VDHGT: A Source Code Vulnerability Detection Method Based on Heterogeneous Graph Transformer

Hongyu Yang[1(✉)], Haiyun Yang[2], and Liang Zhang[3]

[1] School of Safety Science and Engineering, Civil Aviation University of China, Tianjin 300300, China
yhyxlx@hotmail.com
[2] School of Computer Science and Technology, Civil Aviation University of China, Tianjin 300300, China
[3] School of Information, The University of Arizona, Tucson, AZ 85721, USA

Abstract. Vulnerability detection is still a challenging problem. The source code representation method used by the existing vulnerability detection methods cannot fully contain the context information of the vulnerability occurrence statement, and the vulnerability detection model does not fully consider the importance of the context statement to the vulnerability occurrence statement. Aiming at the problems raised above, this paper proposes a source code vulnerability detection method based on the heterogeneous graph transformer. The method proposed in this paper adopts a novel source code representation method—**the vulnerability dependence representation graph**, which includes the control dependence of the vulnerability occurrence statement and the data dependence of the variables involved in the statement. At the same time, this paper builds a graph learning network for vulnerability dependence representation graph based on the heterogeneous graph transformer, which can automatically learn the importance of contextual sentences for vulnerable sentences. To prove the effectiveness of the method in this paper, experiments were carried out on the SARD data set, and the average accuracy rate was 95.4% and the recall rate was 92.4%. The average performance is improved by 4.1%–62.7%.

Keywords: Vulnerability detection · Code representation · Heterogeneous graph transformer · Graph neural network · Code representation graph

1 Introduction

In the past decades, a large number of vulnerability detection methods have been proposed [1]. Most of the methods rely on vulnerability characteristics or template rules manually defined by security experts. These methods not only have a high false positive rate, but also cover a limited number of vulnerabilities, but also template rules need to be dynamically updated with the emergence of new vulnerability patterns [2, 3].

X. Chen et al. (Eds.): CSS 2022, LNCS 13547, pp. 217–224, 2022.
https://doi.org/10.1007/978-3-031-18067-5_16

Sequence-Based Method. Russell et al. [4] used the feature extraction method of sentence emotion classification based on CNN and RNN to detect function-level source code vulnerabilities. Li et al. [5] believe that whether a line of code contains vulnerabilities is related to the context, and the parameters of function calls in the program are often affected by the early or middle, and late stages of the program. Therefore, [5] selects bidirectional long short-term memory (BLSTM) to model the source code. The SySeVR method proposed in reference [2] believes that not every line in the source code is related to the vulnerability characteristics. It selects four points of interest (function call, array index, expression calculation, pointer reference) for code slicing, and then selects neural network models such as bidirectional gate recurrent unit (BGRU), BLSTM, CNN, and so on to model the source code slicing.

Graph-Based Method. Allamanis et al. [6] believe that using the natural language processing (NLP) method for source code vulnerability detection can not solve the problem of long dependence caused by using the same variables and functions far away from the statements where the vulnerability occurs. Therefore, the source code is represented as a joint graph of abstract syntax code (AST), data flow graph (DFG), and control flow graph (CFG). Based on the gated graph neural network [7] (GGNN), the effectiveness of using a graph to represent source code is proved in two scenarios: variable name prediction and variable name misuse detection. The Devign method proposed in [8] adds a natural sequence edge of the code to encode the natural sequence of the source code and uses the code representation of the GGNN learning program joint graph with the convolution module for vulnerability detection.

In the above two methods, sequence-based methods either do not contain syntactic-semantic information or do not explicitly maintain any dependencies in source code slices, making it difficult to learn and reason about source code semantic information. The graph-based method simply uses one or more representation graphs of the source code to represent the source code and does not further analyze the nodes and edges related to the vulnerability statement in the representation graph, so that the model cannot comprehensively learn and reason about the relevant information of the vulnerability statement.

This paper proposes a source code vulnerability detection method based on the heterogeneous graph transformer (VDHGT). VDHGT has the following characteristics:

(1) we believe that the source code statements with vulnerabilities are closely related to the data dependence of variables in the statements and the control structure corresponding to the statements. Therefore, this paper proposes a code representation method named Vulnerability Dependence Representation Graph (VDRG). VDRG distinguishes the dependent edges related to vulnerabilities and the unrelated edges.
(2) VDRG is a heterogeneous graph. Considering that the four different edges may have different characteristic dimension distributions, And there are differences in the importance of VDRG, so heterogeneous graph transformer (HGT) [9] is used to learn and infer the relationship between source code nodes.

2 Overview

The proposed method VDHGT is divided into four stages: VDRG generation stage, nodes embedding stage, graph learning network stage, and vulnerability detection stage. The overall framework of VDHGT is shown in Fig. 1.

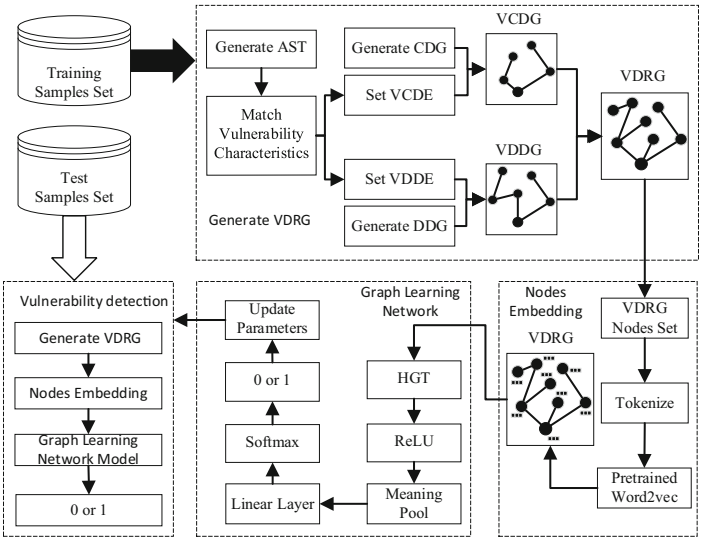

Fig. 1. The framework of VDHGT.

3 VDHGT Method

3.1 Generation of VDRG

The representation of VDRG for function-level source code proposed in this paper can be formally recorded as $f_{vdrg} = (V, E, X, D)$, where V represents the node set and E represents the edge set. X represents the attribute of the node set and saves the node type. The node type can be divided into two types by whether it contains SyVC [2] or not. D represents the attribute of the edge set and saves the edge type. The edge types are divided into four types.

Algorithm1 gives a high-level description of the VDRG generation process. The main idea of algorithm1 is to generate the corresponding VDRG by analyzing the control dependence and data dependence of each SyVC in the function code slice. The input of the algorithm is the function code slice, and the output is the VDRG set.

Algorithm 1 VDRG generation algorithm

Input: Funciton $f = \{s_1, .., s_n\}$,s_i Represents a source code statement

Output: A set S with VDRG as element

$S \leftarrow \emptyset$;
Generate the f_{ast} corresponding to f
Generate the SyVC set S_{SyVC} corresponding to f_{ast} by matching vulnerability syntax features
Generate f_{cdg} corresponding to f
Generate f_{cdg} corresponding to f
for each $SyVC \in S_{SyVC}$ **do**
    Calculate the statement s_{pvs} where SyVC is located
    $f_{vcdg} \leftarrow f_{cdg}$
    $f_{vddg} \leftarrow f_{ddg}$
    $S_{pvs} \leftarrow \emptyset$
    $S_{pvs} \leftarrow S_{pvs} \cup s_{pvs}$
    while S_{pvs} **do**
        Remove element s_{pvs} from S_{pvs}
        for each $edge_{vcdg} \in f_{vcdg}$ **do**
            if $edge_{vcdg}.dst == s_{pvs}.dst$ *and* $edge_{vcdg}.label \neq$'vcde' **then**
                $S_{pvs} \leftarrow S_{pvs} \cup edge_{vcdg}.src$;
                $edge_{vcdg}.label \leftarrow$ 'vcde'
            else if $edge_{vcdg}.label \neq$'vcde' **then**
                $edge_{vcdg}.label \leftarrow$ 'cde'
            end if
        end for
    end while
    for each $var \in s_{pvs}$ **do**
        for each $edge_{vddg} \in f_{vddg}$ **do**
            if $var.name == edge_{vddg}.label$ *and* $edge_{vddg}.label \neq$'vdde' **then**
                $edge_{vddg}.label \leftarrow$ 'vdde'
            else if $edge_{vddg}.label \neq$'vdde' **then**
                $edge_{vddg}.label \leftarrow$ 'dde'
            end if
        end for
    end for
    $f_{vdrg} \leftarrow f_{vcdg} \cup f_{vddg}$;
    $S \leftarrow S \cup f_{vdrg}$
end for

3.2 Node Embedding

The goal of the node embedding stage is to generate an initial representation vector for each node of the VDRG graph. The node embedding phase takes VDRG as the input and goes through the following steps:

(1) Tokenization: in this step, each source code in the data set is divided into a combination of single coincidence numbers. In order not to let the variable name and function name affect the learning of semantic relationships in the program, the variables in each sample are uniformly named *var1*, *var2*, ……, The functions are uniformly named *fun1*, *fun2*, ……

(2) Training word2vec: The phrase generated by the marked function slice is used as the input of word2vec to train a word2vec word encoder.
(3) Generate the node vector representation of VDRG: input the code statement corresponding to each node into the pre-trained word2vec, and obtain the initial vector representation of the node according to the determined parameters through calculation.

3.3 Graph Learning Network and Vulnerability Detection

This paper designs a VDRG graph learning network for source code vulnerability detection based on the heterogeneous graph transformer. The VDRG graph learning network minimizes the loss through the training set and obtains the optimal parameters by fitting the model. After the VDRG generation and node embedding stage, the test sample is input into the pre-trained VDRG graph learning network model in the vulnerability detection stage to obtain the detection result, that is, whether there is a vulnerability in the function slice. Algorithm 2 gives the calculation of VDRG.

Algorithm 2 Calculate the VDRG global representation

Input: $G_{VDRG} = \{V, E, X, D\}$
Output: VDRG global representation H_{VDRG}

$h \leftarrow 2$
for *each* $t \in V$ **do**
  for $s \in N(t)$ **do**
    for $i = 0; i <= h; i++$ **do**
      $K^i(s) \leftarrow K\_Linear^i_{\tau(s)}(H^{init}[s])$
      $Q^i(t) \leftarrow Q\_Linear^i_{\tau(t)}(H^{init}[t])$
      $e \leftarrow (s, t)$
      $ATT\_head^i(s, e, t) \leftarrow (K^i(s) W^{ATT}_{\phi(e)} Q^i(t)^T) \cdot \frac{\mu_{<\tau(s),\phi(e),\tau(t)>}}{\sqrt{d}}$
    end for
    $\mathbf{Attention}_{HGT}(s, e, t) \leftarrow Softmax(\underset{\forall s \in N(t)}{\|} \underset{i \in [1,h]}{} ATT\_head^i(s, e, t))$
    for $i = 0; i <= h; i++$ **do**
      $MSG\_head^i(s, e, t) \leftarrow M\_Linear^i_{\tau(s)}(H^{init}[s]) W^{MSG}_{\phi(e)}$
    end for
    $\mathbf{Message}_{HGT}(s, e, t) \leftarrow \underset{i \in [1,h]}{\|} MSG\_head^i(s, e, t))$
  end for
  $\widetilde{H}[t] \leftarrow \underset{\forall s \in N(t)}{\oplus} (\mathbf{Attention}_{HGT}(s, e, t) \cdot \mathbf{Message}_{HGT}(s, e, t))$
  $H[t] \leftarrow A\_Linear_{\tau(t)}(\sigma(\widetilde{H}[t])) + H^{(init)}[t]$
  $H[t] \leftarrow Relu(H[t])$
end for
$H_{VDRG} \leftarrow \underset{\forall t \in V}{\oplus} H[t]$

4 Experiment and Result Analysis

4.1 Experimental Dataset

The dataset used in this paper is the Software Assurance reference dataset [10] (SARD). The experiment designed in this paper selects the following types of vulnerability datasets from SARD. The specific information is shown in Table 1.

Table 1. Types of source code vulnerabilities contained in the experimental dataset.

CWE type	Name	Counts
CWE78	OS injection	17000
CWE80	XSS	3400
CWE89	SQL injection	14000
CWE129	Improper array index validation	11000
CWE190	Integer overflow	25000
CWE400	Resource exhaustion	7800
CWE789	Out of control memory allocation	7300

4.2 Experimental Results and Analysis

The results of the comparative experiment are shown in Table 2. From the results, it can be seen that the VDHGT method proposed in this paper is superior to other methods in four indexes. The method proposed by Russell et al. processes the source code text into character sequences without considering the context of source code statements. At the same time, it uses the method based on NLP to model the source code, which loses the structured syntax and semantic information in the source code. Therefore, the index value corresponding to the experimental results is the lowest. Vuldeepecker also processes the source code as a sequence model, but it uses BLSTM to build a vulnerability detection model. BLSTM will consider the impact of the pre and post-code statements, which makes its experimental indicators better than those of Russell et al. SySeVR put forward the concepts of program interest points and vulnerability syntax features for the first time and eliminates the statements irrelevant to the vulnerability. Although it also uses the sequence model, it filters the invalid information to a great extent and obtains a more accurate model by learning the dependence information of the vulnerability. Devign is a classic method of modeling the source code as a graph and then using GNN for vulnerability detection. It integrates the AST, CFG, DFG, and natural sequence information of the source code into the program representation diagram, which greatly improves the performance of vulnerability detection.

The reason why the VDHGT method proposed in this paper is superior to other methods is that it not only considers the control dependence and data dependence information of source code statements but also distinguishes the control dependence edges related to and unrelated to vulnerability statements through SyVC so that the later neural network can learn the context of vulnerability more accurately. At the same time, this paper uses the heterogeneous graph transformer to exchange neighbor information of VDRG. It maintains proprietary linear transformation for each VDRG edge type and node type, which greatly enhances the expression ability of the model.

Table 2. Comparison of detection and evaluation indexes between VDHGT method and classical methods

Methods	A	P	R	F1
Russell et al.	78.3	70.6	47.4	56.8
VulDeePecker	84.4	83.2	60	69.8
SySeVR	86.1	84.9	65.1	73.7
Devign	93.4	89.9	87.8	88.8
VDHGT	95.4	91.4	93.3	92.4

5 Conclusion

This paper proposes a vulnerability detection method based on the heterogeneous graph transformer. It obtains the potential vulnerability statements by matching the vulnerability syntax features on the abstract syntax tree corresponding to the function slice and then constructs the vulnerability control dependence graph by analyzing the control dependence of the potential vulnerability statements and the data dependence of the variables contained therein. This source code representation method accurately depicts the context of the vulnerability statement and provides sufficient data basis for the training of the neural network. At the same time, this paper constructs the graph neural network based on the heterogeneous graph transformer. It can maintain different feature dimension distribution for different edge types and node types and learn the key context affecting vulnerability statements through the multi-head attention mechanism, to further improve the accuracy of vulnerability detection.

References

1. Lin, G.J.F., Wen, S.S., Han, Q.L.T.: Software vulnerability detection using deep neural networks: a survey. Proc. IEEE **108**(10), 1825–1848 (2020)
2. Li, Z.F., Zou, D.Q.S., Xu, S.H.T.: SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Depend. Secure Comput. 1–15 (2021)
3. Russell, R.F., Kim, L.S., Hamilton, L.T.: Automated vulnerability detection in source code using deep representation learning. 17th IEEE International Conference on Machine Learning and Applications 2018, ICMLA, vol. 122018, pp. 757–762. IEEE, Piscataway (2018)
4. Wang, H.T.F., Ye, G.X.S., Tang, Z.Y.T.: Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inform. Foren. Secur. **16**, 1943–1958 (2021)
5. Li, Z.F., Zou, D.Q.S., Xu, S.H.T.: Vuldeepecker: A Deep Learning-Based System for Vulnerability Detection. arXiv preprint arXiv 1801, pp. 1681–1695 (2018)
6. Allamanis, M.F., Brockschmidt, M.S., Khademi, M.T.: Learning to Represent Programs with Graphs. arXiv preprint arXiv 1711, pp. 740–756 (2017)
7. Li, Y.J.F., Tarlow, D.S., Brockschmidt, M.T.: Gated Graph Sequence Neural Networks. arXiv preprint arXiv 1511, pp. 5493–5512 (2015)

8. Zhou, Y.Q.F., Liu, S.Q.S., Siow, J.K.T.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. **32**, 1–11 (2019)
9. Hu, Z.N.F., Dong, Y.X.S., Wang, K.S.T.: Heterogeneous graph transformer. In: Proceedings of The Web Conference 2020, WWW, vol. 04202020, pp. 2704–2710.Association for Computing Machinery, New York (2020)
10. NVD: Software assurance reference dataset (2018). https://samate.nist.gov/SRD/index