Hindawi
Security and Communication Networks
Volume 2021, Article ID 3095203, 8 pages
https://doi.org/10.1155/2021/3095203

Research Article

Cross-Platform Binary Code Homology Analysis Based on GRU Graph Embedding

Shen Wang, Xunzhi Jiang, Xiangzhan Yu, and Xiaohui Su

School of Cyberspace Science, Harbin Institute of Technology, Harbin 150001, China

Correspondence should be addressed to Shen Wang; shen.wang@hit.edu.cn

Received 31 July 2021; Revised 21 October 2021; Accepted 1 November 2021; Published 18 December 2021

Academic Editor: Gu Zhaoquan

Binary code homology analysis refers to detecting whether two pieces of binary code are compiled from the same piece of source code, which is a fundamental technique for many security applications, such as vulnerability search, plagiarism detection, and malware detection. With the increase in critical vulnerabilities in IoT devices, homology analysis is increasingly needed to perform cross-platform vulnerability searches. Existing methods for cross-platform binary code homology detection usually convert binary code to instruction sequences and do semantic embedding of the sequences as if they were natural language. However, the gap between natural language and binary code is large, and the spatial features of the binary code are easily lost by directly comparing the semantics. In this paper, we propose a GRU-based graph embedding method to compare the homology of binary functions. First, the attribute control flow graph (ACFG) is built for the assembly function, then the GRU-based graph embedding neural network is used to generate the embedding vector for the ACFG, and finally the homology of the binary code is determined by calculating the distance between the embedding vectors. The experimental results show that our method greatly improves the detection accuracy of negative samples compared with Gemini, the latest method based on graph embedding binary code similarity detection.

1. Introduction

With the rise and development of the Internet of Things technology, more and more embedded devices carry out network communications, and some of the security issues that exist among them have become increasingly prominent. Same as traditional application software, the firmware in many embedded devices also has functional defects, and most embedded device systems contain the same vulnerabilities. Due to the separation of device development and production, device manufacturers contract the development of purchased hardware with drivers to third-party vendors, thus resulting in hardware devices from different vendors that are likely to run the same third-party codebase. Therefore, it is urgent to find a reasonable vulnerability analysis method for embedded device firmware to effectively detect the homology of similar code.

Current binary code homology analysis mainly uses dynamic tracing or static analysis to obtain feature information, such as instruction sequences [1], API call sequences [2], or graph structure features [3]. Sequence information is easier to obtain than graph structure information, so most researchers conduct research on the basis of instruction sequence or API sequence, treat the sequence as a natural language, and use semantic embedding methods to obtain semantic features. However, compared with graph structure information, semantic features usually have larger dimensions leading to lower detection efficiency and lose spatial features in binary code execution, such as function call relationships and basic block call relationships, which are similar when cross-platform. On the other hand, for the matching and analysis of graph structure data, the process of computing homology needs to involve the subgraph isomorphism problem [4], which is an NP-complete problem, and as the graph size increases, the computational complexity grows exponentially, and the efficiency cannot be effectively guaranteed. Recently, Xu et al. [5] proposed a neural network-based approach, Gemini, which shows great

advantages. First, each basic block in the attribute control flow graph (ACFG) is transformed into a manually selected feature, then the embedding of the graph is generated using Structure2vec [6], and finally a Siamese architecture is added to the binary function to calculate the similarity score and reduce the loss. Although this method outperforms traditional methods in terms of accuracy and speed, it is still not effective enough in terms of graph embedding.

We propose an improvement to Gemini by combining the GRU (gate recurrent unit) [7] module in graph embedding. GRU is a recurrent neural network (RNN), which can efficiently learn the temporal information of vectors by updating gates and resetting gates to forget or retain information [8]. Based on this theory, the vertex embedding vectors generated by Structure2vec are fed into the GRU network at different time steps in a rough calling sequence. Compared with the direct linear addition of the vertex embedding vectors proposed by Gemini [5], the use of GRU preserves the order information of the vertex embedding, which makes the final embedding have more spatial features and more accurate embedding. The main contributions of this paper are as follows:

(i) We use the GRU module to aggregate the vertex embedding in the graph embedding process, which makes the embedding better

(ii) Experiments show that the GRU-based graph embedding method improves the accuracy compared with Gemini's graph embedding method on the same dataset

2. Related Work

2.1. GRU. In 2014, Cho et al. [7] proposed GRU to solve the problems of long-term memory of RNN and gradient in backpropagation by introducing a gating mechanism to control the propagation of gradient information to alleviate the phenomenon of gradient disappearance. The GRU includes two gates: a gate to control update and a gate to control reset. The specific internal structure is shown in Figure 1.

First, the two gating states are calculated by using the hidden layer state h_{t-1} propagated from the previous moment and the input data x_t from the current moment, where r is the gating that controls resetting and z is the gating that controls updating. The formula is shown as follows:

$$r = \sigma\left(W_r \cdot \left[x_t; h_{t-1}\right]\right), \tag{1}$$

$$z = \sigma\left(W_z \cdot \left[x_t; h_{t-1}\right]\right), \tag{2}$$

$$\sigma(l) = \frac{1}{1 + e^{-x}}. \tag{3}$$

After two gates are obtained, the h_{t-1} message propagated at the previous moment is first "reset" using a reset gate, as in the following formula:

$$h'_{t-1} = h_{t-1} \odot r, \tag{4}$$

where $\odot$ is to multiply the corresponding elements in the operation matrix, and the two multiplied matrices are required to be of the same type. Then, h'_{t-1} is spliced with the input data x_t, and a tanh activation function is used to scale the calculation result to the range of [−1, 1], as in the following formula:

$$h' = \tanh\left(W \cdot \left[x_t; h'_{t-1}\right]\right), \tag{5}$$

where h' mainly learns the current input data x_t, which is equivalent to memorizing the state of the current moment, and its principle is similar to the selective memory stage in LSTM.

Finally, GRU "update memory" calculation is carried out, in which two steps of "forgetting" and "remembering" are calculated at the same time as shown in the following formula:

$$h_t = z \odot h_{t-1} + (1 - z) \odot h', \tag{6}$$

where $z \odot h_{t-1}$ means selectively "forgetting" the hidden layer state passed down from the previous moment, i.e., forgetting some unimportant information in h_{t-1}, and $(1 - z) \odot h'$ means selectively "remembering" the h' containing the input node information of the current moment and also forgetting some unimportant information in the h' dimension, i.e., selecting some information in the h' dimension.

In summary, what formula (6) does is to forget some dimensional information of h_{t-1} passed down and add some dimensional information input by the current node. Among them, "forgetting" and "selection" are linked; that is, the GRU selectively forgets the incoming dimensional information and compensates by using the weights in the h' containing the current input to maintain a constant state. Compared with LSTM [9], GRU has one less gate and fewer parameters than LSTM, but it can achieve performance equivalent to LSTM [10, 11].

2.2. Binary Code Homology Analysis. Since there are a large number of binaries compiled from the same source code in programs with different CPU architectures, homology-based binary code similarity comparison methods have been studied for vulnerability discovery. Pewny et al. [12] transformed different instruction codes into an intermediate language, used hash to generate a summary of the basic blocks of vulnerabilities, and combined it with the control flow structure for graph matching to achieve accurate homology-based vulnerability discovery. However, the work requires feature extraction of the input and output of each basic block, which has a large performance overhead. Eschweiler et al. [13] proposed a graph matching method based on lightweight syntactic features such as the number of arithmetic and invocation instructions and used function-level feature preanalysis before matching to improve the search efficiency. The work still relies on the graph matching model, which has a large overhead, and performance bottlenecks are not completely solved. Therefore, Feng et al. [14] proposed a firmware homology vulnerability discovery method based on ACFG embedding matching, which can quickly discover cross-platform IoT device firmware

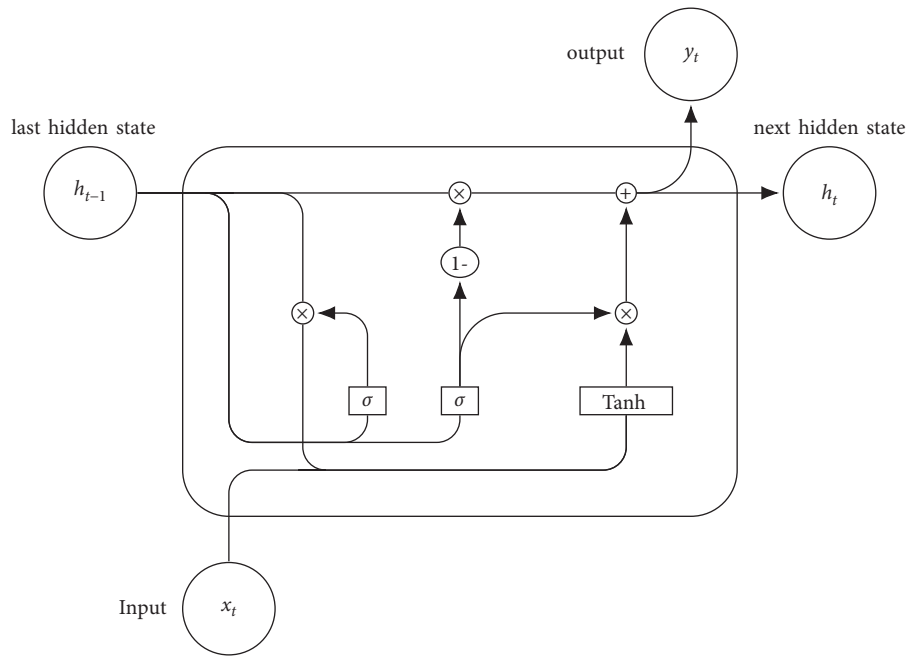

Figure 1: Internal structure of GRU.

vulnerabilities by encoding ACFG into feature vectors through a codebase and indexing them using local sensitive hashes. This work achieves rapid discovery of homologous vulnerabilities for certain scale IoT devices, but there is still a large time overhead in embedding the required codebase in the training graph, and the quality of the codebase is affected by the training set. To address this problem, Xu et al. [5] proposed a neural network-based cross-platform similar code detection method, where the training time is reduced from 1 week to 30 min 10 h. Compared with the above methods, it can identify more additional vulnerability codes.

3. GRU-Based Graph Embedding

3.1. Extraction and Vectorized Representation of ACFG. We use IDA pro to firstly disassemble the binary code into assembly code, extract the ACFG of the function in the process of disassembly, then numerically represent each code block in the ACFG according to the seven attributes listed in Table 1, and finally get the corresponding numerical ACFG of the function code. Six of the seven code attributes selected in this paper are block-level attribute information, which is the inherent information of the code, and one is interblock attribute information, which represents the structural information between a code block and other code blocks; they all have cross-platform properties. Finally, the vectorized ACFG is used as the input of the neural network, and an example of an ACFG extraction is shown in Figure 2.

3.2. Vertex Embedding Generation. Since the two basic components of ACFG are vertices and edges, in the process of generating embeddings for each vertex in the graph, neural networks are used to learn the corresponding vertex information and the edge information related to the vertex.

Table 1: Basic block attributes.

Type	Attribute name
Block-level attributes	String constants
	Numeric constants
	No. of transfer instructions
	No. of calls
	No. of instructions
	No. of arithmetic instructions
Interblock attributes	No. of offspring

The embedding is generated for each vertex in the ACFG through T iterations calculation; the purpose is to better learn the topological structure of the ACFG. In each iteration of the calculation process, in addition to inputting the current node to the neural network, it also inputs several nodes in the graph that are connected to the current node into the neural network for learning because the connections between the vertices in the graph reflect the control flow information of each code block in the function code. After several iterative calculations, the neural network will generate an embedding for each vertex in the ACFG. The embedding not only learns the information about the vertices but also learns the long-distance vertex dependence information in the graph.

The formula for the embedding of each vertex during each iteration of the calculation is shown as follows:

$$\mu_v^{(t+1)} = \tanh\left(W_1 x_v + \sigma\left(\sum_{u \in N(v)} \mu_u^{(t)}\right)\right), \quad \forall v \in V, \tag{7}$$

where $N(v)$ represents the set of nodes adjacent to the current computing vertex v in ACFG, $\mu_v^{(t+1)}$ represents the embedding of vertex v in the $t+1$ iteration calculation, and $\sum_{u \in N(v)} \mu_u^{(t)}$ represents a linear summation computation of

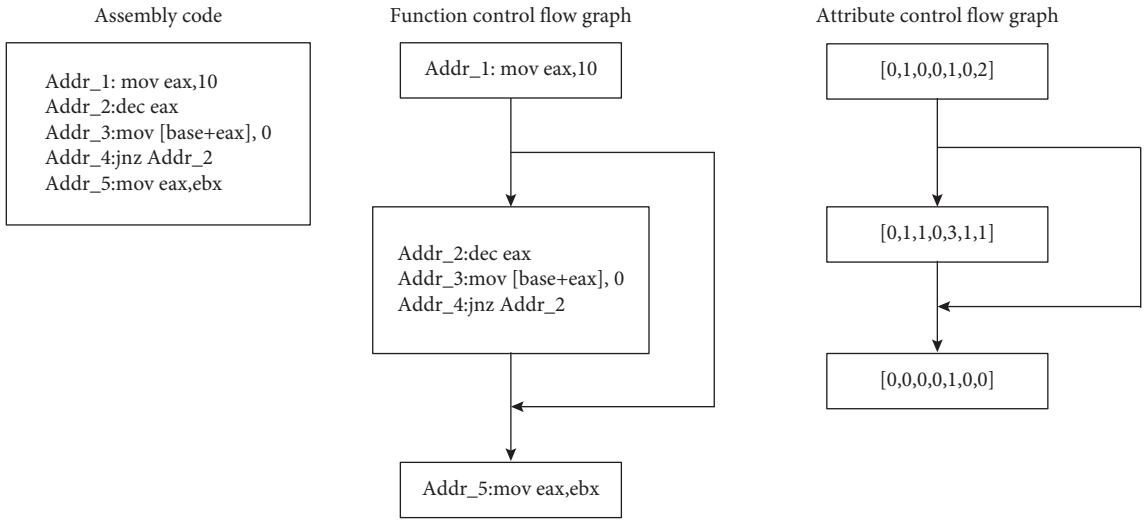

Figure 2: Example of ACFG extraction.

all the vertices associated with the current vertex in the graph, the calculation result of which is used as the input of nonlinear mapping together with the current node. From the above formula, the calculation of vertex embedding is based on the synchronization calculation process of the ACFG topology because the embedding calculation in the next iteration will be performed after the embedding of all vertices in the previous iteration is generated. The deeper the iteration level, the further the vertex information travels along with the graph topology. The algorithm for vertex embedding generation is shown in Algorithm 1.

3.3. Vertex Embedding Aggregation. After T iterations of computation, all vertex embeddings are input into the GRU recurrent neural network according to different time steps, and the hidden layer state output at the last moment will be used as the graph embedding of the ACFG. The graph embedding generation process based on the GRU cyclic neural network is shown in Figure 3.

The basic idea of the GRU-based graph embedding method is to use it to learn the structural information of the ACFG, i.e., the sequence information of the vertices. In the process of generating the vertex embedding, a fully connected neural network is used to learn the vertex information and the topological information of the graph, and then a GRU recurrent neural network is used to learn the hierarchical arrangement information of the vertices in this ACFG to learn the structural information of the function ACFG with cross-platform nature, and finally the state of the hidden layer in the last time step is used as the graph embedding of the ACFG.

3.4. Training Neural Networks with Siamese Architecture. Traditional graph embedding neural network methods are mostly used to solve classification problems; however, the goal of this paper is to perform similarity detection. We perform correlation analysis by combining two identical graph embedding neural networks into a Siamese network structure, the entire correlation framework can make the ACFG of two homologous similar function codes close to each other, and the entire network model can be trained end-to-end to perform similarity detection.

We assume that there exists a relation π that determines the degree of similarity of the codes; given two binary functions f_1 and f_2, then $\pi(f_1, f_2) = 1$ means that they are homologously similar, and $\pi(f_1, f_2) = -1$ means that they are not homologously similar. The core of the code similarity detection problem is to find a mapping that can map a function's ACFG to a numerical vector, so this mapping should supposedly capture enough information with cross-platform characteristics. In this paper, a Siamese architecture consisting of two identical graph embedding neural networks is used to perform the mapping from an ACFG to a numerical vector.

The Siamese architecture takes a pair of ACFG as inputs and then uses a large dataset for end-to-end training to generate a numerical embedding vector for each input ACFG and finally computes the cosine similarity of the two embedding vectors. The neural network embedded in Siamese architecture is shown in Figure 4.

In the figure, g_1 and g_2 are the ACFG extracted from a binary function code to be detected. μ_1 and μ_2 are the numerical embedding vectors generated by the graph embedding neural network for each ACFG. We use cosine similarity to calculate the distance between two vectors, and the calculation formula is as follows:

$$\cos(\mu_1, \mu_2) = \frac{\mu_1 \cdot \mu_2}{\|\mu_1\| \cdot \|\mu_2\|}. \tag{8}$$

In the process of dataset construction, each training sample is composed of a pair of numerical ACFG and a label; if the two binary function codes are homologously similar, the label of the training sample is +1; otherwise, the training label of the sample is −1. After calculating the cosine distance between the two graph embedding vectors, the backpropagation algorithm of the neural network is used to update the parameter weights of each network layer.

(i) Input: ACFG $g = \langle V, E \rangle$.
(ii) **Output:** A sequence of $\{\mu_1^{(T)}, \mu_2^{(T)}, \ldots, \mu_n^{(T)}\}$.
(1) : $\mu_v^{(0)} = 0$, $v \in V$
(2) : **for** $t = 1$ to T **do**
(3) : **for** v in V **do**
(4) : $l_v = \sum_{u \in N(v)} \mu_u^{(t-1)}$
(5) : $\mu_v^{(t)} = \tanh(W_1 x_v + \sigma(l_v))$
(6) : **end for**
(7) : **end for**
(8) : **return** $\{\mu_1^{(T)}, \mu_2^{(T)}, \ldots, \mu_n^{(T)}\}$

Algorithm 1: Vertex embedding generation.

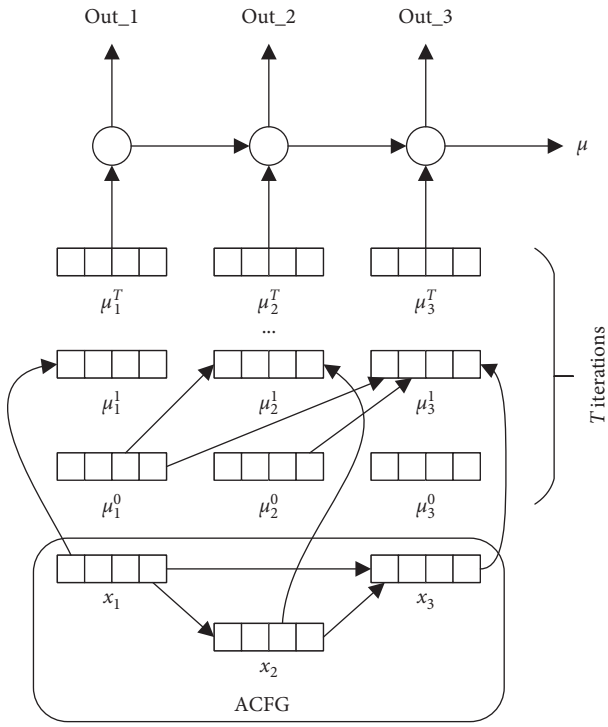

Figure 3: GRU network-based graph embedding generation.

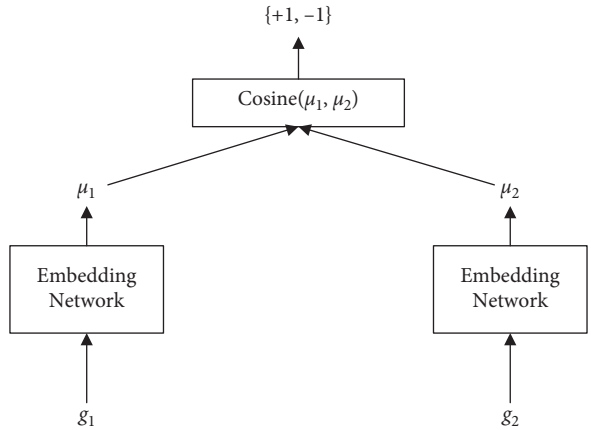

Figure 4: Neural networks embedded in Siamese architecture.

Table 2: The number of ACFG in the dataset.

Platform	Training	Validation	Testing
x86	30,994	3,868	3,973
MIPS	41,477	5,181	5,209
ARM	30,892	3,805	3,966
Total	103,363	12,854	13,148

4. Experimental

4.1. Datasets and Hyperparameters. Since training graph embedding in the neural network requires a large number of homologous and nonhomologous function pairs, it is considered that two pieces of binary code compiled from the same source code are homologous and similar, and vice versa. This paper uses the same large-scale dataset that Gemini uses (called Dataset I in their paper); the training dataset is compiled into binary code by using the GCC compiler, which is open to the OpenSSL family of toolkits that randomly extract several versions of the high-level language source code on different platforms and with different compilation options, and finally extracts a total of 129,365 attributes ACFG. To test the generalization ability of the graph-embedded neural network proposed in this paper, all the ACFG are divided into three disjoint subsets for training, validation, and testing, avoiding the case where binary codes belonging to the same source are partitioned into different collections. This is shown in Table 2.

In this paper, the Adam optimization algorithm is used to minimize the loss function while setting a learning rate of 0.0001, training for 50 epochs, generating graph embeddings with a dimension of 64, and generating vertex embeddings with an iteration of 5, and each batch contains 10 ACFG pairs with attributes to be detected.

4.2. Evaluation Indicators. The main objective of this paper is to detect if two binary codes from different platforms are homologous, so deep learning is used in this topic to solve the binary classification problem. The common evaluation metrics used in the deep learning application problem of binary classification are accuracy, true negative rate, recall, AUC, and so on.

Predictions based on the neural network classifier for samples in the test dataset can be classified into four cases.

(i) True Positive (TP) means that positive cases will be predicted as positive classes

(ii) False Positive (FP) means that negative cases will be predicted as positive classes

(iii) True Negative (TN) means that negative cases will be predicted as negative classes

(iv) False Negative (FN) means that positive cases will be predicted as negative classes

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{P + N}, \tag{9}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{10}$$

$$\text{TNR} = \frac{\text{TN}}{\text{FP} + \text{TN}}. \tag{11}$$

Formula (8) is the formula for the accuracy rate, where P and N represent the number of positive and negative categories, respectively, i.e., the number of correctly classified samples as a proportion of the total number of samples. Accuracy has good evaluation ability on a balanced dataset. Equations (9) and (10) are the formulas for calculating the true rate and true negative rate, respectively, and they can be used as a pretrained neural network's ability to discriminate between positive and negative samples.

Homology detection is a binary classification problem, and in this paper, the output of the supervised learning twin network is the cosine distance of the two attribute control flow graph embedding vectors with values in the range [−1, 1], so if the output of the neural network is greater than 0, it is considered to be predicted as a positive case, otherwise as a negative case.

In this section, comparing our proposed GRU network-based graph embedding with the vector linear aggregation-based graph embedding in Gemini and using the same training dataset for training the same number of epochs, respectively, as well as the same set of tests to finally evaluate the model, the performance of the three different experiments will be evaluated and compared in terms of multiple evaluation metrics below. In the following diagram, linear and GRU will be used to represent the two algorithms based on vector linear clustering and GRU network, respectively.

As shown in Figures 5 and 6, each graph contains 2 polylines, which represent two different graph embedding algorithms of linear and GRU, the horizontal coordinate is the training period epoch, and the vertical coordinate is the TNR and accuracy values, respectively. Each experiment is trained for 50 epochs, and the corresponding values are recorded for every 5 epochs, and finally a trend line graph of the whole training process is plotted. From the graph, it can be seen that the prediction ability of the three algorithms gradually becomes better as the number of training epochs increases. The proposed GRU network-based vector aggregation graph embedding algorithm in this paper outperforms the linear-based vector aggregation graph embedding algorithm in both TNR and accuracy. After training the neural network with 50 epochs, the GRU-based graph embedding algorithm outperforms the vector aggregation-based algorithm by 5% in both metrics.

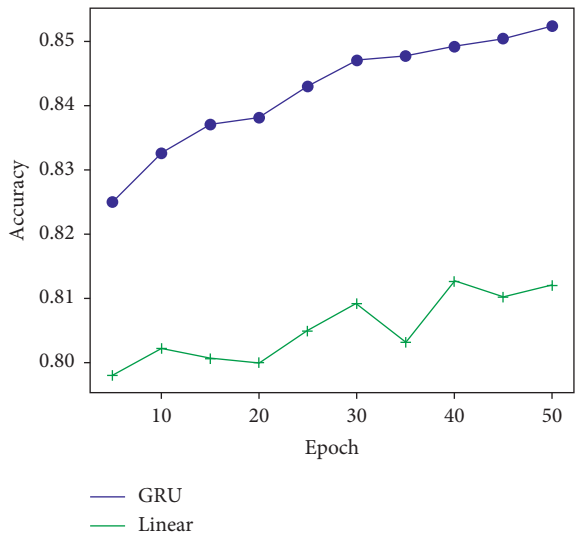

Figure 5: Accuracy comparison between GRU and linear.

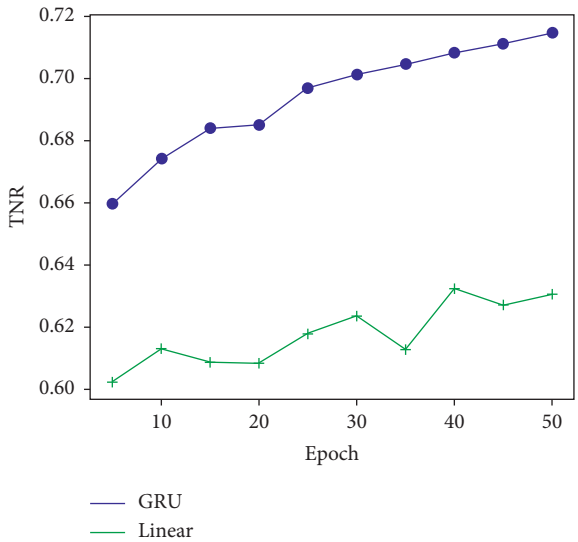

Figure 6: TNR comparison between GRU and linear.

5. Conclusion

In this paper, we propose a GRU-based graph embedding generation method to embed ACFG of different platform binary function codes into vectors to compare the similarity of cross-platform binary functions. Since the GRU network can effectively learn the sequence information of the vectors, based on this theory, the generated vertex embedded vectors are input into the GRU network at different time steps, thus learning the structural information in the ACFG with cross-platform nature. The experimental results show that the Siamese network consisting of two GRU-based graph embedding networks has a higher detection accuracy than the fully connected network-based Siamese network, and our proposed method has a better ability to discriminate between positive and negative samples.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors thank the Information Countermeasure Technique Institute, Harbin Institute of Technology, for the 1080Ti graphics card. This work was supported by National Defense Basic Scientific Research Program of China (grant number. JCKY2018603B006).

References

[1] I. Santos, F. Brezo, X. P. Ugarte, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013.

[2] S. Gupta, H. Sharma, and S. Kaur, "Malware characterization using windows api call sequences," in *Proceedings of the International Conference on Security, Privacy, and Applied Cryptography Engineering*, pp. 271–280, Springer, Salmon Tower Building NY, USA, December 2016.

[3] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen, "Contextual weisfeiler-lehman graph kernel for malware detection," in *Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 4701–4708, IEEE, Vancouver, BC, Canada, July 2016.

[4] D. Eppstein, "Subgraph isomorphism in planar graphs and related problems," in *Graph Algorithms and Applications I*, pp. 283–309, World Scientific, Singapore, 2002.

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

[6] L. Song, *Structure2vec: Deep Learning for Security Analytics over Graphs*, USENIX Association, Atlanta, GA, USA, 2018.

[7] K. Cho, B. V. Merriënboer, C. Gulcehre et al., "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014, https://arxiv.org/abs/1406.1078.

[8] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," in *Proceedings of the 1999 Ninth International Conference on Artificial Neural Networks ICANN 99*, Edinburgh, UK, September. 1999.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "Light gated recurrent units for speech recognition," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 2, pp. 92–102, 2018.

[11] Y. Su and C. C. K. Jay, "On extended long short-term memory and dependent bidirectional recurrent neural network," *Neurocomputing*, vol. 356, pp. 151–161, 2019.

[12] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 709–724, IEEE, San Jose, CA, USA, May 2015.

[13] S. Eschweiler, K. Yakdan, and E. P. Gerhards, "Discovre: efficient cross-architecture identification of bugs in binary code," in *Proceedings of the NDSS*, vol. 52, pp. 58–79, February 2016.

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