

GraphConvDeep: A Deep Learning Approach for Enhancing Binary Code Similarity Detection using Graph Embeddings

Nandish M.*

Dept. of Computer Science & Engineering, JNNCE Shivamogga, Visvesvaraya Technological University, Belagavi – 590018, India

E-mail: nandishbdvt@gmail.com

ORCID iD: <https://orcid.org/0009-0005-7979-0758>

*Corresponding author

Jalesh Kumar

Dept. of Computer Science & Engineering, JNNCE Shivamogga, Visvesvaraya Technological University, Belagavi – 590018, India

E-mail: jalesh_k@yahoo.com

ORCID iD: <https://orcid.org/0000-0001-9600-1398>

Mohan H. G.

Dept. of Computer Science & Engineering, JNNCE Shivamogga, Visvesvaraya Technological University, Belagavi – 590018, India

E-mail: mohan.hg0408@gmail.com

ORCID iD: <https://orcid.org/0000-0001-9064-1749>

Manjunath Sargur Krishnamurthy

JP Morgan & Chase Co., Houston, USA

E-mail: manjunath.skmurthy@yahoo.com

ORCID iD: <https://orcid.org/0009-0001-2598-1090>

Received: 28 August 2024; Revised: 09 March 2025; Accepted: 04 April 2025; Published: 08 June 2025

Abstract: Binary code similarity detection (BCSD) is a method for identifying similarities between two or more slices of binary code (machine code or assembly code) without access to their original source code. BCSD is often used in many areas, such as vulnerability detection, plagiarism detection, malware analysis, copyright infringement and software patching. Numerous approaches have been developed in these areas via graph matching and deep learning algorithms. Existing solutions have low detection accuracy and lack cross-architecture analysis. This work introduces a cross-platform graph deep learning-based approach, i.e., GraphConvDeep, which uses graph convolution networks to compute the embedding. The proposed GraphConvDeep approach relies on the control flow graph (CFG) of individual binary functions. By evaluating the distance between two embeddings of functions, the similarity is detected. The experimental results show that GraphConvDeep is better than other cutting-edge methods at accurately detecting similarities, achieving an average accuracy of 95% across different platforms. The analysis shows that the proposed approach achieves better performance with an area under the curve (AUC) value of 96%, particularly in identifying real-world vulnerabilities.

Index Terms: Binary Code Similarity Detection, Deep Learning, Graph Convolution Networks, Graph Embedding.

1. Introduction

Binary code represents machine-level instructions in a form that computers can execute directly. The goal of similarity detection is to find common patterns or structures between different binary files. Binary code similarity detection involves comparing binary files or executables to identify similarities or differences among them. This process is crucial in various areas, such as cybersecurity, patch detection [1,2], malware analysis [3–5], and software development.

In terms of purpose and applications, BCSD plays a significant role in security by identifying similarities between known and unknown malware to enhance threat detection [6]. BCSD significantly contributes to the security domain by enabling the identification of reused or modified code, which often signals the presence of malware or software vulnerabilities [7–9]. It plays a crucial role in early threat detection by recognizing shared binary code fragments across different malware variants, thereby uncovering hidden connections between cyber-attacks and facilitating comprehensive threat analysis. BCSD enhances malware detection tools by identifying obfuscated or altered code, making it more difficult for intruders to evade security measures [10].

Additionally, it is valuable in software development to analyze similarities between different versions or builds of software for code reuse or plagiarism detection [10,11]. Furthermore, its application in reverse engineering aids in developing patches and security updates by understanding and addressing vulnerabilities. Various techniques and algorithms are employed for binary code similarity detection, including hash functions that create fixed-size hash values for quick comparisons [12,13], sequence matching using algorithms such as longest common subsequence (LCS), control flow graph (CFG) matching [14], and function-level analysis focusing on comparing functions within binaries. Ongoing developments in the BCSD field include the implementation of machine learning (ML) techniques, which are used for more advanced and adaptive similarity analysis, the exploitation of deep learning techniques on neural networks to extract features and perform similarity comparisons, and the utilization of fuzzy matching methods.

A significant amount of research is actively being pursued, resulting in notable advancements and enhancements across this field. Despite these achievements, challenges continue to remain, such as the code obfuscation techniques used by malicious actors and the efficient handling of large datasets for quick and accurate comparisons. As the landscape of cyber threats evolves, continuous advancements in BCSD algorithms are essential to address emerging challenges and support both cybersecurity and software development processes. As deep learning (DL) continues to excel across various domains, such as recommendation systems, anomaly detection, and healthcare, more experts have innovated techniques to identify similarity via neural networks. The novelty of the approach are as follows:

- The approach creates embeddings for binary functions via graph convolutional networks (GCNs), which capture complex relationships, handle the variability present in binary code and adapt to different code structures.
- During model training, a comprehensive cross-platform dataset comprising 129,364 binary files is constructed by compiling the open-source Linux packages across various optimization levels. Compared with state-of-the-art techniques, such as Genius [15], Gemini [16], and DeepDualSD [17], the experimental results demonstrate the good performance of the proposed approach.
- To search for vulnerabilities, a dataset was created that included firmware images and vulnerable functions from the CVE database [18]. By employing similarity detection technique to compare firmware images and known vulnerabilities, proposed approach effectively identified vulnerable firmware images. The experimental results demonstrate that the proposed method has the best accuracy, 95%, in identifying similarities among code fragments, thus demonstrating its effectiveness in cross-platform BCSD.

The subsequent sections of the article are outlined as follows: A literature review of the existing BCSD methods is presented in Section II. Section III explains the proposed DL-based solution. Section IV offers a comparative analysis between the proposed method and other established BCSD techniques. The conclusions of this work are provided in Section V.

2. Literature Survey

In this section, existing methods such as graph matching, deep learning-based methods for BCSD are discussed. Over the past two decades, numerous methodologies have been introduced by researchers to analyze binary code similarity. These approaches have been extensively utilized to address critical issues, including bug identification, malware detection, patch analysis and malware analysis. However, challenges persist within this domain, such as code obfuscation, variability in programming styles, compiler optimizations, and platform-specific characteristics, which can introduce noise and variability in binary code, dynamic behaviors exhibited by software at runtime and cross-platform compatibility. Because of persistent challenges, conducting a comprehensive literature survey in this field is imperative.

Genius [15] proposed a codebook generation approach to solve the scalability challenge present in the bug search approach across different platforms. Xu et al. [16] proposed method, which uses neural networks (NNs) to compute embeddings for each binary and is designed to address the cross-platform BCSD problem. The model outperforms the control flow graph approaches in terms of accuracy and efficiency, demonstrating superior performance in similarity detection tasks. The model leverages a Siamese network for training the embedding network, allowing for the generation of embeddings that capture persistent features of binary functions across different compilers and architectures. The model's adaptability is demonstrated through its ability to be quickly retrained for specific tasks and achieve high accuracy in identifying vulnerable firmware images. Furthermore, the visualization of the embeddings demonstrates the model's effectiveness in preserving information about the source functions across different target architectures, compilers, and optimization levels. The work presented in [17] outlines a dual-attribute embedding approach that combines the semantic and structural attributes of binary code to improve BCSD performance.

Jinxue PENG et al. [19] introduced fast cross-platform BCSD to address the issues of high convolutional computation and structural information loss faced in techniques designed using CFGs and GCNs. The approach employs natural language processing (NLP) for basic block embedding, and for function representation, inductive GNN GraphSAGE (GAE) is used.

The CrossDeep [20] approach discusses cross version binary similarity model-based CNN and encoding methods. The proposed work combines raw byte analysis with CNN to enhance accuracy of similarity detection. The SimInspector [21] method proposes advanced techniques such as the neural machine translation (NMT) model and the graph embedding model. These models extract semantics of binary code, leading to improved accuracy in similarity detection. This can be achieved by training an NMT model [22] and a graph embedding model to automatically extract binary code semantics, which are represented as numerical vectors. This work evaluates the effects of different hyperparameters on model accuracy, providing insights into the optimal settings for achieving the highest performance.

A novel solution for binary code similarity comparison was discussed in [23], which addresses the challenges posed by compilation diversity and code obfuscation. A multiview approach that combines structural, semantic, and syntactic features of functions is extracted for embedding. The technique achieves higher recall than the baseline in vulnerability function search does, highlighting its effectiveness in real-world vulnerability detection. Yuede Ji et al. [24] explained how high accuracy and code coverage challenges are addressed. The efficiency is achieved through two different steps, namely, binary source canonicalization and code similarity calculation. The system uses an Attributed Control Flow Graph (ACFG) to collect the raw features from binary code and employs a GNN model to compute representative embeddings for each attributed graph. Additionally, the work also includes a graph triplet-loss network (GTN) for ranking the similarity of binary functions and supervising the learning process.

A deep learning (DL) approach [25] aims to address limitations such as dependency on engineered syntactic features for similarity comparison. The proposed technique, i.e., BinDeep, uses a recurrent neural network (RNN) DL algorithm to recognize the types of target functions. To measure the similarity between two entities, a Siamese neural network is employed, leveraging a fusion of convolutional neural networks (CNNs) and long short-term memory (LSTM). The model uses instruction embedding with vectorizing the extracted instructions. Yang, S. et al. [26] proposed the semantic resemblance of functions across different architectures via the abstract syntax tree (AST) encoding method. The Tree-LSTM network is used to capture function semantic representation. ASTERIA successfully detects vulnerable functions in Internet of Things (IoT) firmware images, emphasizing its practical application in vulnerability search. The method uses the Siamese architecture to combine two similar Tree-LSTM models for AST similarity detection.

The binary search engine described in [27] matches semantically equivalent and similar functions across different architectures, operating systems, and compilers. A selective inlining approach will learn the whole function semantics, and three filtering processes are narrowed down to match the candidate target functions. The selective inlining technique greatly enhances the overall matching accuracy, particularly for ARM binaries. Problems such as dynamic analysis difficulties for firmware, handling a single architecture and using semantic similarity for analyzing large code bases are addressed in discovRE [28]. The technique discovRE identified POODLE and Heartbleed vulnerabilities of Android images in approximately 80 ms.

Existing solutions primarily rely on individual features, which limits their ability to capture the complex relationships within program behavior, resulting in lower accuracy. In contrast, the Attributed Control Flow Graph (ACFG) integrates both structural and syntactic features information, providing a more comprehensive representation. The choice of the Attributed Control Flow Graph (ACFG) feature in GCN, helps to improve accuracy in similarity detection. For binary structural features, GCNs are an optimal choice due to their ability to effectively capture and propagate structural dependencies within graph-based data. The choice of the specific GCN model was based on its effectiveness in capturing structural relationships within graph-structured data. GCNs have been widely used for node classification and graph-based learning tasks due to their ability to aggregate information from neighboring nodes. Hence, GraphConvDeep, a Graph CNN, is proposed.

3. Proposed Approach

This section outlines the main ideas of the proposed approach. The overview of proposed model is provided in subsection 3.1. The feature extraction from the binaries is explained in subsection 3.2. Subsection 3.3 presents the computation of ACFG embedding. Finally, the use of the Siamese architecture in the proposed model is explained in subsection 3.4.

3.1. Solution Overview

Figure 1 shows an overview of the proposed technique GraphConvDeep. The block level and interblock parameters are obtained from the source and target binaries. This preprocessing stage, which extracts features, is called feature extraction. The extracted features are converted into ACFG. The ACFG encompasses block-level attributes, interblock attributes for every node, the total node count in the graph, source filename, and edge details within the graph. The ACFG is an input to the GraphConvDeep Embedding Model, which generates graph embedding, i.e., a high-dimensional vector based on the CFG of each binary function. The model computes the vector by passing through four steps. The steps include node degree computation, neighborhood aggregation, linear transformation of node features and a message

passing mechanism. In the first step, the degrees of every node in a graph are computed by adding self-loops to the graph, providing valuable information about the node's connectivity within the graph structure. The primary reason for adding self-loops in GCNs is to ensure that each node's own feature representation is also considered during the aggregation step. In the second step, every node representation is updated on the basis of information from neighboring nodes in a graph. Neighborhood aggregation allows nodes to gather and integrate information from their local environment, resulting in refined and context-aware representations that are crucial for various tasks, such as linear transformation and message passing mechanisms. The input feature matrix undergoes a linear transformation during the third step, which is the forward pass of a GCN layer. A linear transformation stage aims to project the feature vectors into a new space, potentially of higher dimensionality. The transformation enables the model to acquire more complex representations of the input data. The final step is the message passing mechanism in GCNs, where nodes exchange and gather information to revise their representations by utilizing the graph structure and input features. At the GCN output later, the final graph embeddings are computed based on the aggregated and transformed feature vectors obtained from the previous stages. The final graph embedding from the output layer is passed to the Siamese architecture for comparison. The Siamese architecture [29] takes two embeddings of different binaries and generates a similarity score via a similarity measure. Specifically, the proposed model mainly consists of three stages: feature extraction, ACFG embedding and similarity computing, which will be discussed in the following sections.

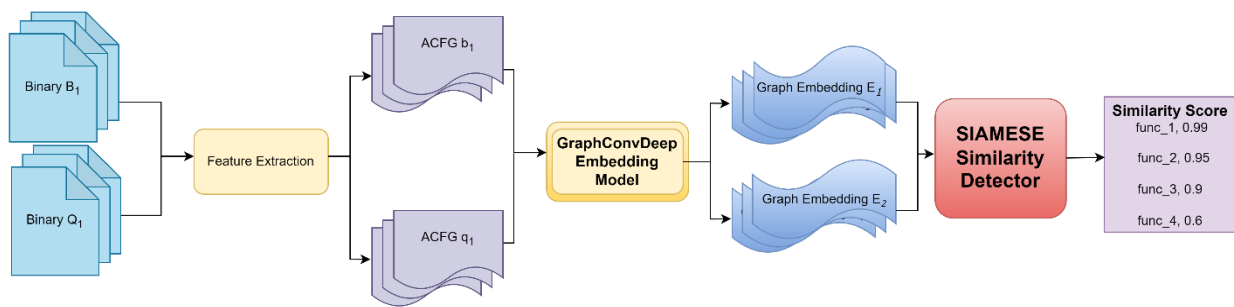


Fig.1. Overview of the proposed approach, GraphConvDeep

3.2. Feature Extraction

In binary code similarity detection, both block-level parameters and interblock parameters [15] are essential for capturing the structural and behavioral characteristics of the binary code. Table 1 lists the basic block parameters, which include both block level and interblock parameters. The block level and interblock parameters are as follows:

Block-level parameters: These attributes refer to characteristics or properties associated with a block of code. Curly braces {} in languages such as C, C++, or Java typically contain a group of statements for creating a block of code.

- String literals: String literals are fixed values of characters (text) that do not change during program execution. For example, "Hello, World!" is a string constant.
- Numeric literals: Numeric literals are fixed values of numbers that do not change during program execution. For example, 42 is a numeric constant.
- No. of Transfer Instructions: Transfer instructions refer to statements that control the flow of the program, such as conditionals (if, if-else, switch) and loops (while do-while for). The transfer instructions count can provide insights into the program's control flow complexity.
- No. of Calls: This attribute typically refers to the number of functions or method calls within a block of code. It indicates how often other pieces of code are invoked.
- No. of Instructions: This attribute represents the total count of instructions within a block. It provides a measure of the size or complexity of the code.
- No. of Arithmetic Instructions: Arithmetic instructions refer to basic mathematical operations such as summation, subtraction, product calculation, and division. The number of arithmetic instructions helps to evaluate the computational complexity of a block of code.

Interblock attributes: These attributes involve relationships or characteristics between different blocks of code.

- No. of Offspring: In the context of code analysis, the offspring count usually refers to the number of blocks that are directly executed as an output of the current block. It provides insight into how control flow is distributed.

Figure 2 depicts the partial control flow graph and ACFG of the Heartbleed vulnerability function present in the OpenSSL Linux package. The seven features extracted are shown in the attributed control flow graph of vulnerability function.

Table 1. Basic block parameters

Block-level parameters	Interblock parameters
String literals	Offspring Count
Numeric literals	
Transfer Instructions Count	
Calls Count	
Instructions Count	
Arithmetic operations Count	

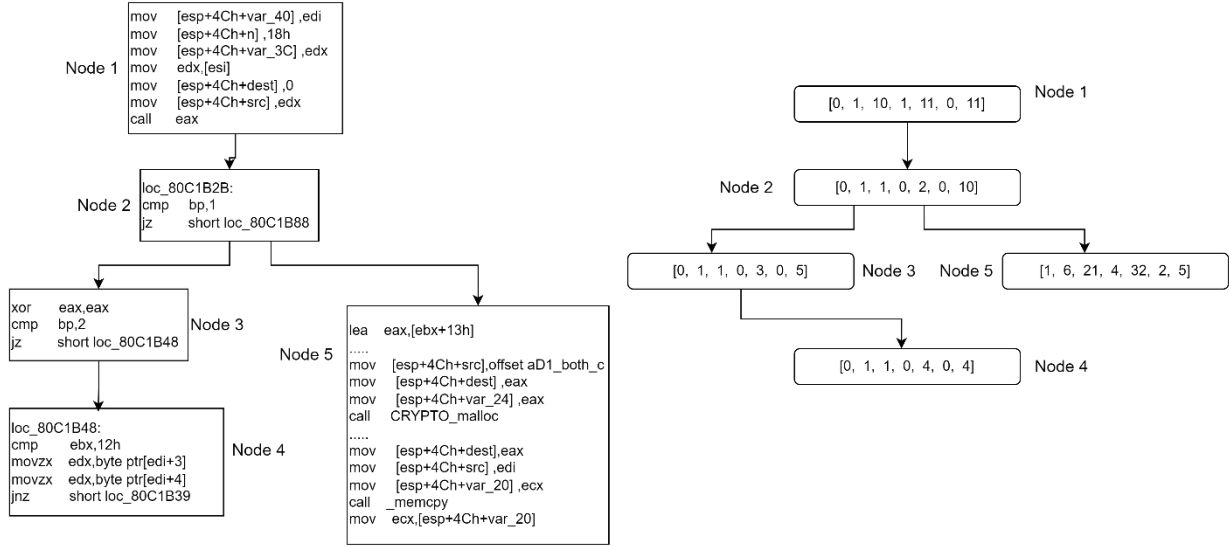


Fig.2. Partial CFG of the dtls1_process_heartbeat (heartbleed vulnerability) function and its ACFG

3.3. ACFG Embedding

The features extracted are used to compute embedding of each node in a graph and then one final graph embedding. The graph embedding generation process aims to generate an embedding vector for the input graph. It starts by processing each vertex in the graph, calculating node degrees and normalization factors, applying a linear transformation to the associated features, message propagation and final embedding computation. Algorithm 1 represents the detailed process of embedding computation. The input is an Attributed Control Flow Graph (ACFG) represented as $G = (V, E, f)$ where V is the set of vertices (nodes), E is the set of edges (connections between nodes) and f_v is the feature vector of each node v . The algorithm iterates over L convolution layers. For each node v in the graph, the following computations occur:

Algorithm 1: Graph Embedding Generation

Input: ACFG $G = (V, E, f)$
Set of Vertices: V , Set of Edges: E ,
Set of basic-block features of each vertex in the graph: f

Hyperparameters: Learnable Weight Matrix: W

Output: Final Graph Embedding: $\phi(G)$

- for** $t=1$ **to** No. of Convolution Layers (L)
- for each** node $v \in V$ **do**
- $d_i = \sum_{j \in N(i)} A_{ij}$ (Compute Node Degree)
- $\hat{A}_{ij} = \frac{A_{ij}}{\sqrt{d_i d_j}}$ (Compute Normalization Factor using node degrees)
- $H^{(l+1)} = (\hat{A} \cdot H^{(l)} \cdot W^{(l)})$ (Linear Transformation of Node Features f)
- $m_v = \sum_{u \in N(v)} \hat{A}_{uv} \cdot \hat{A}_v \cdot H^{(l+1)}$ (Message Passing)
- $H' = \sigma(m_v)$ (Apply Activation Function $RELU(\sigma)$)
- end for**
- end for**
- $\phi(G) = \text{Aggregate}(H')$ (Aggregation of all the node embeddings)
- return** $\phi(G)$

A. Node Degree Calculation

For each node v_i in the graph, calculate its degree d_i :

$$d_i = \sum_{j \in \mathcal{N}(i)} A_{ij} \quad (1)$$

where A_{ij} is the adjacency matrix element indicating the connection between nodes i and j , $\mathcal{N}(i)$ denotes the set of neighboring nodes of v_i . This stage helps in understanding the structural role of each node, distinguishing between highly connected nodes (hubs) and isolated nodes.

B. Normalization Factors

Calculate the normalization factor for each node v_i using the degrees of the nodes

$$\hat{A}_{ij} = \frac{A_{ij}}{\sqrt{d_i d_j}} \quad (2)$$

where $\hat{A}_{ij}$ is the normalized adjacency matrix. Normalization factors prevent overemphasis on high-degree nodes, reducing bias in message aggregation, improves numerical stability and convergence during training.

C. Linear Transformation

Apply a linear transformation to the features of each node. Let $X \in \mathbb{R}^{N \times F}$ represent the input feature matrix, where N is the no. of nodes and F is the no. of features per node. The linear transformation is defined as:

$$H^{(l+1)} = (\hat{A} \cdot H^{(l)} \cdot W^{(l)}) \quad (3)$$

where $H^{(l+1)}$ is the updated feature for node v_i at the next layer $l + 1$. $H^{(l)}$ is the hidden feature matrix at layer l , with $H^{(0)} = X$, $W^{(l)} \in \mathbb{R}^{F \times F'}$ is the trainable weight matrix at layer l , $\sigma(\cdot)$ is a non-linear activation function and $\hat{A}$ is the normalized adjacency matrix from the previous step.

D. Message Propagation

The process of message passing [30] and feature updating in a GCN begins after a linear transformation applied to each node's features. This transformation prepares the features for aggregation. With each layer, nodes aggregate information from their immediate neighbors, and by stacking layers, nodes can capture information from increasingly larger neighborhoods in the graph. The key idea is that every node collects information from its neighbors, and the node representations are updated through a neural network layer.

$$m_v = \sum_{u \in \mathcal{N}(v)} \hat{A}_u \cdot \hat{A}_v \cdot \mathcal{H}^{(l+1)} \quad (4)$$

At each iteration of message passing, a hidden embedding is updated for every node based on information collected from its graph neighborhood. Figure 3 depicts the message passing mechanism, where each node in the GCN calculates its embedding by gathering information from its neighbors.

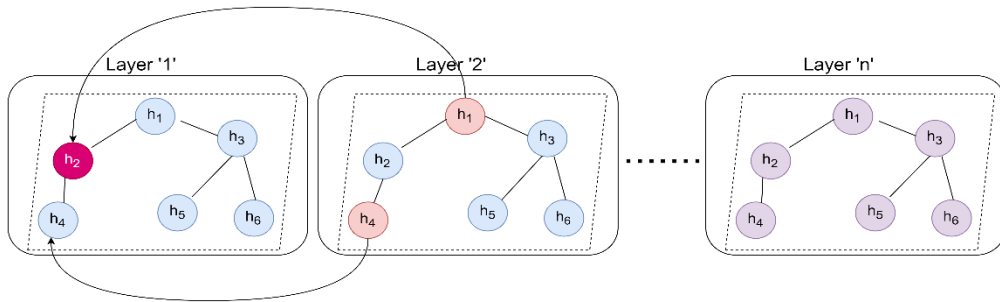


Fig.3. Neural message passing mechanism in the GCN

E. Embedding Computation

The rectified linear unit (ReLU) activation function is performed elementwise at the output of the message passing operation. This stage introduces nonlinearity into the network and helps in dimensionality reduction, making the computation more efficient.

$$H' = \sigma(m_v) \quad (5)$$

The final embedding vector $\phi(G)$ for the entire graph can be obtained by aggregating the node embeddings,

$$\phi(G) = \text{Aggregate}(H') \quad (6)$$

where H' is the output feature matrix from the final layer L , and the average is the aggregation function. Figure 4 illustrates the graph embedding computation in GCN. The process begins with the initial node features, followed by message passing where each node aggregates information from its neighbors. The process is repeated for the ' L ' number of convolution layers. Finally, an aggregation function processes the final layer's output, resulting in an updated embedding vector of the input graph. The updated embedding vector can be useful for finding matching functions.

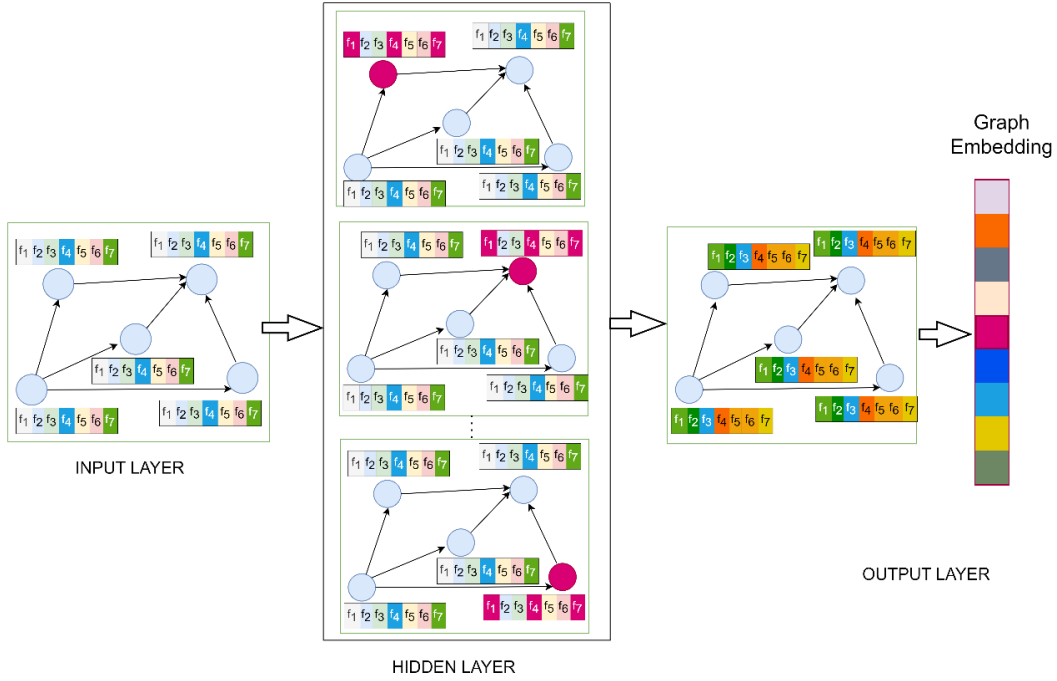


Fig.4. Graph convolutional network (GCN) architecture utilized for generating graph embedding

3.4. Siamese Similarity Detection

The Siamese neural network is a deep learning architecture designed for similarity learning, where two identical subnetworks process different inputs while sharing the same parameters. Each subnetwork extracts feature representations from its respective input, ensuring that the learned features remain consistent across both networks. Siamese architecture [29] refers to a design pattern or architectural style in which two identical or similar structures are conjoined or connected in some way. The Siamese architecture in BCSD refers to the use of neural network architectures inspired by Siamese networks to detect similarities or dissimilarities between binary code samples. Siamese networks are neural network architectures characterized by two identical subnetworks known as twins, which share identical parameters and are trained simultaneously on pairs of input data. This architecture is particularly effective in binary similarity detection, where structural patterns in binaries can be analyzed and compared to identify relationships, making it useful for tasks like malware detection and software lineage analysis. The Siamese architecture is the best fit for binary code similarity detection because it effectively learns and compares structural and semantic patterns between binary representations. Unlike traditional classification models that assign discrete labels, the Siamese network focuses on measuring the similarity between two inputs, making it ideal for comparing binary files without requiring an extensive labeled dataset.

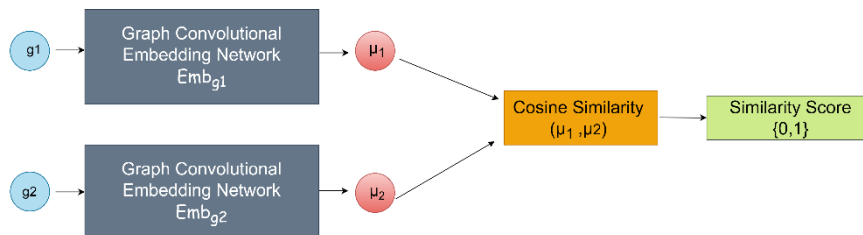


Fig.5. Siamese architecture

In the GraphConvDeep model, the Siamese layer takes two embeddings from different binaries and computes the similarity score based on the similarity metrics, cosine similarity and jaccard similarity coefficient metric. The Siamese architecture employed in the proposed method using cosine similarity is depicted in Figure 5. The cosine similarity among two binary code samples is calculated by the dot product of their normalized feature vectors and dividing the result by the product of their Euclidean norms. Mathematically, it can be expressed in Eq. 7.

$$\text{Cosine Similarity (A,B)} = \frac{A \cdot B}{|A| \cdot |B|} \quad (7)$$

where:

- A and B are the normalized binary feature vectors representing two binary code samples.
- $\cdot$ represents the dot product operation.
- |A| and |B| denote the Euclidean norms of vectors A and B, respectively.

The Jaccard similarity coefficient is a measure of similarity between two vectors. It is defined as the size of the intersection divided by the size of the union of the two vectors. Mathematically, it can be expressed as:

$$\text{Jaccard Coefficient(A,B)} = \frac{|A \cap B|}{|A \cup B|} \quad (8)$$

where:

- $|A \cap B|$ is the no. of elements in the intersection of vectors A and B,
- $|A \cup B|$ is the no. of elements in the union of vectors A and B

4. Evaluation

This section focuses on the analysis of the experimental results obtained by the proposed model and the comparative analysis with the cutting-edge methods.

4.1. Experimental Setups

Using the PyTorch backend library and the Keras [31] platform, the proposed method was developed in Python. The experiments were conducted on an HP Z4 G5 Workstation - 1x Intel Xeon Octa-core w3-2435 (8 Core) server running the Windows 10 operating system with a processor speed of 3.10 GHz, 512 GB solid-state drive (SSD) secondary memory, 16 GB SDRAM DDR5 internal memory and NVIDIA T1000 4 GB graphics. Binaries are disassembled using the Binary Ninja open source tool [32]. The plugin in the IDA Pro tool is utilized to extract the ACFG from binary files [33].

4.2. Dataset

The proposed model is evaluated on 3 datasets. Dataset 1 is derived from OpenSSL across multiple compiler optimizations and CPU architectures, ensuring a balanced set of function pairs for supervised learning in binary similarity analysis. Dataset 2 consists of function graphs extracted from diverse firmware images of multiple vendors, categorized by size to evaluate efficiency and reduce bias in analysis. Dataset 3 includes vulnerable functions from real-world security exploits, compiled under different optimization settings to study patterns across software from various vendors.

Dataset 1: The dataset was prepared by selecting a popular Linux package, OpenSSL (v3.0.0 to v3.3) [34]. OpenSSL is a widely used open-source cryptographic library. OpenSSL is used in binary similarity analysis because many binaries include its cryptographic functions, making it useful for detecting code reuse, software lineage, and security vulnerabilities. OpenSSL functions have well-known signatures, making it easier to compare binaries based on function-level similarity. Following the acquisition of the package source code, each program is compiled using GCC and Clang compiler with four optimization stages (O0, O1, O2, O3) and three CPU architectures (ARM, x86 and MIPS). The matched and unmatched function pair counts used in training and testing are reasonably balanced to aid in supervised learning. Functions are randomly selected from several binaries with distinct function names to obtain the negative training examples. Positive and negative samples are extracted and stored in different files with similar and dissimilar labels, respectively. For the experimental observations, 129364 samples were considered. This selection ensures a fair comparison, as other baseline approaches have also utilized same Linux package in their evaluations. The ACFG counts across the different platforms used for training and testing are displayed in Table 2. The sample consists of 38,794 X86, 38,610 ARM and 51,960 MIPS (Microprocessor without Interlocked Pipelined Stages) binaries. To assess the method's binary performance, three distinct subsets of the entire dataset are used for testing and training. For ablation studies, Dataset 1, which comprises 129364 functions, is further subdivided into three subsets of across different compilers, optimization levels and architectures. The complete dataset is partitioned into two subsets, with 30% allocated for testing and 70% allocated for training.

Table 2. ACFG Counts in the Dataset 1

	X86	ARM	MIPS	TOTAL
Training	27155	27027	36372	90554
Testing	11639	11583	15588	38810
TOTAL	38794	38610	51960	129364

Dataset 2: A dataset containing ACFGs of varying sizes, based on the vertex count in each graph, is constructed to assess efficiency. Specifically, ten firmware images are chosen at random from Dataset II used in [15]. With sizes varying from 1 to 1,306 vertices, 54,200 ACFGs are derived from these 10 firmware images. The firmware images are from Dlink, TPlink, Netgear, Cisco, vendors. This diversification helps mitigate bias and ensures evaluation across a broader range of binaries. To ensure that every ACFG in a set is the same size, these ACFGs are arranged into sets according to size. Eighteen ACFGs are chosen at random from each set that has more than 18, with the remaining ACFGs being eliminated. For the dataset II, 2,785 ACFGs are ultimately obtained.

Dataset 3: The dataset includes 121 vulnerable functions of four different vendors obtained from the CVE database [18]. The functions are compiled using GCC and Clang compiler with four optimization stages (O0, O1, O2, O3). The dataset is designed to analyze vulnerable function patterns from real-world exploits. Binaries are extracted from documented security vulnerabilities in widely used software. The functions were selected based on severity, exploitability, and frequency of occurrence, which could introduce biases. Even though Dataset 1 was prepared from OpenSSL package, Dataset 2 from specific vendors, and Dataset 3 from a vulnerability database, biases exist in terms of dataset size and vendor representation. However, each dataset was designed separately to support different types of analysis, ensuring targeted evaluations rather than direct comparisons.

4.3. Evaluation Metrics

The performance metrics like precision rate (PR), accuracy (Acc), F1 score (F1), average running time (ART) and recall rate (RR) are used to analyze the performance of the proposed model. Metrics are well-suited for binary similarity detection because they effectively quantify the model's ability to distinguish between similar and dissimilar binary files. These metrics are defined as follows:

- Acc measures the overall correctness of the model but can be misleading in imbalanced datasets. Since binary similarity detection may involve imbalanced pairs (e.g., more dissimilar pairs than similar ones), accuracy alone is insufficient.
- RR is critical in this context as it indicates the proportion of actual similar binaries that are correctly identified. A high recall ensures that most similar binaries are detected, which is essential for malware analysis, vulnerability detection, and software lineage tracking.
- PR evaluates how many of the detected similar binaries are truly similar. A high precision prevents false positives, ensuring that detected similar binaries are genuinely related.
- F1 balances precision and recall, making it a more reliable metric in scenarios where both false positives and false negatives are costly. Since binary similarity detection involves trade-offs between missing similar binaries and wrongly identifying unrelated binaries as similar, the F1-score provides a comprehensive evaluation.
- The total running time (TRT) and the ART are employed to evaluate efficiency. The TRT is the sum of running time per round, and the ART is defined in Eq. 13, where γ is the training round number.

$$\text{Acc} = \frac{TP+TN}{TP+TN+FP+FN} \quad (9)$$

$$\text{PR} = \frac{TP}{TP+FP} \quad (10)$$

$$\text{RR} = \frac{TP}{TP+FN} \quad (11)$$

$$\text{F1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (12)$$

$$\text{ART} = \frac{\text{TRT}}{\gamma} \quad (13)$$

These metrics are essential for assessing the performance of classification techniques and provide insights into different aspects of their effectiveness, such as overall accuracy, ability to correctly classify positive instances, and ability to identify all relevant instances. Table 3 summarizes the PR, Acc, F1 and RR values achieved by GraphConvDeep on the testing dataset. The metric values are tabulated over different numbers of epochs.

4.4. Performance Analysis

Dataset 1 is considered for the experiment. 129,364 ACFGs are present in the dataset, 90,554 ACFG's are used to train the model and 38,810 ACFG's are used for testing. The values in Table 3 show that the best accuracy score is 0.9592, and the recall rate is 0.9448 after 100 epochs. As the no. of epochs increases, the best accuracy and recall rate are observed in the model. The F1_score is 0.9390, and the precision rate is 0.9334, which means that a higher F1 score and precision rate indicate that the model makes more accurate positive predictions while minimizing false positives, which generally signifies better performance in binary classification tasks. Figure 6 shows the different performance metric results of the proposed approach method.

Table 3. Performance metric results achieved by proposed approach on the testing dataset 1

Epoch	Acc	RR	PR	F1
10	0.9272	0.9109	0.9292	0.9199
20	0.9216	0.9263	0.9345	0.9303
30	0.9263	0.9295	0.9392	0.9343
40	0.9325	0.9234	0.9292	0.9262
50	0.9358	0.9316	0.9307	0.9311
60	0.9443	0.9263	0.9392	0.9327
70	0.9482	0.9309	0.9308	0.9308
80	0.9443	0.9463	0.9492	0.9477
90	0.9591	0.9339	0.9434	0.9386
100	0.9592	0.9448	0.9333	0.9390

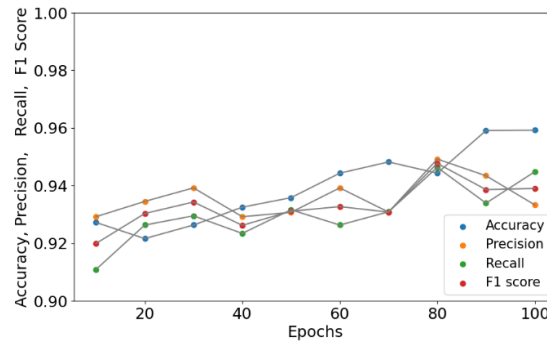


Fig.6. Performance metric results of the proposed method

Accuracy in Different Function size: The statistical analysis is conducted by taking function size (number of nodes in ACFG) present in Dataset 1 across a total number of target functions with similarity detection accuracy. In Table 4, the first column delineates the range of function sizes, and the second column enumerates the no. of functions falling within each size range. The function sizes span from 5-900, with the majority being less than 50. In the third column, the accuracy of similarity detection corresponding to functions of varying sizes is presented. The accuracy falls within the range of 0.9397-0.9592, indicating notably consistent and reasonably accurate detection across the spectrum of function sizes. Notably, there was no discernible trend suggesting diminishing accuracy with increasing function size.

Table 4. Function size distribution in the dataset and detection accuracy achieved in proposed approach

Function Size	Function #	Detection Accuracy
{5,50}	51446	0.9592
{50,100}	3249	0.9554
{100,200}	1078	0.9499
{200,300}	224	0.9588
{300,400}	94	0.9473
{400,500}	91	0.9397
{500,900}	81	0.9492

Evaluation of the hidden units in proposed approach: The no. of hidden units in neural networks directly influences how well they can recognize and interpret complex patterns in data. Increasing the no. of hidden units can enhance the network's capability to capture intricate relationships within the data, potentially leading to improved performance metrics such as accuracy. Selecting an appropriate number of hidden units is crucial for optimizing the balance between model

complexity and generalization ability, ultimately impacting the neural network's performance. Our experiments investigating the impact of varying the no. of hidden units in GNNs reveal significant effects on the results. Ranging from 1 to 25 different hidden units on the GraphConvDeep approach, the findings are illustrated in Figure 7(a). The number of hidden units in neural networks can also affect their running time. Generally, increasing the no. of hidden units typically leads to longer training and inference times because of the greater computational complexity involved in processing more parameters and performing more calculations. This increased computational demand arises from the need to propagate information through additional layers and units, resulting in longer processing times. Figure 7(b) indicates that the ART of the proposed method after 20 epochs remains stable. The experimental results indicate that there is a consistent trend as the hidden units increases. Acc, RR, PR and F1 generally improve, whereas ART remains relatively stable in the range of 29-36. After 20 epochs, the four performance metrics listed in Table 5 remain stable. Based on thorough experimental analysis, it is evident that selecting 20 as the number of hidden units represents an optimal choice.

Table 5. Performance metric results achieved by proposed approach for different hidden units

Hidden Units	Acc	PR	RR	F1	ART(s)
1	0.9240	0.9430	0.9310	0.93696	29
5	0.9359	0.9490	0.9390	0.94397	31
10	0.9351	0.9459	0.9490	0.94745	35
15	0.9321	0.9377	0.9469	0.94228	28
20	0.9591	0.9494	0.9331	0.94118	28
21	0.9592	0.9594	0.9441	0.95169	33
22	0.9593	0.9454	0.9433	0.94435	34
23	0.9584	0.9424	0.9453	0.94385	35
24	0.9583	0.9434	0.9633	0.95325	36

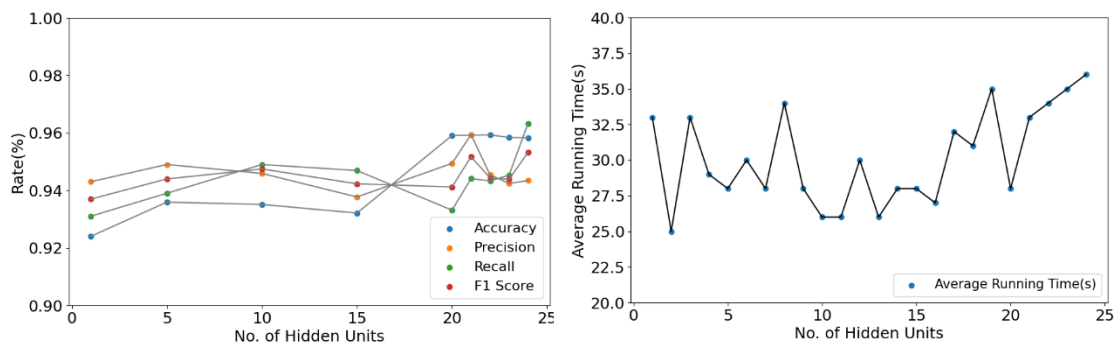


Fig.7. (a) Hidden unit vs performance metric (b) Hidden unit vs ART

4.5. Hyperparameters

This section evaluates the efficacy of hyperparameters in the proposed approach by analyzing the influence of training epochs and loss [35].

- **Number of Graph Convolutional Layers:** Graph Convolutional Networks likely involves multiple layers of graph convolutional operations. The depth of these layers can impact the model's capacity to capture complex relationships in the data. The number of layers created in the proposed approach is 21. Based on the observed experimental results, the best performance was achieved with 21 layers in the proposed approach.
- **Hidden Dimensionality:** The number of hidden layer representations in the graph convolutional layers. This parameter determines the dimensionality of the embedding space and can significantly affect model performance. The no. of hidden layers used in the proposed approach is 20.
- **Activation Functions:** The selection of activation functions used in the neural network layers includes ReLU, leaky ReLU, and sigmoid. The ReLU is used in the hidden layers of the GCN because it is simple and computationally efficient and avoids the vanishing gradient problem. Sigmoid functions are employed in the output layer of binary classification problems where the objective is to produce probabilities. The sigmoid function produces outputs in the range (0, 1), which can be interpreted as probabilities.
- **Training Epochs:** The model underwent training for 200 epochs, with evaluations conducted over the test set every 10 epochs to measure both the AUC and loss. The findings are depicted in Figure 8(a) and Figure 8(b). Figure 8(b) shows a gradual improvement in the AUC value, which stabilizes around epoch 35, whereas the loss value rapidly decreases and remains relatively stable after the same epoch. From the analysis shown in Figure 8(a) and Figure 8(b), it is evident that the model can be trained faster to obtain improved efficiency.

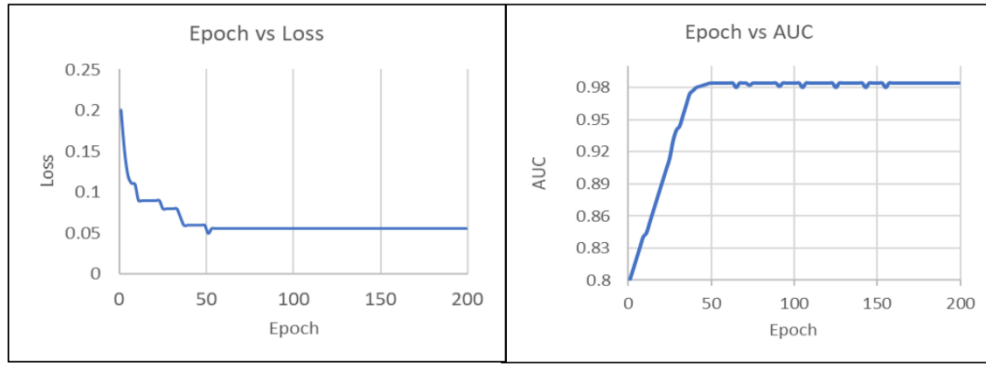


Fig.8. (a) Loss vs. epochs (b) AUC vs. epochs

4.6. Comparative Analysis

The binaries of X86, ARM, and MIPS platforms present in Dataset I are selected. The proposed method is evaluated on ACFG pairs from the dataset. [16] and [17] are used as the standard baseline methods for comparison, and their accuracies are assessed on the same testing dataset. The receiver operating characteristic (ROC) curves of the three methods are displayed in Figure 9. The proposed approach ROC is shown to be closer to the top and left borders, obtaining a substantial area under the curve (AUC) value (average 96% across platforms), indicating that the proposed technique is more accurate. Table 6 lists the different AUC values obtained via the three different approaches. In the ARM vs. x86 comparison, the result of proposed approach is 5% higher than that of DeepDual-SD [17] and 6.1% higher than that of Gemini [16]. The result of GraphConvDeep is 4% higher than that of Gemini in the case of MIPS vs ARM. DeepDual-SD and proposed approach are almost equal in the cases of MIPS vs ARM and x86 vs MIPS. It is evident from the results that proposed approach achieves better performance than the other two approaches do in the case of the x86 vs ARM comparison.

Table 6. Performance comparison of the proposed approach with other cross platform methods

Platforms	Gemini [16]	DeepDual-SD [17]	Proposed Approach
ARM vs x86	0.93	0.94	0.98
MIPS vs ARM	0.92	0.93	0.95
x86 vs MIPS	0.92	0.94	0.94
Average	0.93	0.94	0.96

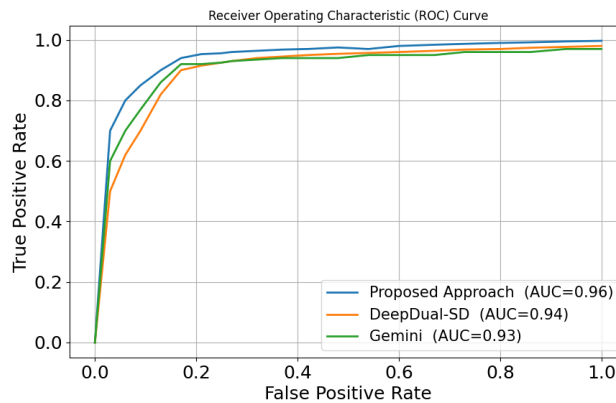


Fig.9. ROC curves of the proposed approach using different models

Efficiency of embedding models: The efficiency of LSTM+GAE [19], LSTM+GAT [19], Structure2Vec DNN [36] and proposed approach for embedding generation is evaluated using Dataset II. The deployment of an approach hinges significantly on its inference time. Therefore, the total runtime and test time of the four methods are recorded in Table 7. The dimension is identified by the embedding scheme used. The TRT is scaled according to the dataset size. The experimental results indicate a positive correlation between the input dimension and the inference time when identical GNN models are employed. The running and test time of LSTM+GAE [19] are 3227 and 75.5 s, respectively, when the embedding dimension size is 100. The Structure2Vec DNN method with a dimension size of 50 has a running time of 7373 and a test time of 345.6 seconds. The proposed method has a running time of 2495 s and a test time of 32 s with a dimension size of 75. The results suggests that the proposed model takes less time than the other three approaches do.

The first 25 epochs running times of the three different models are depicted in Figure 10. The time overhead is lower in the GraphConvDeep technique than in the other two approaches. The experimental results shows that baseline approaches are slow, while the proposed method demonstrates superior speed and flexibility.

Table 7. Time analysis comparison of the proposed approach with existing embedding models

Scheme	Dimension	Running time (s)	Test time (s)
LSTM+GAE[19]	100	3227	75.5
LSTM+GAT [19]	100	3886	56.8
Structure2Vec DNN[36]	50	7373	345.6
Proposed Approach	75	2495	32.1

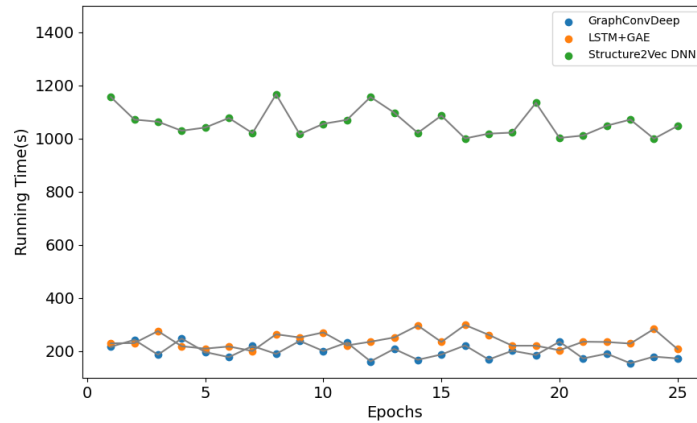


Fig.10. Running times of proposed approach and other embedding models for the first 25 epochs

Vulnerability Detection Analysis: The technique is evaluated on firmware images encountered in the real world. The two vulnerabilities were selected from the Dataset III. The two vulnerabilities used in the Genius [15] and Gemini [16] techniques are CVE-2015--1791 and CVE-2014--3508. In the case of CVE-2015--1791, Genius has 28% precision in detecting vulnerable firmware images, which are exclusively from D-Link and Belkin. The Gemini technique identified 42 true positives with 85% precision from four different vendors, such as Tomato by Shibby, DD-wrt, ZyXEL and D-Link, which were included in the top 50 results. The proposed method is able to detect 45 true positives from all vendors considered for evaluation. This was possible because of the task-specific retraining approach using a GCN. In the case of CVE-2015-3508 vulnerability, the control flow graph does not change much, so it is difficult to find. Among the top 50 results, Genius identified 24 firmware images as vulnerable, achieving a precision rate of 48%. Gemini detected vulnerabilities in 41 out of the top 50 firmware images, achieving a precision rate of 82%. The proposed technique was successful in detecting 46 firmware images from the top 50 results with a precision of 95%. The precision rates and numbers of vulnerable firmware images detected in the top 50 results of Genius [15], Gemini [16] and the proposed techniques are shown in Table 8. By employing the proposed technique, organizations can effectively manage and mitigate security risks associated with CVE-listed vulnerabilities.

In IoT security, where firmware is often closed source and reverse engineering is complex, proposed similarity detection model significantly enhances vulnerability detection efficiency. Proposed model allows vulnerabilities found in one architecture (e.g., ARM) to be efficiently mapped to others (e.g., MIPS, x86), reducing the need for redundant manual analysis. Cross-platform similarity detection solution enhances vulnerability detection in IoT firmware by enabling seamless identification of security flaws across different architectures, compilers, and optimizations. It helps in proactive risk assessment, cross-platform security consistency, and automated identification of both known and structurally similar vulnerabilities.

Table 8. Comparison of precision rate and vulnerable firmware identifications of the proposed approach with other cross-platform methods

Technique	CVE Vulnerable Name	Precision	No. of vulnerable firmware images detected out of Top 50
Genius [15]	CVE-2015-1791	28%	14
	CVE-2014-3508	48%	24
Gemini [16]	CVE-2015-1791	84%	42
	CVE-2014-3508	82%	41
Proposed Method	CVE-2015-1791	94%	45
	CVE-2014-3508	95%	46

4.7. Ablation Study

Table 9. The Ablation models of proposed approach

	Features		
Model	Cross Compiler	Cross Optimization	Cross Architectures
Base	✓	✓	✓
A1	✓		✓
A2		✓	✓
A3	✓	✓	
A4			✓
A5		✓	
A6	✓		

Table 9 outlines the ablation models of the proposed approach, focusing on the inclusion or exclusion of three key features: cross compiler, cross Optimization, and cross architectures. The Base model incorporates all three features, representing the full version of the proposed approach. Models A1 through A6 systematically exclude one or more features to evaluate their impact on performance. A1 excludes cross Optimization, A2 excludes cross compiler, and A3 excludes cross architectures. The remaining models (A4, A5, and A6) include only one feature, cross architecture, cross optimization and cross compiler respectively, allowing for a deeper understanding of each component's individual contribution.

Table 10. Ablation study of proposed approach on dataset 1 and dataset 2

	Dataset 1				Dataset 2			
Model	Accuracy	Precision	Recall@1	F1 Score	Accuracy	Precision	Recall@1	F1 Score
Base	0.95	0.94	0.93	0.93	0.96	0.94	0.95	0.94
A1	0.93	0.93	0.92	0.92	0.91	0.92	0.93	0.92
A2	0.94	0.93	0.91	0.91	0.92	0.93	0.91	0.91
A3	0.91	0.89	0.91	0.89	0.83	0.84	0.86	0.84
A4	0.92	0.91	0.93	0.91	0.78	0.88	0.79	0.83
A5	0.89	0.86	0.89	0.87	0.89	0.86	0.89	0.87
A6	0.87	0.78	0.79	0.78	0.81	0.73	0.78	0.75

Table 10 presents the results of an ablation study, comparing the performance of various models on Dataset 1 and Dataset 2 across four metrics. The Base model, which includes all three features (Cross Compiler, Cross Optimization, and Cross Architectures), achieves the highest performance on both datasets, with Dataset 2 showing slightly better accuracy (0.96) and recall (0.95). In contrast, Model A6, which only includes Cross Compiler, consistently exhibits the lowest performance across all metrics for both datasets, highlighting the insufficiency of relying solely on a single feature. The high performance of the Base model suggests that the combination of all three features is crucial for optimal results, as they likely complement each other in enhancing the model's ability to generalize across different datasets. On the other hand, the poor performance of Model A6 reflects the limitations of excluding Cross Optimization and Cross Architectures, which are key to improving the model's adaptability and precision. Cross compilation ensures the model runs optimally on different hardware platforms without loss of precision but due to optimization techniques used, leading to inefficient execution and lower accuracy. A5 model may be highly optimized but restricted to a specific platform, making it difficult to deploy across different hardware architectures. Due to optimization techniques and the compilation of binaries for different hardware, only the cross-architecture feature suffers in accuracy and performance. This study highlights the importance of a comprehensive feature set, where the exclusion of critical components like Cross Architectures (as in A4) or Cross Optimization (as in A1) leads to notable drops in performance, particularly on Dataset 2. Dataset 1 is the most flexible for cross-compilation, optimization, and architecture analysis since it is open-source and can be compiled for different hardware platforms. This allows for broader adaptability and performance tuning across various architectures. However, in Dataset 2, only Cisco binaries are suitable for all analyses. The other binaries are designed for specific hardware and architectures, limiting their flexibility for in-depth analysis. This restriction results in lower performance due to the lack of optimizations and cross-platform adaptability. The varying results across the datasets underscore the need for a balanced inclusion of features to maintain robustness across different scenarios.

5. Conclusions

BCSD is essential for various purposes, such as plagiarism detection, software reuse and maintenance, vulnerability detection, and reverse engineering. BCSD plays a crucial role in enhancing code quality, security, and reliability across

various domains. Many techniques have been proposed to increase the detection accuracy of BCSD. However, conventional methods face several challenges because they manually construct features and slow graph matching algorithms. In this work, a novel cross-platform GCN-based solution, i.e., GraphConvDeep, which uses graph convolutional networks to generate graph embeddings for binaries, is proposed. The GraphConvDeep technique outperforms traditional techniques in terms of similarity detection accuracy. The utilization of the proposed model demonstrated a high degree of AUC, i.e., 0.96, in identifying similarities among binary code snippets. The proposed technique is successful in identifying vulnerable firmware images in real-world hardware, and firmware has proven that it has significant contributions to the field of cybersecurity.

References

- [1] Xu Y, Xu Z, Chen B, Song F, Liu Y, Liu T. "Patch based vulnerability matching for binary programs", In: ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, Inc, pp. 376–387, 2020. DOI:10.1145/3395363.3397361
- [2] Yang S, Xu Z, Xiao Y, Lang Z, Tang W, Liu Y, Shi Z, Li H, Sun L, "Towards Practical Binary Code Similarity Detection: Vulnerability Verification via Patch Semantic Analysis", ACM Transactions on Software Engineering and Methodology, Vol.32, No.6, pp.1-29, 2023. DOI:10.1145/3604608
- [3] Hu X, Chiuueh T, Shin KG, "Large-scale malware indexing using function-call graphs", In: Proceedings of the 16th ACM conference on Computer and communications security. ACM, pp. 611–620, 2009. DOI:10.1145/1653662.1653736
- [4] Venkatasubramanian M, Lashkari AH, Hakak S, "IoT Malware Analysis Using Federated Learning: A Comprehensive Survey", IEEE Access , Vol. 11, pp. 5004–5018, 2023. DOI:10.1109/ACCESS.2023.3235389
- [5] Venkatasubramanian M, "Federated Learning Assisted IoT Malware Detection Using Static Analysis", In: Proceedings of the 2022 12th International Conference on Communication and Network Security, pp. 191 – 198, 2019. DOI:10.1145/3586102.3586131
- [6] HaddadPajouh H, Dehghantanha A, Khayami R, Choo K-KR, "A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting", Future Generation Computer Systems, Vol. 85, pp. 88–96, 2018. DOI:10.1016/j.future.2018.03.007
- [7] Gao J, Yang X, Fu Y, Jiang Y, Sun J, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary", In: ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Association for Computing Machinery, pp. 896–899, 2018. DOI:10.1145/3238147.3240480
- [8] Luo Z, Wang P, Wang B, Tang Y, Xie W, Zhou X, Liu D, Lu K, "VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search", In: 30th Annual Network and Distributed System Security Symposium, NDSS 2023. The Internet Society, pp.1-18, 2023. DOI:10.14722/ndss.2023.24415
- [9] Cheng X, Wang H, Hua J, Xu G, Sui Y, "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network", ACM Transactions on Software Engineering and Methodology, Vol. 30, No. 3, pp.1-33, 2021. DOI:10.1145/3436877
- [10] Luo L, Ming J, Wu D, Liu P, Zhu S, "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection", IEEE Transactions on Software Engineering, Vol. 43, No. 12, pp. 1157–1177, 2017. DOI:10.1109/TSE.2017.2655046
- [11] 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. ACM, pp. 389–400, 2014. DOI:10.1145/2635868.2635900
- [12] Jin W, Chaki S, Cohen C, Gurfinkel A, Havrilla J, Hines C, Narasimhan P, "Binary Function Clustering Using Semantic Hashes", In: 2012 11th International Conference on Machine Learning and Applications, IEEE, pp. 386–391, 2012. DOI:10.1109/ICMLA.2012.70
- [13] Qiao Y, Yun X, Zhang Y, "Fast Reused Function Retrieval Method Based on Simhash and Inverted Index", In: 2016 IEEE Trustcom/BigDataSE/ISPA, IEEE, pp. 937–944, 2016. DOI:10.1109/TrustCom.2016.0159
- [14] Gao D, Reiter MK, Song D, "BinHunt: Automatically Finding Semantic Differences in Binary Programs", Information and Communications Security. ICICS 2008, Vol. 5308, pp. 238–255, 2008. DOI:10.1007/978-3-540-88625-9_16
- [15] 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. ACM, pp. 480–491, 2016. DOI:10.1145/2976749.2978370
- [16] 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, ACM, pp. 363–376, 2017. DOI:10.1145/3133956.3134018
- [17] Guo J, Zhao B, Liu H, Leng D, An Y, Shu G, "DeepDual-SD: Deep Dual Attribute-Aware Embedding for Binary Code Similarity Detection", International Journal of Computational Intelligence Systems, Vol. 16, No. 35, pp.1-11, 2023. DOI:10.1007/s44196-023-00206-9
- [18] "National Vulnerability Database, National Institute of Standards and Technology", Accessed: 13 Apr 2023. [Online]. Available: <https://nvd.nist.gov/>
- [19] Peng J, Wang Y, Xue J, Liu Z, "Fast Cross-Platform Binary Code Similarity Detection Framework Based on CFGs Taking Advantage of NLP and Inductive GNN", Chinese Journal of Electronics 33, Vol. 33, No. 1, pp.128–138, 2024. DOI:10.23919/cje.2022.00.228
- [20] N. M, J. Kumar and M. H. G, "CrossDeep: A Hybrid Approach For Cross Version Binary Code Similarity Detection," 2024 Fourth International Conference on Multimedia Processing, Communication & Information Technology (MPCIT), Shivamogga, India, 2024, pp. 242-247, 2024. DOI:10.1109/MPCIT62449.2024.10892660.
- [21] Zhu X, Jiang L, Chen Z, "Cross-platform binary code similarity detection based on NMT and graph embedding", Mathematical Biosciences and Engineering, Vol. 18, No. 4, pp. 4528–4551, 2021. DOI:10.3934/mbe.2021230

- [22] 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 2019 Network and Distributed System Security Symposium, Internet Society, pp.1-15, 2019. DOI:10.14722/ndss.2019.23492
- [23] Wang Y, Jia P, Huang C, Liu J, He P, “Hierarchical Attention Graph Embedding Networks for Binary Code Similarity against Compilation Diversity”, Security and Communication Networks 2021, pp.1–19, 2021. DOI:10.1155/2021/9954520
- [24] Ji Y, Cui L, Huang HH, BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. ACM, pp.702–715, 2021. DOI:10.1145/3433210.3437533
- [25] 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 , Vol. 168, pp.1-8, 2021. DOI:10.1016/j.eswa.2020.114348
- [26] 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: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 224–236, 2021. DOI:10.1109/DSN48987.2021.00036
- [27] Chandramohan M, Xue Y, Xu Z, Liu Y, Cho CY, Tan HBK, “BinGo: cross-architecture cross-OS binary search”, In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 678–689, 2016. DOI:10.1145/2950290.2950350
- [28] Eschweiler S, Yakdan K, Gerhards-Padilla E, “discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code”, In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society, pp.1-15, 2016. DOI:10.14722/ndss.2016.23185
- [29] Bromley J, Guyon I, Lecun Y, Sickinger E, Shah R, Bell A, Holmdel L, “Signature Verification using a “Siamese” Time Delay Neural Network”, NIPS'93: Proceedings of the 6th International Conference on Neural Information Processing Systems, pp. 737-744, 1994.
- [30] Li X, Cheng Y , “Understanding the message passing in graph neural networks via power iteration clustering”, Neural Networks , Vol. 140, pp.130–135, 2021. DOI:10.1016/j.neunet.2021.02.025
- [31] “F. Chollet Keras: The Python Deep Learning library”, Accessed: 15 Mar 2023. [Online]. Available: <https://ui.adsabs.harvard.edu/abs/2018ascl.soft06022C/abstract>
- [32] “Binary Ninja”. Accessed: 1 Jan 2023. [Online]. Available: <https://binary.ninja/>
- [33] “The IDA Pro Disassembler and Debugger”, Accessed: 18 Jan 2023. [Online]. Available: <https://hex-rays.com/ida-pro/>
- [34] “OpenSSL”, Accessed: 15 Aug 2023. [Online]. Available: <https://github.com/openssl/openssl>.
- [35] Ito N, Hashimoto M, Otsuka A, “Feature Extraction Methods for Binary Code Similarity Detection Using Neural Machine Translation Models”, IEEE Access 11, Vol. 11, pp. 102796–102805, 2023. DOI:10.1109/ACCESS.2023.3316215
- [36] Massarelli L, Di Luna GA, Petroni F, Querzoni L, Baldoni R, “Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis”, In: Workshop on Binary Analysis Research Society, pp.1-11, 2019. DOI:10.14722/bar.2019.23020

Authors’ Profiles



Nandish M is currently working as Assistant Professor in the department of Computer Science and Engineering, JNNCE Shivamogga and pursuing his Ph.D in Visvesvaraya Technological University, Belagavi in the field of Cyber Security. He received his M.Tech. in Computer Science and Engineering from Visvesvaraya Technological University, India in 2014. His research interests include Machine Learning, Deep Learning, Artificial Intelligence and Cybersecurity.



Dr. Jalesh Kumar is the Professor and Head of Computer Science and Engineering department at JNN College of Engineering, Shimoga. He received his Ph.D. in Computer Science and Engineering in 2018 from Visvesvaraya Technological University, Belagavi. His research interests include Computer Security, Computer Vision, and Cryptography.



Mohan H G is currently working as Assistant Professor in the department of Computer Science and Engineering, JNNCE Shivamogga and pursuing his Ph.D in Visvesvaraya Technological University, Belagavi in the field of Cyber Security. He received his M.Tech. in Computer Science and Engineering from Visvesvaraya Technological University, India in 2014. His research interests include Semantic Networks, Deep Learning and Cybersecurity.



Manjunath Sargur Krishnamurthy has completed his Bachelors of Engineering in Computer Science and Engineering from Visvesvaraya Technological University (VTU), Belagavi, Karnataka. Currently working as Vice President - Sr Lead Software Engineer in JP Morgan Chase & Co, Houston, USA, with 18+ years of experience and a strong passion for securing digital communications and protecting digital assets. He is a patent holder on System and Method for Selective Dynamic Encryption (US 11-120141). He has strong research interests in machine learning, quantum computing and information security.

How to cite this paper: Nandish M., Jalesh Kumar, Mohan H. G., Manjunath Sargur Krishnamurthy, "GraphConvDeep: A Deep Learning Approach for Enhancing Binary Code Similarity Detection using Graph Embeddings", International Journal of Computer Network and Information Security(IJCNIS), Vol.17, No.3, pp.72-88, 2025. DOI:10.5815/ijcnis.2025.03.05