Hierarchical Attention Graph Embedding Networks for Binary Code Similarity against Compilation Diversity

Yan Wang, Peng Jia, Cheng Huang, Jiayong Liu, and Peisong He

School of Cyber Science and Engineering, Sichuan University, Chengdu, China

Correspondence should be addressed to Peng Jia; pengjia@scu.edu.cn

Received 19 March 2021; Revised 13 July 2021; Accepted 29 July 2021; Published 10 August 2021

Copyright © 2021 Yan Wang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Binary code similarity comparison is the technique that determines if two functions are similar by only considering their compiled form, which has many applications, including clone detection, malware classification, and vulnerability discovery. However, it is challenging to design a robust code similarity comparison engine since different compilation settings that make logically similar assembly functions appear to be very different. Moreover, existing approaches suffer from high-performance overheads, lower robustness, or poor scalability. In this paper, a novel solution HBinSim is proposed by employing the multiview features of the function to address these challenges. It first extracts the syntactic and semantic features of each basic block by static analysis. HBinSim further analyzes the function and constructs a syntactic attribute control flow graph and a semantic attribute control flow graph for each function. Then, a hierarchical attention graph embedding network is designed for graph-structured data processing. The network model has a hierarchical structure that mirrors the hierarchical structure of the function. It has three levels of attention mechanisms applied at the instruction, basic block, and function level, enabling it to attend differentially to more and less critical content when constructing the function representation. We conduct extensive experiments to evaluate its effectiveness and efficiency. The results show that our tool outperforms the state-of-the-art binary code similarity comparison tools by a large margin against compilation diversity clone searching. A real-world vulnerabilities search case further demonstrates the usefulness of our system.

1. Introduction

Binary code similarity comparison is used for detecting whether two given binary functions are similar. It plays an essential role in many computer security applications, including code clone detection [1–4], malware classification [5–8], security patch analysis [9, 10], and vulnerability discovery [11–13]. However, with the increase of various options for compilation settings, the binary code similarity comparison faces some severe challenges. First, different compilers (e.g., GCC (https://gcc.gnu.org) and Clang (https://clang.llvm.org)) generate cross-compiler binaries. Second, different compiler optimization levels yield cross-optimization-level binaries. Third, the source code itself may have version updates that yield cross-version binaries. These binaries by nature are similar because they have the same root. Existing solutions could address these challenges (cross compiler, cross-optimization level, and cross version) to some extent but perform poorly in code obfuscation binaries.

Traditional binary code similarity comparison solutions heavily rely on a specific syntactic or structural feature of binary code, i.e., N-grams or N-perms [14], tree based [15, 16], and graph based [11, 17, 18]. However, these approaches that only considering the syntax of instructions are expensive and cannot handle all the syntax changes due to different compiler settings. Moreover, utilizing graph-isomorphism (GI) theory [19, 20] to compare control flow graphs (CFGs) of function is time consuming and lacks polynomial-time solutions.

Recent works have leveraged the advance of machine learning to compare binary code similarity [12, 13, 21–23]. These methods learn higher-level numeric feature representations from control flow graphs or assembly
instructions. Thus, they have higher accuracy and better scalability than the traditional methods [12, 22, 24, 25], but they still have some limitations.

First, insufficient function information is captured for binary code similarity comparison against compilation diversity, leading to a high false-positive rate. Existing work includes combining the syntactic and structural information of function [13], combining the semantic and structural information of function [26]. However, the multiview information of function represents the characteristics from different dimensions. As a result, existing solutions are somewhat resistant to some compilation differences but perform poorly in code obfuscation binaries. Moreover, most of the semantic features extracted in the learning-based methods consider the whole assembly instruction as one token [23, 26–28]. However, each instruction may involve different registers, immediate values, and memory addresses, causing severe OOV (Out-of-Vocabulary) problems. Some schemes separate the opcode and operands of the instruction as a single token [21, 22] to address the aforementioned problem. Asm2Vec [21] does not further normalize these tokens. The normalization rules in DEEPBINDIFF [22] use “ptr” to represent pointers, but there are many instructions based on memory addressing that cannot be normalized. For example, instruction “cmp dword [rbp − 0x10c], 0x10,” where operand “[rbp − 0x10c]” cannot be processed, so “[rbp − 0x10c]” will be treated as a token. There are still many similar tokens, such as “[rip + 0x10b2]” and “[rip + 0x10e2],” which may lead to an OOV problem if the register or offset value in the changes. Therefore, none of these methods can effectively solve the OOV problem.

Second, the intuition is that not all parts of a function are equally relevant for binary code similarity comparison. For example, the syntactic and semantic features of each function will have different weights for the similarity comparison in different scenarios, and each basic block has a different importance to the function. Moreover, each instruction has a different importance to the basic block. The current research work does not consider these characteristics but only processes each instruction or each basic block with the same importance [21–23, 28]. Although i2v_attention [26] considers the attention mechanism during instruction aggregation, it merely considers the different importance of different instructions without considering the different importance of each basic block.

1.1. Our Approach. In this paper, a solution HBinSim is proposed to address the aforementioned questions. In particular, HBinSim first learns function embeddings via the proposed hierarchical attention graph embedding network. Each learned embedding represents a specific function by carrying both the syntactic, semantic, and structural information of the function. Then, these embeddings are used to efficiently and accurately calculate the similarities among functions.

To achieve this goal, the syntactic and semantic information of each basic block is first extracted. The extracted syntactic information is mainly numerical, which can be directly converted into a vector. For processing semantic information, assembly instruction is separated into the opcode and operands as a single token. Then, these tokens are normalized based on different addressing modes. Thus, more characteristic tokens can be retained, and the word embedding model constructed can eliminate the OOV problem in this way. Finally, we refer to [29] and propose a hierarchical attention graph embedding network to highlight the importance of different instructions, basic blocks, and the syntactic and semantic information in function. This network constructs multilevel attention models based on graph embedding network and determines the relevant sections involves modeling the interactions of the instructions, not just their presence in isolation. Attention serves two benefits: not only does it often result in better performance but it also provides insight into which instructions, basic block, and syntactic and semantic features contribute to the similarity comparison, which can be of value in applications and analysis. Finally, a robust function embedding vector is generated for similarity comparison.

A proof-of-concept HBinSim is implemented and compared with existing state-of-the-art binary clone search approaches. Our experiments demonstrate that HBinSim soundly outperforms state-of-the-art techniques i2v_attention and Gemini for cross compiler, cross-optimization level, cross version, and code obfuscation clone searching. The results further demonstrate the efficacy of HBinSim on vulnerability search based on a publicly available vulnerability dataset and show that HBinSim can achieve zero false positives and 100% recalls.

1.2. Contributions. Our contributions are as follows:

1) A general framework is designed for binary code similarity comparison against compilation diversity, which relies on the multiview feature of function, including syntactic, semantic, and structural information from raw bytes, to generate robust function embedding.

2) A new neural architecture, named hierarchical attention graph embedding network, is proposed, which has a hierarchical structure that mirrors the hierarchical structure of the function. This network has three levels of attention mechanisms applied at the instruction, basic block, and function level, enabling it to attend differentially to more and less critical content when constructing the function representation.

3) An extensive evaluation shows that HBinSim (https://github.com/wtwofire/HBinSim) could outperform state-of-the-art binary similarity comparison tools for clone search under different compilation settings, including cross compiler, cross version, cross-optimization level, and code obfuscation. Also, we conduct a vulnerability search case study on a publicly available vulnerability dataset, where HBinSim achieves higher recall than baseline.
This paper is organized as follows. Section 2 discusses the literature. Section 3 systematically integrates graph embedding into a clone search process. Section 4 describes the methodology. Section 5 presents our experiment. Section 6 discusses the limitations and concludes the paper.

2. Related Work

Clone detection, or plagiarism detection, has long been a focus of software engineering and security research. The initial research was to detect code clones at the source code level [30–34]. However, since we do not always have access to source code and lacking symbolic information complicates the process of comparing code at the binary level, clone search techniques that work on binary code are essential. Existing binary code similarity comparison techniques can be roughly divided into two classes: traditional approaches and learning-based ones.

2.1. Traditional Approaches. Traditional approaches rely on various raw features directly extracted from binary for code similarity comparison. N-grams or N-perms [14] are two early approaches for the similarity code search. They adopt the binary sequence or mnemonic code matching without understanding the semantics of code [35], so they cannot tolerate the opcode reordering issue caused by different compilations. To further improve the accuracy, the tracelet-based approach [36] captures execution sequences as features for code similarity checking, which can address the issue of opcode changes. A related concept is used by David et al. in [37], where functions are divided into pieces of independent code, called strands. The matching between functions is based on how many statistically significant strands are similar. Intuitively, a strand is significant if it is not statistically common. Some rely on the tree-based approach [15, 16]. TEDEM [16] introduces tree edit distances to measure code similarity at the basic blocks level, which is costly for matching and does not handle all syntactical variation. Besides, the most widely used method is graph based [11, 17, 18, 24]. Zynamics BinDiff [17] and BinSlayer [18] adopt the graph isomorphism algorithm to quantify the similarity between control flow graphs for code search. Pewny et al. [11] proposed a solution where each vertex of a CFG is represented with an expression tree; similarity among vertices is computed using the edit distance between the corresponding expression trees. DiscovRE [24] utilizes the prefiltering to boost CFG-based matching process by eliminating unnecessary matching pairs. However, these methods are susceptible to CFG changes such as flattening. Others compare the semantics of binary code using symbolic execution and theorem prover, such as BinHunt [38] and iBinHunt [39], but they are computation expensive and thus not applicable for large codebases.

2.2. Learning-Based Approaches. Compared with traditional methods, some learning-based methods have been proposed recently. A binary function can have structural and textual properties. Being inspired by natural language processing (NLP) technology, different techniques have been proposed by Fend et al. [12] and others [13, 21, 22, 40–42] to model the structural and textual aspects of a binary function as an embedding vector or to compute the similarity using deep neural networks. Genius [12] forms attributed CFGs and calculates the similarity via graph embeddings generated through comparing with a set of representative graphs named codebook. Xu et al. [13] proposed a GNN-based model is called Gemini, which gets better results than the previous methods. Nevertheless, these methods use some manually selected statistical features to represent basic blocks, which may not contain enough semantic information. aDiff [25] uses a Siamese network [43] with CNN to generate function embeddings. It eliminates the need for manually crafted features. Roberto et al. [26] proposed various feature embedding generation schemes of the basic block based on the similar Siamese network to calculate similarity. Yu et al. [44] propose combining semantic, structural, and order features of functions to generate the embedding of functions for similarity comparison.

There are many works to consider assembly instructions as features. Being inspired by natural language processing (NLP) technology, and they consider each instruction as a word and then use word vector models such as one-hot, n-gram, word2vec, or BERT to transform assembly instructions into vector representations [13, 23, 26, 28]. However, assembly instruction is different from the normal text data in the sense that an assembly snippet has complex internal formats. Asm2Vec [21] splits an instruction into an opcode and up to two operands and adopts an unsupervised learning approach to generate token embeddings. Nevertheless, it does not use the normalization method for assembly instruction. DEEPBINDIFF [22] and Codee [45] also consider the division of instructions into opcodes and operands, and they adopt a normalization strategy for dealing with operands. BinDeep [46] applies a more fine-grained normalization strategy. It classifies the common operands into 8 different categories. In addition to dividing the opcode and operands of the instructions into words, Instruction2Vec [47], PalmTree [48], and BinDiff _MN_ [49] apply a more fine-grained strategy, and they subdivide the operands into smaller parts to alleviate the OOV problem.

In the choice of learning algorithm, deep learning algorithms can effectively extract feature values based on powerful nonlinear feature representation than traditional algorithms. Many works are based on RNN or CNN neural networks to train similarity comparison models [23, 25, 26, 28, 49]. The authors in [26, 28, 49] also introduce the attention mechanism to reflect the different importance of different instructions or basic blocks for similarity detection. Nevertheless, these algorithms do not consider the structure information of binary code. One of the most widely used techniques to analyze programs is by transforming them into graphs [50]. A graph is a mathematical structure used to represent the relationship and connection between objects termed nodes. Recently, graph neural networks have emerged to deal with these structural data. Therefore, these graph neural networks have also been applied to the binary code similarity comparison field. Gemini [13] and
i2v_attention [26] build an attribute control flow graph input to the graph embedding network to train. Bin2vec [51] constructs a GCN network [52] to train program graphs. FuncGNN [53] combines attention mechanism with GraphSAGE [54] to train CFG. Xiang et al. [55] propose a multilevel graph matching network (MGMN) framework for computing the graph similarity between any pair of graph-structured objects in an end-to-end fashion.

Since documents have a hierarchical structure (words form sentences and sentences form a document) [29], and graphs can be clustered into successively more compact graphs at different hierarchies for similarity computation [56]. This hierarchical perspective has also been introduced into the field of binary code analysis. Gibert et al. [57] proposes a hierarchical convolutional network for malware classification. It has two levels of convolutional blocks applied to the mnemonic level and the function level. Yan et al. [58] also considered the two-level hierarchical structure (elements form instruction and instruction form program) for binary software vulnerability detection. This hierarchical structure is introduced to consider better the spatial correlation of different levels of binary executables.

3. Approach Overview

The system architecture of HBinSim is shown in Figure 1. The system takes the binary code of two functions as input and outputs the graph embedding representations of these functions. Then, the cosine distance of two embedding vectors is calculated to determine whether two binary functions are similar.

The whole system comprises three major components: (1) feature extraction; (2) hierarchical attention graph embedding network; (3) similarity comparison. Feature extraction is responsible for generating three pieces of information: syntactic features, semantic features, and structural features. Once generated, the syntactic and semantic features are sent to the basic block embedding generation component to generate the basic block embeddings (semantic embedding and syntactic embedding) based on the NLP techniques. We construct a syntactic attribute control flow graph and a semantic attribute control flow graph for each function based on the generated two types of embeddings and structural features. Next, these two types of graphs are then used as input, respectively, to the graph embedding network to generate the embedding for each graph. The generated function semantic embedding and syntactic embedding are aggregated by the attention mechanism to generate the final function embedding. Finally, the cosine distance of two embedding vectors is calculated to determine whether two binary functions are similar.

4. Methodology

This section discusses how we extract features and transform them into high-level vector representations suitable for salable and accurate binary code clone search.

4.1. Feature Extraction. The main challenge in binary code similarity comparison is that the compilation process can produce different binary code representations for the same source code. So, the robust feature should capture the compilation and obfuscation transformations that it can handle. Two functions can be considered similar if their syntax, semantics, and structure are similar [59]. However, the syntactic feature is least robust, which is sensitive to simple changes in the binary code, e.g., register reallocation, instruction reordering, and replacing instructions with semantically equivalent ones. The structural feature is robust against multiple syntactical transformations but sensitive to transformations that change code structure, such as code inlining or code obfuscation. The semantic feature is robust against semantics-preserving transformations, despite changes to the code syntax and structure. However, it still leads to performance degradation in some compilation settings, e.g., different compile optimization levels and code obfuscation. Therefore, we consider extracting multiview features of function to resist compilation differences, combining the function’s syntactic, semantic, and structural features instead of a single feature.

4.1.1. Syntactic Feature. The syntactic feature mainly describes the syntactic attribute of code representation. These syntactic attributes can be expressed in numeric information or “metadata” within the function, e.g., the number of instructions. By referring to features used in previous works [12, 13, 24, 60] and performing a series of code clone experiments for different feature sets, we finally determine to use 8 types of features in Genius [12] as the syntactic feature of each basic block, which are shown in Table 1. As shown, the first six features are related to the instruction of the basic block. Offspring represents each node’s descendants in a graph, and betweenness represents the betweenness centrality of each node in a graph. To extract syntactic features, we first extract each basic block in the function, along with attributes in Table 1 for each basic block, and store them as the syntactic features associated with the basic block.

4.1.2. Semantic Feature. The intuition is that a function has semantics. The compiler translates the function’s source code to a target platform, where it can use several optimizations. However, the underlying semantics must remain the same. To extract these semantics in function, we treat the function as a document, each basic block as a sentence in the document, and each instruction as a word. Then, the semantic feature is extracted based on an unsupervised learning method used in natural language processing.

However, there is an OOV problem where terms that appear during testing do not appear during training in natural language processing. Assembly instruction is composed of opcode and a certain number of operands, and the range of operands is vast. Many OOV terms are generated when training with assembly instructions as the smallest
observed that different instructions, basic blocks, and syntactic and semantic features. Second, it is
contains syntactic and semantic features). Second, it is
similarities and the corresponding semantic features of each function will be used as a training corpus to train the NLP model, and the corresponding semantic features of each function will be obtained based on it.

4.2. Hierarchical Attention Graph-Embedding Network. A
hierarchical attention graph embedding network is designed to capture multilevel insights about the function structure. First, the function has a hierarchical structure (instructions form basic blocks, basic blocks form function, and function contains syntactic and semantic features). Second, it is observed that different instructions, basic blocks, and syntactic and semantic information of function in a function are
differentially informative. Moreover, the importance of instructions and basic blocks is highly context dependent, i.e., the same instruction or basic block may be differentially crucial in different functions. Also, the syntactic and semantic features of function play different roles in distinguishing functions in different scenarios. To include sensitivity to this fact, a hierarchical attention graph embedding network is proposed.

The overall architecture of the hierarchical attention graph embedding network is shown in Figure 2. It consists of several parts: an instruction-level attention layer, a graph neural network layer, a basic block-level attention layer, and a function-level attention layer. The purpose of attention mechanism used at the instruction level is to obtain the semantic embedding of basic block $bb_{emb}$ ($u$ is the instruction-level attention parameter), and the syntactic embedding of basic block $bb_{emb}$ itself is the basic-block level. After the Syntactic Attributed Control Flow Graph (SyACFG) and the Semantic Attributed Control Flow Graph (SeACFG) of function are obtained, they are input to the graph neural network layer, respectively. The output of the graph neural network is aggregated by the basic block-level attention mechanism to generate the function syntactic embedding $f_{sy}$, and the function semantic embedding $f_{se}$, respectively ($k_1$ and $k_2$ are the block-level attention parameters). Finally, the function embedding $f \in \mathbb{R}^{m\times l}$ ($m$: output embedding dimension) is generated through the function-level attention layer ($h$ is the function-level attention parameter). We describe the details of different components in the following sections.

4.2.1. Basic Block Syntactic Embedding. In the feature extraction, a total of 8 features are extracted to represent the syntactic features of the basic block. Each feature in the basic block will be counted, and then, they will be arranged in order as a numerical vector. The order is [No. of string constants, No. of numeric constants, No. of transfer instructions, No. of calls, No. of instructions, No. of arithmetic instructions, No. of offsprings, Betweenness]. These vectors represent the basic block syntactic embedding. As shown in Figure 3, when a basic block is given, the opcode and operands of each instruction are classified and counted, and the two structural features of the

| Table 1: Basic-block attributes. |
|----------------------------------|
| Type                           | Attribute name         |
| Block-level attributes          | No. of string constants |
|                                 | No. of numeric constants |
|                                 | No. of transfer instructions |
|                                 | No. of calls |
|                                 | No. of instructions |
| Interblock attributes           | No. of offsprings |
|                                 | Betweenness |

![Feature Extraction](image1)

![Hierarchical Attention Graph Embedding Network](image2)

![Similarity Comparison](image3)

![Figure 1: Overview of HBInSim.](image4)
Basic blocks in the CFG are also calculated. Then, the computed value is combined into a vector representation, which represents the basic block syntactic embedding $bb_{embsy} \in \mathbb{R}^{d \times 1}$ ($d$: syntactic embedding dimension).

The Syntactic Attributed Control Flow Graph (SyACFG) of the function can be obtained by combining the basic block syntax embedding with the control flow graph. It can be denoted as $g_{sy} = \langle \mathcal{V}, \mathcal{E}, \mathcal{X} \rangle$, where $\mathcal{V}$ is a set of basic blocks; $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges representing the connections between these basic blocks; $\mathcal{X}$ denotes the set of syntactic features for each basic block.

### 4.2.2. Basic Block Semantic Embedding

HBinSim also takes into account the generation of semantic embedding for each basic block. The whole process consists of two subtasks: token embedding generation and basic block semantic embedding generation. More specifically, a token corpus is firstly constructed, which is used to train a token embedding model derived from the Word2Vec algorithm [61]. Next, the basic block semantic embedding is generated based on token embedding. Figure 4 shows the four major steps for the basic block semantic embedding generation. Among them, the token embedding generation needs to be implemented only
once. After the token embedding model training is completed, the subsequent generation of the basic block semantic embedding only needs to start with the Normalization step.

(1) Random Walk. When distilling the semantics of each token, we would like to make use of the instructions around it as its context. However, if all tokens in a basic block are used as a training corpus, the trained token will not contain the context information of the other basic blocks. Moreover, if all tokens in a function are used as a training corpus, then the order of the tokens in the corpus is not a valid execution path. To address this problem, we refer to the DEEPBINDIFF [22] method by serializing ICFGs to extract control flow dependency information of the other basic blocks. Moreover, if all tokens in a basic block are used as a training corpus, then the order of the tokens in the corpus is not a valid execution path. To address this problem, we refer to the DEEPBINDIFF [22] method by serializing ICFGs to extract control flow dependency information. As depicted in the Random Walks step in Figure 4, take random walking in ICFGs so that each walk contains one possible execution path of the binary. Then, each random walk from the binary is used as a separate sequence of instructions, which is then used in turn for training.

(2) Normalization. The raw instructions may have some differences due to different compilation settings, such as choosing different compiler versions resulting in different registers used in the operands. When the code is cloned, some constant values or strings in the original code may be modified. In this case, if the raw instruction is used as a token for training, a serious OOV phenomenon will occur. To effectively alleviate these differences and reduce the probability of OOV, we do not use the raw instruction but treat the operands and opcode in the instruction as separate tokens and filter it as follows: (1) all strings are replaced with the special symbol "STR;" (2) all immediate values are replaced with the special symbol "HIMM;" (3) all general registers are renamed according to their lengths; (4) if the operand belongs to the memory type, determine whether the operand is based on the base addressing, if not, use "[MEM]" instead of the operand, if it is based on the base addressing without combined with index addressing, the operand is replaced with "[normalized register + HIMM].” Otherwise, it is indicated by "[normalized register + index * HIMM + HIMM].” The register represented by the index also needs to be normalized by the third rule. As example the instruction "mov eax, 0x10" becomes "mov,” “reg8," “HIMM,” "cmp dword [rbp – 0x10c], 0x10" becomes "cmp,” “[reg8 + HIMM],” “HIMM.”

(3) Token-Embedding Model. The normalized token cannot be directly used as the input of the graph neural network. Instead, these tokens need to be converted into vector representations. In this section, a representation model will be train and produce a numeric vector for each token. Figure 5 shows the structure of the token embedding model. It is different from the original Word2Vec CBOW [61] model, which uses words around a target word as context. In our case, we consider instructions around each token as its context.

Each token is mapped into a numeric vector \( \vec{v}_t \in \mathbb{R}^d \) and another numeric vector \( \vec{v}_p \in \mathbb{R}^{2 \times d} \), \( \vec{v}_t \) is the vector representations of token \( t \). After training, it represents a token’s lexical semantics. \( \vec{v}_p \) is used for token prediction. All \( \vec{v}_t \) is initialized to a random value between \([-1, 1]\).

Given an instruction sequence \( \delta (\text{seq}) \), which is represented as a list of instructions \( \delta (\text{seq}) = \{1: j\} \), where \( i_j \) is one of them. An instruction \( i_n \) contains a list of operands \( d (i_j) \) and one opcode \( R (i_j) \). Their concatenation is denoted as its list of tokens \( \mathcal{T} (i_j) = \mathcal{P} (i_j) \| d (i_j) \), where \( \| \) denotes concatenation. For \( \delta (\text{seq}) \), we collect the current instruction \( i_j \), its previous instruction \( i_{j-1} \), and its next instruction \( i_{j+1} \). If the current instruction is at the block boundary (e.g., the first instruction in the block), then only one adjacent instruction will be considered as its context. For each target token \( t_j \in \mathcal{T} (i_j) \), which belongs to the current instruction, the objective of the model is to maximize the average log probability \( \mathcal{J} (t_j) \) in Equation 1 as follows:

\[
\mathcal{J} (t_j) = \frac{1}{\sum_{j=1}^{16} (\text{seq})} \sum_{\delta (\text{seq})} \sum_{t_j} \log P (t_j | \text{in}_{j-1}, \text{in}_{j+1}).
\]

(1)

The vector representation \( \mathcal{C} \mathcal{T} (i_j) \in \mathbb{R}^{2 \times d} \) of the neighbor instruction in is calculated by averaging the vector representations of its operands and concatenating the averaged vector with the vector representation of the opcode. It can be formulated as

\[
\mathcal{C} \mathcal{T} (i_j) = \bar{\vec{v}}_{\mathcal{P} (i_j)} \| \frac{1}{d (i_j)} \sum_{t_j} \vec{v}_t.
\]

(2)
Recall that $\mathcal{P}(\bullet)$ denotes an opcode, $\mathcal{V}_{\mathcal{P}(\text{in})} \in \mathbb{R}^d$ is vector representation of the opcode $\mathcal{P}(\bullet)$, $t_h \in \mathcal{A}(\text{in})$ is an operand, and $\mathcal{V}_{t_h} \in \mathbb{R}^d$ is its vector representation. By averaging $\mathcal{E}(\text{in}_{j-1})$ and $\mathcal{E}(\text{in}_{j+1})$, $\delta(\text{in}_j)$ models the joint memory of neighbor instructions:

$$\delta(\text{in}_j) = \frac{1}{2} (\mathcal{E}(\text{in}_{j-1}) + \mathcal{E}(\text{in}_{j+1})).$$

Given $\delta(\text{in}_j)$, the probability term in equation (1) can be rewritten as follows:

$$P(t_{\text{in}_j} \vert \text{in}_{j-1}, \text{in}_{j+1}) = P(\mathcal{V}_{t_{\text{in}_j}} \vert \delta(\text{in}_j)).$$

where $P(\cdot)$ denotes the hierarchical softmax function and $\mathcal{V}_{t_{\text{in}_j}}$ is obtained by querying the dictionary established previously.

(4) Instruction-Embedding Generation. An instruction in $\text{in}_j$ contains a list of operands $\mathcal{A}(\text{in}_j)$ and one opcode $\mathcal{P}(\text{in}_j)$. In assembly code, different opcodes have different importance. For example, “mov” and “push” instructions are common in basic blocks, but they are less important than the information brought by “call” instructions. To distinguish the importance of these opcodes, we refer to the TF-IDF model [62] used DEEPBINDIFF to adjust the weights of opcodes by adopting a weighting strategy.

Therefore, the calculation of instruction embedding $\text{inst}_{\text{in}_j}$ is mainly divided into three steps: (1) opcode embedding $\mathcal{V}_{\mathcal{P}(\text{in}_j)}$ multiplies by its TF-IDF weight $\mathcal{W}_{\text{in}_j}$; (2) the average of operand embeddings $\mathcal{V}_{t_h} \in \mathcal{A}(\text{in}_j)$; (3) concatenating the calculated opcode embedding and operand embedding, as depicted in

$$\text{inst}_{\text{in}_j} = \mathcal{V}_{\mathcal{P}(\text{in}_j)} \mathcal{W}_{\text{in}_j} + \frac{1}{|\mathcal{A}(\text{in}_j)|} \sum_{t_h} \mathcal{V}_{t_h}.$$

(5) Attention-Based Basic Block Semantic Embedding Generation. The previous step generates $|\mathcal{F}(\text{bb}_i)|$ ($|\mathcal{F}(\text{bb}_j)$: the set of instruction embeddings $\text{inst}_{\text{in}_j}$ in basic block $\text{bb}_i$) are aggregated to basic block embeddings by employing the attention mechanism. The motivation is that different instructions will make various contributions to the representation of the basic block. Thus, the semantic embedding of a basic block $\text{bb}_i$ is formulated as

$$\text{bb}_i = \frac{1}{|\text{inst}_{\text{in}_j}|} \sum_{\text{inst}_{\text{in}_j}} \alpha_{\text{inst}_{\text{in}_j}},$$

where $\text{bb}_i \in \mathbb{R}^{d \times 1} (d$ is the feature embedding dimension), $\text{inst}_{\text{in}_j}$ is the instruction embedding, and $\alpha_{\text{inst}_{\text{in}_j}}$ indicates the importance of different embeddings, which is formulated as

$$\alpha_{\text{inst}_{\text{in}_j}} = \frac{\exp(\text{LeakyReLU}(u^T \text{inst}_{\text{in}_j}))}{\sum_{\text{inst}_{\text{in}_j}} \exp(\text{LeakyReLU}(u^T \text{inst}_{\text{in}_j}))},$$

where LeakyReLU denotes leaky version of a Rectified Linear Unit and $u \in \mathbb{R}^{d \times 1}$ is the attention parameter.

The Semantic Attributed Control Flow Graph (SeACFG) of the function can be obtained by combining the basic block semantic embedding with the control flow graph. It can be denoted as $\mathcal{g}_c = \langle \mathcal{F}, \mathcal{E}, \mathcal{X} \rangle$, where $\mathcal{F}$ is a set of basic blocks; $\mathcal{E} \subseteq \mathcal{F} \times \mathcal{F}$ is a set of edges representing the connections between these basic blocks; $\mathcal{X}$ denotes the set of semantic features for each basic block.

4.2.3. Attention-Based Graph-Embedding Network. The neural network model is the most important part of binary code similarity comparison. Since functions are
represented as two graphs (SyACFG and SeACFG), it is critical to learn accurate function features from the graphs. Moreover, deep learning models such as convolutional neural network (CNN) and recurrent neural network (RNN) cannot be used directly for graph-structured data because their raw input data are usually a picture or a textual description. To apply to our binary code similarity comparison scenario, the neural network dedicated to graph topology is required. Dai et al. [63] proposed a structure2vec graph embedding network, which is an effective and scalable approach for graph-structured data representation. It can encode vertex features and the connection relationship of the edges in the graph as the embedding vector to represent the function. So, we adapt Structure2vec [63] using the parameterization of [13].

The architecture of the graph embedding network is shown in Figure 2. If the embedding representation of graph \( g \) is obtained, the vector representation \( \mu \) (\( p \) dimensional) of each node \( v_i \) after iteration in round \( T \) needs to be computed first. After each step, the network generates a new \( \mu \) vector for each vertex in the graph, taking into account both the vertex features and graph-specific characteristics.

Denote \( \mathcal{N}(v_i) \) to be represented as the set of neighbors of vertex \( v_i \) in graph \( g \). The embedding \( \mu^{(0)} \) of each vertex is randomly initialized. The vertex vector \( \mu_i \) is updated at each round as follows:

\[
\mu^{(t+1)}_i = \mathcal{F} \left( x_{v_i}, \sum_{u \in \mathcal{N}(v_i)} \mu^{(t)}_u \right), \quad \forall v_i \in \mathcal{V},
\]

where \( \mathcal{V} \) is a set of vertex and \( \mathcal{F} \) is a nonlinear function:

\[
\mathcal{F} \left( x_{v_i}, \sum_{u \in \mathcal{N}(v_i)} \mu^{(t)}_u \right) = \tanh \left( W_1 x_{v_i} + \sigma \left( \sum_{u \in \mathcal{N}(v_i)} \mu^{(t)}_u \right) \right),
\]

where \( x_{v_i} \) is a \( d \)-dimensional vector for basic block embedding \( (x_{v_i} = bb_{emb}_{u} \) and \( bb_{emb}_{v} \), \( W_1 \) is a \( d \times p \) matrix, and \( p \) is the embedding size as explained above. \( \sigma \) is a non-linear function:

\[
\sigma (l) = P_1 \times \text{ReLU} \left( P_2 \times \ldots \times \text{ReLU} \left( P_n l \right) \right),
\]

where \( \sigma (l) \) is an \( n \) layers fully connected neural network and \( P_i (i = 1, \ldots, n) \) is a \( p \times p \) matrix.

In [13], the final graph embedding \( \overrightarrow{g} \) is obtained by aggregating the vertex vector \( \mu \) after \( T \) rounds as follows:

\[
\overrightarrow{g} = W_2 \sum_{t \in \mathcal{V}} \mu^{(t)},
\]

where \( W_2 \) is another \( p \times p \) matrix used to transform the final graph embedding vector.

However, instead of taking this approach, an attention-based mechanism is proposed for aggregating together the vertex \( \mu \) vectors after \( T \) rounds. The motivation is that different basic blocks will make various contributions to the final representation of the function. Thus, the graph \( g \) embedding \( \overrightarrow{g} \) is formulated as

\[
\overrightarrow{g} = \sum_{i \in \mathcal{V}} a_i \mu^{(T)}_i,
\]

where \( \overrightarrow{g} \in \mathbb{R}^{m \times 1} \) (\( m \): output embedding dimension) and \( a_i \) indicates the importance of different vertex \( \mu \) vectors. \( a_i \) is similar to \( a_{in} \) calculation.

### 4.2.4. Attention-Based Function-Embedding Generation

After the graph embedding network being trained, the function semantic embedding \( \overrightarrow{g}_s \) and function syntactic embedding \( \overrightarrow{g}_y \) can be obtained. To measure the different importance of the syntactic and semantic features, these two types of embedding will be aggregated based on the attention mechanism to generate the final function embedding representation, see Figure 2. We denote the set of embeddings of a function as \( \mathcal{F} = \{ \overrightarrow{g}_s \cup \overrightarrow{g}_y \} \) and the final function embedding \( \overrightarrow{f} \) as

\[
\overrightarrow{f} = \sum_{g \in \mathcal{F}} a_j g_j,
\]

where \( h \in \mathbb{R}^{m \times 1} \) is the attention parameter.

#### 4.3. Similarity Comparison

The input of binary code similarity comparison should be a pair of functions, and the output is their degree of similarity. The Siamese architecture [43] works well for this scenario. It mainly consists of two identical subnetworks, i.e., hierarchical attention graph embedding network. Each subnetwork takes a processed feature as its input and then outputs the embedding representation of the feature. In particular, the two subnetworks share all the parameters. The joining neuron is used to measure the distance between these two output embedding representations of the two subnetworks and output the similarity value ranging from \( -1 \) to \( 1 \). Therefore, the similarity calculation process of our model is that we input a function pair \( \langle f_1, f_2 \rangle \), and the embedding vector \( \langle \overrightarrow{f}_1, \overrightarrow{f}_2 \rangle \) of the function will be obtained through the same hierarchical attention graph embedding network. These vectors are compared using cosine similarity as the distance metric, with the following formula:

\[
\text{similarity} \left( \overrightarrow{f}_1, \overrightarrow{f}_2 \right) = \frac{\sum_{i=1}^{n} \overrightarrow{f}_1[i] \cdot \overrightarrow{f}_2[i]}{\sqrt{\sum_{i=1}^{n} \overrightarrow{f}_1[i] \cdot \sum_{i=1}^{n} \overrightarrow{f}_2[i]}},
\]

where \( \overrightarrow{f} [i] \) indicates the \( i \)th component of the vector \( \overrightarrow{f} \).

Moreover, we give a set of \( K \) pairs of ACFGs \( \langle f_1, f_2 \rangle \). Then, the ground truth pairing information \( \alpha_{ij} \in \{+1, -1\} \) is assigned according to whether two functions are similar, where \( \alpha_{ij} = +1 \) indicates that the two functions are similar and \( \alpha_{ij} = -1 \) otherwise. The model is trained by minimizing the following mean square error loss function:
In this section, we evaluate HBinSim in terms of accuracy, efficiency, and scalability. First, the experimental settings are described. Next, the impact of preprocessing on the out-of-vocabulary token and the effectiveness of our proposed model are examined. We then evaluate whether HBinSim can successfully detect the similar function compiled by different compilers, different binary versions, different compiler optimization levels, and different obfuscation techniques. Finally, the effectiveness of HBinSim in real-world vulnerability detection is verified through vulnerability function search, and the efficiency of HBinSim is evaluated.

5.1. Experimental Setup. Our experiments are performed on a moderate desktop computer running Ubuntu 18.04 LTS operating system with two Intel Xeon E5-2680v4 CPUs (56 cores in total) running at 2.40 GHz, 4 TB HDD, 128 GB memory, and four GeForce GTX 2080 Ti GPU cards. In the experiment, the Radare2 (Radare2 is an open-source framework (https://rada.re/n/radare2.html) that supports multiple architectures and supports interaction with multiple scripting languages) framework is used for static analysis.

5.1.1. Data Preparation. The multiple datasets will be built based on three popular programs: Coreutils (https://www.gnu.org/software/coreutils/), Diffutils (https://www.gnu.org/software/diffutils/), and Findutils (https://www.gnu.org/software/findutils/) to evaluate the effectiveness of HBinSim in different compilation settings. Moreover, the various versions of these binaries also will be collected, including Coreutils (v5.93, v6.4, v7.6, v8.1, and v8.30), Diffutils (v2.8, v3.1, v3.3, v3.4, and v3.6), and Findutils (v4.6, v4.41, and v4.233). The details of the datasets are

(1) Dataset I: we compile all programs using GCC v5.4 with four different compiler optimization levels (O0, O1, O2, and O3) to produce binaries equipped with different optimization techniques. This dataset is first used for token embedding model training and then to evaluate the effectiveness of HBinSim.

(2) Dataset II: for the cross-compiler scenario, we select three binaries (including Coreutils v8.30, Findutils v4.6, and Diffutils v3.6) and use the GCC v5.4 and CLANG v4.0.1 compiler with the O2 optimization level to obtain the binaries compiled by different compilers.

(3) Dataset III: for the cross-version scenario, we split dataset I from four cross-version datasets based on the same optimization level (O0, O1, O2, and O3).

(4) Dataset IV: we select binaries (including Coreutils v8.30, Findutils v4.6, and Diffutils v3.6) that use the GCC compiler with four different compiler optimization settings. Then, cross-optimization level dataset is constructed by combining two different of the four compiler optimization settings. Finally, 6 cross-optimization level dataset are constructed.

(5) Dataset V: for code obfuscation scenario, we choose the popular obfuscator-LLVM (O-LLVM) [64], whose obfuscation strategies include three different techniques and their combinations: Instruction Substitution (SUB), Control Flow Flattening (FLA), and Bogus Control Flow (BCF). O-LLVM heavily modifies the original assembly code and breaks the CFG and basic blocks integrity. By design, most of the static features are oblivious to obfuscation. The selected binaries (including Coreutils V8.30, Findutils V4.6, and Diffutils V3.6) are compiled first without applying any obfuscating techniques. Then, different obfuscation techniques (including SUB, FLA, BCF, and SUB+FLA+BCF) are used to recompile the binaries. Finally, the data is extracted from these five types of binaries to build the dataset.

(6) Dataset VI: finally, a publicly available vulnerability dataset presented in [37] is used as a case study to demonstrate the usefulness of HBinSim in practice.

To get the ground truth of searching functions, we create a certain number of pairs of two kinds from each dataset: similar pairs (labeling +1), obtained by pairing together two functions originated by the same source code, and dissimilar pairs (labeling −1), obtained by randomly pairing functions not derive from the same source code. We split pairs into three disjoint subsets for training, validation, and testing. The proportion of these three subsets is roughly set at 3:1:1.

5.1.2. Baselines. With the datasets above, we compare HBinSim with two state-of-the-art baselines: Gemini [13], which is a clone search model based on syntactic features and structural features, and i2v_attention [26], which is a clone search model based on semantic features and structural features.

5.1.3. Parameters’ Set. When training the word2vec model, to ensure the completeness of the basic block coverage in the binary and each random walk can carry enough control flow information, we configure each basic block to contain at least 2 random walks and the length of each random walk is 5.

In all of the experiments, we trained the hierarchical attention graph embedding network using a batch size of 150, learning rate of 0.001, Adam optimizer, feature vector size is 128, the number of rounds is 2, and the number of layers in Structure2Vec is 2. Therefore, we manipulate each vertex to contain the same number of instructions, i.e., fixing the instruction length in each basic block to 100 by either padding with a special instruction or by truncation. Padding
vectors are all zeros. To make CFG contain more graph structural information, we exclude functions with a basic block number less than 5. The number of basic blocks included in the function is mostly within 100. We choose the maximum number of basic blocks to be 100, removing CFGs larger than this threshold to avoid sparse features due to padding.

Under the current parameter settings, we compared the performance of neural networks in the model through computational complexity, in terms of FLOPs (floating-point operations, in no. of multiply adds) and numbers of parameters. HBinSim is 1.55x and 1.26x more than Gemini and i2V_attention in FLOPs, respectively. The number of trainable parameters of HBinSim is about 2x that of i2V_attention and Gemini. The cause of this phenomenon is that multiple features and attention mechanisms are considered in HBinSim. Detailed results can be found in Table 2.

5.1.4. Metrics. We test our system using the area under curve and precision as evaluation metrics. The AUC is equivalent to the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example [65]. Therefore, classifiers with larger AUC values have better generalization performance. The precision shows the percentage of correct matching pairs among all the known pairs (similarity and dissimilarity). In the vulnerability search experiment, we use the recall metric to represent the percentage that the vulnerability function is correctly matched.

5.2. Evaluation on Out-of-Vocabulary Token. To verify that our proposed method can alleviate the OOV challenge, we evaluate its impact and seek to understand: (a) the size of the vocabulary (the number of columns in the instruction embedding matrix) generated by different methods, (b) the number of OOV cases in later token embedding generation, and (c) the distribution of tokens in OOV cases.

To this end, we use dataset I to generate a corpus that contains 14,192,918 assembly instructions and 38,024,234 opcodes and operands. We then counted the number of unique tokens with different methods in this corpus. i2v is a method proposed by Baldoni et al. [26] for building vocabularies by treating each normalized instruction as a token. The DEEPBINDIFF approach is to treat the preprocessed operand and opcode as a token, respectively. The results are shown in Table 3. As shown, the sizes of the vocabularies built by i2v and DEEPBINDIFF are 424,795 and 63,007, respectively. However, the size of the vocabulary built by our method is only 238, which is 0.056% and 0.38% of the size of the previous two methods.

Next, investigate the number of OOV cases, i.e., unseen token, in later token embedding generation. We select OpenSSL (v1.1.1) binary compiled by the clang compiler at the default optimization level (O3) that have never appeared in the previous corpus, which containing 666,687 instructions and 1,809,594 operands and opcodes. We then count the number of unseen tokens that do not exist in the vocabulary and show the result in Table 3. It shows that only 1.06% of unseen tokens appeared in later token embedding generation of HBinSim, which is much smaller than i2v (20.88%) and DEEPBINDIFF (3.89%). From the above two aspects of testing, we can find that our proposed model can obtain higher instruction coverage with a smaller vocabulary size, which indicates that our proposed token-embedding model is more effective and leads to smaller OOV cases.

We analyze the OOV tokens in each method in the test corpus and count the occurrence frequency of each token. The results are shown in Figure 6. The OOV tokens appearing less than 5 times account for 92.0% and 92.2% among all the OOV tokens generated by i2v and DEEPBINDIFF, respectively. Therefore, the token embedding model of HBinSim will be denser and have less space complexity and time complexity than other models. In addition, OOV generated by i2v contains a lot of “X_cmp_cl,[rdx ∗ 1 + 0x10],” “X_imul_esi,[rbx ∗ 1 + 32], HIMM,” and other similar tokens. There are also a large of “[rip + 0 × 10b2],” “[rip + 0x10e2],” and other similar tokens in the OOV generated by DEEPBINDIFF, in which the registers and offset values can be changed arbitrarily, which is the main reason for a large amount of OOV generated by these two methods. The tokens in OOV generated by our method are opcode of assembly instructions, such as “Xadd” and “Mfence”. There is the fact that the number of opcodes in any instruction set is 10^7 orders of magnitude, and the number of memory offsets contained in the operands is on the 10^4 orders of magnitude. If it is base addressing with index addressing, the number of operands for memory types can reach 10^7 orders of magnitude. So, when our model normalizes offset values, registers, etc., that can completely cover all instructions in a smaller token corpus, the size of the vocabulary is only on 10^2 orders of magnitude.

| Method      | FLOPs   | No. of parameters |
|-------------|---------|------------------|
| Gemini      | 501.78k | 50.18k           |
| i2v_attention| 619.57k | 62.15k           |
| HBinSim     | 779.80k | 116.61k          |

| Categories | i2v [26] | DEEPBINDIFF [22] | HBinSim |
|------------|---------|------------------|---------|
| Train      |         |                  |         |
| #Token number | 14,192,918 | 38,024,234 | 38,024,234 |
| #Vocabulary size | 424,795 | 63,007 | 238 |
| Test       |         |                  |         |
| #OOV count | 139,201 | 70,497           | 19,121  |
| #OOV size  | 39982   | 9787             | 118     |
| #OOV       | 20.88%  | 3.89%            | 1.06%   |
5.3. Effectiveness of the Model. In this experiment, the effectiveness of our proposed model for code clone search based on dataset I is evaluated. The evaluation mainly answers the following three questions: (1) whether the token embeddings we trained can be used for actual code clone search; (2) is it useful to apply attention mechanisms to the model; (3) whether the method combining syntactic and semantic features is more effective than the methods only considering syntactic features or semantic features. To this end, the following HBinSim variation models are proposed and compared with the other two baseline models.

1. HBinSim-vs: this model only considers the semantic features of the basic blocks to construct the instruction embeddings. The average is used when the instruction embeddings are aggregated.

2. HBinSim-as: this model only considers the semantic features of basic blocks and adopts the attention mechanism in the aggregation process of instruction embedding.

3. HBinSim-aa: this model only considers the semantic features of basic blocks, and the attention mechanism is used both in the aggregation of instruction embeddings and Structure2Vec node embeddings.

4. HBinSim-aav: this model considers both syntactic and semantic features and adopts the attention mechanism when the instruction embedding and Structure2Vec node embedding aggregate. The average method is used for the aggregation of function syntactic embedding vectors and function semantic embedding vectors.

Figure 7 illustrates the ROC curves for our model and baseline approaches in the code clone search. HBinSim outperforms not only benchmark Gemini and i2v_attention but also the other four HBinSim variation models. The AUC and precision of similarity comparison results are shown in Figure 8. HBinSim-vs, which is based on our proposed token embedding model, has higher clone search capabilities than i2v_attention, which is based on semantic features and uses the i2v token embedding model. The precision is improved by 6.3%. The results show that our proposed token embedding model can be better used for clone search.

Our model uses the attention mechanism at three levels: instruction embeddings’ aggregation, basic block embeddings’ aggregation, and function embeddings’ aggregation. For instruction embeddings’ aggregation, the attention-based HBinSim-as improves precision by 1.3% compared to HBinSim-vs without the attention mechanism. For basic block embeddings’ aggregation, the performance of HBinSim-aa that uses the attention mechanism is also improved by 0.1% compared to HBinSim-as without the attention mechanism. For function embedding aggregation, the precision improvement of HBinSim based on the attention mechanism is tiny compared to HBinSim-aav without the attention mechanism, which also shows that semantic information and syntactic information play an essential role in the function clone search. The above three comparison experiments show that using the attention mechanism can make the model pay more attention to the critical parts, thus ignoring the influence of some unnecessary information to improve the clone search performance of the model.
For the combination of semantic information and syntactic information, the precision of HBinSim-aav combining the two features is improved by 1% compared with HBinSim-aa based only on semantic features. The result indicates that we extract information of more dimensions of function which can help us better recognize functions. Moreover, as shown in Figure 8, our proposed model and its variants perform improved performance compared to Gemini and i2v_attention baselines. Compared with Gemini and i2v_attention, HBinSim improves precision by 4.7% and 8.8%, respectively. Besides, we observe that Gemini performed better than i2v_attention, which shows that syntactic information is more effective than semantic information in some scenarios.

5.4. Impact of Compilation Diversity. To demonstrate the effectiveness of HBinSim against compilation diversity, we evaluate HBinSim under cross compiler, cross version, cross-optimization level, and code obfuscation scenarios, respectively.

5.4.1. Cross Compiler. In this experiment, we benchmark the clone search performance of HBinSim and baseline under cross-compiler conditions (GCC v5.4.0 and CLANG v4.0.1) on dataset II.

Figure 9 shows the ROC curves of the code clone search for all models in cross-compiler binaries. The dashed line is the ROC for Gemini and i2v_attention; the continuous line is the ROC for HBinSim. As each curve is close to the left-hand and top border, our models have good AUC. The AUC values of HBinSim, Gemini, and i2v_attention are 0.987, 0.945, and 0.951, respectively. Overall, the improvement is 4.2% and 3.6% of HBinSim with respect to Gemini and i2v_attention, respectively. Here, i2v_attention is more resistant to cross-compiler differences than Gemini. In general, our proposed model is much less sensitive than other tools, which demonstrates its robustness in across compiler.

5.4.2. Cross Version. In this experiment, we benchmark the clone search performance against different versions of binaries in dataset III. We report the AUC and precision results for each tool under cross version with assigned optimization levels in Table 4.

As shown, HBinSim outperforms Gemini and i2v_attention across-version of four datasets in terms of AUC and precision. The best results are for datasets at the O0 optimization level; HBinSim improves the precision by 10.9% and 10.3% over Gemini and i2v_attention. In four different cross version of the datasets, HBinSim has an average precision improvement of 7.2% and 7.1% compared with Gemini and i2v_attention. We can also observe that the results of Gemini and i2v_attention in AUC and precision are not much different. This result shows that syntactic information and semantic information are of similar importance in cross-version binary code search. Moreover, the performance of HBinSim shows that it can achieve higher AUC and precision with the inclusion of syntactic and semantic information.

5.4.3. Cross-Optimization Level. In this experiment, we benchmark the clone search performance against different optimization levels with the GCC compiler. We test every combination of two different compiler optimization settings on dataset IV and perform six times’ experiments. The AUC and precision results are reported in Table 5.

As shown in Table 5, HBinSim significantly outperforms i2v_attention and Gemini in all situations. It indicates that HBinSim is robust against heavy syntax modifications and intensive inlining introduced by the compiler. We can know the
following fact: a higher optimization level contains all optimization strategies from the lower level. In the greatest difference between two optimization levels’ case (O0–O3), the learned representation can correctly match more than 92.7% of assembly functions in precision. It improves the precision by 17.8% and 17.5% over Gemini and i2v_attention. In the best situation (O2–O3), HBinSim reaches the AUC value of 0.992 and the precision of 0.927. On average, HBinSim achieves the AUC value of 0.988 and the precision of 0.909 in detecting clones among four compiler optimization options. BU_he precision increased, respectively, by 14.3% and 12.7% compared with the baseline. Nevertheless, it is much less sensitive than the other tools, demonstrating its robustness in the cross-optimization level. The result also shows that cross-optimization-level binary clone search is more complicated than cross-version clone search since the AUC rates are reduced. The main reason is that the compiler optimization techniques could significantly transform the binaries.

5.4.4. Code Obfuscation. In this experiment, the impact of obfuscation techniques on HBinSim as well as Gemini and i2v_attention approaches will be investigated on dataset V. We test every combination between codes that have not applied any obfuscation techniques and codes that applied different obfuscation techniques, the situation where all data are mixed. The existing clone search for obfuscated code is performed between the obfuscated codes and the codes without any obfuscation techniques applied. However, our datasets allow clone search between different obfuscation techniques.

In Table 6, “None” represents code that not applied any obfuscation techniques, “SUB” denotes code that applied SUB obfuscation techniques, “BCF” denotes code that applied BCF obfuscation techniques, “FLA” denotes code that applied FLA obfuscation techniques, “SUB + FLA + BCF” denotes code that applied all the obfuscation options above, and “Mix” denotes code that includes all the codes above. Table 6 shows the results of clone search in code obfuscation. SUB breaks the sequence by adding instructions in between. However, all baselines can still recover more than 75% of clones in precision since the syntactic and semantic features can resist these interferences, and the graph structure is not heavily modified. HBinSim can improve 14.1% and 12.6% precision compared with the Gemini and i2v_attention against assembly instruction substitution. Instructions are replaced with their equivalent form, which still shares similar lexical semantic to the original. HBinSim well captures this information.

After applying BCF or FLA obfuscation, the performance is reduced compared to the applied SUB obfuscation technique. The main reason is that both methods modify the control flow graph, e.g., adding a large number of irrelevant random basic blocks and branches or using complex hierarchy of new conditions as switches. So, Gemini and i2v_attention, which are based on fewer feature dimensions, have lower performance. HBinSim can still achieve the lowest AUC value of 0.951 and the lowest precision of 0.842 in code applied BCF or FLA obfuscation. It improves precision by 15.2% and 11.2% over Gemini and i2v_attention.

After applying all the obfuscation techniques, HBinSim can still recover around 0.835 precision of assembly functions. Although the AUC and precision are reduced compared to when each obfuscation technology is used alone, it is improved, respectively, by 18.7% and 9.9% precision compared with Gemini and i2v_attention. When all the data
are mixed, our model can effectively recover precision of 0.901 from different obfuscation techniques. Moreover, the average performance of i2v_attention is 3.5% higher than that of Gemini in precision in the obfuscated code scenario, with a best-case improvement of 8.8%, which shows that semantic information is more effective in resisting obfuscation technology than syntactic information.

When the obfuscation technique is applied to code, changes in the instruction and structure can lead to a degradation in the performance of the model, especially when the structure is changed, e.g., inserting junk code and faking basic blocks. Inserted junk basic blocks or noise instructions follow the general syntax of random assembly code. HBinSim can correctly pinpoint and identify critical patterns from noise because it captures the multiview features of the function against compilation diversity.

5.5. Application to Real Vulnerability Search. In this case study, we apply HBinSim on a publicly available vulnerability dataset presented in dataset VI to evaluate its performance in actually recovering the reuse of the vulnerabilities in functions. The dataset contains several vulnerable binaries compiled with 11 compilers in the families of clang, gcc, and icc. The total number of different vulnerabilities is 8 (cve-2011-0444, cve-2014-0160, cve-2014-4877, cve-2014-6271, cve-2014-7169, cve-2014-9295, cve-2015-3456, and cve-2015-6862). We disassembled the dataset with Radare2, obtaining 3,005 binary functions. We performed a lookup for each of the 8 vulnerabilities, computing the recall on each result. Finally, we averaged these performances over the 8 queries.

The results of our experiments are reported in Figure 10. The results show that HBinSim outperforms Gemini and i2v_attention in tests. For \( k = 20 \), our maximum recall is 84.5% (vs. 74.1% recall of i2v_attention and 70.7% recall of Gemini, with an increment of performance of 10.4% and 13.8%, respectively). For \( k = 200 \), HBinSim reaches the average recall of 94.6%, while i2v_attention reaches a recall of 82.6%, and Gemini is only 76.7%. In addition, when \( k = 154 \), HBinSim can correctly detect all the vulnerability functions, while the other two models are not fully detected at \( k = 200 \).

To better understand our vulnerability search, we analyze in detail one of the vulnerability query examples. The query for this experiment was the CVE-2015-6826 vulnerable procedure from ffmpeg.2.6.4-rv34 compiled with Clang 3.5. Each bar in Figure 11 represents a single target procedure, and the height of the bar represents the similarity score (normalized) against the query. The specific compiler vendor and version were noted below the graph (on the X-axis) and the source of the package and the version above it. Bars filled in green represent procedures originating from the same code as the query (i.e., "CVE-2015-6826") but vary in the compilation. All unrelated procedures were filled blue.

As we can see from the results in Figure 11, our method gives high scores to all other similar versions of the CVE-2015-6826 procedure, despite them being compiled using different compilers, different compiler versions. A gap of 0.346 in the similarity score exists between the true positives from the rest of the procedures (0.999 for the icc 15 compiled procedure of ffmpeg.2.6.4-rv34 vs. 0.653 for the Coreutils.8.23 normal procedure compiled with icc 15). It is important to note that we will not try to establish a fixed threshold to evaluate the quality of these results. As mentioned, this clean separation between the true positives and
the false positives is not always possible. Instead, this result and others are evaluated according to the produced ranking. The result in Figure 11 receives a ROC = 1.0 score as it puts all of the true positives at the top of the ranking.

5.6. Efficiency. We then evaluate the efficiency of HBinSim, which can be split into three: feature extraction and processing time, embedding generation time, and similarity comparison time.

5.6.1. Feature Extraction and Processing Time. Static analysis needs to extract the attribute control flow graph of each function in the binaries and normalize the syntactic and semantic features. The average time required for this processing is 0.2 s/kB.

5.6.2. Embedding Generation Time. The most massive part of HBinSim is the embedding generation. On average, it takes 6.423 ms to generate one function embedding.

5.6.3. Similarity Comparison Time. We calculate the similarity comparison time based on the experiment of vulnerability search. There are a total of 58 functions to be searched, which are divided into 8 times, each time with 3005 functions to be calculated. The total time spent on 8 searches is 791 ms.

6. Discussion and Conclusions

We did not discuss all the different compilation settings, such as across-compiler version and across architecture. For the cross-compiler version, the vulnerability search experiment dataset contains the functions compiled by different versions of the compiler, which can prove that our method can resist the differences between the cross-compiler version. For the cross architecture, we only need to build a token corpus in different architectures and train token vectors in different architectures. Then, our model can be completely migrated to cross-architecture scenarios.

As for the generation of function embedding, a hierarchical attention graph embedding network is proposed based on Structure2Vec. However, there are still many other graph neural networks that can be used, such as GCN [52], GAN [66], or MPNN [67]. We will leave the evaluation of HBinSim for this case as the future work.

In this paper, a robust and accurate binary code clone search approach is proposed and named HBinSim, which learns vector representation of an assembly function by discriminating it from the others. To precisely match the functions against compilation diversity, the syntactic, semantic, and structural information of function are extracted. Among them, we use token construction methods based on different addressing models to address the issue of OOV effectively. Then, a hierarchical attention graph embedding network is proposed, which integrates multilevel attention mechanisms with a graph-embedding network which progressively builds a function-embedding vector. It consists of three steps: first, the important instruction-embedding vectors are aggregated into basic block-embedding vectors. Then, the important basic block syntactic- and semantic-embedding vectors are aggregated into function syntactic- and semantic-embedding vectors. Finally, the different important syntactic- and semantic-embedding vectors of function are aggregated into function-embedding vectors, which carry both the syntactic, semantic, and structural information of function. We conduct extensive experiments on binary code clone search with various compilation settings considering different compilers, compiler optimization levels, binary versions, and obfuscation techniques. Our results suggest that HBinSim is accurate and robust against severe changes in the assembly instructions and
control flow graph. We also conduct a vulnerability search case study on a publicly available vulnerability dataset, where HBinSim achieves higher recall than baselines.

Data Availability

The data used to support the findings of the study are available at https://github.com/wtwofire/HBinSim.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China, under Grant 2017YFB0802900, the China Postdoctoral Science Foundation (2020M673276), and the Fundamental Research Funds for the Central Universities (2020SCU12066).

References

[1] A. Saebjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, “Detecting code clones in binary executables,” in Proceedings of the 18th International Symposium on Software Testing and Analysis, pp. 117–128, Association for Computing Machinery, New York, NY, USA, July 2009.

[2] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “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, pp. 389–400, Association for Computing Machinery, New York, NY, USA, November 2014.

[3] Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, “Program characterization using runtime values and its application to software plagiarism detection,” IEEE Transactions on Software Engineering, vol. 41, pp. 925–943, 2015.

[4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection,” IEEE Transactions on Software Engineering, vol. 43, pp. 1157–1177, 2017.

[5] K. Iwamoto and K. Wasaki, “Malware classification based on extracted API sequences using static analysis,” in Proceedings of the Asian Internet Engineering Conference, pp. 31–38, Association for Computing Machinery, New York, NY, USA, November 2012.

[6] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware android malware classification using weighted contextual API dependency graphs,” in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1105–1116, Association for Computing Machinery, New York, NY, USA, November 2014.

[7] J. Jang, M. Woo, and D. Brumley, “Towards automatic software lineage inference,” in Proceedings of the 22nd USENIX Security Symposium, pp. 81–96, USENIX Association, Washington, DC, USA, August 2013.

[8] X. Hu, T. Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” in Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 611–620, Association for Computing Machinery, New York, NY, USA, August 2009.

[9] Y. Li, W. Xu, Y. Tang, X. Mi, and B. Wang, “Semhunt: identifying vulnerability type with double validation in binary code,” in Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering, pp. 491–494, July 2017.

[10] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, “SPAIN: security patch analysis for binaries towards understanding the pain and pills,” in Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 462–472, IEEE, Buenos Aires, Argentina, July 2017.

[11] J. Pewny, B. Garmany, R. Gawlik, C. Rossov, and T. Holz, “Cross-architecture bug search in binary executables,” in Proceedings of the 2015 IEEE Symposium on Security and Privacy, pp. 709–724, IEEE, San Jose, CA, USA, July 2015.

[12] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, “Scalable graph-based bug search for firmware images,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 480–491, ACM, Vienna, Austria, October 2016.

[13] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376, Association for Computing Machinery, New York, NY, USA, August 2017.

[14] W. M. Khoo, A. Myncroft, and R. Anderson, "Rendezvous: a search engine for binary code," in Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 329–338, IEEE, San Francisco, CA, USA, October 2013.

[15] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in Proceedings of the 2006 13th Working Conference on Reverse Engineering, pp. 253–262, IEEE, Benevento, Italy, October 2006.

[16] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossov, "Leveraging semantic signatures for bug search in binary programs," in Proceedings of the 30th Annual Computer Security Applications Conference on ACSAC’14, pp. 406–415, ACM Press, New Orleans, LA, USA, 2014.

[17] Zynamics BinDiff, https://www.zynamics.com/bindiff.html, 2020.

[18] M. Bourquin, A. King, and E. Robbins, "BinSlayer: accurate comparison of binary executables," in Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop on-PPREW’13, pp. 1–10, ACM Press, Rome, Italy, December 2013.

[19] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," Sstic, vol. 5, pp. 1–13, 2005.

[20] H. Flake, "Structural comparison of executable objects," in Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, pp. 161–173, Gesellschaft für Informatik e.V., Bonn, Germany, July 2004.

[21] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489, IEEE, San Francisco, CA, USA, September 2019.

[22] Y. Duan, X. Li, J. Wang, and H. Yin, “DEEPBINDIFF: learning program-wide code representations for binary differencing," in
Proceedings of the 2020 Network and Distributed System Security Symposium, April 2020.

[23] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in Proceedings 2019 Network and Distributed System Security Symposium, 2019.

[24] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovering cross-architecture identification of bugs in binary code," in Proceedings of the 2016 Network and Distributed System Security Symposium, February 2016.

[25] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao et al., "Diff-version binary code similarity detection with DNN," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering-ASE 2018, pp. 667–678, ACM Press, Montpellier, France, September 2018.

[26] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in Proceedings of the 2019 Workshop on Binary Analysis Research, May 2019.

[27] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing inspired binary code analysis," in Proceedings of the NDSS Workshop on Binary Analysis Research (BAR), February 2019.

[28] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Function representations for binary similarity," IEEE Transactions on Dependable and Secure Computing, p. 1, 2021.

[29] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1480–1489, San Diego, CA, USA, June 2016.

[30] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilingual token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, pp. 654–670, 2002.

[31] L. Jiang, G. Mishchergh, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in Proceedings of the 29th International Conference on Software Engineering (ICSE’07), pp. 96–105, IEEE, Minneapolis, MN, USA, May 2007.

[32] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in Proceedings of the 28th Annual Computer Security Applications Conference - ACSAC’12, p. 359, December 2012.

[33] F. Gauthier, T. Lavoie, and E. Merlo, "Uncovering access control weaknesses and flaws with security-discordant software clones," in Proceedings of the 29th Annual Computer Security Applications Conference on—ACSAC’13, pp. 209–218, ACM Press, New Orleans, LA, USA, December 2013.

[34] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire OS distributions," in Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp. 48–62, San Francisco, CA, USA, December 2012.

[35] K. McEnamul, A. Walenstein, A. Lakhota, and L. Parida, "Malware phylogeny generation using permutations of code," Journal in Computer Virology, vol. 1, pp. 13–23, 2005.

[36] Y. David and E. Yahav, "Tracelet-based code search in executables," SIGPLAN Notices, vol. 49, pp. 349–360, 2014.

[37] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," SIGPLAN Notices, vol. 51, pp. 266–280, 2016.

[38] D. Gao, M. K. Reiter, and D. Song, BinHunt: Automatically Finding Semantic Differences in Binary Programs, Information and Communications Security, Springer, Berlin, Heidelberg, 2008.

[39] J. Ming, M. Pan, and G. Debin, "Binary hunting with interprocedural control flow," in Information Security and Cryptology-ICISC Springer, Berlin, Heidelberg, 2012.

[40] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," Proceedings of the ACM on Programming Languages, vol. 4, 2020.

[41] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: predicting debug information in stripped binaries," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1667–1680, ACM, Toronto Canada, October 2018.

[42] J. Lacomis, P. Yin, E. Schwartz et al., "DIRE: a neural approach to decompiled identifier naming," in Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 628–639, IEEE, San Diego, CA, USA, November 2019.

[43] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in Proceedings of the 6th International Conference on Neural Information Processing Systems, pp. 737–744, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, January 1993.

[44] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: semantic-aware neural networks for binary code similarity detection," AAAI, vol. 34, pp. 1145–1152, 2020.

[45] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and Z. P. Codee, "A tensor embedding scheme for binary code search," IEEE Transactions on Software Engineering, p. 1, 2021.

[46] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, "BinDeep: a deep learning approach to binary code similarity detection," Expert Systems with Applications, vol. 168, Article ID 114348, 2021.

[47] J. L. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in Proceedings of the KSII The 9th International Conference on Internet (ICONI) 2017 Symposium, Vientien, Laos, December 2017.

[48] X. Li, Q. Yu, and H. Yin, "Palmtree: learning an assembly language model for instruction embedding," 2021, https://arxiv.org/abs/2103.03809.

[49] S. Ullah and H. Oh, "BinDiffNN: learning distributed representation of assembly for robust binary differencing against semantic difference," IEEE Transactions on Software Engineering, p. 1, 2021.

[50] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2018, https://arxiv.org/abs/1711.00740.

[51] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan, "Bin2vec: learning representations of binary executable programs for security tasks," 2021, https://arxiv.org/abs/2002.03388.

[52] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017, https://arxiv.org/abs/1609.02907.

[53] A. Nair, A. Roy, and K. Meinke, "FuncGNN: a graph neural network approach to program similarity," in Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11, ACM, Bari, Italy, August 2020.
[54] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 1025–1035, Curran Associates Inc., Red Hook, NY, USA, December 2017.
[55] X. Ling, L. Wu, S. Wang et al., "Multi-level graph matching networks for deep graph similarity learning," 2021, https://arxiv.org/abs/2007.04395.
[56] H. Xiu, X. Yan, X. Wang, J. Cheng, and L. Cao, "Hierarchical graph matching network for graph similarity computation," 2020, https://arxiv.org/abs/2006.16551.
[57] D. Gibert, C. Mateu, and J. Planes, "A hierarchical convolutional neural network for malware classification," in Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8, IEEE, Budapest, Hungary, July 2019.
[58] H. Yan, S. Luo, L. Pan, and Y. Zhang, "HAN-BSVD: a hierarchical attention network for binary software vulnerability detection," Computers & Security, vol. 108, Article ID 102286, 2021.
[59] I. U. Haq and J. Caballero, "A survey of binary code similarity," 2019, https://arxiv.org/abs/1909.11424.
[60] S. Alrabaee, L. Wang, and M. Debbabi, "BinGold: towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs)," Digital Investigation, vol. 18, pp. S11–S22, 2016.
[61] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in Advances in Neural Information Processing Systems 26, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., Curran Associates, Inc., Red Hook, NY, USA, 2013.
[62] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," Journal of Documentation, vol. 28, pp. 11–21, 1972.
[63] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in Proceedings of the 33rd International Conference on International Conference on Machine Learning, pp. 2702–2711, JMLR.org, New York, NY, USA, March 2016.
[64] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—software protection for the masses," in Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection, pp. 3–9, IEEE, Florence, Italy, August 2015.
[65] T. Fawcett, "An introduction to ROC analysis," Pattern Recognition Letters, vol. 27, pp. 861–874, 2006.
[66] P. Velicković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in Proceedings of the International Conference on Learning Representations, Vancouver, Canada, May 2018.
[67] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in Proceedings of the 34th International Conference on Machine Learning, pp. 1263–1272, JMLR.org, Sydney, Australia, August 2017.