

Article

Fus: Combining Semantic and Structural Graph Information for Binary Code Similarity Detection

Yanlin Li ^{1,2} , Taiyan Wang ^{1,2}, Lu Yu ^{1,2} and Zulie Pan ^{1,2,*} ¹ College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China² Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China

* Correspondence: panzulie17@nudt.edu.cn

Abstract

Binary code similarity detection (BCSD) plays an important role in software security. Recent deep learning-based methods have made great progress. Existing methods based on a single feature, such as semantics or graph structure, struggle to handle changes caused by the architecture or compilation environment. Methods fusing semantics and graph structure suffer from insufficient learning of the function, resulting in low accuracy and robustness. To address this issue, we proposed Fus, a method that integrates semantic information from the pseudo-C code and structural features from the Abstract Syntax Tree (AST). The pseudo-C code and AST are robust against compilation and architectural changes and can represent the function well. Our approach consists of three steps. First, we preprocess the assembly code to obtain the pseudo-C code and AST for each function. Second, we employ a Siamese network with CodeBERT models to extract semantic embeddings from the pseudo-C code and Tree-Structured Long Short-Term Memory (Tree LSTM) to encode the AST. Finally, function similarity is computed by summing the respective semantic and structural similarities. The evaluation results show that our method outperforms the state-of-the-art methods in most scenarios. Especially in large-scale scenarios, its performance is remarkable. In the vulnerability search task, Fus achieves the highest recall. It demonstrates the accuracy and robustness of our method.



Academic Editor: Andreas Mauthe

Received: 10 August 2025

Revised: 17 September 2025

Accepted: 23 September 2025

Published: 24 September 2025

Citation: Li, Y.; Wang, T.; Yu, L.; Pan, Z. Fus: Combining Semantic and Structural Graph Information for Binary Code Similarity Detection. *Electronics* **2025**, *14*, 3781. <https://doi.org/10.3390/electronics14193781>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: BCSD; CodeBERT; Tree LSTM; integration; neural network; vulnerability detection

1. Introduction

Binary code similarity detection (BCSD) plays an important role in software security. It is widely used in tasks like 1-day vulnerability discovery [1–16], malware detection [17–19], third-party library detection [20,21], software plagiarism detection [22,23], and patch analysis [24–26]. It determines whether the function under test has a known vulnerability by calculating the similarity between the function under test and the vulnerability function. The success of deep learning in BCSD can be attributed to its powerful representation learning ability, which has been proven effective in capturing complex patterns and relationships in data and learning meaningful representations of assembly code. Despite their improved performance, existing methods have several limitations.

First, the semantic-based approach regards the code of a function as text, focusing only on the code semantics while neglecting the structural information of the function. Semantic-based methods typically take the machine code of a function, assembly code, pseudo-C code, or other textual information as the function features, and use Word2Vec [27], Transformer, or

other Natural Language models for learning, such as SAFE [28], Asm2Vec [29], jTrans [30], and CLAP [31]. They achieved excellent results. However, in scenarios where there are significant differences in semantic information due to variations in the compilation environment or architecture, it will affect the accuracy of the detection results. As shown in Figure 1, the two functions have different names of variables, which leads to a decrease in semantic similarity.

<pre> void __fastcall child_list_free(child_list *list) { child_list *v1; child_list *next; child_list *v3; if (list) { v1 = list; do { next = v1->next; iso_file_source_unref(v1->file); v3 = v1; v1 = next; free(v3); } while (next); } } </pre>	<pre> void __fastcall child_list_free(__int64 a1) { __int64 v1; __int64 v2; void *v3; if (a1) { v1 = a1; do { v2 = *(_QWORD *)(v1 + 8); iso_file_source_unref(*(IsoFileSource **)v1); v3 = (void *)v1; v1 = v2; free(v3); } while (v2); } } </pre>
---	---

Figure 1. An example with a similar structure, but due to different variable identifiers, its semantics seem to have significant differences. Both of these functions are from the “child_list_free” function in the binkit gnu-debug xorriso 1.4.8 binary file. The left one is for the mipseb_64 architecture, and the right one is for the x86_64 architecture.

Second, graph-based methods focus solely on the graph structure information of the function, while ignoring its semantic information. The graph-based methods [8,9,16,32,33] usually use the graph structure information of functions as the feature, such as control flow graphs (CFGs), data flow graphs (DFGs), Abstract Syntax Trees (ASTs), etc. They have performed well in some scenarios. However, when there are differences in the compilation environment or architecture, it may lead to significant structural variations, which will affect the accuracy of the detection results. As shown in Figure 2, the two functions have a reduced structural similarity because of the different implementation methods of the “if-else” statement. A comparison of the control flow graphs of these two functions is shown in Figure 3.

Finally, methods based on semantics and graph structure usually fuse the text semantic features with the graph structure features to conduct detection [34–36].

Existing methods have low robustness in diverse real-world scenarios. In terms of semantics, the existing methods mostly take the assembly code as the function feature, overly focusing on the basic instructions of the underlying implementation, such as stack instructions and register operations, while ignoring the actual semantics of the functions. We have written two functions implemented in the C code ourselves. As shown in Figure 4, the function on the left implements the function of adding two numbers, and the function on the right implements the function of subtracting two numbers. The only difference between the two codes lies in the assembly instruction marked in yellow. The other assembly instructions are highly similar, which introduces a lot of noise when judging the similarity by the semantic model. In terms of graph information representation, CFG is

often used as a feature. In complex scenarios such as cross-architecture, the robustness of the graph structure is insufficient, resulting in a decline in the accuracy of the method. As shown in Figure 3, these are the functions compiled from the same source code for different architectures, and their CFGs have differences. Due to their insufficient learning of semantic and structural features, they have not achieved satisfactory results.

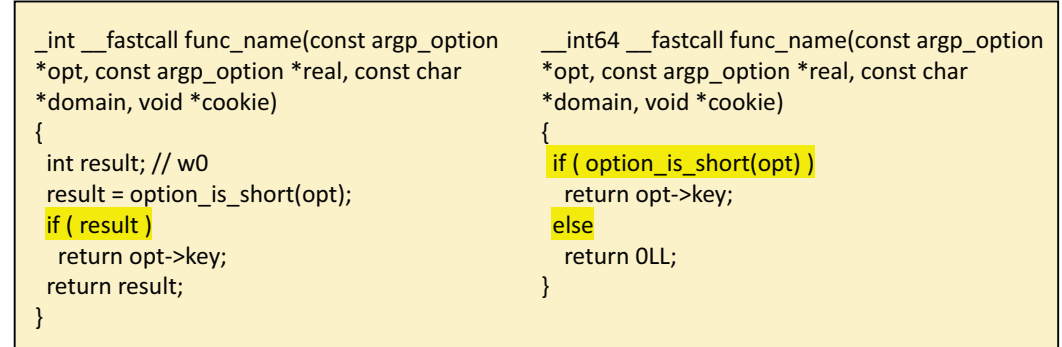


Figure 2. An example where the semantics are highly similar, but the graph structures vary due to different implementation methods of “if else”. Both of these functions are derived from the “until_short” function in the binkit gnu-debug cpio 2.12 binary file. On the left is the arm_64 architecture, and on the right is the mipseb_64 architecture.

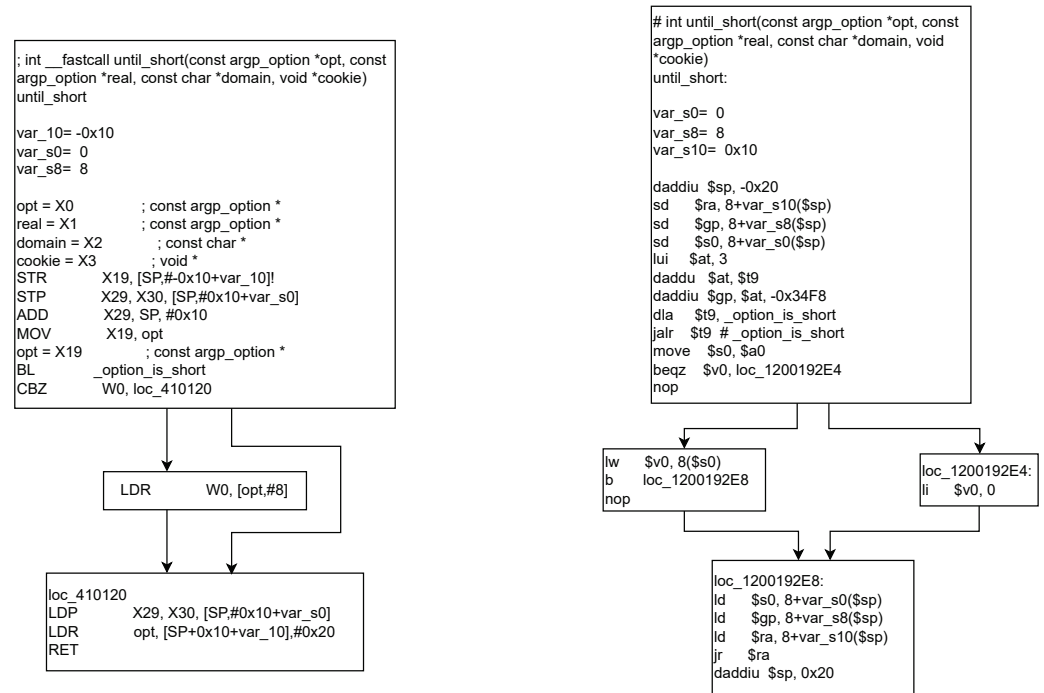


Figure 3. The control flow graphs corresponding to the functions in Figure 2.

To address these limitations, we propose Fus, a method that integrates the semantic information learned from the pseudo-C code and the graph structure information learned from the AST to obtain robust vector representations for functions, thereby enhancing robustness. We adopt an integrated approach, taking into account both semantic and graph structure information, which alleviates the problem of being unable to handle changes in the compilation environment and architecture when using a single feature. We choose the pseudo-C code as the semantic feature of the function and the AST as the structural feature of the function to study the representation of functions fully. The pseudo-C code is obtained through the decompilation process, which filters out the details of underlying instructions and retains sufficient semantic information, providing a cleaner semantic representation

for model learning. As shown in Figure 5, the pseudo-C code has no underlying call stack and instructions for register operations. Within the function body, there are only variable definitions and statements for actually calling the function. When making the similarity judgment, a lot of interference was reduced. As shown in Figure 4, this is the corresponding assembly instruction for Figure 5. There are a large number of call stacks and operation register instructions within the function body, which cause a lot of interference in the similarity judgment of the function. The AST is generated based on the pseudo-C code, and the pseudo-C code is obtained through disassembly and decompilation processes. It is independent of the specific hardware architecture and thus can exhibit better stability across different architectures.

<pre> ; int __fastcall main(int argc, const char **argv, const char **envp) public main main proc near var_C= dword ptr -0Ch var_8= dword ptr -8 var_4= dword ptr -4 ; __unwind { endbr64 push rbp mov rbp, rsp sub rsp, 10h mov [rbp+var_C], 11h mov [rbp+var_8], 9 mov edx, [rbp+var_8] mov eax, [rbp+var_C] mov esi, edx mov edi, eax call sum mov [rbp+var_4], eax mov eax, [rbp+var_4] mov esi, eax lea rax, format ; "Sum: %d\n" mov rdi, rax ; format mov eax, 0 call _printf mov eax, 0 leave retn ; } // starts at 1161 main endp _text ends </pre>	<pre> ; int __fastcall main(int argc, const char **argv, const char **envp) public main main proc near var_C= dword ptr -0Ch var_8= dword ptr -8 var_4= dword ptr -4 ; __unwind { endbr64 push rbp mov rbp, rsp sub rsp, 10h mov [rbp+var_C], 11h mov [rbp+var_8], 9 mov edx, [rbp+var_8] mov eax, [rbp+var_C] mov esi, edx mov edi, eax call jian mov [rbp+var_4], eax mov eax, [rbp+var_4] mov esi, eax lea rax, format ; "Cha: %d\n" mov rdi, rax ; format mov eax, 0 call _printf mov eax, 0 leave retn ; } // starts at 115F main endp _text ends </pre>
--	---

Figure 4. Comparison of assembly instructions for two different functions. It is from our previous work, HLEn.

<pre> int __fastcall main(int argc, const char **argv, const char **envp) { int v4; // [rsp+Ch] [rbp-4h] v4 = sum(17LL, 9LL); printf("Sum: %d\n", v4); return 0; } </pre>	<pre> int __fastcall main(int argc, const char **argv, const char **envp) { int v4; // [rsp+Ch] [rbp-4h] v4 = jian(17LL, 9LL); printf("Cha: %d\n", v4); return 0; } </pre>
--	---

Figure 5. Comparison of pseudo-C codes.

To effectively acquire semantic information, we utilize a Siamese network to integrate two CodeBERT models, allowing them to learn from the pseudo-C code. To effectively acquire the structural information, we train the Tree LSTM in the same manner to learn from the AST. We use these two models to calculate the similarity of the function, and then take the sum of the two similarities as the final function similarity. We select seven scenarios: XA, XO, XC, XA + XO, XA + XC, XO + XC, and XA + XO + XC. Here, XA represents the cross-architecture, XO represents cross-optimization options, and XC represents the cross-compiler. The remaining scenarios are combinations of these three. We use these scenarios to evaluate the robustness of the method in various situations. We construct datasets with sizes of 2, 10, 32, 128, 512, 1000, and 10,000 to evaluate the performance of the

method under different data scales. The results show that Fus outperforms baselines in terms of MRR and Recall@1 in most scenarios. As the size of the function pool increases, the performance of Fus remains very good. We also conduct evaluations in real vulnerability search tasks, and Fus is superior to baseline methods.

In summary, our contributions are as follows:

- We present a novel method named Fus, which integrates semantic information and graph structure information to encode functions. We use the Siamese neural network to train the CodeBERT model to learn the semantic information from the pseudo-C code, and use the same training method to train the Tree LSTM to learn the structure information from the AST. We use the similarity obtained by integrating the two models as the final similarity value of the function.
- We implement a prototype of Fus. For the model training, we construct a dataset across optimization options for the semantic model training and a dataset across architectures for the graph model training. Both of these datasets contain 20,000 pairs of functions. We compare our model against the baselines. The evaluation results show that Fus outperforms all of them in most scenarios.
- We evaluate the vulnerability search application. We collect 140,000 firmware functions as the pool of functions to be tested and 3 CVEs as query functions. Experimental results show that Fus achieves the best recall compared with the baselines.

2. Background

2.1. CodeBERT

CodeBERT is a large-scale bimodal pre-trained model developed by Microsoft Research based on the Transformer architecture. It is specifically designed for joint modeling of Programming Languages (PLs) and Natural Language (NL). It was pre-trained on a massive dataset of code–comment pairs using a hybrid objective function that combines Masked Language Modeling (MLM) and Replaced Token Detection (RTD).

The core strength of this model lies in its ability to simultaneously understand the semantic meaning of a source code alongside its related Natural Language descriptions. Consequently, it has demonstrated state-of-the-art performance on numerous software engineering tasks such as code search, code generation, and code defect detection, establishing itself as a foundational model in the domain of code intelligence.

2.2. Tokenization

Tokenization is a fundamental preprocessing step in both Natural Language Processing (NLP) and code processing. Its purpose is to break down a raw character sequence (such as a text paragraph or a snippet of code) into smaller, meaningful units called tokens. For code processing, tokenization typically employs one of the following two strategies:

- Subword Tokenization: This approach, using algorithms like Byte-Pair Encoding (BPE) or WordPiece, effectively handles the Out-of-Vocabulary (OOV) problem common in both natural and programming languages. It decomposes unknown or rare words into known subword units (e.g., decomposing “unfamiliar” into “un” and “familiar”).
- Rule-Based Tokenization: When processing a source code, language-specific grammatical rules (e.g., for keywords, operators, identifiers) can be used for more precise tokenization. This process is analogous to the lexical analysis stage of a parser.

CodeBERT utilizes the same tokenizer as RoBERTa, which employs the Byte-Pair Encoding (BPE) algorithm, enabling it to process both Natural Language and programming code efficiently.

2.3. Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of a source code. It expresses the grammatical structure of a programming language in a tree format, where each node denotes a construct in the source code (e.g., an expression, a statement, a declaration), as shown in Figure 6.

The AST is a core data structure in compiler design. It is generated after lexical and syntactic analysis, stripping away unnecessary details from the source code (like whitespace, comments, and explicit parentheses) and focusing solely on the structure and content of the code.

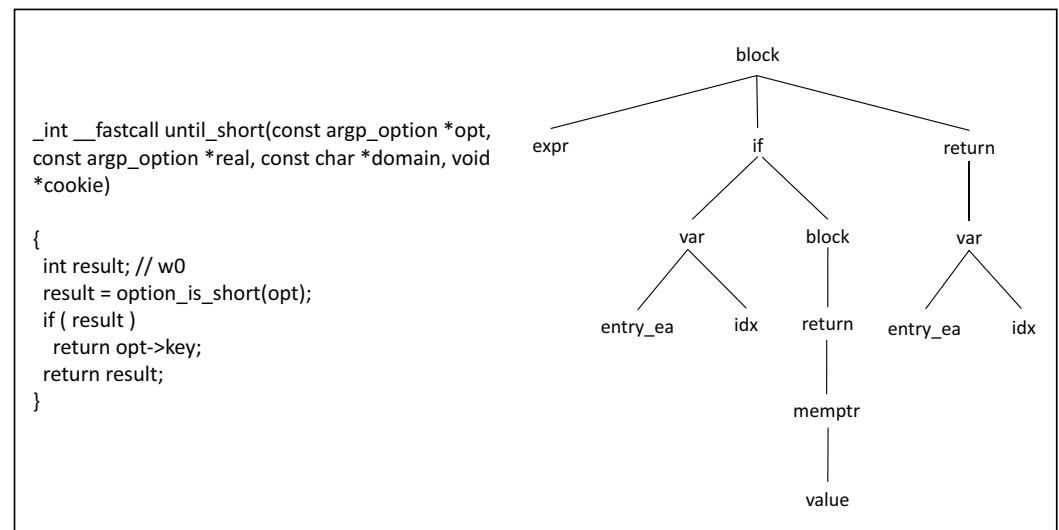


Figure 6. The correspondence between source code and AST. This function is from the “until_short” function in the binkit gnu-debug cpio 2.12 binary file. It is the arm_64 architecture. On the left is the source code, and on the right is the corresponding AST.

3. Related Works

With the development of neural networks, learning-based methods have become mainstream. The existing intelligent BCSD methods mainly fall into three categories: those based on semantics, those based on graphs, and those based on the fusion of semantics and graph structures.

3.1. Semantic-Based

With the development of NLP, the BCSD field has been inspired. It regards the machine code of the function [11], the assembly code [28–30], and the pseudo-C code [37] as text, and uses NLP models such as Word2Vec [27], PV-DM, and Bert [38] to learn vector representations. Finally, it encodes the text into embedding vectors. This method only takes into account the sequence of instructions and the relationships between them, but ignores the actual execution information of the program, such as control flow relationships. It cannot adapt well to the impacts brought about by changes in the compilation environment.

3.2. Graph-Based

With the development of graph neural networks, BCSD methods have begun to treat graph structure information as the features of functions for learning. They usually take the CFG [8], DFG, and AST [39] of the function as features, and use graph neural networks such as structure2vec and Tree-LSTM to learn the vector representations of the graph. This method only takes into account the information of the graph structure, while ignoring the instruction semantics within the basic blocks. Genius [8] and Gemini [16] propose to

extract statistical features manually as the basic blocks of features, but these features are insufficient to capture the semantic information of the instructions.

3.3. Fusion-Based

The existing methods that balance semantic and graph structure can be mainly classified into two categories. One approach is to use different models to calculate the similarity of function pairs, and then combine the similarities to obtain the final similarity result [40]. The other approach is to use different models to represent the function separately, and then fuse the embedded vectors to obtain the final function representation vector [36]. However, they do not fully grasp the semantics and structural features of the functions, which results in poor performance.

Overall, the method that takes into account both semantic and graph structure provides the most comprehensive information and is more robust than using only a single type of information. How to effectively represent semantic information and graph structure information, and how to integrate them well, is a question worth exploring.

4. Methodology

4.1. Overview of the Framework

To address the challenges outlined in Section 1, we proposed Fus, which integrates semantic and graph structure information to encode functions and enhances the robustness of the method. As shown in Figure 7, our method is mainly divided into three steps: data preprocessing, model training, and similarity comparison. First, we perform data preprocessing. We use disassembly tools, IDA Pro [41], to disassemble and decompile the functions, obtaining the pseudo-C code and AST of the functions, which are the semantic information and graph structure information. Second, we conduct model training. We train the CodeBERT model using the Siamese neural network, learning the vector representations of functions from the pseudo-C code. We train the Tree LSTM model in the same way, learning the vector representations of functions from the AST. Finally, we proceed to the stage of similarity calculation. We try two fusion methods: vector concatenation and weighted summation. The experimental results showed that the weighted summation method performed better (Section 6.4). Therefore, in this paper, we choose to use the weighted summation method to calculate the final similarity. We use the trained CodeBERT model and the Tree LSTM model to encode the pseudo-C code and the AST of the function, respectively. Then we calculate the cosine similarity for each of the two models. We add the two similarities together to obtain the final similarity of the function.

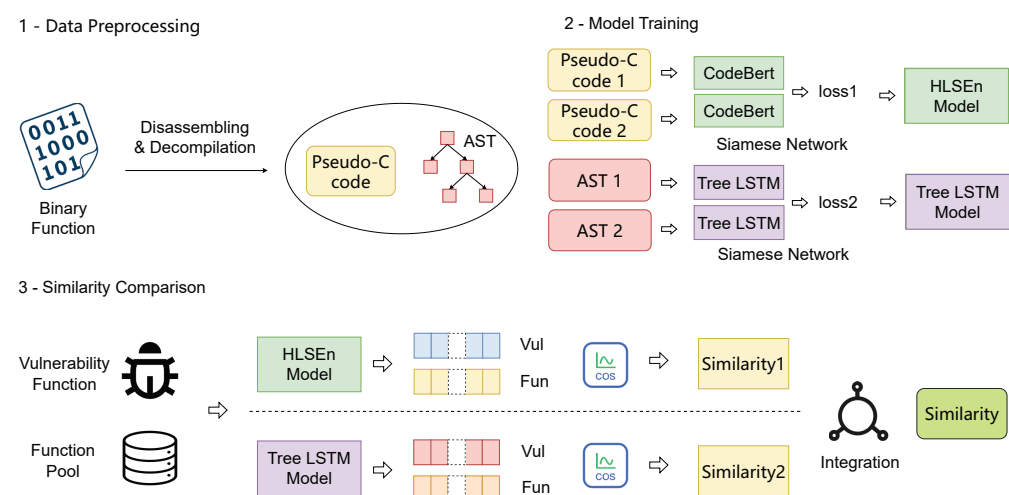


Figure 7. System overview.

4.2. Data Preprocessing

The existing semantic-based methods mostly take the assembly code as the feature of functions. The assembly code contains many low-level instructions for implementing functions. When the NLP model is learning, it pays excessive attention to these instructions. However, these instructions do not represent the true semantics of the function and can lead to incorrect model predictions. Our previous work, HLSEn, demonstrates that the pseudo-C code can effectively represent the semantics of functions. We choose to use the pseudo-C code as the characteristic of the function.

The existing graph-based methods, including those based on control flow graphs or those combining data flow and control flow graphs, have achieved very good results. However, [5] demonstrates that the CFG is greatly influenced by different architectures. Asteria [39] believes that the AST can demonstrate better stability in different architectures. This is because the AST is generated based on the pseudo-C code. The pseudo-C code is obtained by disassembling and decompiling the assembly instructions, and it is independent of the structure. Therefore, when encoding the graph information of the function, we choose to use the AST as the feature of the function.

During the data preprocessing stage, we use a disassembler tool to extract the pseudo-C code and AST of all functions from the binary file.

4.3. Model Training

4.3.1. Semantic-Based

This work incorporates the semantic features from our prior model, HLSEn, which was designed to measure code similarity by learning a semantic-aware representation of functions. HLSEn uses the pseudo-C code as the semantic feature of functions and trains the CodeBERT model using a Siamese network. This approach has achieved very good results. The pseudo-C code of the function will contain the function name. To prevent the model from paying more attention to the function names, we replace them with “func_name”. CodeBERT is a dual-modal extended version of Bert, and it has demonstrated an extremely high level of performance in understanding the source code. In the current work, we directly utilize the high-quality semantic embeddings model, HLSEn, for fusion.

4.3.2. Graph-Based

RNN has been widely applied due to its ability to handle sequences of any length. However, during the training process of the Recurrent Neural Network (RNN), there are problems of gradient disappearance and explosion, which make it difficult for it to learn long-term dependencies. Therefore, Long Short-Term Memory (LSTM) was developed. LSTM can handle sequential information, but it is unable to process data with a tree structure. To address the issue of training with tree-structured data as input, the Tree-Structured LSTM is proposed. The nodes of the Tree LSTM are the indexes corresponding to opname. Based on the statistics, we assign opname the numbers ranging from 0 to 79, as shown in Table 1. Based on this table, we convert opname into corresponding numbers. If opname is not in the table, we will uniformly convert it to “80”.

Table 1. The corresponding table for converting opname to index.

Index	Opname	Index	Opname
0	idx	40	shl
1	var	41	bor
2	entry_ea	42	sge
3	expr	43	for

Table 1. Cont.

Index	Opname	Index	Opname
4	asg	44	fnum
5	value	45	sgt
6	num	46	memref
7	block	47	offset
8	if	48	sle
9	ptr	49	switch
10	cast	50	postinc
11	obj	51	udiv
12	string	52	empty
13	add	53	fmul
14	return	54	predec
15	goto	55	comma
16	eq	56	umod
17	lnot	57	asgsub
18	ne	58	bnot
19	ref	59	fadd
20	sub	60	asgmul
21	case	61	asgbor
22	call	62	continue
23	xor	63	asgxor
24	band	64	neg
25	preinc	65	fsub
26	memptr	66	tern
27	ult	67	sshr
28	break	68	asgshl
29	land	69	asgband
30	mul	70	fdiv
31	asgadd	71	helper
32	ushr	72	asgumod
33	ugt	73	asgushr
34	lor	74	asm
35	while	75	sizeof
36	slt	76	postdec
37	do	77	asgudiv
38	uge	78	fneg
39	ule	79	str

4.3.3. Siamese Network

After encoding in both semantic-based and graph-based aspects, the Siamese network is used for training to obtain the semantic encoding model and the graph encoding model, respectively. The input of the Siamese neural network is $\langle V_1, V_2, \text{and Label} \rangle$, and it measures the similarity of the two inputs by comparing the distance between the two input feature vectors. V_1 and V_2 are vectors representing the pseudo-C code or AST encoding of the model. The label represents whether the two input functions are similar. If Label is 1, it indicates that these two functions are similar. If Label is 0, it indicates that these two functions are not similar. The two sub-networks integrate into the Siamese network share weights during training. This training method enables two similar functions to be closer in the vector space, while dissimilar functions are farther apart in the vector space. As shown in Equations (1) and (2), we use the cosine distance as the distance between two vectors. When Label is 1, it indicates that the function is a positive sample. At this time, the value of the loss function is the square of the cosine distance between the two vectors, $\cos\text{Dist}^2(V_1, V_2)$. The smaller the cosine distance of the positive sample, the smaller the

loss. When Label is 0, it indicates that the function is a negative sample. At this point, if the cosine distance is greater than the Margin, the loss is 0. When the cosine distance is less than the Margin, the value of the loss is the difference between the Margin and the cosine distance. The smaller the difference, the smaller the loss, and at this time, the cosine distance is larger. The Margin represents the minimum distance between two negative samples. This loss can make the cosine distance of positive samples smaller, while the cosine distance of negative samples becomes larger, and is greater than the Margin. Since the range of the cosine distance is $[0, 2]$, in this study, we set $\text{Margin} = 1$. This value can effectively separate positive and negative samples and achieve a balance between training stability and performance.

$$\text{cosDist}(V_1, V_2) = 1 - \frac{V_1 \cdot V_2}{\|V_1\| \|V_2\|} \quad (1)$$

$$\text{Loss} = \text{label} \cdot \text{cosDist}^2(V_1, V_2) + (1 - \text{label}) \cdot \max(\text{Margin} - \text{cosDist}(V_1, V_2), 0)^2 \quad (2)$$

4.4. Similarity Comparison

There are three main methods for calculating code similarity based on distance, matching algorithms, and multiclass performance metrics. Methods based on distance include entropy (such as Shannon entropy, behavioral entropy, and string entropy), cosine, longest common subsequence, edit, Euclidean, fingerprint comparison, K-L divergence, Hamming distance, Gaussian kernel, and distance between n-grams and context. Methods based on matching algorithms include function-name matching and string matching. Methods based on multiclass performance metrics mainly include precision, recall, and F-measure (F1-score) [42]. Cosine similarity not only has high computational efficiency and can quickly handle large-scale comparison tasks between high-dimensional feature vectors, but also its computational results are naturally normalized within the $[-1, 1]$ interval, providing a unified and interpretable scale for setting similarity thresholds and classification decisions. We also referred to the excellent ideas in previous papers and ultimately chose cosine similarity as the method for measuring similarity [9,16,28–30,36,43,44].

We encode the functions respectively using the trained CodeBERT model and the Tree LSTM model. A function can obtain vector encodings in both semantic and graphical aspects. The integration method we employ is adding similarities. We use the trained CodeBERT model to vector-encode the functions, and then calculate the similarity between two functions using cosine similarity. We also use the trained Tree LSTM model to calculate the similarity between the two functions. Finally, we add the similarities together to obtain the final similarity between the two functions. The larger the cosine similarity value is, the higher the degree of similarity is. Conversely, the smaller the value is, the lower the degree of similarity is.

We separately calculate the similarity between the vulnerability function and the target function, and then rank the target function according to the similarity. We believe that the higher the ranking of a function, the greater the probability that it is a vulnerability function.

5. Experimental Setup

5.1. Experiment Settings

When training CodeBERT, we set batch size = 16, epoch = 100, learning rate = 5×10^{-5} , and margin = 1.0. When training the Tree LSTM, we set the batch size = 1024, the number of epochs = 100, the learning rate = 2.4×10^{-3} , and the margin = 1.0.

5.2. Datasets

For these three datasets, Dataset-1, Dataset-2, and Dataset-3, we obtain the function pairs that meet the conditions in a random manner from binary files. The four datasets we constructed are publicly available in <https://github.com/Geyuxiyu2/Fus> accessed on 9 September 2025. Dataset-1 and Dataset-2 are from Trex [45], while Dataset-3 is from binkit [46]. We have checked the datasets to ensure that the datasets used for training, Dataset-1 and Dataset-2, and that used for evaluation, Dataset-3, are not duplicated.

Dataset-1. This dataset is used for training CodeBERT. It is sourced from the Trex [45] project. This dataset covers multiple common software projects, such as ImageMagick, gmp, sqlite, zlib, Putty, and popular tool suites like binutils, coreutils, diffutils, and findutils. The dataset includes various instruction set architectures, including X86-32, X86-64, MIPS-32, MIPS-64, ARM-32, and ARM-64 versions, and also includes multiple compilation optimization options such as O0, O1, O2, and O3. We constructed this dataset in a random manner. It consists of 20,000 pairs of functions, among which 10,000 are positive sample pairs and 10,000 are negative sample pairs. The dataset consists of pairs of functions of XO.

Dataset-2. This dataset is used for training the Tree LSTM. It is also sourced from the Trex [45] project. It consists of 20,000 pairs of functions, among which 10,000 are positive sample pairs and 10,000 are negative sample pairs. The dataset is composed of function pairs of XA and XA + XO.

Dataset-3. This dataset is used to evaluate our method, Fus. The binary files come from binkit [46]. This dataset supports multiple architectures such as X86-32, X86-64, MIPS-32, MIPS-64, ARM-32, and ARM-64, and the optimization options cover O0, O1, O2, and O3. It includes seven scenarios: XA, XO, XC, XA + XO, XA + XC, XO + XC, and XA + XO + XC. There are 10,000 pairs of functions for each scenario.

Dataset-4. This is used to evaluate the performance of Fus in real vulnerability search scenarios. As shown in Table 2, we select three vulnerability functions as the query functions for the search. Each vulnerability function is compiled into different architectures (ARM, X86-64) and compilation optimization options (O0, O1, O2, O3, Os), resulting in a total of 10 variants. We respectively take one variant of a CVE as the query function and the other nine as the truth value functions, and combine them with Dataset-3 as the functions to be tested.

Table 2. Information of vulnerability function.

CVE	Binary	Version	Function
CVE-2023-42363	busybox	v1.36.1	xasprintf()
CVE-2021-42382	busybox	v1.34.0	getvar s()
CVE-2021-42381	busybox	v1.34.0	hash_init()

5.3. Baselines

- **SAFE [28].** Safe leverages word2vec [27] for the automated embedding of assembly instructions by passing traditional feature extraction methods.
- **GMN [32].** GMN employs graph embedding techniques to encode functions into vector representations.
- **CLAP [31].** The method utilizes LLM-generated training examples that map assembly code to Natural Language, optimizing vector embeddings for functional representation.

5.4. Evaluation Metrics

Let $\mathcal{F} = \{\phi_1, \phi_2, \dots, \phi_n\}$ be a collection of binary functions with their associated ground-truth mappings $\mathcal{G} = \{\phi_1^*, \phi_2^*, \dots, \phi_n^*\}$. For a query function $\phi_i \in \mathcal{F}$, the binary similarity analysis task involves identifying the top K most analogous functions from

$\mathcal{G}$, ordered by their similarity measures. The rank of the ground-truth match ϕ_i^* in the retrieved results is denoted as $\text{Rank}(\phi_i^*)$.

We define an indicator variable $\mathbb{I}_i^{(K)}$ to represent whether the correct correspondence appears within the top K results for the i -th query, as follows:

$$\mathbb{I}_i^{(K)} = \begin{cases} 1 & \text{if } \text{Rank}(\phi_i^*) \leq K \\ 0 & \text{otherwise} \end{cases}$$

The retrieval accuracy is then evaluated using the following two metrics:

$$\text{Recall@K} = \frac{1}{n} \sum_{i=1}^n \mathbb{I}_i^{(K)}$$

$$\text{MRR} = \frac{1}{n} \sum_{i=1}^n \frac{1}{\text{Rank}(\phi_i^*)}$$

Recall@K measures the proportion of correct matches of the target appearing in the first K positions of the retrieval results among all queries. This metric directly reflects the ability of the method to successfully recall the true match within the first K results. MRR (Mean Reciprocal Ranking) calculates the average of the reciprocals of the rankings of the true matching targets for all queries. This metric is more sensitive to the ranking positions and can better reflect the overall ability of the method to place the correct match in the most prominent positions.

6. Evaluation

Our evaluation aims to answer the following questions.

RQ1. How does the performance of Fus compare with the baselines?

RQ2. How do Fus and baseline perform at different function pool sizes?

RQ3. How does Fus perform in real vulnerability detection?

RQ4. How effective is the integration strategy?

6.1. BCSD Performance (RQ1)

We randomly selected 32 pairs of functions and 10,000 pairs of functions from each scenario of XA, XO, XC, XA + XO, XA + XC, XO + XC, and XA + XO + XC in Dataset-3 as evaluation datasets. This enables our method and the baselines to be evaluated at different levels of difficulty.

Small Pool (Pool Size = 32). As shown in Tables 3 and 4, Fus has the highest MRR and Recall@1 in all seven scenarios compared with the baselines. This proves that Fus can adapt to various complex environments. Compared with our individual method, HLSEn and Tree LSTM, in the XC scenario, Fus are slightly lower than HLSEn. This is an acceptable result.

Large Pool (Pool Size = 1000). As shown in Tables 5 and 6, compared with the baselines, Fus achieves the highest MRR and Recall@1 in all seven scenarios. It proves that Fus can also adapt to various complex environments in large-scale scenarios. Compared with our single method HLSEn and Tree LSTM, Fus achieves the best results in terms of MRR and Recall@1. It proves that our integrated method has higher accuracy and recall than individual methods.

The experiments in this section can prove that Fus has high accuracy on both small-scale test sets and large-scale test sets. Moreover, its outstanding performance in various scenarios also demonstrates that Fus can adapt to various complex situations.

Table 3. The MRR of different BCSD methods (pool size = 32). The bold numbers represent the maximum values for each column.

Method	XA	XO	XC	XA + XO	XA + XC	XO + XC	XA + XO + XC
SAFE	0.249	0.605	0.725	0.075	0.196	0.529	0.137
GMN	0.373	0.559	0.905	0.249	0.495	0.403	0.286
CLAP	0.141	0.731	0.908	0.066	0.166	0.738	0.129
HLSEn	0.779	0.914	0.969	0.828	0.906	0.801	0.839
Tree LSTM	0.740	0.764	0.816	0.590	0.793	0.705	0.681
Fus	0.893	0.927	0.940	0.855	0.984	0.864	0.849

Table 4. The Recall@1 of different BCSD methods (pool size = 32). The bold numbers represent the maximum values for each column.

Method	XA	XO	XC	XA + XO	XA + XC	XO + XC	XA + XO + XC
SAFE	0.156	0.469	0.625	0.000	0.094	0.375	0.031
GMN	0.219	0.438	0.844	0.062	0.375	0.219	0.156
CLAP	0.031	0.594	0.875	0.000	0.031	0.594	0.031
HLSEn	0.656	0.844	0.938	0.750	0.844	0.719	0.781
Tree LSTM	0.656	0.656	0.750	0.469	0.719	0.625	0.562
Fus	0.844	0.875	0.906	0.781	0.969	0.812	0.781

Table 5. The MRR of different BCSD methods (pool size = 10,000). The bold numbers represent the maximum values for each column.

Method	XA	XO	XC	XA + XO	XA + XC	XO + XC	XA + XO + XC
SAFE	0.023	0.152	0.321	0.001	0.010	0.191	0.002
GMN	0.068	0.180	0.461	0.012	0.035	0.145	0.011
CLAP	0.014	0.271	0.567	0.001	0.005	0.187	0.002
HLSEn	0.374	0.428	0.641	0.347	0.367	0.402	0.260
Tree LSTM	0.376	0.280	0.509	0.204	0.267	0.238	0.169
Fus	0.508	0.459	0.671	0.389	0.430	0.415	0.332

Table 6. The Recall@1 of different BCSD methods (pool size = 10,000). The bold numbers represent the maximum values for each column.

Method	XA	XO	XC	XA + XO	XA + XC	XO + XC	XA + XO + XC
SAFE	0.014	0.125	0.273	0.000	0.006	0.156	0.001
GMN	0.047	0.148	0.410	0.005	0.022	0.120	0.005
CLAP	0.010	0.217	0.508	0.001	0.003	0.143	0.001
HLSEn	0.307	0.364	0.591	0.278	0.301	0.336	0.189
Tree LSTM	0.307	0.226	0.444	0.149	0.210	0.189	0.118
Fus	0.431	0.389	0.617	0.316	0.356	0.348	0.252

6.2. The Effects of Pool Size on Performance (RQ2)

The previous section demonstrates that the size of the function pool can affect the accuracy of the method. To further explore the impact of the size of the function pool on the accuracy of the method, we construct function pools of sizes 2, 10, 32, 128, 512, 1000, and 10,000, and evaluate the method.

The results are shown in Figure 8. In most scenarios, as the pool size increases, Fus outperforms all other methods. It indicates that our method can be adapted to different testing scales and will be more robust in practical applications. As the size of the function pool increases, the MRR of other methods drops sharply, while our method experiences a slower decline. It also demonstrates the superiority of our method.

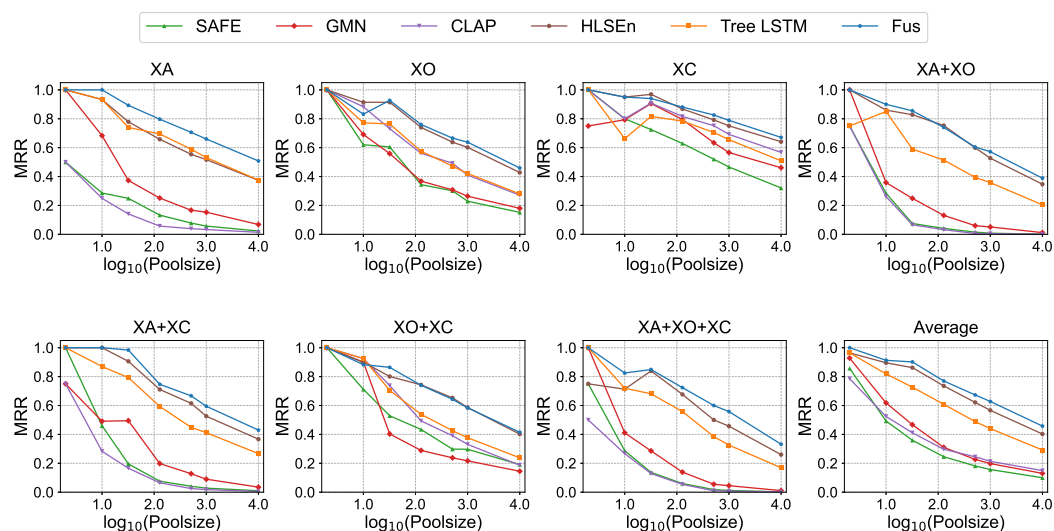


Figure 8. The performance of different BCSD methods in different function pool sizes.

6.3. Real-World Vulnerability Search (RQ3)

The detection of known vulnerabilities is an important application scenario of BCSD. We use Dataset-4 as the evaluation dataset. Each time, one variant of a CVE is selected as the query function, and the other nine variants of that CVE are used as truth functions. The other CVEs, together with 70,000 pairs of functions from Dataset-3, form the function pool. Therefore, our function pool contains 140,029 functions. This is a large-scale function pool that is extremely challenging.

When each CVE is used as a query function, the other nine variants are truth functions, so we use Recall@9 as the metric to evaluate the recall of each query. We take the average value of the Recall@9 obtained from 10 queries of a specific CVE as the Ave Recall@9 for that CVE. We calculated the Ave Recall@9 for each CVE, as shown in Figure 9. Our method, Fus, outperforms baselines and achieves the highest Ave Recall@9. Compared with the individual methods of HLSEn and Tree LSTM, our integrated method performs best, indicating the effectiveness of our approach.

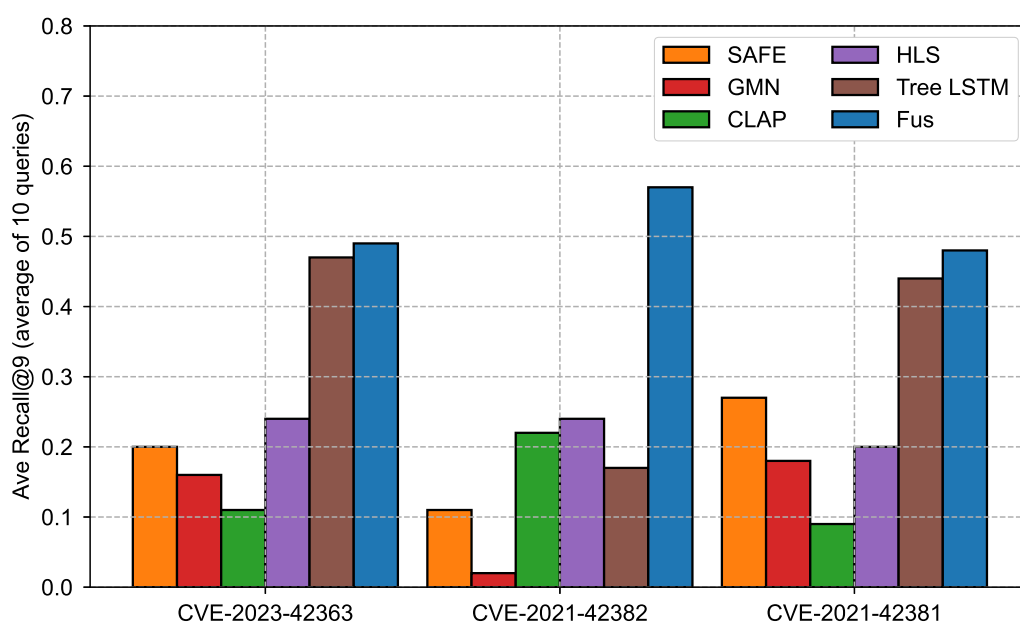


Figure 9. Recall@9 of real-world vulnerability search.

6.4. The Effectiveness of the Integration Method (RQ4)

In this section, we will conduct ablation experiments on the integrated method. There are many ways to integrate the semantic-based method HLSEn and the graph structure information-based method Tree LSTM. In this article, the proposed Fus fusion method is to add the similarity calculated by HLSEn and Tree LSTM as the final similarity. There is another integration method, which we name as Concat. This method involves vector encoding of the HLSEn and Tree LSTM functions separately, and then concatenating the vectors to form the final embedding vector of the function. The cosine similarity of the embedded vectors is used as the similarity between functions.

We conduct a comparative evaluation of the two integration methods, namely, Fus and Concat. In the function pool with pool size set to 10,000, we calculate the MRR and Recall@1 for both methods. The experimental results are shown in Figure 10. In various scenarios, the MRR and Recall@1 metrics of Fus are superior to those of Concat. This demonstrates the effectiveness of our integration method.

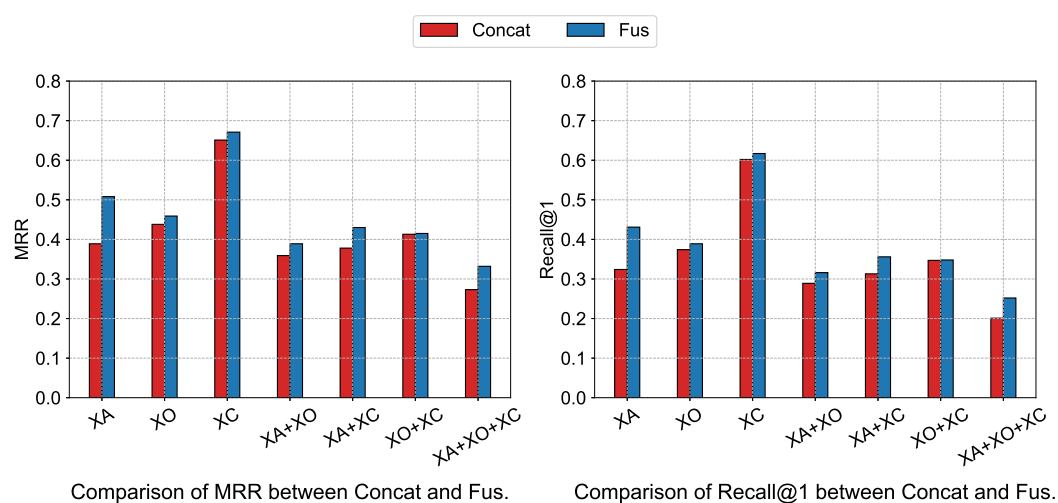


Figure 10. Comparison of MRR and Recall@1 between Concat and Fus (pool size = 10,000).

7. Conclusions

In this paper, we propose Fus, which integrates semantic and graph structure information to encode functions. We adopt an integrated approach, not relying on the unilateral information of the function, which enhances the accuracy and practicability of the method. The experimental results show that our method outperforms baselines in most of the BCSD tasks. Our integrated method, Fus, is also superior to the single methods based on semantics and graph structure. This indicates that the integrated method is superior to the single method, with higher accuracy and greater stability. Meanwhile, our single method HLSEn and Tree LSTM, in comparison with the baselines, also demonstrate excellent performance. It indicates that the pseudo-C code and AST have excellent stability and are excellent features for function representation. However, the integration method in this paper is static and fixed. It lacks the ability for dynamic decision making and self-optimization. It cannot perform multi-angle reasoning, verification, and iterative analysis like security experts, which limits the automation level and scalability of vulnerability detection. With the development of large language models, we will explore more intelligent integration methods to enhance performance in the future. For instance, we can design a collaborative decision-making mechanism based on multi-agent systems. Four types of role-based intelligent agents, namely, “analysis–comparison–reasoning–verification”, are introduced. We can assess the similarity of functions from multiple perspectives together. Each intelligent agent independently assesses based on its own capabilities, collaborates with others, and

dynamically adjusts its judgment strategies through a feedback mechanism. It simulates the multi-angle reasoning and knowledge integration in the human analysis process to achieve a more flexible and intelligent vulnerability detection decision-making process.

Author Contributions: Conceptualization and methodology, Y.L. and T.W.; validation, Y.L. and L.Y.; formal analysis, Y.L., Z.P., and T.W.; investigation, Y.L., Z.P., and T.W.; resources, data curation, and visualization, Y.L., T.W., and L.Y.; supervision, project administration, and funding acquisition, Z.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original data presented in this study are openly available at <https://github.com/Geyuxiyu2/Fus>, accessed on 9 September 2025.

Acknowledgments: The authors thank the support from the College of Electronic Engineering, National University of Defense Technology.

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

Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract Syntax Tree
BCSD	binary code similarity detection
CFG	control flow graph
DFG	data flow graph
CVEs	Common Vulnerabilities and Exposures

References

1. Ahn, S.; Ahn, S.; Koo, H.; Paek, Y. Practical binary code similarity detection with bert-based transferable similarity learning. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; pp. 361–374.
2. David, Y.; Partush, N.; Yahav, E. Statistical similarity of binaries. *ACM SIGPLAN Not.* **2016**, *51*, 266–280. [CrossRef]
3. David, Y.; Partush, N.; Yahav, E. Similarity of binaries through re-optimization. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain, 18–23 June 2017; pp. 79–94.
4. David, Y.; Partush, N.; Yahav, E. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Not.* **2018**, *53*, 392–404. [CrossRef]
5. David, Y.; Yahav, E. Tracelet-based code search in executables. *ACM SIGPLAN Not.* **2014**, *49*, 349–360. [CrossRef]
6. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. Discovre: Efficient cross-architecture identification of bugs in binary code. In Proceedings of the Ndss, San Diego, CA, USA, 21–24 February 2016; Volume 52, pp. 58–79.
7. Feng, Q.; Wang, M.; Zhang, M.; Zhou, R.; Henderson, A.; Yin, H. Extracting conditional formulas for cross-platform bug search. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 346–359.
8. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
9. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 896–899.
10. Huang, H.; Youssef, A.M.; Debbabi, M. Binsequence: Fast, accurate and scalable binary code reuse detection. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 155–166.
11. Liu, B.; Huo, W.; Zhang, C.; Li, W.; Li, F.; Piao, A.; Zou, W. α diff: Cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 667–678.
12. 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 Proceedings of the NDSS, San Diego, CA, USA, 27 February–3 March 2023.

13. Marcelli, A.; Graziano, M.; Ugarte-Pedrero, X.; Fratantonio, Y.; Mansouri, M.; Balzarotti, D. How machine learning is solving the binary function similarity problem. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 2099–2116.
14. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724.
15. Shirani, P.; Collard, L.; Agba, B.L.; Lebel, B.; Debbabi, M.; Wang, L.; Hanna, A. B in a rm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, 28–29 June 2018; Proceedings 15; Springer: Berlin/Heidelberg, Germany, 2018; pp. 114–138.
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, Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376.
17. Cesare, S.; Xiang, Y.; Zhou, W. Control flow-based malware variant detection. *IEEE Trans. Dependable Secur. Comput.* **2013**, *11*, 307–317. [\[CrossRef\]](#)
18. Hu, X.; Shin, K.G.; Bhatkar, S.; Griffin, K. MutantX-S: Scalable malware clustering based on static features. In Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), San Jose, CA, USA, 26–28 June 2013; pp. 187–198.
19. Kim, T.; Lee, Y.R.; Kang, B.; Im, E.G. Binary executable file similarity calculation using function matching. *J. Supercomput.* **2019**, *75*, 607–622. [\[CrossRef\]](#)
20. Tang, W.; Luo, P.; Fu, J.; Zhang, D. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 104–115.
21. Zhu, X.; Wang, J.; Fang, Z.; Yin, X.; Liu, S. BBDetector: A precise and scalable third-party library detection in binary executables with fine-grained function-level features. *Appl. Sci.* **2022**, *13*, 413. [\[CrossRef\]](#)
22. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 389–400.
23. 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 Trans. Softw. Eng.* **2017**, *43*, 1157–1177. [\[CrossRef\]](#)
24. Hu, Y.; Zhang, Y.; Li, J.; Gu, D. Cross-architecture binary semantics understanding via similar code comparison. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 57–67.
25. Kargén, U.; Shahmehri, N. Towards robust instruction-level trace alignment of binary code. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 342–352.
26. Xu, Z.; Chen, B.; Chandramohan, M.; Liu, Y.; Song, F. Spain: Security patch analysis for binaries towards understanding the pain and pills. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 462–472.
27. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781. [\[CrossRef\]](#)
28. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Baldoni, R.; Querzoni, L. Safe: Self-attentive function embeddings for binary similarity. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, 19–20 June 2019; Proceedings 16; Springer: Berlin/Heidelberg, Germany, 2019; pp. 309–329.
29. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 472–489.
30. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jtrans: Jump-aware transformer for binary code similarity. *arXiv* **2022**, arXiv:2205.12713.
31. Wang, H.; Gao, Z.; Zhang, C.; Sha, Z.; Sun, M.; Zhou, Y.; Zhu, W.; Sun, W.; Qiu, H.; Xiao, X. CLAP: Learning transferable binary code representations with natural language supervision. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Vienna, Austria, 16–20 September 2024; pp. 503–515.
32. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In Proceedings of the International Conference on Machine Learning, PMLR, Long Beach, CA, USA, 9–15 June 2019; pp. 3835–3845.
33. Yang, S.; Dong, C.; Xiao, Y.; Cheng, Y.; Shi, Z.; Li, Z.; Sun, L. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 1–40. [\[CrossRef\]](#)

34. Massarelli, L.; Di Luna, G.A.; Petroni, F.; Querzoni, L.; Baldoni, R. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In Proceedings of the 2nd Workshop on Binary Analysis Research (BAR), San Diego, CA, USA, 24 February 2019; pp. 1–11.
35. Duan, Y.; Li, X.; Wang, J.; Yin, H. Deepbindiff: Learning Program-Wide Code Representations for Binary Diffing 2020. Available online: <https://www.ndss-symposium.org/ndss-paper/deepbindiff-learning-program-wide-code-representations-for-binary-diffing/> (accessed on 22 September 2025).
36. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1145–1152.
37. Liu, Z.; Tang, Q.; Nie, S.; Wu, S.; Zhang, L.F.; Tang, Y. KEENHash: Hashing Programs into Function-Aware Embeddings for Large-Scale Binary Code Similarity Analysis. *Proc. ACM Softw. Eng.* **2025**, *2*, 801–824. [\[CrossRef\]](#)
38. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*; Volume 1 (Long and Short Papers); Association for Computational Linguistics: Minneapolis, MN, USA, 2019; pp. 4171–4186.
39. Yang, S.; Cheng, L.; Zeng, Y.; Lang, Z.; Zhu, H.; Shi, Z. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, 21–24 June 2021; pp. 224–236.
40. Alrabaee, S.; Shirani, P.; Wang, L.; Debbabi, M. Fossil: A resilient and efficient system for identifying foss functions in malware binaries. *ACM Trans. Priv. Secur. TOPS* **2018**, *21*, 1–34. [\[CrossRef\]](#)
41. Ida Pro. Available online: <https://hex-rays.com/ida-pro> (accessed on 9 September 2025).
42. Ebad, S.A.; Darem, A.A.; Abawajy, J.H. Measuring Software Obfuscation Quality—A Systematic Literature Review. *IEEE Access* **2021**, *9*, 99024–99038. [\[CrossRef\]](#)
43. He, H.; Lin, X.; Weng, Z.; Zhao, R.; Gan, S.; Chen, L.; Ji, Y.; Wang, J.; Xue, Z. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 1759–1776.
44. Jiang, L.; An, J.; Huang, H.; Tang, Q.; Nie, S.; Wu, S.; Zhang, Y. Binaryai: Binary software composition analysis via intelligent binary source code matching. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13.
45. Pei, K.; Xuan, Z.; Yang, J.; Jana, S.; Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv* **2020**, arXiv:2012.08680.
46. Kim, D.; Kim, E.; Cha, S.K.; Son, S.; Kim, Y. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Trans. Softw. Eng.* **2022**, *49*, 1661–1682. [\[CrossRef\]](#)

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