computers

MDPI

Article

Distributed Representation for Assembly Code

Kazuki Yoshida *, Kaiyu Suzuki and Tomofumi Matsuzawa *

Department of Information Sciences, Tokyo University of Science, Yamazaki, Chiba 278-8510, Japan
* Correspondence: yoshida9134@gmail.com (K.Y.); t-matsu@is.noda.tus.ac.jp (T.M.)

Abstract: In recent years, the number of similar software products with many common parts has been increasing due to the reuse and plagiarism of source code in the software development process. Pattern matching, which is an existing method for detecting similarity, cannot detect the similarities between these software products and other programs. It is necessary, for example, to detect similarities based on commonalities in both functionality and control structures. At the same time, detailed software analysis requires manual reverse engineering. Therefore, technologies that automatically identify similarities among the arge amounts of code present in software products in advance can reduce these oads. In this paper, we propose a representation earning model to extract feature expressions from assembly code obtained by statically analyzing such code to determine the similarity between software products. We use assembly code to eliminate the dependence on the existence of source code or differences in development anguage. The proposed approach makes use of Asm2Vec, an existing method, that is capable of generating a vector representation that captures the semantics of assembly code. The proposed method also incorporates information on the program control structure. The control structure can be represented by graph data. Thus, we use graph embedding, a graph vector representation method, to generate a representation vector that reflects both the semantics and the control structure of the assembly code. In our experiments, we generated expression vectors from multiple programs and used clustering to verify the accuracy of the approach in classifying similar programs into the same cluster. The proposed method outperforms existing methods that only consider semantics in both accuracy and execution time.

Keywords: representation earning; graph data; classification problems; software; security; machine earning

Citation: Yoshida, K.; Suzuki, K.; Matsuzawa, T. Distributed Representation for Assembly Code. *Computers* **2023**, *12*, 222. https://doi.org/10.3390/computers12110222

Academic Editor: Paolo Bellavista

Received: 11 August 2023
Revised: 25 October 2023
Accepted: 25 October 2023
Published: 1 November 2023

1. Introduction

In recent years, technologies for determining software similarity have been attracting attention. This is due to the increase in the number of variants of malware that have been modified from existing malware and the increase in the number of similar software products that share common functions and control structures through the reuse and theft of source code [1]. These variants of malware and similar software products cannot be detected by existing matching methods. Therefore, it is necessary to conduct reverse engineering analysis to identify the similarities in function and control structure. However, reverse engineering requires manual work, and dynamic analysis of actual behavior requires an analysis environment such as a sandbox [2]. Therefore, by automatically identifying similarities between software products from static analysis results in advance, the software products that need to be reverse engineered can be selected and the manual workload can be greatly reduced.

There are two approaches to detecting similarities between different programs, one is targeting source code and the other is targeting binaries. In source-code-targeted approaches, code clones, which are pieces of code that match or are similar to each other in the source code, are used [3,4]. Methods for detecting code clones include detecting matching or similar sequences of characters or ines [5,6], matching or similar sequences of tokens [7,8], and matching or similar abstract syntax trees [9]. However, this approach requires source code, and software source code is not always available. It may also be dependent on coding style and the development anguage.

Binary-targeted approaches include binary difference comparison and similarity digests. The binary difference comparison approach extracts and compares the common parts or differences between two input binaries. For example, the ength of the ongest common segment (LCS [10]) of an executable code sequence can be determined using a method that graphs the control structure of a program and compares these features [11]. However, while binary difference comparison can reduce the number of analysis targets by extracting differences between two binaries, it is unsuitable for comparing arge amounts of data at the same time due to its characteristics. A similarity digest is a data digest that preserves the similarity of the input; typical methods include ssdeep [12] and sdhash [13]. Since ssdeep generates digests while dividing the input string, partial comparisons between inputs are possible. Meanwhile, sdhash can compare the features it retains with a Bloom filter that stores the features extracted from the input string. However, while similarity digests can handle arge amounts of data simultaneously, their output is often a variable-length string. For ease of comparison and applicability to other tasks, the output should be a fixed-length vector representation.

In this study, as a method for detecting similarities between different software products and programs, we use a distributed representation of assembly code obtained by statically analyzing the executable code of software products and programs. Distributed representation is a method for embedding discrete objects, such as inguistic symbols, into a vector space and representing them as real-valued vectors; typical methods include Word2Vec [14] for words, Doc2Vec [15] for sentences, and Code2Vec for snippets of program code [16]. In the proposed method, the assembly code and the control flow graph (CFG) (Figure 1), which represents the control structure of the program based on the assembly code, are represented as vectors using a distributed representation of graph data known as graph embedding [17–21], which is a distributed representation of graph data. This enables a fixed-length vector representation that reflects the meaning and control structure of the target. If the semantics and control structures of different programs are similar, the generated vectors will be close in vector space. This allows the comparison of expression vectors to identify similar software products and programs. In addition, by using assembly code obtained from executable code, the approach is not dependent on the existence of source code or a common development anguage by the target software products. In our experiments, we evaluate the proposed method by classifying the obtained expression vectors among programs derived from a single program that have partial modifications to the source code of that program but no significant differences in the overall control flow.

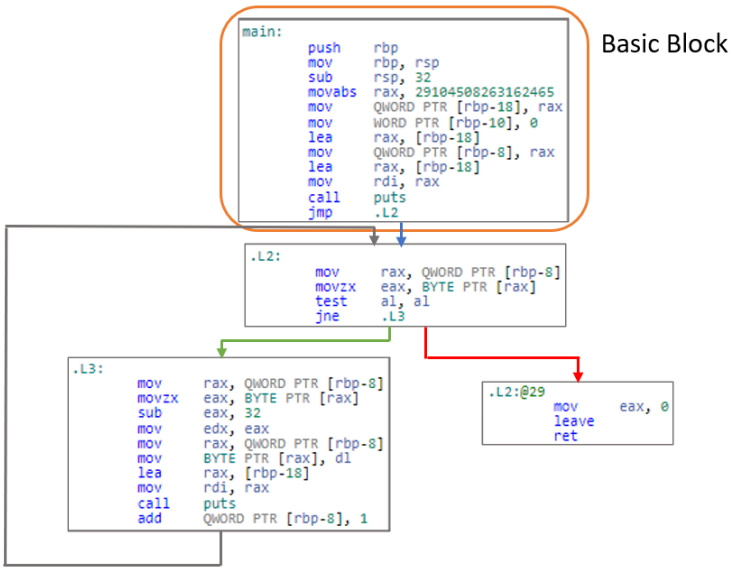

Figure 1. Control Flow Graph.

2. Related Works

2.1. Representation Learning Model for Assembly Code

Asm2Vec [22] is an assembly code representation earning model created by Ding et al. In this study, Ding et al. used an expression earning model based on the syntax of assembly code examples to detect assembly functions in the same source code that have been changed in syntax or control structure by compiler options or obfuscation as clones based on semantic similarity.

Asm2Vec earns a representation of assembly code based on the PV-DM model [15]. In contrast to plain text, which is placed sequentially, here, assembly code contains a specific syntax and a structure different from the text. Therefore, Ding et al. represent the assembly code of each assembly function in CFG, and create multiple sequences based on its syntax. First, the approach selectively expands the calling function for a function call instruction in an assembly function. Second, for the caller-extended CFGs obtained, sequences are created by the following two methods:

2.1.1. Edge Coverage

In edge coverage, edges are sampled randomly until all the edges of the CFG are covered, and the assembly code of the basic blocks at both ends are concatenated to create a sequence.

2.1.2. Random Walks

Random walks randomly select movable blocks starting from the entrance block and follow the CFG. The assembly code of the passed blocks are concatenated to make a sequence.

The repository RP contains the assembly functions used to train the embedding. After creating a sequence for each function in the repository RP, a neural network is used to maximize the next og probability over the entire repository RP.

$$R = RP\mathcal{S}(f_s)\mathcal{I}(seq_i)\mathcal{T}(in_j)\sum_{f_s}\sum_{seq_i}\sum_{in_j}\sum_{t_c}\log P(t_c|f_s, in_{j-1}, in_{j+1}) \tag{1}$$

$$\begin{aligned}\mathcal{S}(f_s) =& seq[1:i],\ f_s \in RP\\ \mathcal{I}(seq_i) =& in[1:j],\ seq_i \in \mathcal{S}(f_s)\\ \mathcal{T}(in_j) =& OP(in_j)||\mathcal{A}(in_j),\ in_j \in \mathcal{I}(seq_i)\\ & t_c \in \mathcal{T}(in_j)\end{aligned}$$

RP : A repository containing the assembly functions;
$\mathcal{S}(f_s)$: Set of sequences generated from the function f_s in the repository RP;
$\mathcal{I}(seq_i)$: Set of instructions contained in the *i-th* sequence seq_i of function f_s;
$\mathcal{T}(in_j)$: Set of tokens (operands and opcodes) contained in the *j-th* instruction in_j of the sequence seq_i;
$OP(in_j)$: Opcode of instruction in_j;
$\mathcal{A}(in_j)$: Set of operands in instruction in_j;
t_c: Current target token in instruction in_j.

Equation (1) represents the og probability of detecting a token t_c at the current instruction in_j from adjacent instructions in_{j-1} , in_{j+1} for each sequence seq_i of the assembly function f_s , in order. Maximizing this value means predicting the current instruction from the embedding vector of the assembly function and adjacent instructions (Figure 2). Using this model, Asm2Vec enables the creation of a distributed representation that captures the semantics of pieces of assembly code. If the embedding vector of the token t is $\vec{v_t} \in \mathbb{R}^d$, then instruction i is treated as a concatenation of tokens, and the embedding vector is $\vec{v_i} \in \mathbb{R}^{2\times d}$, and the embedding vector of function f_s is $\vec{\theta_{f_s}} \in \mathbb{R}^{2\times d}$. Furthermore, d is a parameter specified by the user and represents the dimension of the embedding.

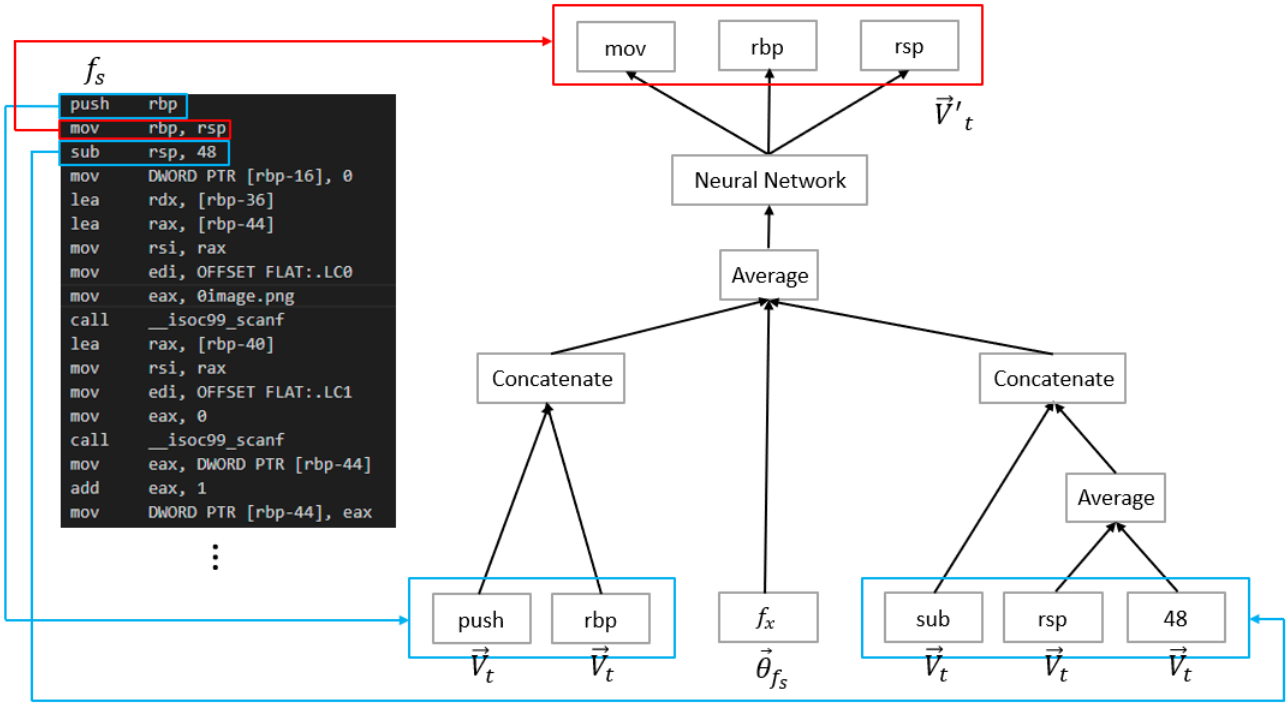

Figure 2. Representation earning model of Asm2Vec.

3. Proposed Method

3.1. Overview

In many cases, similar programs derived from one program have only partial modifications, additions, or deletions from the original source code. The proposed method aims for vector representations that are similar between programs that have many of these common parts.

Asm2Vec uses a representation earning model based on the syntax of assembly code to realize a distributed representation that captures the semantics of the code. In contrast, in a static random walk, it is difficult to create a sequence that correctly reflects the number of oop processes, actual behaviors such as I/O-based processing, and the overall control structure. In addition, the accuracy and stability of the results depend on the number of random walks, and increasing the number of random walks may increase the execution time. Therefore, because CFGs are similar between similar programs with only partial modifications, the proposed method solves these problems through embedding and graph embedding for basic blocks and creating a distributed representation method for assembly code that enables comparison of semantic and structural similarities. In this study, we use only non-obfuscated assembly code. This maintains the integrity of the control flow and the basic blocks of the assembly code.

3.2. Generate Target Graph

In this study, the entire executable code of the program is the target of embedding. The following steps are then used to generate the graph to be embedded from the assembly code:

3.2.1. Function Call Expansion of Main Routine (Main Function)

In the main function, function calls are expanded recursively, and the function call instruction is replaced with the destination (Figure 3). The recursive function may be expanded no more than once. Standard ibrary functions are not expanded as they have sufficient semantics for earning embedding and only user-defined functions that exist in the executable code are targeted.

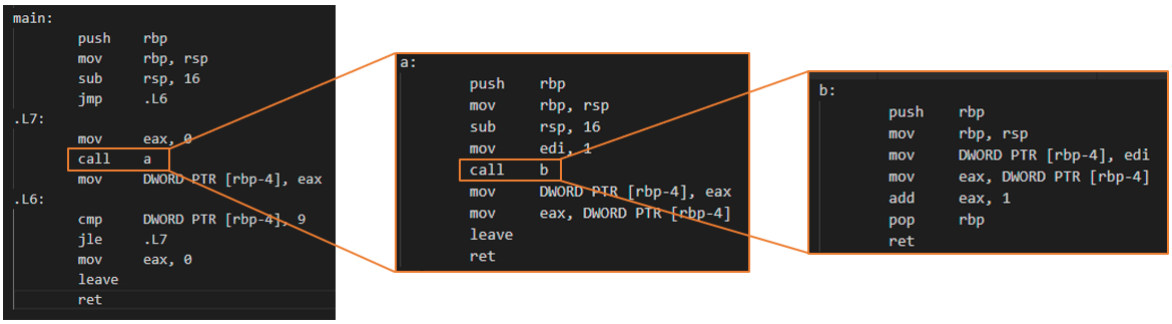

Figure 3. Recursive call expansion.

3.2.2. Vector Representation of Each Basic Block of the Main Function

Using Asm2Vec, we obtain a vector representation of the d-dimension for each basic block of the CFG of the main function that expands the function call. As the basic block does not contain branches or branch destinations and each instruction is placed in order, it can be treated as a sequence without randomness. Figure 4 shows the target graph of the CFG of the main function thus obtained. Herein, each node of the CFG has a d-dimensional vector representation of the basic block.

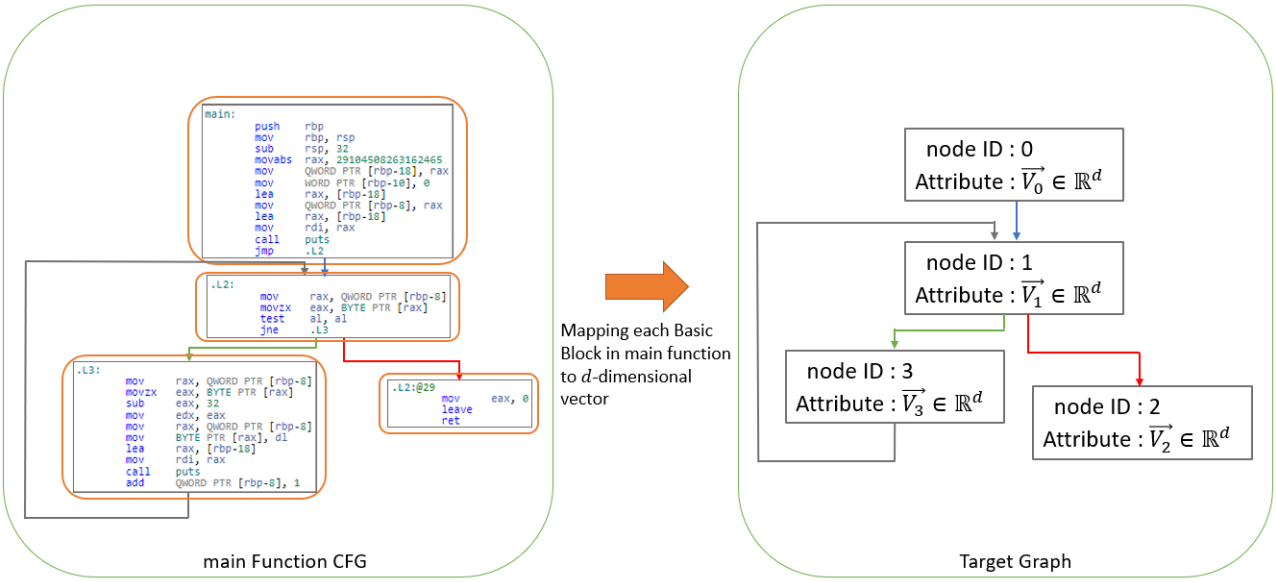

Figure 4. Generate target graph from CFG of the main function.

3.3. Graph Embedding Methods

The target graph to be embedded is a directed graph where each node has a vector of dimension d as an attribute. We use the vector representation method of this graph generated using FEATHER [23].

3.3.1. FEATHER

FEATHER [23] is a node-embedding method, created by Rozemberczki et al. that uses a complex functional characteristic function [24] with random walk probability weights to represent the feature distribution near a node. The characteristic function is a function that shows the characteristics of the distribution of a random variable and is an analog of the Fourier transform of the probability density.

FEATHER has the following features:

- It is possible to embed three types of information into the hybrid: neighbor node information, structural roles, and node attributes (metadata possessed by the nodes);

- The resulting vector representation of each node can be used as a representation of the entire graph by pooling it with a permutation-invariant function.

Based on these features, we obtain the representation vectors of the nodes of the target graph using FEATHER. Then, we calculate the representation vector of the whole graph using the arithmetic mean of the vectors as a pooling function. This is used as a distributed representation of the assembly code.

The following is the procedure for embedding a node.

3.3.2. Characteristic Functions and Node-Embedding on Graphs in FEATHER

There is an attributed graph, $G = (V, E)$. The nodes of G have a random variable X defined as a feature vector $X \in \mathbb{R}^{|V|}$, where X_v is the feature vector of node $v \in V$. When the distribution of this feature is described in the neighborhood of $u \in V$, the characteristic function of X of the source node u at the characteristic function evaluation point $\theta \in \mathbb{R}$ is defined by the following equation, where i is an imaginary unit:

$$\mathrm{E}[e^{i\theta X}|G, u] = \sum_{w \in V} P(w|u) \cdot e^{i\theta X_w} \tag{2}$$

In Equation (2), the affiliation probability $P(w|u)$ represents the strength of the relationship between the source node u and the target node $w \in V$. The real and imaginary parts of Equation (2) are defined by the following equations, respectively.

$$\mathrm{Re}(\mathrm{E}[e^{i\theta X}|G, u]) = \sum_{w \in V} P(w|u) \cos(\theta X_w) \tag{3}$$

$$\mathrm{Im}(\mathrm{E}[e^{i\theta X}|G, u]) = \sum_{w \in V} P(w|u) \sin(\theta X_w) \tag{4}$$

Here, u and w are parameterized by the transition probabilities of random walks on the r scale. Let $\{v_j, v_{j+1}, \ldots, v_{j+r}\}$ be a sequence of nodes in a random walk on G. When describing the distribution of features in the neighborhood of $u \in V$ reachable by a random walk (r scale) from u to r steps, we define each by the real and imaginary parts of the characteristic function as follows:

$$\mathrm{Re}(\mathrm{E}[e^{i\theta X}|G, u, r]) = \sum_{w \in V} P(v_{j+r} = w|v_j = u) \cos(\theta X_w) \tag{5}$$

$$\mathrm{Im}(\mathrm{E}[e^{i\theta X}|G, u, r]) = \sum_{w \in V} P(v_{j+r} = w|v_j = u) \sin(\theta X_w) \tag{6}$$

where $P(v_{j+r} = w|v_j = u) = P(w|u)$ in Equations (5) and (6) is the probability that a random walk starting from source node u will reach target node w within r steps. If the adjacency matrix of G is $\mathbf{A}$ and the order matrix is $\mathbf{D}$, the normalized adjacency matrix is defined as $\widehat{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}$. For a pair of source and target nodes u, w and a scale r, $P(v_{j+r} = w|v_j = u)$ can be represented by the r power of the normalized adjacency matrix. That is, $\widehat{\mathbf{A}}^r_{u,w} = P(v_{j+r} = w|v_j = u)$. From this we obtain the following equation:

$$\mathrm{Re}(\mathrm{E}[e^{i\theta X}|G, u, r]) = \sum_{w \in V} \widehat{\mathbf{A}}^r_{u,w} \cos(\theta X_w) \tag{7}$$

$$\mathrm{Im}(\mathrm{E}[e^{i\theta X}|G, u, r]) = \sum_{w \in V} \widehat{\mathbf{A}}^r_{u,w} \sin(\theta X_w) \tag{8}$$

The characteristic functions expressed in Equations (7) and (8) have the following properties

- The real part is an even function and the imaginary part is a radix;
- The range of both parts is $[-1, 1]$;
- Nodes with different structural roles have different characteristic functions.

For efficient computation, we extract d points from the domain of the characteristic function and evaluate the characteristic function at the point described by the evaluation point vector $\Theta \in \mathbb{R}$. We define the r-scaled random walk weight characteristic function of the entire graph as the complex matrix-valued function $C\mathcal{F}(G, X, \Theta, r) \to \mathbb{C}^{|V| \times d}$. The real and imaginary parts of this function are described by the following matrix-valued functions, respectively:

$$\mathrm{Re}(C\mathcal{F}(G, X, \Theta, r)) = \widehat{\mathbf{A}}^r \cos(X \otimes \Theta) \tag{9}$$

$$\mathrm{Im}(C\mathcal{F}(G, X, \Theta, r)) = \widehat{\mathbf{A}}^r \sin(X \otimes \Theta) \tag{10}$$

Each of these matrices corresponds to a node, and if two rows are similar, it means that the corresponding nodes have similar distributions in scale r. This describes the feature distribution around the node.

Next, the algorithm evaluates the characteristic function of the graph at all scales up to r for each node and each feature when there are k node features, that is, when the set of node features is defined as $\mathcal{X} = \{X^1, \ldots, X^k\}$, the following Algorithm 1 evaluates the characteristic function of the graph at all scales up to r for each node and each feature. For simplicity, each value of the characteristic function evaluation vector $\Theta^{i,r} \in \mathbb{R}^d, i \in \{1, \ldots, k\}$ are assumed to be the same.

1. Initialize the real part $\mathbf{Z}_{\mathrm{Re}}$ and imaginary part $\mathbf{Z}_{\mathrm{Im}}$ of the embedding (lines 1 and 2);
2. Repeat for each of the k different node features on a scale up to r (lines 3 and 4);
3. At the first scale, compute the outer product $\mathbf{H}$ of the feature under consideration and the corresponding evaluation point vectors (lines 6 and 7);
4. Take the cosine and sine of $\mathbf{H}$ (lines 8 and 9);
5. Using the normalized adjacency matrix, compute the real part $\mathbf{H}_{\mathrm{Re}}$ and imaginary part $\mathbf{H}_{\mathrm{Im}}$ of the characteristic function evaluation (lines 10 and 11);
6. Add matrices to the real and imaginary parts of the embedding (lines 12 and 13);
7. Once the characteristic function of each feature is evaluated for each scale, concatenate and return the real and imaginary parts of the embedding (lines 14 and 15).

Algorithm 1: Node-embedding algorithm on FEATHER.

Data: $\widehat{\mathbf{A}}$—Normalized adjacency matrix.
$\mathcal{X} = \{x^1, \ldots, x^k\}$—Set of node feature vectors.
$\widetilde{\Theta} = \{\Theta^{1,1}, \ldots, \Theta^{1,r}, \Theta^{2,1}, \ldots, \Theta^{k,r}\}$—Set of evaluation point vectors.
r—Scale of empirical graph characteristic function.
Result: Node-embedding matrix $\mathbf{Z}$

1 $\mathbf{Z}_{\mathrm{Re}} \leftarrow$ Initialized Real Features()
2 $\mathbf{Z}_{\mathrm{Im}} \leftarrow$ Initialized Imaginary Features()
3 **for** i in 1 : k **do**
4 **for** j in 1 : r **do**
5 **for** l in 1 : j **do**
6 **if** $l = 1$ **then**
7 $\mathbf{H} \leftarrow x^i \otimes \Theta^{i,j}$
8 $\mathbf{H}_{\mathrm{Re}} \leftarrow \cos(\mathbf{H})$
9 $\mathbf{H}_{\mathrm{Im}} \leftarrow \sin(\mathbf{H})$
10 $\mathbf{H}_{\mathrm{Re}} \leftarrow \widehat{\mathbf{A}}\mathbf{H}_{\mathrm{Re}}$
11 $\mathbf{H}_{\mathrm{Im}} \leftarrow \widehat{\mathbf{A}}\mathbf{H}_{\mathrm{Im}}$
12 $\mathbf{Z}_{\mathrm{Re}} \leftarrow [\mathbf{Z}_{\mathrm{Re}} \mid \mathbf{H}_{\mathrm{Re}}]$
13 $\mathbf{Z}_{\mathrm{Im}} \leftarrow [\mathbf{Z}_{\mathrm{Im}} \mid \mathbf{H}_{\mathrm{Im}}]$
14 $\mathbf{Z} \leftarrow [\mathbf{Z}_{\mathrm{Im}} \mid \mathbf{Z}_{\mathrm{Re}}]$
15 Output $\mathbf{Z}$.

3.3.3. Pooling of Each Node

The real and imaginary parts of the mean pooled r-scale random-walk-weighted characteristic function are defined by the following equation:

$$\mathrm{Re}(\mathrm{E}[e^{i\theta X}|G,u,r]) = \sum_{u\in V}\sum_{w\in V}\widehat{\mathbf{A}}^{r}_{u,w}\frac{\cos(\theta X_w)}{|V|} \tag{11}$$

$$\mathrm{Im}(\mathrm{E}[e^{i\theta X}|G,u,r]) = \sum_{u\in V}\sum_{w\in V}\widehat{\mathbf{A}}^{r}_{u,w}\frac{\sin(\theta X_w)}{|V|} \tag{12}$$

The functions expressed in Equations (11) and (12) allow characterization and comparison based on the structural properties of the entire graph.

4. Experiment and Results

4.1. Experiment Overview

In our experiments, we use an experimental dataset to classify similar programs and evaluate the results. The experimental dataset consists of assembly code generated from the publicly available solution source code of the programming contest "Kyopuro tenkei 90 mon [25]". The solution code for the same problem by the same author have many common parts due to reuse, and their control flows are similar. Therefore, we perform K-means clustering with cos similarity as the distance function on the set of expression vectors obtained from the experimental dataset, and measure the accuracy with which the solutions of the same creator are classified into the same cluster.

4.2. Experimental Data Sets

The experimental dataset consists of 32 assembly source files generated from the solution code.

Each assembly source file is written in assembly code and named according to the following naming convention:

- One capital alphabet etter to indicate the solution creator;
- Issue no.;
- Sequential numbers for multiple solutions;
- File extension (.s).

Each assembly code is output from the source code of the solution method written in C, using the x-86-64gcc12.2 compiler, without using any compiler options.

Each value in Table 1 is the similarity between the assembly code of each file in the dataset, expressed by the following equation.

$$\text{Similarity: } S = (1 - \text{Normalized Levenshtein distance})$$

The Levenshtein distance [26] is defined as the minimum number of steps required to transform one string into another by inserting, deleting, or replacing a single character in one of the two strings, and is the distance that indicates the difference between the two strings. The normalized Levenshtein distance is normalized to a value between 0 and 1 by dividing the Levenshtein distance by the number of characters in the onger of the two strings. Similarity S takes a value between 0 and 1 and represents the degree of agreement between the two strings.

From Table 1, it can be seen that the similarity between the assembly code generated from the solutions of the same author is higher than that of different authors, and the solutions of the same author are similar.

Table 1. Similarity of the assembly code for each file in the data set.

	A_1_0.s	**B_1_0.s**	**C_1_0.s**	**D_2_0.s**	**E_2_0.s**	**F_3_0.s**	**G_3_0.s**
A_1_0.s	**1.0**	0.59	0.71	0.65	0.56	0.66	0.64
A_1_1.s	**0.95**	0.60	0.71	0.65	0.57	0.66	0.63
A_1_2.s	**0.93**	0.60	0.71	0.65	0.57	0.66	0.63
A_1_3.s	**0.86**	0.59	0.73	0.64	0.50	0.65	0.65
A_1_4.s	**0.85**	0.59	0.74	0.64	0.50	0.65	0.65
B_1_0.s	0.59	**1.0**	0.58	0.60	0.51	0.56	0.55
B_1_1.s	0.59	**1.0**	0.58	0.60	0.51	0.56	0.55
B_1_2.s	0.57	**0.92**	0.56	0.58	0.48	0.56	0.53
B_1_3.s	0.57	**0.88**	0.55	0.57	0.49	0.53	0.53
C_1_0.s	0.71	0.58	**1.0**	0.66	0.49	0.64	0.64
C_1_1.s	0.70	0.58	**0.99**	0.66	0.49	0.64	0.64
C_1_2.s	0.72	0.58	**0.98**	0.66	0.50	0.64	0.64
C_1_3.s	0.71	0.57	**0.96**	0.65	0.49	0.63	0.63
C_1_4.s	0.72	0.58	**0.98**	0.66	0.51	0.65	0.64
D_2_0.s	0.65	0.60	0.66	**1.0**	0.50	0.61	0.61
D_2_1.s	0.65	0.60	0.66	**0.99**	0.50	0.61	0.61
D_2_2.s	0.65	0.60	0.66	**0.99**	0.50	0.61	0.61
D_2_3.s	0.65	0.59	0.65	**0.97**	0.51	0.61	0.60
E_2_0.s	0.56	0.51	0.49	0.50	**1.0**	0.54	0.50
E_2_1.s	0.55	0.50	0.50	0.50	**0.93**	0.53	0.49
E_2_2.s	0.55	0.50	0.50	0.50	**0.93**	0.54	0.49
E_2_3.s	0.54	0.51	0.49	0.50	**0.97**	0.53	0.50
F_3_0.s	0.66	0.56	0.64	0.61	0.54	**1.0**	0.77
F_3_1.s	0.67	0.56	0.64	0.60	0.55	**0.98**	0.79
F_3_2.s	0.66	0.56	0.63	0.61	0.55	**0.94**	0.77
F_3_3.s	0.67	0.56	0.64	0.61	0.55	**0.95**	0.78
F_3_4.s	0.67	0.56	0.64	0.61	0.55	**0.95**	0.78
G_3_0.s	0.64	0.55	0.64	0.61	0.50	0.77	**1.0**
G_3_1.s	0.65	0.55	0.64	0.61	0.51	0.78	**0.98**
G_3_2.s	0.65	0.56	0.63	0.61	0.50	0.76	**0.95**
G_3_3.s	0.65	0.56	0.64	0.61	0.51	0.77	**0.96**
G_3_4.s	0.64	0.54	0.64	0.60	0.49	0.77	**0.96**

Bolded numbers in the table indicate the similarity of assembly codes generated from the same author's solution, which is higher than the others.

4.3. Used Models

The following models(Tables 2 and 3) are used in the experiments (Figure 5). All these models take as input the CFG of the calling expanded main function in each assembly file.

1. Model_1
   - **Embedded target**: CFG with node attributes;
     Each node has the following node features:
     - **Node neighborhood**: Normalized adjacent matrix;
     - **Structural information**: Logarithmic transformation of the order of each node and clustering coefficients;
     - **Node attribute**: 400-dimensional (each token: 200 dimensions) representation vector obtained from the corresponding basic block using Asm2Vec (Reduced to 16 dimensions using the Python ibrary Scikit-Learn's Truncated SVD [27]).
   - **Embedding method**: Pooling on the arithmetic mean after each node embedding using FEATHER;
   - **Embedded dimension**: 4000-dimensional;
   - **Number of evaluation points** Θ: 25;

- **Scale of random walk** r: 5.

2. Model_2
   - **Embedded target**: CFG without node attribute;
     Each node has the following node features:
     - **Node neighborhood**: Normalized adjacent matrix;
     - **Structural information**: Logarithmic transformation of the order of each node and clustering coefficients.
   - **Embedding method**: Pooling on the arithmetic mean after each node embedding using FEATHER;
   - **Embedded dimension**: 500-dimensional;
   - **Number of evaluation points** Θ: 25;
   - **Scale of random walk** r: 5.
3. Model_3
   - **Embedded target**: CFG without node attribute;
     Each node has the following node features:
     - **Node neighborhood**: Normalized adjacent matrix;
     - **Structural information**: Logarithmic transformation of the order of each node and clustering coefficients.
   - **Embedding method**: Pooling on the arithmetic mean after each node embedding using FEATHER;
   - **Embedded dimension**: 4000-dimensional;
   - **Number of evaluation points** Θ: 25;
   - **Scale of random walk** r: 40.
4. Model_4
   - **Embedded target**: Assembly Code;
   - **Embedding Method**: Asm2Vec;
   - **Embedded dimension**: 500-dimensional(each token: 250-dimensional);
   - **Number of random walks** n: 10 .
5. Model_5
   - **Embedded target**: Assembly Code;
   - **Embedding Method**: Asm2Vec;
   - **Embedded dimension**: 4000-dimensional(each token: 2000-dimensional);
   - **Number of random walks** n: 10.
6. Model_6
   - **Embedded target**: Assembly Code;
   - **Embedding Method**: Asm2Vec;
   - **Embedded dimension**: 500-dimensional (each token: 250-dimensional);
   - **Number of random walks** n: 50.

The clustering coefficients are a measure of the ratio of edges between adjacent nodes of a node, and the higher the average of the cluster coefficients of all nodes in a graph, the denser the network. The following equation expresses the cluster coefficient c_u of a node u in an edge-unweighted graph.

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)} \tag{13}$$

$deg(u)$: Order of u;
$T(u)$: Number of triangles formed by connecting u and the edges of adjacent nodes

Table 2. Details of each model (Model_1, 2, 3).

	Model_1	**Model_2**	**Model_3**
Embedded target	CFG with node attributes	CFG without node attribute	CFG without node attribute
Embedding method	FEATHER	FEATHER	FEATHER
Embedded dimension	4000	500	4000
Number of evaluation points	25	25	25
Scale of random walk	5	5	40

Table 3. Details of each model (Model_4, 5, 6).

	Model_4	**Model_5**	**Model_6**
Embedded target	Assembly Code	Assembly Code	Assembly Code
Embedding method	Asm2Vec	Asm2Vec	Asm2Vec
Embedded dimension	500	4000	500
Number of random walks	10	10	50

For all Asm2Vec training, the number of iterations is 10. The exit condition of the random walk in Asm2Vec is the arrival at the exit block of the CFG or the arrival at the basic block that has already been passed.

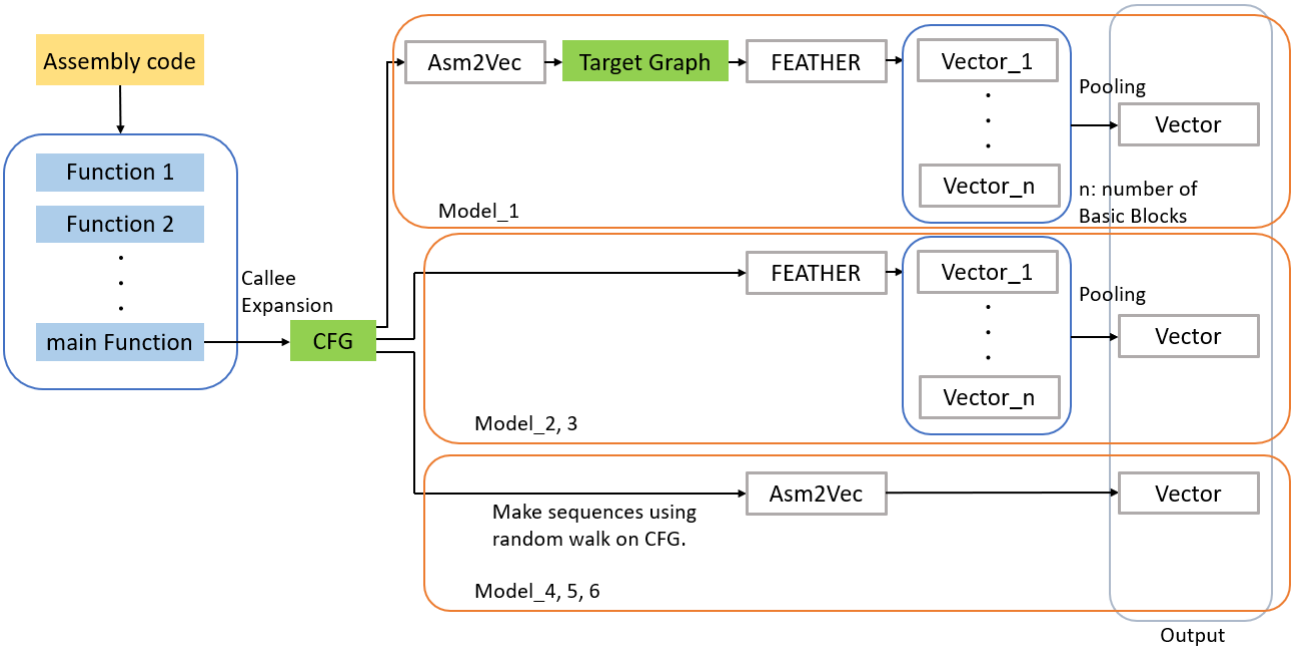

Figure 5. Flow of each model.

4.4. Evaluation Index

We evaluate whether the proposed method can embed vectors generated from similar programs in close proximity in the vector space.

The following are used as evaluation indices.

4.4.1. Precision Measure and Recall Measure

To evaluate the clustering results, we calculate precision and recall measure over pairs of cluster elements [28]. Precision and recall measure are expressed by the following equations:

$$precision = \frac{TP}{TP + FP} \tag{14}$$

$$recall = \frac{TP}{TP + FN} \tag{15}$$

TP: Number of pairs of elements with the same abel that are classified into the same cluster as a result of clustering;
FP: Number of pairs of elements with different abels that are classified into the same cluster as a result of clustering;
FN: Number of pairs of elements with the same abel that are classified into different clusters as a result of clustering.

Precision is calculated as the fraction of pairs correctly put in the same cluster, and recall is the fraction of actual pairs that were identified.

4.4.2. Normalized Mutual Information

The amount of information $H(U)$, $H(V)$ for objects U and V with the same N elements is expressed by the following equations:

$$H(U) = -\sum_{i=1}^{|U|} P(i) \log(P(i)) \tag{16}$$

$$H(V) = -\sum_{j=1}^{|V|} P(j) \log(P(j)) \tag{17}$$

$$P(i) = \frac{|U_i|}{N}, P(j) = \frac{|V_j|}{N} \tag{18}$$

When the probability that a randomly selected object is classified into both classes U_i and V_j is $P(i,j) = |U_i \cap V_j|/N$, the mutual information $MI(U,V)$ and normalized mutual information $NMI(U,V)$ are expressed as follows:

$$MI(U,V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i,j) \log\left(\frac{P(i,j)}{P(i)P(j)}\right) \tag{19}$$

$$NMI(U,V) = \frac{MI(U,V)}{\text{mean}(H(U), H(V))} \tag{20}$$

Mutual information content is a function that measures the agreement between two assignments, ignoring permutations, and normalized mutual information content is the normalized mutual information content with the score scaled between 0 (no mutual information) and 1 (perfect correlation). This index evaluates how well the expected number of clusters and the number of elements in each cluster agree with the correct answer.

4.4.3. Execution Time

The execution time is the elapsed time from the input of the CFG of the main function to the output of the clustering result.

4.5. Results of the Experiments

Table 4 shows the results of clustering with the number of clusters $k = 7$. Each value is the average of five runs.

Table 4. Results of the experiments.

	Model_1	Model_2	Model_3	Model_4	Model_5	Model_6
Precision measure	0.85	0.79	0.812	0.507	0.506	0.621
Recall measure	0.929	0.844	0.862	0.635	0.633	0.707
Normalized Mutual Information	0.954	0.895	0.929	0.657	0.673	0.785
Execution time (s)	35.239	0.0359	0.0657	153.044	462.574	774.089

From Table 4, it can be seen that the proposed method, Model_1, has the highest values for the precision measure, the recall measure, and the amount of normalized mutual information; while, in terms of execution time, Model_2 was the fastest.

5. Evaluation and Discussion

As the results presented in Table 4 show, the proposed method, Model_1, has the highest values for the precision measure, the recall measure, and the amount of normalized mutual information. The results also show that the desired clustering results are obtained. This means that, with the proposed method, the representation vectors of similar pieces of assembly code are mapped most closely in the vector space. Furthermore, in each model, the recall measure was higher than the precision measure. This is because the False Negative (FN) value was ower than the False Positive (FP) value in each case. This means that cases in which elements of the same cluster are missed are ess ikely to occur than cases in which elements of different clusters are detected incorrectly.

Figures 6–8 visualize the most frequent results among the clustering results of Model_1, Model_3, and Model_4. Each representation vector is dimensionally reduced to two dimensions using principal component analysis, and plotted with coloring for each cluster. From the results of Model_4 and 5, Asm2Vec did not show a significant performance change between 500 and 2000 embedding dimensions, and Model_1, 2, and 3 outperformed the random-walk-based Asm2Vec in all indices for the same embedding dimensions. Comparison between Model_4 and Model_6 shows that increasing the number of random walk sequences improves performance, but increases the execution time roughly approximately inearly. Asm2Vec was used to retrieve the attributes in Model_1, and embedding by basic blocks can be performed in a shorter time than embedding from the whole CFG. Thus, applying graph embedding to CFGs for the classification of similar programs whose overall control flow is not significantly changed, as carried out in this experiment, is effective.

We also compared Model_1, 2, and 3. A comparison of the results of Model_1 and Model_3 shows that adding attributes to the nodes and incorporating them into the embedding process results in better embedding than in the case where only the information obtained from the graph (neighborhood and structural information) is considered. Meanwhile, the extraction of attributes and their dimension reduction increase the execution time significantly. In addition, from the comparison between Model_2 and Model_3, it can be seen that the accuracy of embedding can be improved by increasing the random walk scale r in FEATHER, as more neighborhood and structural information can be incorporated. Here, in FEATHER, when the number of node features is multiplied by n and the scale is multiplied by m, the dimension of embedding becomes $n \times m$ times arger. As the embedding dimension increases, the number of computations required for comparison between representation vectors increases, which may increase the time required for comparison or make comparison difficult. Therefore, when increasing the scale r in a graph with node attributes, it is important to appropriately reduce the dimension of the node feature matrix.

One of the concerns of the proposed method is that Asm2Vec, which is used to extract node attributes, is designed for a single assembly anguage and cannot be applied to different architectures. Therefore, in terms of execution time and generality, embedding without considering node attributes may be superior in some cases.

Thus, the proposed method can identify similar programs with similar source code that have not undergone major changes in the control flow. Furthermore, the recall measure has a higher value than the precision measure, which is useful in situations where we want to reduce oversights rather than false positives. Thus, the proposed approach can be used to detect source code copying and variants of malware derived from existing malware.

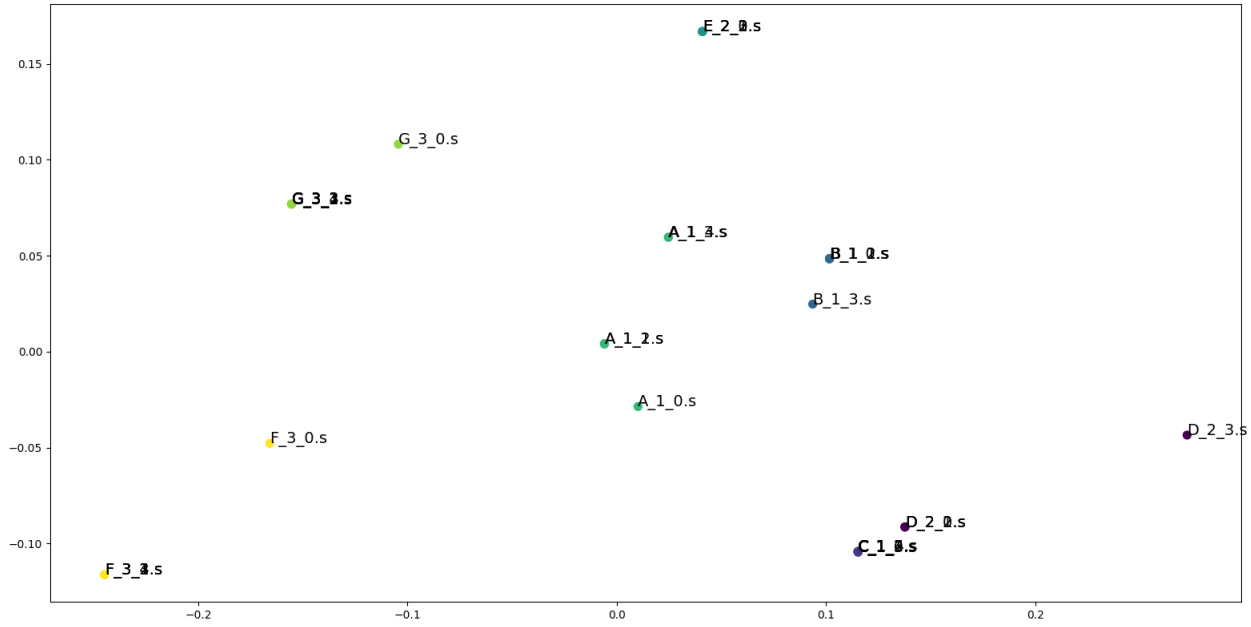

Figure 6. Visualization of the most frequent results in Model_1. Dots of the same color are in the same cluster.

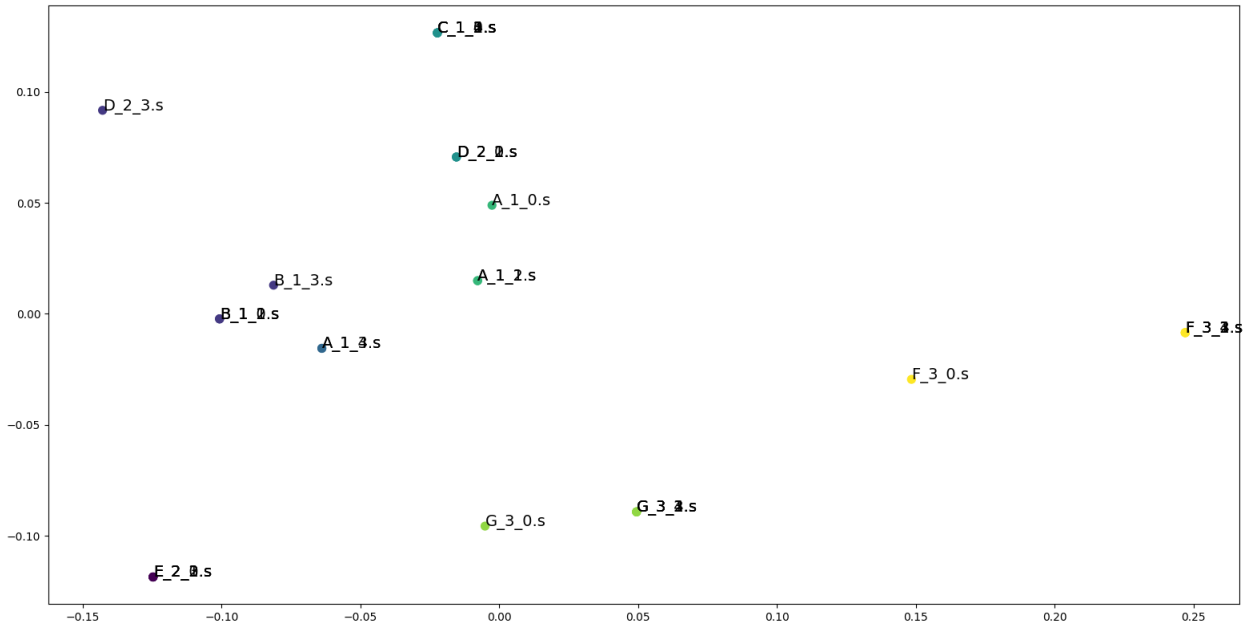

Figure 7. Visualization of the most frequent results in Model_3. Dots of the same color are in the same cluster.

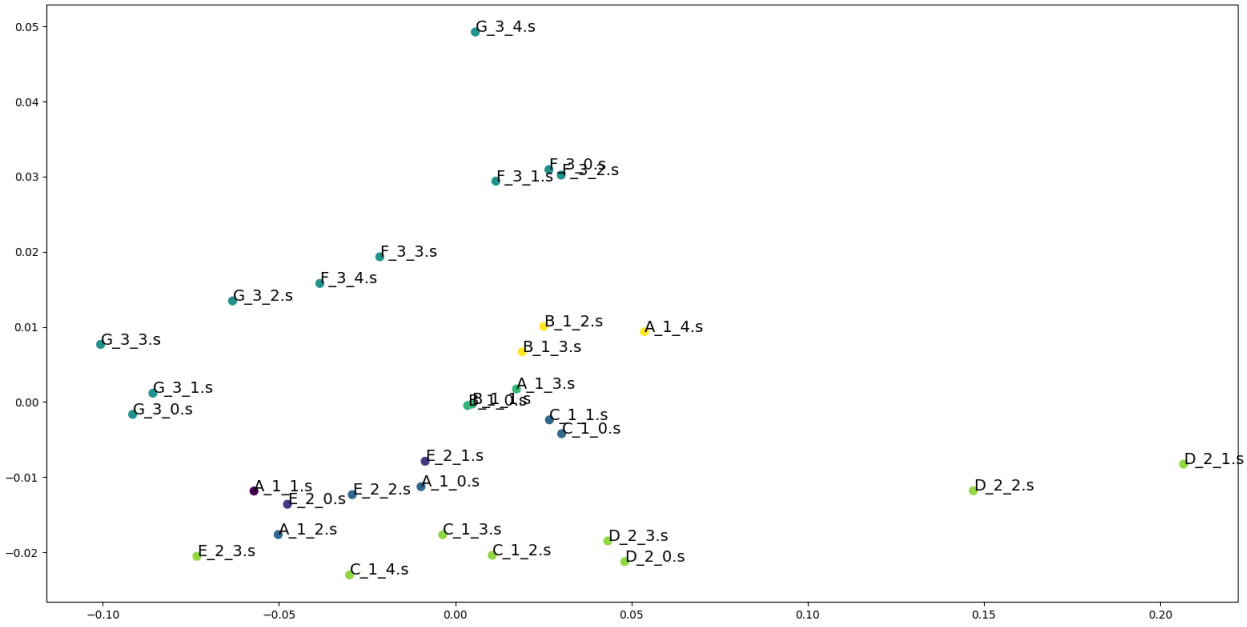

Figure 8. Visualization of the most frequent results in Model_4. Dots of the same color are in the same cluster.

6. Conclusions

In this study, we created a distributed representation method of assembly code that can be used to detect similar programs and software products, the occurence of which has increased in recent times due to source code reuse and plagiarism. The proposed method combines Asm2Vec, an existing assembly code distributed representation method, with FEATHER, a graph embedding method, based on the fact that the CFG of each program is similar between similar programs. Asm2Vec performs embedding based on a earning model based on the syntax of assembly code, capturing the semantics of assembly code. FEATHER is also capable of hybrid embedding that incorporates information on neighboring nodes of the graph, structural information, and metadata possessed by each node. Therefore, we used Asm2Vec to create a vector representation of the basic blocks corresponding to each node of the CFG, and FEATHER and a vector representation of the graph with these basic blocks as node attributes. In the experiment, we performed vector representation of the assembly code generated from solution code submitted to a programming contest using multiple distributed representation methods including the proposed method. Next, we performed clustering of the obtained representation vectors to measure the ratio of solutions by the same creator that are classified into the same cluster. Then, we used precision measure, recall measure, and normalized mutual information as evaluation indices of clustering accuracy. The experimental results show that the clustering accuracy and execution time of the proposed method are better than those of the random-walk-based Asm2Vec, indicating the effectiveness of graph embedding. Furthermore, the recall measure was higher than the precision measure. Thus, the proposed method can determine similarity based on the semantics and control structure of software and programs, independent of the existence of source code or the use of different development anguages. This approach can be applied to identify variants of malware derived from existing malware or plagiarized software. However, it should be noted that assembly code is always required to generate CFGs and acquire node attributes; thus, if accurate assembly code cannot be obtained due to software obfuscation, the proposed method will not work correctly. In the future, we will implement obfuscation measures and validate the utility of the proposed approach with more data.

Author Contributions: Conceptualization, K.Y., K.S., and T.M.; methodology, K.Y., K.S., and T.M.; software, K.Y.; validation, K.Y.; formal analysis, K.Y.; investigation, K.Y.; data curation, K.Y.; writing—original draft preparation, K.Y.; writing—review and editing, K.Y.; supervision, T.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All data has been present in main text.

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

References

1. M-Trends Report. 2020. Available online: https://www.mandiant.com/sites/default/files/2021-09/mtrends-2020.pdf (accessed on 24 March 2023)
2. Yuta, T.; Terada, M.; Matsuki, T.; Kasama, T.; Araki, S.; Hatada, M. Datasets for Anti-Malware Research. *SIG Tech. Rep.* **2018**, *38*, 1–8.
3. Yamamoto, T.; Matsushita, M.; Kamiya, T.; Inoue, K. Similarity Metric CSR Using Code Clone Detection Tool. *Softw. Sci.* **2001**, *101*, 25–32.
4. Okahara, S.; Manabe, Y.; Yamauchi, H.; Monden, A.; Matsumoto, K.; Inoue, K. Experimentally Deriving Probability of Program Piracy based on Length of Code Clone. *IEICE Tech. Rep.* **2008**, *108*, 7–11.
5. Basit, H.A.; Puglisi, S.J.; Smyth, W.F.; Turpin, A. Efficient token based clone detection with flexible tokenization. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Dubrovnik, Croatia, 3–7 September 2007; pp. 513–516.
6. Selim, G.m.k.; Foo, K.C.; Zou, Y. Enhancing Source-Based Clone Detection Using Intermediate Representation. In Proceedings of the 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 227–236.
7. Gode, N.; Koschke, R. Incremental Clone Detection. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 24–27 March 2009; pp. 219–228.
8. Hummel, B.; Jürgens, E.; Heinemann, L.; Conradt, M. Index-based code clone detection: Incremental, distributed, scalable. In Proceedings of the 26th IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010; pp. 1–9.
9. Chilowicz, M.; Duris, É.; Roussel, G. Syntax tree finger printing for source code similarity detection. In Proceedings of the 17th IEEE International Conference on Program Comprehension, Vancouver, BC, Canada, 17–19 May 2009; pp. 243–247.
10. Iwamura, M.; Itoh, M.; Muraoka, Y. Automatic malware classification system based on similarity of machine code instructions. *Inf. Process. Soc. Jpn.* **2010**, *51*, 1–11.
11. Flake, H. Structural Comparison of Executable Objects. In Proceedings of the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2004), Dortmund, Germany, 6–7 July 2004; pp. 161–173.
12. Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig.* **2006**, *3*, 91–97. [CrossRef]
13. Roussev, V.; Quates, C. Content triage with similarity digests: The M57 case study. *Digit. Investig.* **2012**, *9*, S60–S68. [CrossRef]
14. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed Representations of Words and Phrases and Their Compositionality. In Proceedings of the 26th International Conference on Neural Information Processing Systems, Lake Tahoe, NA, USA, 5–10 December 2013.
15. Le, Q.; Mikolov, T. Distributed representations of sentences anddocuments. In Proceedings of the 31st International Conference on International Conference on Machine Learning, Beijing, China, 21–26 June 2014.
16. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* **2018**, *3*, 1–29. [CrossRef]
17. Perozzi, B.; Al-Rfou, R.; Skiena, S. Deepwalk: Online earning of social representations. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014; pp. 701–710.
18. Grover, A.; Leskovec, J. node2vec: Scalable feature earning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Nowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 855–864
19. Ahmed, N.K.; Rossi, R.; Lee, J.B.; Willke, T.L.; Zhou, R.; Kong, X.; Eldardiry, H. Learning role-based graph embeddings. *arXiv* **2018**, arXiv:1802.02896.
20. Liao, L.; He, X.; Zhang, H.; Chua, T. Attributed social network embedding. *IEEE Trans. Knowl. Data Eng.* **2018**, *30*, 2257–2270. [CrossRef]
21. Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; Liu, Y.; Jaiswal, S. graph2vec: Learning Distributed Representations of Graphs. *arXiv* **2017**, arXiv:1707.05005.
22. Ding, S.H.H.; Fung, B.C.M.; Charland, P. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 19–23 May 2019.

23. Rozemberczki, B.; Sarkar, R. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In Proceedings of the CIKM '20: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, Virtual Event, Ireland, 19–23 October 2020; pp. 1325–1334.
24. DasGupta, A. Characteristic functions and applications. In *Probability for Statistics and Machine Learning*; Springer: New York, NY, USA, 2011 ; pp. 292–322.
25. Kyopuro Tenkei 90 Mon. Available online: https://atcoder.jp/contests/typical90 (accessed on 24 March 2023)
26. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
27. Halko, N.; Martinsson, P.-G.; Tropp, J.A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *Siam Rev.* **2011**, *53*, 217–288. [CrossRef]
28. Banerjee, A.; Krumpelman, C.; Ghosh, J.; Basu, S.; Mooney, R.J. Model-based overlapping clustering. In Proceedings of the KDD '05: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, Chicago, IL, USA, 21–24 August 2005; pp. 532–537