performed on six platforms using gcc 7.0 with optimization levels (O0, O1, O2 and O3). Dataset3 includes a total of 373,993 functions and is used to test the flexibility and generality of Breg.
- **Dataset4:** Dataset4 consists of real CVE vulnerability functions extracted from openssl. Initially, multiple CVE vulnerability functions from openssl 1.0.2d are compiled for four different platforms, creating a vulnerability library. Then, openssl 1.0.2h is compiled as the target sample for vulnerability detection on two platforms. Dataset4 is used to evaluate the practicality of Breg in the context of binary vulnerability detection.

When generating the dataset, for each binary function, we select another function with the same name but compiled in a different environment as a positive pair ($label = 1$). Then, we randomly select a function with a different name as a negative pair ($label = -1$). This way, we obtain a dataset consisting of 50% positive and 50% negative function pairs. In the function pair dataset, we use 80% for the training set and 10% each for the validation and test sets.

4.2. Experimental Setup

Baselines and configuration. We select Gemini, GMN, Instruction2vec, and Palmtree as baselines for our experiments, setting their outputs to 128-dimensional vectors.

During the pretraining of Breg, we configure it with $Layers = 12$, $Heads = 8$, $Hiddens = 128$, and set the batch size to 512. We use the Adam [19] optimizer with a learning rate of 0.00005 and train it for over 1.5 million iterations. For the training of Breg, we set the batch size to 32, use the Adam optimizer with a learning rate of 0.0005, and train it for 20 epochs.

Environment. We train and evaluate Breg on a server equipped with two Intel Xeon Gold 6234 CPUs (Intel Corporation, Santa Clara, CA, USA), an NVIDIA Quadro RTX 5000 GPU (NVIDIA Corporation, Santa Clara, CA, USA), and 128GB of RAM.

Evaluation metrics. Following the previous work [5,20,21], the binary code similarity detection problem can be viewed as a binary classification problem, where the model needs to determine whether the input function pairs are similar. In the effectiveness and generality evaluation, we use the following metrics: Precision (P), Recall (R), F1 score (F), Accuracy (A), and AUC. TP , FP , TN , and FN represent the number of true positives, false positives, true negatives, and false negatives, respectively. The AUC value corresponds to the area under the ROC curve. In the practicality evaluation, we use MRR@10 (Mean Reciprocal Ranking) as the performance metric. The $rank_i$ denotes the ranking of the correct answer for the query i . The evaluation metrics are defined as follows:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN} \quad (13)$$

$$A = \frac{TP + TN}{TP + TN + FN + FP}, F = \frac{2 * P * R}{P + R} \quad (14)$$

$$MRR@10 = \frac{1}{10} \sum_i^{10} \frac{1}{rank_i} \quad (15)$$

In the context of binary code similarity detection, a True Positive (TP) indicates that our model correctly identified a pair of functionally equivalent functions as similar. A False Positive (FP) occurs when the model incorrectly labels a pair of dissimilar functions as similar. Consequently, Precision reflects the model's reliability—a high precision means that, when Breg predicts a “similar” match, we can be highly confident it is correct, which is critical for minimizing false alarms in vulnerability search. Recall measures the model's ability to find all truly similar pairs. A high recall is essential for ensuring that known vulnerabilities are not missed during security analysis. The F1-score balances these two concerns. Accuracy provides an overall measure of correctness, while the AUC evaluates the model's ranking capability across all classification thresholds. In our task, a high F1-score is desirable, with their relative importance depending on the specific security scenario. Different models use their own distance functions. During the evaluation, we set the threshold for Breg to 0.295, GMN's threshold to 0.302, Gemini's threshold to 0.412, Palmtree's threshold to 0.895, and Instruction2vec's threshold to 0.453.

4.3. Effectiveness

The effectiveness evaluation reflects the theoretical performance of the model in common scenarios. Therefore, we choose the most representative dataset (Dataset2) to compare and assess Breg's performance. We first train Breg and the baselines to convergence on the training set of Dataset2 and then compare their performance on the test set. Figure 4A shows the ROC curve for the effectiveness evaluation, and Table 1 presents the performance metrics.

Table 1. Comparison of performance metrics in the Dataset2. The best results are highlighted in **bold**.

Model	Accuracy	Precision	Recall	F1-Score	AUC
Breg	0.8747	0.8624	0.8741	0.8682	0.9652
Breg(MLM)	0.8713	0.8624	0.8711	0.8667	0.9487
GMN	0.8653	0.8702	0.8653	0.8677	0.9441
Gemini	0.8547	0.8683	0.8357	0.8572	0.9387
Palmtree	0.7425	0.8028	0.7431	0.7718	0.9162
Instruction2vec	0.8245	0.8545	0.8250	0.8395	0.9379

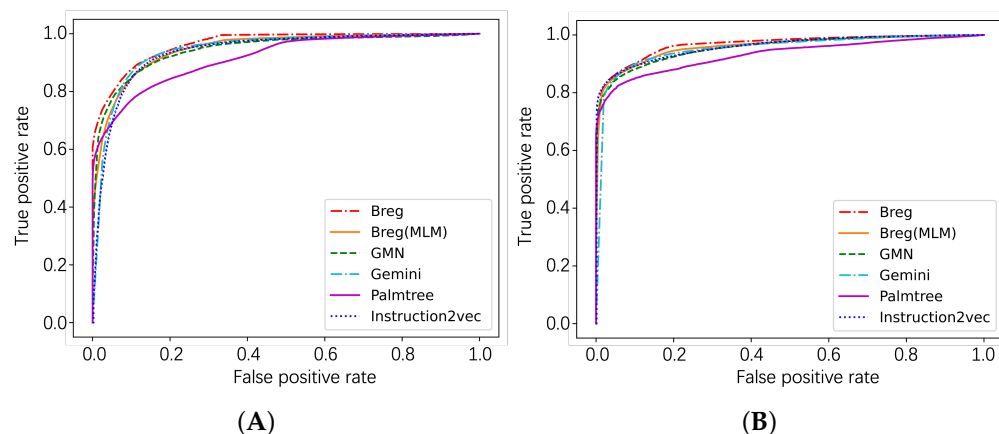


Figure 4. ROC curve in the different datasets. (A) ROC curve in the Effectiveness Evaluation. (B) ROC curve in the Generality Evaluation.

To further validate the effectiveness of our approach and its stability across different compilation environments, we evaluate multiple subsets from the Dataset2 test set. These subsets are compiled with the same compiler but come from six different compilation platforms and four different compilation optimization levels (O0, O1, O2, and O3). The results are depicted in Figure 5.

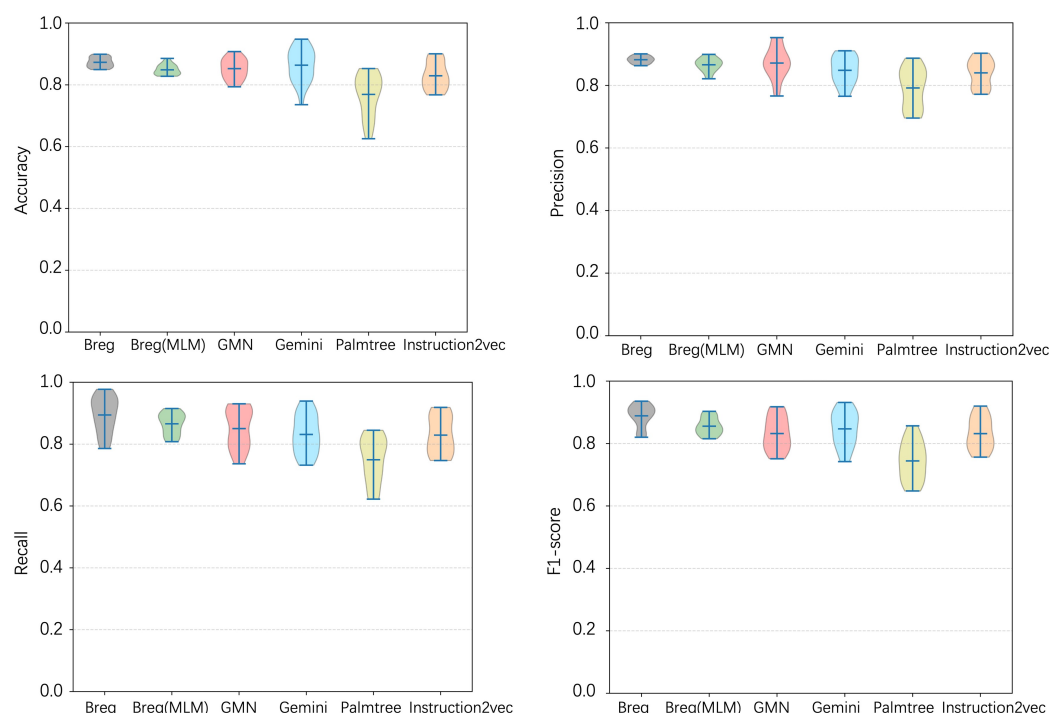


Figure 5. Accuracy, Precision, Recall, and F1-score for subsets of the Dataset2 testset compiled in different environments.

Instead of showing a single aggregate score, Figure 5 presents the distribution of performance metrics (Accuracy, Precision, Recall, F1-score) across all these subsets. This allows us to see not only the average performance but also the variance, which is critical for assessing a model's practical reliability. In the results shown in Figure 5, the stability of Breg and Breg(MLM) surpasses that of the other models, even though their recall exhibits significant fluctuations. Palmtree's performance exhibited significant fluctuations, and its average value is the lowest among the baselines.

4.4. Generality

We evaluate Breg’s performance on Dataset3 to simulate the assessment of the model’s performance in different detection scenarios. Additionally, we conduct cross-testing experiments by assessing the model trained on Dataset2 on the Dataset3 test set using the same strategy applied to the Dataset2 test set.

The ROC curves for the experiments are shown in the Figure 4B, and the performance metrics for the experiments are presented in Table 2. The results of the cross-experiment analysis are shown in Table 3.

The results in Table 2 show that even in another completely different dataset, Breg still achieves excellent performance, demonstrating its ability to generalize and its versatility.

Table 2. Comparison of performance metrics in Dataset3. The best results are highlighted in **bold**.

Model	Accuracy	Precision	Recall	F1-Score	AUC
Breg	0.8940	0.8985	0.8955	0.8970	0.9681
Breg(MLM)	0.8813	0.8924	0.8911	0.8917	0.9599
GMN	0.8840	0.8973	0.8931	0.8952	0.9555
Gemini	0.8822	0.8857	0.8840	0.8848	0.9532
Palmtree	0.7578	0.7997	0.7537	0.7861	0.9316
Instruction2vec	0.8822	0.8840	0.8822	0.8821	0.9561

Table 3. Performance metrics when trained on Dataset2 and tested on Dataset3. The best results are highlighted in **bold**.

Model	Accuracy	Precision	Recall	F1-Score	AUC
Breg	0.8778	0.8839	0.8778	0.8808	0.9495
Breg(MLM)	0.8637	0.8731	0.8772	0.8752	0.9518
GMN	0.8427	0.8581	0.8427	0.8503	0.9300
Gemini	0.8649	0.8639	0.8441	0.8539	0.9561
Palmtree	0.7578	0.7997	0.7578	0.7782	0.9316
Instruction2vec	0.8859	0.8791	0.8759	0.8775	0.9586

4.5. Practicality

As shown in Table 4, our Dataset4 contains 9 CVE vulnerabilities and 10 vulnerable functions. We compile vulnerable functions separately on four platforms: arm32, mips32, x86, and x64. We also compile the functions to be detected on arm32 and mips32 platforms. It’s important to note that the vulnerable functions and the functions to be detected are not the same; they come from different versions of openssl. We conduct vulnerability retrieval for the functions to be detected on both arm32 and mips32 platforms using Dataset4, and the results are shown in Table 5.

Table 4. CVE vulnerability functions.

CVE Number	Vulnerability Function
CVE-2014-3508	OBJ_obj2txt()
CVE-2022-0778	BN_mod_sqrt()
CVE-2023-0215	BIO_new_NDEF ()
CVE-2019-1563	CMS_decrypt_set1_pkey(),PKCS7_dataDecode()
CVE-2023-0466	X509_VERIFY_PARAM_add0_policy()
CVE-2021-3712	ASN1_STRING_set()
CVE-2016-2176	X509_NAME_oneline()
CVE-2016-2182	BN_bn2dec()
CVE-2021-23841	X509_issuer_and_serial_hash()

The results in Table 5 indicate that Breg performs the best in the practicality evaluation. Palmtree, which only learns semantic features, is useful only when retrieving functions compiled from the same platform. This may be because, when learning only semantic features, the model is overly sensitive to instruction formats.

Table 5. Comparison of MRR@10 in Practicality Evaluation on Dataset4. The best results are highlighted in **bold**.

Model	ARM32				MIPS32			
	arm32	mips32	x64	x86	arm32	mips32	x64	x86
Breg	1.000	0.8333	1.000	0.8500	0.8500	0.8833	0.9333	0.8583
Breg(MLM)	0.9333	0.8583	1.000	0.8200	0.8500	0.8583	0.8533	0.7833
GMN	0.7667	0.6450	0.6700	0.7833	0.5921	0.7333	0.4308	0.3839
Gemini	0.9000	0.7283	0.7417	0.6867	0.6417	0.7950	0.5733	0.5683
Palmtree	0.8750	0.3790	0.2990	0.2929	0.2962	0.7850	0.2762	0.2929
Instruction2vec	1.000	0.8200	0.8833	0.7950	0.7367	0.8833	0.8533	0.7950

4.6. Visual Representation

Visual representation evaluation assesses the model's ability to retain source function information when generating function embeddings. We compile these six randomly chosen functions into nine different copies on arm32, mips32, and x86 platforms using gcc 7.0 with optimization levels O0, O1, and O2. We use T-Distributed Stochastic Neighbor Embedding (T-SNE) [22] to reduce the dimensionality of binary function embedding vectors to two-dimensional space. Figure 6 presents the results of the visualization standard evaluation, with the legend providing the names of the selected functions.

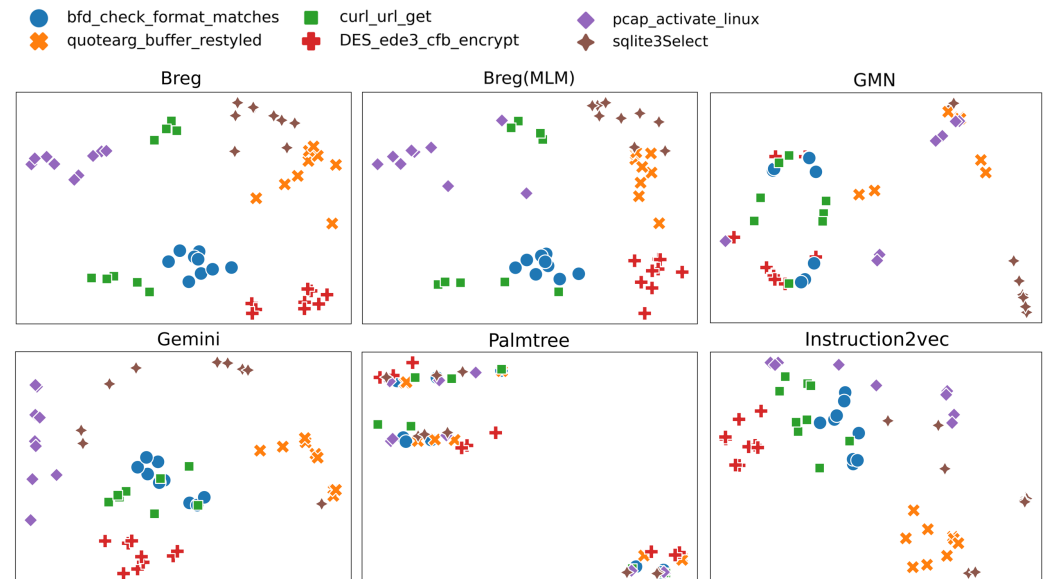


Figure 6. The visualization representation of Breg and other models.

4.7. Model Complexity Analysis

To provide a comprehensive comparison of the computational requirements of different models, we analyze the model complexity in terms of the number of parameters, training time per epoch, and GPU memory usage. The results are summarized in Table 6.

As shown in Table 6, Breg has the highest parameter count (15.2M) among all compared models, which is expected given its dual-component architecture combining BERT and the CGNN. The training time and GPU memory usage of Breg are also the highest, reflecting

its more complex architecture and the computational overhead of cross-graph feature extraction. Breg(MLM), which uses only the MLM pre-training task, shows slightly reduced training time while maintaining the same parameter count.

Table 6. Model complexity comparison.

Model	Parameters (M)	Training Time (Hours/Epoch)	GPU Memory (GB)
Breg	15.2	3.5	8.2
Breg(MLM)	15.2	3.2	8.2
GMN	8.7	2.1	6.5
Gemini	6.3	1.8	5.2
Palmtree	12.5	2.8	7.3
Instruction2vec	4.2	1.2	4.1

Among the baseline models, Instruction2vec has the lowest complexity across all metrics, as it relies on a simpler embedding approach without graph structural analysis. Gemini and GMN, being graph-based methods, show moderate complexity, while Palmtree, which also employs a pre-trained language model, has a higher parameter count and memory requirements comparable to Breg(MLM).

These complexity metrics should be considered alongside the performance results presented in previous sections, as Breg’s higher computational cost is justified by its superior performance in binary code similarity detection tasks.

5. Limitation

In this paper, we do not consider the compiler’s name mangling mechanism and code obfuscation techniques, so Breg primarily focuses on non-obfuscated functions. Handling obfuscated code remains a challenging yet important direction for future work. We plan to extend Breg to improve its robustness in such scenarios. Potential solutions include incorporating techniques for fuzzy matching of semantically equivalent basic blocks [23] and leveraging insights from cross-architecture semantic similarity approaches such as INNEREYE [24]. These enhancements are expected to broaden the practical applicability of Breg in real-world security analysis involving obfuscated binaries.

Additionally, Breg exhibits certain limitations in terms of computational performance and scalability. The pre-training phase, which involves processing a large corpus of binary instructions (e.g., over 150 million instructions), requires substantial computational resources and time. Although the model achieves high accuracy in similarity detection, its training and inference efficiency could be further optimized for deployment in resource-constrained environments or real-time analysis scenarios. Moreover, the current CGNN-based structural feature extraction, while effective, may face scalability challenges when processing extremely large or densely connected control flow graphs, as the cross-graph attention mechanism involves pairwise node comparisons.

We are also concerned with how to further enhance the performance and efficiency of Breg. First, approaches such as model compression without significant performance degradation are proposed to accelerate training, as presented in Xu et al.’s work [25]. MiniLM [26] employs deep self-attention knowledge distillation to transfer semantic information and expedite training. Regarding cross-graph feature extraction, methods like the MGNN [27] calculate graph similarity by aggregating information from each node of one graph to the other whole graph. Additionally, the H2MN [28] learns graph representations from the perspective of hypergraphs and performs subgraph matching on each hyperedge to capture rich substructure similarities within the entire graph. These works serve as inspiration for our future directions.

6. Related Work

Traditional approaches. Many existing works are seeking to map binary functions to low-dimensional representations, which include fuzzy hash computations or embedding vectors. Unlike traditional cryptographic hash algorithms, fuzzy hash algorithms are intentionally designed to map similar input values to similar hashes, making the hash values of similar inputs as close as possible [29,30].

Another form of low-dimensional representation relies on embedding vectors, which refer to using fixed-length vectors to represent input functions. This approach typically requires the use of intermediate code representations such as the CFG and the AST [15] as an abstract representation of input code, manually extracting feature vectors from the AST to obtain function embedding vectors. Genius [6] converts CFGs into high-level numerical feature vectors. VulDetector [31] slices the control flow graph of functions by identifying vulnerability-sensitive keywords, reducing the size of the graph without affecting security-related semantics.

Learning-based binary similarity detection approaches are generally divided into semantic-based learning and structure-based learning approaches. Siamese networks [32,33], as a similarity-based neural network architecture, are widely employed in the field of binary similarity.

Semantic based learning. MalConv [13] employs this method to process the entire raw byte stream of a program, classifying malware and performing coarse-grained value set analysis. To enhance instruction understanding, DeepVSA [12] constructs an abstract memory model, recovers variables in executable code represented by abstract addresses, and statistically computes the set of values that abstract addresses may contain for each instruction.

Additionally, many studies treat binary code as text and leverage existing NLP techniques to address binary function similarity problems, such as Asm2vec [34] and Instruction2vec [35], which are based on Word2vec [36], a widely used model in the NLP field. SAFE [37] allowing semantic mappings from different architectures to share the same embedding space without manual feature extraction, thus learning cross-architecture similarities. OrderMatters [38], Palmtree [21], and BinShot [39] design various tasks to pre-train BERT models on certain binary characteristics to generate basic block embeddings, while Trex [40] uses layered Transformers and masked language modeling tasks to learn approximate program execution semantics and then transfer this knowledge to identify semantically similar functions.

Structural based learning. Another research direction builds upon code-structured graph embedding machine learning approaches. These are particularly suitable for capturing characteristics based on functional CFGs, which inherently stem from the distinct flow structures present in different binaries. These embeddings can be generated using machine learning models such as Structure2vec [7] and GNN [9]. Gemini [5] is the first to incorporate GNN into the field of binary similarity. There are also variants of GNNs, such as the graph match network (GMN) [20].

7. Conclusions

In this paper, we introduce Breg, a binary code similarity detection model that combines semantic features and cross-graph structural features using BERT and CGNN. Addressing the unique characteristics of binary instructions, we propose a new BERT pre-training task called an ISP within the Breg framework and train it in conjunction with cross-graph structural features. Experimental results demonstrate that our model outperforms existing models in terms of theoretical performance and practicality. It also exhibits robust generalization capabilities, making it resilient to variations in architecture, compilers, and other compilation environments.

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

Funding: This research was funded by the Ministry of Science and Technology of the People's Republic of China, the Research on Digital Identity Trust Systems for Massive Heterogeneous Terminals in Road Traffic Systems (Grant No. 2022YFB3104400).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

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

References

1. 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.
2. Cui, A.; Costello, M.; Stolfo, S. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. Available online: <https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/when-firmware-modifications-attack-case-study-embedded-exploitation/> (accessed on 23 October 2025).
3. Brosch, T.; Morgenstern, M. Runtime packers: The hidden problem. Available online: https://www.av-test.org/fileadmin/pdf/publications/blackhat_2006_avtest_presentation_runtime_packers-the_hidden_problem.pdf (accessed on 23 October 2025).
4. 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), Boston, MA, USA, 10–12 August 2022; pp. 2099–2116.
5. 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.
6. 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.
7. Song, L. Structure2vec: Deep Learning for Security Analytics over Graphs. 2017. Available online: <https://www.usenix.org/conference/scainet18/presentation/song> (accessed on 23 October 2025).
8. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
9. Zhou, J.; Cui, G.; Hu, S.; Zhang, Z.; Yang, C.; Liu, Z.; Wang, L.; Li, C.; Sun, M. Graph neural networks: A review of methods and applications. *AI Open* **2020**, *1*, 57–81. [CrossRef]
10. Qiu, X.; Sun, T.; Xu, Y.; Shao, Y.; Dai, N.; Huang, X. Pre-trained models for natural language processing: A survey. *Sci. China Technol. Sci.* **2020**, *63*, 1872–1897. [CrossRef]
11. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Volume 30.
12. Guo, W.; Mu, D.; Xing, X.; Du, M.; Song, D. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019.
13. Raff, E.; Barker, J.; Sylvester, J.; Brandon, R.; Catanzaro, B.; Nicholas, C.K. Malware detection by eating a whole exe. In Proceedings of the Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018.
14. Liu, B.; Huo, W.; Zhang, C.; Li, W.; Li, F.; Piao, A.; Zou, W. α diff: 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.
15. Jiang, L.; Mishnerghi, G.; Su, Z.; Glondou, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105. [CrossRef]
16. 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.

17. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–37. [\[CrossRef\]](#)
18. Hindle, A.; Barr, E.T.; Gabel, M.; Su, Z.; Devanbu, P. On the naturalness of software. *Commun. ACM* **2016**, *59*, 122–131. [\[CrossRef\]](#)
19. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* **2017**, arXiv:1412.6980. [\[CrossRef\]](#)
20. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In Proceedings of the International Conference on Machine Learning, PMLR, Long Beach, CA, USA, 9–15 June 2019; pp. 3835–3845.
21. Li, X.; Qu, Y.; Yin, H. Palmtree: Learning an assembly language model for instruction embedding. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 15–19 November 2021; pp. 3236–3251.
22. Van der Maaten, L.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.
23. 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.
24. Zuo, F.; Li, X.; Young, P.; Luo, L.; Zeng, Q.; Zhang, Z. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In Proceedings of the 2019 Network and Distributed System Security Symposium. Internet Society, San Diego, CA, USA, 24–27 February 2019. [\[CrossRef\]](#)
25. Xu, C.; Zhou, W.; Ge, T.; Wei, F.; Zhou, M. BERT-of-Theseus: Compressing BERT by Progressive Module Replacing. *arXiv* **2020**, arXiv:2002.02925. [\[CrossRef\]](#)
26. Wang, W.; Wei, F.; Dong, L.; Bao, H.; Yang, N.; Zhou, M. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. *arXiv* **2020**, arXiv:2002.10957. [\[CrossRef\]](#)
27. Ling, X.; Wu, L.; Wang, S.; Ma, T.; Xu, F.; Liu, A.X.; Wu, C.; Ji, S. Multilevel graph matching networks for deep graph similarity learning. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *34*, 799–813. [\[CrossRef\]](#) [\[PubMed\]](#)
28. Zhang, Z.; Bu, J.; Ester, M.; Li, Z.; Yao, C.; Yu, Z.; Wang, C. H2mn: Graph similarity learning with hierarchical hypergraph matching networks. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, Singapore, 14–18 August 2021; pp. 2274–2284.
29. Dullien, T. Searching Statically-Linked Vulnerable Library Functions in Executable Code, 18 December 2018. Available online: <https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html> (accessed on 23 October 2025).
30. Pagani, F.; Dell’Amico, M.; Balzarotti, D. Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis. In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, Tempe, AZ, USA, 19–21 March 2018; pp. 354–365.
31. Cui, L.; Hao, Z.; Jiao, Y.; Fei, H.; Yun, X. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Trans. Inf. Forensics Secur.* **2020**, *16*, 2004–2017. [\[CrossRef\]](#)
32. Melekhov, I.; Kannala, J.; Rahtu, E. Siamese network features for image matching. In Proceedings of the 2016 23rd International Conference on Pattern Recognition (ICPR), Cancun, Mexico, 4–8 December 2016; pp. 378–383.
33. Sun, H.; Cui, L.; Li, L.; Ding, Z.; Hao, Z.; Cui, J.; Liu, P. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Comput. Secur.* **2021**, *110*, 102417. [\[CrossRef\]](#)
34. Ding, S.H.H.; Fung, B.C.M.; 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. [\[CrossRef\]](#)
35. Lee, Y.; Kwon, H.; Choi, S.H.; Lim, S.H.; Baek, S.H.; Park, K.W. Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN. *Appl. Sci.* **2019**, *9*, 4086. [\[CrossRef\]](#)
36. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781. [\[CrossRef\]](#)
37. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Querzoni, L.; Baldoni, R. Function representations for binary similarity. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2259–2273. [\[CrossRef\]](#)
38. 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.
39. Ahn, S.; Ahn, S.; Koo, H.; Paek, Y. Practical binary code similarity detection with bert-based transferable similarity learning. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; pp. 361–374.
40. Pei, K.; Xuan, Z.; Yang, J.; Jana, S.; Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv* **2020**, arXiv:2012.08680.

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.