Liu *et al. Cybersecurity* (2023) 6:29
https://doi.org/10.1186/s42400-023-00160-1

Cybersecurity

RESEARCH

Open Access

Graph neural network based approach to automatically assigning common weakness enumeration identifiers for vulnerabilities

Peng Liu[1,4], Wenzhe Ye[1,4], Haiying Duan[2], Xianxian Li[1,4*], Shuyi Zhang[1,4], Chuanjian Yao[1,4] and Yongnan Li[3]

Abstract

Vulnerability reports are essential for improving software security since they record key information on vulnerabilities. In a report, CWE denotes the weakness of the vulnerability and thus helps quickly understand the cause of the vulnerability. Therefore, CWE assignment is useful for categorizing newly discovered vulnerabilities. In this paper, we propose an automatic CWE assignment method with graph neural networks. First, we prepare a dataset that contains 3394 real world vulnerabilities from Linux, OpenSSL, Wireshark and many other software programs. Then, we extract statements with vulnerability syntax features from these vulnerabilities and use program slicing to slice them according to the categories of syntax features. On top of slices, we represent these slices with graphs that characterize the data dependency and control dependency between statements. Finally, we employ the graph neural networks to learn the hidden information from these graphs and leverage the Siamese network to compute the similarity between vulnerability functions, thereby assigning CWE IDs for these vulnerabilities. The experimental results show that the proposed method is effective compared to existing methods.

Keywords Vulnerability categorization, CWE, Graph representation, GNN

Introduction

Due to the increasing reliance on software in modern society and the increasing number of newly reported software vulnerabilities every day, software vulnerabilities have become an important issue in network security. The MITRE Corporation has presented CVE (Common Vulnerabilities and Exposures) (https://cve.mitre.org/) since January 1999, a public vulnerability database to report vulnerabilities discovered in various software including operating systems and web browsers. As of September 15, 2022, 184,784 vulnerabilities have been reported according to CVEDetails (https://www.cvedetails.com/). To better understand these vulnerabilities, CWE (https://cwe.mitre.org/), short for Common Weakness Enumeration, was proposed to classify vulnerabilities, which denotes the common weakness shared by a set of vulnerabilities.

Generally, in a vulnerability report, a vulnerability is assigned a unique CVE ID and is categorized into one CWE. For example, the famous Heartbleed vulnerability is assigned with CVE-2014-0160, and is categorized into CWE-119, which denotes failure to constrain operations within the bounds of a memory buffer. Note that the CWE ID reveals the common cause of many vulnerabilities sharing the same weakness. CVE-2018-14438 and CVE-2017-3733 are two vulnerabilities in different programs, but they both belong to CWE-20, which states that the program does not validate or incorrectly

*Correspondence:
Xianxian Li
lixx@gxnu.edu.cn
[1] Key Lab of Education Blockchain and Intelligent Technology, Ministry of Education, Guangxi Normal University, Guilin 541004, China
[2] School of Software, Beihang University, Beijing 100000, China
[3] School of National Security, People's Public Security University of China, Beijing 1000000, China
[4] Guangxi Key Lab of Multi-Source Information Mining and Security, Guangxi Normal University, Guilin 541004, China

validates the input. Categorizing vulnerabilities can help us identify the cause of weakness, which further guides us to fix the vulnerability quickly. Therefore, it is critical to assign an accurate CWE ID for a newly discovered vulnerability.

Currently, CWE assignment is performed by security experts. However, according to CVEDetails, the number of vulnerabilities was 894 in 1999, and is increased to 20,169 in 2021. The sharp increase definitely imposes heavy work load on security experts. In addition, there exist as many as 927 CWE IDs. Such a number makes manual classification error-prone, especially when some CWE IDs are highly similar, e.g., buffer underflow (CWE-124) and memory buffer errors (CWE-1218). In addition, for some vulnerabilities, the expert is not sure which CWE ID is more appropriate for the vulnerability; as a result, the CWE ID in the vulnerability report is missing, e.g., CVE-2022-33936 (https://cwe.mitre.org/) and CVE-2022-32552 (https://www.cvedetails.com/cve/CVE-2022-32552/), just name a few. To this end, it is becoming important to automatically assign the CWE ID for vulnerabilities.

Recent studies have viewed CWE ID assignment as a classification problem and have explored the use of machine learning methods to classify vulnerabilities (Das et al. 2021) at the level of function for C/C++programs. Generally, they treat the source code of the vulnerability as text and then employ the methods of natural language processing to learn the features from the text for classification. However, a programing language is more logical and structured than the natural language. In addition, the snippets of the vulnerabilities generally take a small piece of the source code to function. Thus, it is difficult for these existing studies to effectively learn semantic information. In addition, these studies perform valuations on synthetic datasets where a number of vulnerabilities are simple and highly similar. Therefore, they perform poorly in real-world scenarios.

In this paper, we aim to propose an automated approach to classify vulnerabilities (i.e., CWE assignment) with deep neural networks. To achieve this goal, we need to address three problems and challenges. First, we must characterize the vulnerability code with an appropriate format that can easily expose its syntax and semantics. Second, we need a deep neural network to learn well from the code representations. Third, since existing vulnerability datasets are simple and contain few vulnerabilities sharing the same CWE, we need to collect a vulnerability dataset to help us reveal the similarities within the same class of CWEs and the differences across different CWEs.

To this end, we propose a new method to assign CWE IDs for vulnerabilities with a graph neural network and Siamese network. The key idea behind this is that the vulnerability snippets sharing the same type of CWE exhibit similar syntax and semantics, which can be exploited by deep neural networks to learn this hidden information for classification. More specifically, we first prepare a dataset that contains 3394 real world vulnerabilities from Linux, OpenSSL, Wireshark and other software. Then, we extract statements with vulnerability syntax features from these vulnerabilities and use program slicing to slice them according to the categories of syntax features. On top of slices, we use Joern, a graph representation of code, to represent these slices with graphs that characterize the data dependency and control dependency between statements. Finally, we employ the graph neural network to learn the hidden information from these graphs and leverage the Siamese network to compute the similarity between vulnerability functions, thereby performing assigning CWE IDs for these vulnerabilities.

On our prepared dataset with 3394 CWEs across 8 CWE types, our method outperforms existing methods such as CNN (Convolutional Neural Network), GCN (Graph Convolutional Networks), LSTM (Long short-term memory) in terms of precision, recall, f1-score and accuracy upon CWE assignment.

To summarize, our main contributions are as follows:

(1) First, we present a dataset consisting of 3394 real world vulnerabilities. Not only the source code of the whole function, but also the vulnerability snippets are pinpointed. We plan to release the dataset after the paper is published.
(2) Second, a graph-based method is proposed to characterize vulnerability from different aspects, which helps reveal the syntactic and semantic information more precisely.
(3) Third, we present a deep neural network model that combines a graph neural network and a Siamese network, where the former learns the hidden information from graphs and the latter computes the similarity between vulnerabilities for classification.
(4) Fourth, a set of experiments is performed to demonstrate the effectiveness of the proposed method.

Related work

In recent years, many approaches have been proposed to classify CVEs into CWEs. DeLooze (2004) proposed CVE classification with four common classes of attacks (Denial of Service, Deception, Reconnaissance, and Unauthorized Access) using a Self-Organizing Map and achieved a 75% success rate on testing data in 2004 before the CWE list was published. Wita and Teng-Amnuay (2005) focused on 388 CVE entries and candidates from

Linux, and manually generated a vulnerability profile for Linux system with the CVE severity level. Neuhaus and Zimmermann (2010) explored the security trend within each cybersecurity cluster derived by the unsupervised Latent Dirichlet Allocation (LDA) learning model upon CVE entries. Rahman and Yeasmin (2013) proposed an adaptive bug classification method on a prepared CVE dataset using Naïve Bayes and Bayes net, which achieved 80.64% and 81.50% of accuracy, respectively. Aivatoglou et al. (2021) used the description of CVEs on the NVD as the dataset and processed the dataset using an NLP-based approach to classify CWEs using three tree-based machine learning models, with the best result being an accuracy of 76% for ten classes of CWEs using XGBoost. Na et al. (2017) also used the descriptions of CVEs on the NVD website to classify CWEs. They selected the largest number of CWE categories for three experiments, namely 1) the top three CWE categories, 2) the top five CWE categories, and 3) the top ten CWE categories, with final accuracy rates of 95.2%, 84.2%, and 75%, respectively. Although Aivatoglou et al. (2021) also used the description of CVE to classify CWE, they proposed the concept of a hierarchical classification of CWE based on the level of CWE and achieved good results. Wang et al. (2022) used word2vec to embed the CVE descriptions, which are then fed into the CNN and GRU models for processing.

Dam et al. (2018) characterized the vulnerability source code in the AST, and then embed the tree nodes of the AST through a tree-based LSTM to obtain a vector representation, which was then fed into a traditional classifier for vulnerability detection. Zhou et al. (2019) used AST and PDG to characterize the vulnerability function as a graph for processing, using statements, identifiers, and direct values as nodes of the graph and relations between nodes as edges, although they use graph neural networks for vulnerability detection. Xiao et al. (2020) detected whether a vulnerability has been fixed in a vulnerability function by characterizing the vulnerability source code in the form of AST and PDG to calculate the signature in the vulnerability function and the patch function. Sun et al. (2021) used similarity comparisons of vulnerable functions for vulnerability detection, they compare the similarity of the vulnerable function with the function that has been fixed. Cui et al. (2020) proposed the concept of a Weighted Featute Graph (WFG) to assign values to nodes in the WFG based on the number of occurrences of the type of AST and to detect the presence of vulnerabilities in the function using the WFG of the vulnerable function and the repaired function. Li et al. (2021b) employed program dependency graph (PDG) to characterize the source code of vulnerability by aggregating five vulnerability-related features, and then feed the PDGs to FA-GCN for vulnerability detection and classification. Our approach is inspired by the above methods, but the differences are: 1) The purpose is different, The goal of the above methods is to detect vulnerabilities, while our goal is to classify vulnerabilities; 2) The code characterization methods are different, The above methods use AST or CFG or treat the source code directly as text, while our approach slices the graph on the basis of PDG to generate graphs with vulnerability features; 3) Our approach embeds the graphs with GCN and then compares the similarity by Siamese network to classify the vulnerability functions.

As seen from the above, the existing research on CWE classification is still based on the NLP approach, rather than classifying CWE based on the vulnerability function, which is the main purpose of the work on vulnerability functions as a dataset in performing vulnerability detection. Different from these methods, we classify CWEs on a dataset of vulnerability source code and characterize the code (mainly the vulnerability snippets) as a graph. By using the graph convolutional network and Siamese network, the semantics of graphs can be well learned and the similarity between two functions is computed to assign newly discovered functions with CWE. The experimental results show the effectiveness of our proposed method.

Background and problems

Background

A vulnerability is a cyber-security term that refers to a weakness, defect, or security bug in computer systems that leaves information security exposed to a threat. It can be exploited by an attacker to gain unauthorized access (e.g., steal sensitive data) or even perform arbitrary actions (e.g., install malware) on a computer system.

CVE (Common Vulnerabilities and Exposures) is a list of publicly known vulnerabilities and exposures. Each CVE ID (in the form of CVE-Year-Number) on the list is a record upon a finding of a specific vulnerability or exposure in software, rather than a general class of vulnerabilities or exposures. For example CVE-2014-0160 denotes the Heartbleed bug in the dtls1 process heartbeat() function of d1both.c and t1 lib.c of OpenSSL-1.0.1. CWE (Common Weakness Enumeration) refers to types of weaknesses, each of which is shared by many vulnerabilities. For example, CWE-416 denotes use after free, which refers to the attempt to access memory after it has been freed (e.g., CVE-2015-2546 and CVE-2020-3947). CWE-415 denotes that double free when free() is called more than once with the same memory address as an argument (e.g., CVE-2015-2419, CVE-2018-8460). Note that many CWEs have a parent–child relationship, e.g., CWE-415 and CWE-416 are children of CWE-825.

Once a vulnerability is newly discovered, it is reported publicly. Many databases, such as CVEDetails NVD, manage the reported vulnerabilities across a broad range of programs. A vulnerability report provides detailed information on vulnerability. As shown in Fig. 1, which comes from the report of CVE-2016-2842, it explains the CVSS (Common Vulnerability Scoring System) Score, which denotes the severity of vulnerability. Then, it describes the impact on confidentiality, integrity and availability. In addition, it provides the type of vulnerability, which denotes it potential harm to the system, e.g., denial of service and overflow. Finally, it provides a CWE ID, e.g., 119 which stands for failure to constrain operations within the bounds of a memory buffer.

As stated before, the CWE ID for a vulnerability may be incorrectly provided or even missing, mainly because i) the number of CWE IDs is as many as 924, ii) manual assignment would make mistakes especially for CWE IDs that are similar, and iii) the vulnerability is complex and cannot find an appropriate CWE ID. Therefore, it is critical to assign CWE ID for a newly discovered vulnerability automatically and accurately.

Problems and challenges

Assigning CWE ID for a newly discovered vulnerability automatically and accurately faces several key problems and challenges.

First, how can vulnerable information be revealed? The vulnerability snippet, which denotes the vulnerable information, is only a few lines of code. It generally takes a small piece in the whole function, which may contain dozens or even hundreds of lines. Therefore, it hides deeply in the source code. Considering that the CWE ID denotes the common weakness for many vulnerabilities, how to reveal these small snippets is critical for accurate CWE assignment.

Second, how can we learn the semantics of vulnerability snippets? The semantics of vulnerability snippets are important for CWE assignment, i.e., understanding the weakness of vulnerability. Although there exist many approaches to learn the semantics of the whole function, how to accurately learn the semantics of a smaller fragment in the view of vulnerability is still a major challenge.

Third, how can high performance be achieved on real world dataset? Different from synthetic datasets, real world datasets generally contain only a few vulnerabilities for each CWE, which further limits the learning ability of deep neural networks. Thus, we need a method to effectively learn syntax and semantics from a few samples and then use them for classification.

Our approach

In this section, we present the basic design of our proposed method. As shown in Fig. 2, Data Collection first prepares a dataset containing real world vulnerabilities, in which each vulnerability is tagged with a CWE ID. On our prepared dataset, Code Characterization represents each vulnerability with a vulnerability graph, that more accurately characterizes the syntactic and structural dependencies by combining code property graphs such as AST, PDG, CFG, etc. Then, the similarity model feeds these graphs into a graph neural network and Siamese network to compute the similarity between them. Finally, CWE Assignment labels the vulnerability with a CWE ID by comparing the similarities. In the following, we will describe each component in detail.

Data collection

Although there exist several well-known vulnerability datasets, we decide to build a new dataset on our own for the following reasons. First, many existing datasets are synthetic (Li et al. 2021a), and some vulnerability

– CVSS Scores & Vulnerability Types

CVSS Score	10.0
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files
Integrity Impact	Complete (There is a total compromise of system integrity. There is a comp compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exi:
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Denial Of Service Overflow
CWE ID	119

Fig. 1 Vulnerability report of CVE-2016-2842 (https://www.cvedetails.com/cve/CVE-2016-2842/)

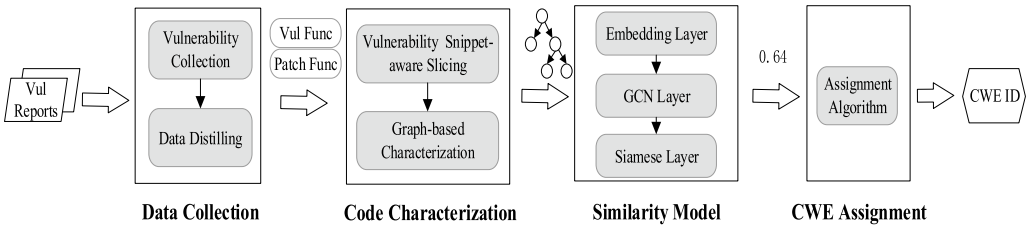

Fig. 2 Overall architecture of our proposed method

functions and vulnerability snippets are highly similar, which amplifies the performance of CWE assignment methods. Second, the number of vulnerabilities of the same CWE is relatively small, i.e., dozens or even fewer. Meanwhile, the vulnerability snippets in real-world programs are generally more diverse, mainly due to different developers having different coding styles. Therefore, we need to collect a dataset that helps reveal the similarities of vulnerabilities that belong to the same CWE, and the differences in vulnerabilities that belong to different CWEs. We prepare the dataset following the steps below.

Vulnerability collection We prepare the dataset from two sources. The first source is the CVEDetails website (https://www.cvedetails.com/cve/CVE-2022-32552/), a well-known vulnerability management system. We employ a web-crawling framework Scrapy to extract the information from the vulnerability reports in CVEDetails, including the CVE-ID, CWE-ID, the file name and function name associated with the vulnerability, and the software version that is affected by the vulnerability. Then, using the function name, file name, and software, the source code of vulnerability at the function level can be extracted using LLVM (Low Level Virtual Machine), a famous open-source compiler (Lattner et al. 2004). Meanwhile, since we collect many different versions of software affected by vulnerability, we therefore obtain many functions for one vulnerability. This will help us to better learn the similarities or differences among vulnerability snippets, as will be detailed in the similarity model ("Similarity Model" Section). In addition, we can also obtain many patched functions from the software with newer versions. Another data source is the Vulncode-db website (https://www.vulncode-db.com/), which records thousands of vulnerabilities as well as their patches. Vulncode-db includes the CVE-ID, CWE-ID, and link of the patch. Similarly, we use the Scrapy framework to crawl the webpage referred to by the patch link to extract the vulnerability function.

Data distilling The prepared dataset may contain redundant functions for the same vulnerability, when the function across different versions of software remains unchanged. These redundant functions would result in the overfitting of the deep learning model. To exclude these functions, we first remove the comments that are irrelevant to the vulnerability and then compute the hash value of the whole function code. For functions with the same hash value, only one copy is kept. In addition, to accurately pinpoint the vulnerability snippets that are important for CWE assignment, we employ the diff tool to find the difference between the vulnerability function and the associated patched function. It is worth noting that the difference may not only record the vulnerability snippets but also contain codes used for code improvement. To this end, we resort to the vulnerability descriptions and manually filter out the snippets where the real vulnerabilities are located to improve the accuracy of snippets.

Code characterization

The vulnerability functions in the prepared dataset are in the form of source code. Although they can be fed into deep learning models such as LSTM (Long short-term memory), the hidden semantics related to vulnerability are difficult to be reveal. To this end, we first use slicing to locate codes more related to the vulnerability snippets so that the smaller pieces of code can more accurately represent the weakness, as shown in Fig. 3. Then, we employ Joern (https://joern.io/) to represent the sliced piece of code with syntax and dependency semantics, which has shown to be effective in characterizing codes. In this way, the characteristics of different types of weakness in the vulnerability function will be highlighted.

Vulnerability snippet-aware slicing We locate the vulnerability snippets when preparing our dataset, which

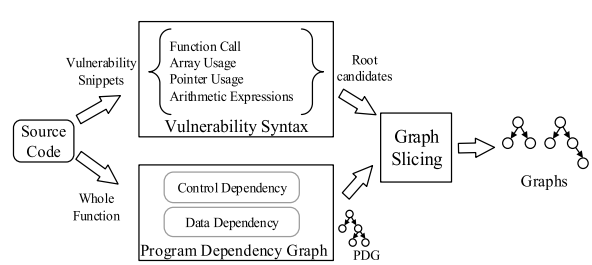

Fig. 3 Code characterization with graphs

are in the form of lines of code. The problem here is that a snippet is still coarse-grained since it contains multiple statements and identifiers. For further location, we extract the vulnerability syntax features from the snippets, e.g., the pointer and sensitive function. More specifically, we use the vulnerability signature statements collected by the code static analysis tool Checkmarx (https://checkmarx.com/), which provides many common vulnerability syntax rules for C/C++ programs. Vulnerability syntax rules contain 93.6% of the vulnerability programs in the NVD database, so the classification of vulnerability syntax rules and the vulnerabilities included are comprehensive. For example, functions including strcpy, strcat, gets and the scanf family are more likely to expose buffer overflow risks. fprintf, snprintf, and syslog may lead to format string problems. exec, system, and popen may lead to potential shell metacharacter dangers. The vulnerability syntax features fall into four categories, i.e., function call, array usage, pointer usage, and arithmetic expressions. Note that slicing from these features may not ensure the existence of vulnerability, but it can significantly exclude unnecessary code statements that are irrelevant to vulnerability. The vulnerability syntax features are extracted by traversing the nodes in the AST (abstract syntax tree) of the function. If the AST node satisfies the syntax statement that implements the defined vulnerability, the statement corresponding to that node will be extracted. Then, rooted from the statement, slicing will be performed to obtain a piece of code, which will be detailed in the next part.

Graph based characterization For each vulnerability function, the graph-based static analysis tool Joern (https://checkmarx.com/) is employed to generate the graph. Here, the graph is a program dependence graph, i.e., PDG, which is the combination of a control flow graph (CFG) and a data flow graph (DFG). In a CFG, a node denotes a basic block of successive statements, and an edge denotes the control transfer between blocks. In a DFG, a node denotes an identifier or variable while an edge denotes the relationship between two identifiers or variables. In this way, the PDG is able to express both the control dependencies and data dependencies, which are useful for exposing the semantics of codes.

On top of the PDG, the previously collected statements matching vulnerability syntax features will be traversed. Upon traversal, slicing is performed to obtain a piece of code that has a data dependency or control dependency relationship with the root statement. More concretely, rooted in the statement, bread first search (BFS) is performed on the graph. One node, i.e., a block, variable or statement, that is directly or indirectly connected to the root node, is reserved. Meanwhile, the other nodes that are unrelated to the root statement will be dropped. Finally, a smaller piece of code is generated. Note that there exist four types of syntax features and each type records multiple statements. Thus, multiple pieces would be generated for each function. To characterize the function concisely, we merge the pieces whose root statements are of the same type of syntax features, and finally obtain four pieces, Each piece corresponds to one type (i.e., function call, array usage, pointer usage, or arithmetic expression). Thus, each vulnerability function can be represented by up to four graphs, each of which characterizes the function (mainly the vulnerability snippets) from a certain aspect.

The process of graph characterization can be seen in Fig. 4. First, the statements of vulnerability features are found in the source code of the function, and then a graph focused on vulnerability features is generated in the PDG. The PDG was generated by the function based on the node where the vulnerability feature statement is located, and this node was the root node, as determined by forward traversal and backward traversal to remove nodes and edges that are irrelevant to the vulnerability feature statement. Figure 4 shows only the process of graph characterization of a vulnerability feature in a function, and a vulnerability function may generate multiple vulnerability feature statements to generate multiple graphs, and then finally aggregate multiple graphs into one graph based on the PDG of the function.

Similarity model

To compute the similarity of two vulnerability functions for CWE assignment, it is important to learn the semantics of graphs of functions. The semantics are distributed in two aspects. One is the semantic information of each node in a graph, which expresses the operational intention of one or several lines of code. The other is the structural information, which expresses the dependencies between codes in terms of both control and data.

To this end, we propose a deep learning model for better learning the semantics of graphs and computing the similarity between two vulnerability functions, as shown in Fig. 4. The model takes two functions as input, each of which is represented by four graphs, and outputs a similarity value. Simply put, it first employs the embedding layer to embed each node of the graph, which learns the operational intention, and then calls the GCN layer to embed the whole graph by learning the structural information. In this way, considering that each function is represented by up to four graphs, the model merges the four graph embeddings into the final function embedding. Finally, it employs the Sia-mese Network to compute the similarity between two function embeddings. The following paragraphs will describe each step in detail (Fig. 5).

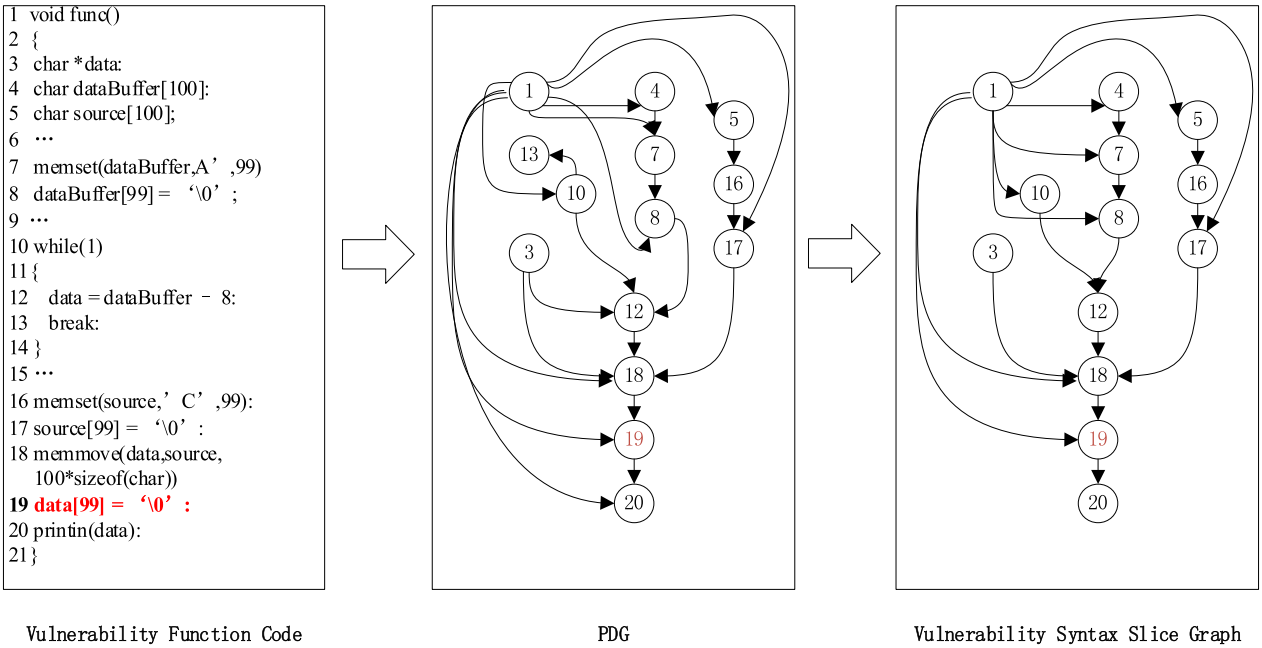

Fig. 4 Code characterization in a slice graph

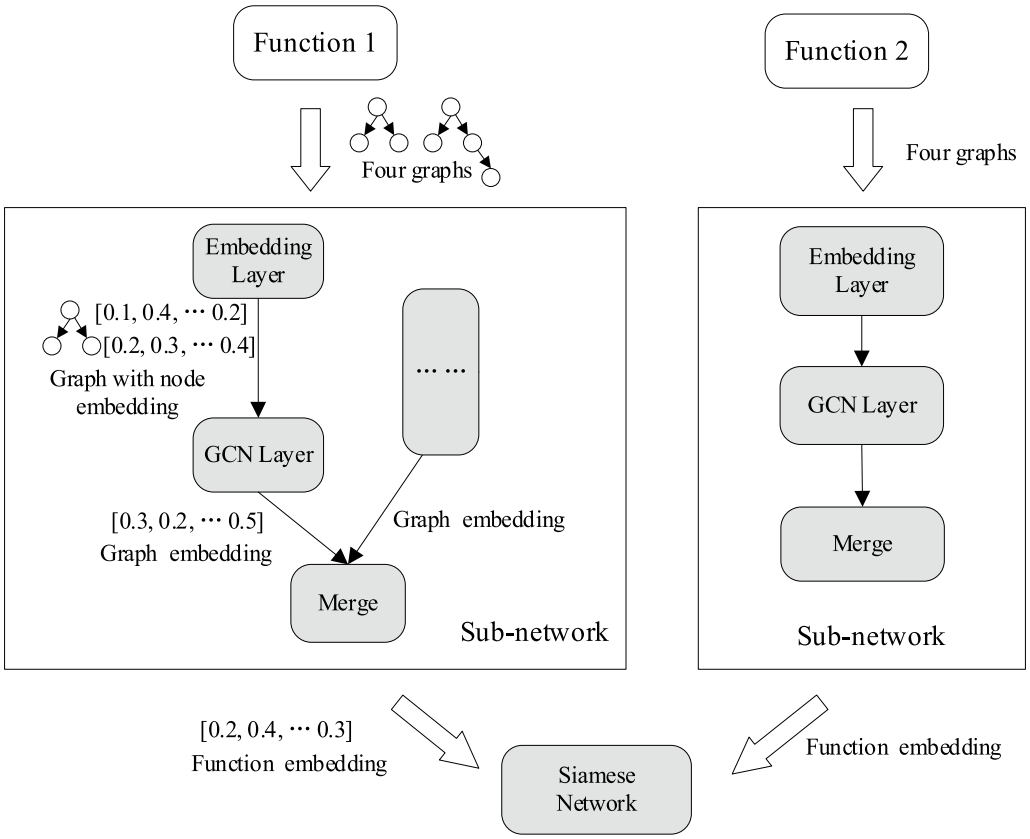

Fig. 5 Similarity calculation with deep neural networks

Embedding layer This layer is responsible for learning the semantic information of nodes in a graph. One possible method is to use Word2vec (Mikolov et al. 2013), as performed in many other studies. However, a node here generally contains one or multiple lines of code, and each line contains many identifiers and variables with variable length. Thus, we use doc2vec (Le et al. 2014) to create a numeric representation of the codes in a node, regardless of its length. Doc2vec is more capable of embedding sentences than Word2vec, which can directly embed multiple lines of code into a vector, not just one line of code, and in practice, there are few cases where a node contains multiple lines of code. After embedding, a node is represented by a vector.

GCN layer In a graph, the edges represent the control dependency or data dependency between nodes that have been embedded with vectors. To learn the structural information, we use a GCN (Graph Convolutional Network) (Kipf and Welling 2016) to process the graphs of a vulnerability function. Note that each function is represented by up to four graphs. The GCN layer takes four graphs as input, produces one embedding for each graph, and then outputs one final function embedding by merging the four graph embeddings (here we use the average of four embeddings).

Siamese model. Siamese networks are twin networks used for metric learning, i.e., they are composed of two identical subnetworks sharing the same weights (Neculoiu et al. 2016). Here, each subnetwork is actually a GCN. To learn the similarity between the vulnerability functions, we first need to pair the generated function embeddings. If the two functions belong to the same CWE, then they are considered similar, and the associated label y is 1. Otherwise, we consider them different and set the label to 0. The goal of Siamese training is to maximize the similarity between functions of the same CWE while minimizing the similarity between functions belonging to different CWEs. Specifically, let $S_{G_1G_2}$ denotess the similarity of a pair $\{G_1, G_2, y\}$ (the notations used in the similarity model are listed in Table 1), which is calculated by Eq. 1

$$S_{G_1G_2} = 1 - D \tag{1}$$

where D refers to the Euclidean distance of graphs G_1 and G_2 and is calculated by Eq. 2.

$$D = \frac{R_1 \cdot R_2}{R_1 \times R_2} \tag{2}$$

Then, the loss L is calculated by Eq. 3 as follows.

$$L \leftarrow \frac{1}{2N}\sum_{n=1}^{N} yD^2 + (1-y)\max(margin - D, 0)^2 \tag{3}$$

Table 1 The notations in the similarity model

G	**Graphs of vulnerability functions**
X	Statements of nodes of G
$S_{G_1G_2}$	Similarity between two functions G_1, G_2
W	Embedding matrix of all words in statements
A	Adjacency matrices of graphs G
E	Words embedding matrix of X
R	The embedding of graphs G
D	Distance between two functions
N	Number of function pairs
L	Contrastive of two functions

For a set of vulnerability function graphs, the total loss L(G), i.e., the sum of loss for all pairs, is calculated by Eq. 4.

$$L(G) = \sum \left(L\left(G_i, G_j, y\right)\right) \tag{4}$$

CWE assignment

Now, we have trained a similarity model to obtain the similarity between two vulnerability functions. With this model, we can perform CWE assignment by the similarities. Simply put, for a vulnerability function with a labeled CWE ID and a newly discovered function, if their similarity is larger than a predefined threshold T, then the new function is labeled with the CWE ID of the known vulnerability function. Otherwise, it is not in the same category, and the subsequent calculation will continue by paring it with vulnerability functions of other categories of CWE until one CWE ID is identified. If a CWE ID cannot be determined because all the similarities are below the threshold, we then leave the assignment to the security experts. A newly reported vulnerability is input to the trained model; if this vulnerability belongs to the trained CWE type, it will be assigned correctly; if the vulnerability belongs to a new CWE type, then this vulnerability will not be assigned a CWE ID in the model, and the vulnerability will be handed over to a security expert for further research to assign a CWE.

Algorithm 1 describes how we label the new function CWE assignment. In Lines 1–2, we extract the adjacency matrix A of the edge relations and the code statements X in the nodes from vulnerability function graphs. In Lines 3–7, we initialize the embedding matrix of the whole corpus W and the embedding matrix of graph node E_1, E_2. Then, matrices E_1 and E_2 are determined according the graph nodes X_1 and X_2. In Line 8, the GCN model takes embedding matrix E_1, E_2 and adjacency matrix A_1, A_2 as inputs and obtains the embedding of the whole graph R.

Finally, we use cosine similarity to calculat the distance between R_1 and R_2, and the distance is predicted as the output.

Algorithm 1. CWE assignment using similarity model.

Input: Vulnerability function G_1 and a testing function G_2

Output: CWE ID of testing function.

X_1, A_1 *in* G_1

X_2, A_2 *in* G_2

Initialize the embedding matrix of words W

$E_1 \leftarrow \varnothing$, $E_2 \leftarrow \varnothing$

for $x_1 \in X_1$, $x_2 \in X_2$ do

$e_1 \leftarrow W[x_1]$, $e_2 \leftarrow W[x_2]$

append e_1 to E_1, e_2 to E_2

$R_1 = GCN(A_1, E_1)$, $R_2 = GCN(A_2, E_2)$

$D = \frac{R_1 \cdot R_2}{\|R_1\| \times \|R_2\|}$

$S_{G_1 G_2} \leftarrow 1 - D$

If $S > T$: $CWE \leftarrow CWE$ of G_1

else $CWE \leftarrow Unknown$

return *CWE ID*

Implementation

In this section, we describe how we implement the proposed method in detail, mainly how we characterize the code and how we set up the similarity model.

Dataset preparation

Our prepared dataset contains 3394 CVEs in total. Of these, 498 CVEs are from Linux, OpenSSL, and Wireshark of the CVE-Details website, and 2896 CVEs are from multiple software programs and are from the Vulcode-db website. These CVEs are distributed in eight CWEs, whose detailed descriptions are reported in Table 2. We collect CVEs from three popular open-source projects, i.e., Linux kernel, OpenSSL, and Wireshark of the CVE-Details website, and multiple programs from the Vulcode-DB website. Considering that the machine learning models generally require a larger dataset for training classifiers, where there exist a number of samples for each type. Thus, when preparing the dataset, we dropped the categories with samples less than 100. Finally, we obtain 3394 CVEs in total, of which 498 are from CVE-Details and 2896 are from Vulcode-DB. These CVEs are distributed in eight CWEs, whose detailed descriptions are reported in Table 2. In the future, we will collect CVEs from more projects to cover more types of CWEs."

Code characterization

We use the SySeVR framework to represent the vulnerability functions (Li et al. 2021a). More specifically, we use the vulnerability syntax rules provided on Checkmarx to filter out the vulnerability syntax features in vulnerability functions. The rules of Checkmarx cover 93.6% of the vulnerability programs collected on SARD and are useful in identifying potential security issues. In addition, the vulnerability syntax features can be broadly classified into four types: function call, array usage, pointer usage, and arithmetic expressions. We use these features to match the nodes in the abstract syntax tree that are related to the located vulnerability snippets. Then, these matched nodes will be used as root candidates for slicing.

Based on these root candidates that match vulnerability syntax features in the function, graph slicing is performed on the PDG graph generated by Joern. Each vulnerability syntax feature was traversed forward and backward on the PDG, and the program slices containing all the related statements were traversed. For each function, the slices of the same type of syntax feature are then aggregated together based on the PDG graph, where data dependencies and structural dependencies between statements are added to the program slices.

Table 2 The CWE distribution in our dataset

CWE	Percent	Description
20	0.26	Improper input validation
119	0.206	Improper restriction of operations within the bounds of a memory buffer
125	0.139	Out-of-bounds read
189	0.064	Numeric errors
200	0.12	Exposure of sensitive information to an unauthorized actor
264	0.086	Permissions, privileges, access controls
399	0.084	Resource management errors
476	0.087	NULL pointer dereference

Similarity model with deep neural network

We implemented the similarity model on PyTorch. For encoding the statements inside a node of graph, the key hyperparamenters in Doc2vec are the dimension of the word vector is 30, the window size is 5, the training algorithm is skip-gram, and the threshold for configuring which higher-frequency words are randomly downsampled is 0.0001.

To feed the graph into the GCN layer, an adjacency matrix is used to represent the edges of the graph, and a feature matrix is used to represent the nodes of the graph. Each row denotes the embedding with doc2vec. For a graph with m nodes, the size of the adjacency matrix is m*m, and the size of the feature matrix is m*s, where s denotes the length of the embedding.

Evaluation

Experimental setup

We evaluate our methods on our prepared dataset, consisting of 3394 CVEs across 8 CWEs. We perform three different configurations in the experiments, mainly in terms of the number of CWE types: (1) the top 2 categories of CWE with the highest number of CVEs, i.e., CWE-20 and CWE-119. In this case, 2261 pairs (i.e., two vulnerability functions) are generated. (2) The top 5 categories of CWE with the highest number of CVEs, i.e., CWE-20, CWE-119, CWE-200, CWE-476, CWE-399, generating a total of 12,498 pairs. (3) All 8 categories of CWE, in which 25,372 pairs are generated. Under these configurations, the percentage of the same pairs (label is 1) and different pairs (label is 0) are almost similar; as a result the set is balanced. We use 90% of the pairs as the training set and the other 10% as the testing set. We compare our method with other machine learning models, including Support Vector Machine (SVM) (Russell 2010) and Naïve Bayes (Fukushima 1980) and common deep neural networks such as Convolutional Neural Network (CNN) (Shi et al. 2015), Long Short-Term Memory (LSTM) (Shi et al. 2015) and Graph Convolutional Network (GCN) (Kipf and Welling 2016), on the same dataset. These models take as input the embedding of Sysevr (Li et al. 2021a), which is a graph-based embedding method proposed recently.

Our method mainly consists of graph-based characterization and similarity model. Therefore, we compare our method with two types of models. First, two vulnerability representation models, SySeVR (Li et al. 2021a) and IVDetect (Li et al. 2021b). Second, five machine learning-based classification models that fed with the same embedding, including Support Vector Machine (SVM) (Russell 2010) and Naïve Bayes (Fukushima 1980) and common deep neural networks such as Convolutional Neural Network (CNN) (Shi et al. 2015), Long Short-Term Memory (LSTM) (Shi et al. 2015) and Graph Convolutional Network (GCN) (Kipf and Welling 2016), on the same dataset. These models take as input the embedding of Sysevr (Li et al. 2021a), which is a graph-based embedding method proposed recently.

We measure the effectiveness of our model in terms of widely used metrics: accuracy(A), precision(P), recall(R), and F1-score(F1). When the dataset is unbalanced, precision, recall and F1-score are more reliable metrics in comparison to accuracy. Specifically, accuracy metric equals to the correct predictions divided by the total examples in the dataset, precision for each class corresponds to the true positives, recall for each class equals the true positives by the number of examples that should have been identified as positive and finally, the F1-score for each class is the harmonic mean of the Precision and Recall. For each of the metrics precision, recall and F1-score, the overall result of each metric corresponds to the average of the respective metric for each class.

Performance on CWE assignment

Comparison with similarity models

We first compare our method with 5 machine learning-based similarity models. We present the results by different configurations, which have different number of CWE types.

Configuration 1: 2 CWE types Table 3 compares the results of different methods for the data distributed over 2 CWE types, i.e., CWE-20 and CWE-119. As we can see, our method performs best in recall, F1-score and accuracy. Note that the GCN method, which takes the whole graph as input, performs the second best among all methods, showing that the graph-based source code characterization is effective. Nevertheless, our method improves naïve GCN by 0.07 in f1-score, mainly contributing to i) vulnerability-snippet based slicing that captures vulnerability-related code well and ii) the Siamese network that effectively learns the similarity between two functions.

It is worth noting that LSTM achieves the highest precision, which is probably because LSTM takes the raw source code as input so that it can learn the semantics of the entire code. Thus, if two functions are highly similar in both length and semantics, it is able to identify them as similar. However, it may miss some potentially similar functions, which are similar in the small vulnerability snippets, and thus fail to assign the correct CWE ID. As a result, the recall of LSTM is low, i.e., only 0.78 as shown in the table, so the f1-score is accordingly low.

Configuration 2: 5 CWE types Table 4 compares the results of different methods for the data distributed over 5 CWE types, i.e., configuration 2. Similarly, our method outperforms the other methods in f1-score and accuracy. Given that the F1-score is a harmonic measurement of precision and recall, our method is more effective than other methods in assigning CWE for vulnerabilities.

Table 3 Results for 2 CWE types

Model	P	R	F1	A
SVM	0.78	0.47	0.59	0.657
Naïve Bayes	0.56	0.69	0.62	0.658
CNN	0.79	0.68	0.73	0.853
LSTM	0.94	0.78	0.85	0.895
GCN	0.74	0.89	0.81	0.926
Our method	0.83	0.94	0.88	0.953

Table 4 Results for 5 CWE types

Model	P	R	F1	A
SVM	0.43	0.32	0.37	0.435
Naïve Bayes	0.54	0.23	0.32	0.579
CNN	0.68	0.89	0.77	0.752
LSTM	0.86	0.78	0.82	0.794
GCN	0.76	0.77	0.76	0.810
Our method	0.83	0.83	0.83	0.885

Table 5 Results for 8 CWE types

Model	P	R	F1	A
SVM	0.25	0.45	0.32	0.354
Naïve Bayes	0.53	0.36	0.43	0.457
CNN	0.43	0.53	0.47	0.651
LSTM	0.82	0.62	0.71	0.712
GCN	0.65	0.56	0.6	0.774
Our method	0.77	0.73	0.75	0.818

Configuration 3: 8 CWE types Table 5 compares the results of different methods for the data distributed over 8 CWE types, i.e., configuration 3. Nevertheless, our method achieves the best recall, f1-score and accuracy among these methods, proving the effectiveness of our method in CWE assignment.

It should be noted that the performance drops when the number of CWE types increases, e.g., the f1-score for 2 CWE types is 0.88 and drops to 0.83 for 5 CWE types and then 0.75 for 8 CWE types. This is because of two reasons. First, with the increase in the number of types, deep learning-based methods are prone to degradation, which is still a challenging problem. Second, some CWE types are more general and involve more problems, and some CWE types are closely related and have parent–child relationships, making many vulnerability functions hard to assign correctly. For example, CWE-264 denotes weakness related to permissions, privileges, and access control, covering a broad range of problems. CWE-125 (out-of-bounds read) is a child of CWE-119. The same results can be found in Fig. 6, where the AUC value (area under the ROC curve) decreases when the number of CWE types increases. We will study the performance improvement on more CWE types in our future work.

Comparison with representation models

To demonstrate the effectiveness of the proposed vulnerability graph representation, we compared it with two vulnerability representation frameworks. The first one is SySeVR (Li et al. 2021a), which uses a graph

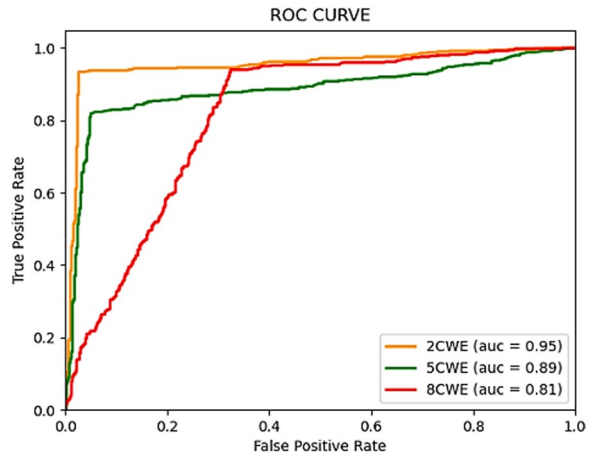

Fig. 6 ROC curve of our method for different configurations

Table 6 Comparison experiments (Accuracy)

	2CWE	5CWE	8CWE
SySeVR	0.876	0.785	0.647
IVDetect	0.853	0.710	0.713
Our approach	0.953	0.885	0.818

representation to remove code statements irrelevant to the vulnerability, and the processed source code is input to the BLSTM model; the second one is IVDetect (Li et al. 2021b), which characterizes the function into a graph based on five features in the vulnerability, and finally input to FA-GCN(Feature-Attention Graph Convolutional Networks) to perform the vulnerability classification task.

As can be seen from the Table 6, our method is better than the comparison methods. The reasons are: 1) Our graph characterization method targets the specific location where the vulnerability is located rather than the whole function and removes code statements that are irrelevant to vulnerability features, which makes the generated vulnerability graph more focused on the representation of vulnerability features. 2) The graph is non-Euclidean space data, which can clearly reflect the information in the code language, and combined with graph neural networks and siamese networks, it can better distinguish between different categories of vulnerabilities differences, thus classifying vulnerabilities more effectively.

Results per CWE with our method

We further explore whether the CWE assignment exhibits bias to certain CWE types, mainly to show which type of CWE our method is more suitable for. From Tables 7,

Table 7 Assignment performance for 2 CWE types

	P	R	F1	A
CWE-20	0.8	0.94	0.86	0.933
CWE-119	0.85	0.93	0.88	0.971

Table 8 Assignment performance for 5 CWE types

	P	R	F1	A
CWE-20	0.71	0.77	0.74	0.831
CWE-119	0.88	0.87	0.87	0.853
CWE-125	0.9	0.88	0.88	0.924
CWE-189	0.9	0.91	0.90	0.916
CWE-200	0.73	0.75	0.74	0.901

Table 9 Assignment performance for 8 CWE types

	P	R	F1	A
CWE-20	0.61	0.57	0.59	0.743
CWE-119	0.89	0.81	0.85	0.921
CWE-125	0.79	0.53	0.63	0.786
CWE-189	0.93	0.94	0.93	0.809
CWE-200	0.65	0.84	0.73	0.831
CWE-264	0.61	0.57	0.59	0.704
CWE-399	0.83	0.94	0.88	0.864
CWE-476	0.88	0.65	0.74	0.886

8, and 9, it can be seen that the performance of our method on different CWE types is different. For example, the performance on CWE-119 is always higher than that of CWE-20 in these tables. The assignment for CWE-264 is much poorer than other CWEs. We suspect that the reason is as follows.

CWE-20 denotes improper input validation, while CWE-119 denotes improper restriction of operations within the bounds of a memory buffer. It can be seen that CWE-119 is focused on memory buffer-related vulnerabilities, which is more focused than CWE-20, which covers more types involving input validation. Remember that we use four types of vulnerability syntax features upon code characterization, two of which are array usage and pointer usage. Therefore, our method can more precisely capture the memory buffer related vulnerability snippets and thus performs better in assigning functions belonging to CWE-119.

CWE-264, as stated before, denotes weakness related to permissions, privileges, and access control, covering a broad range of problems. Similarly, it involves more general types and vulnerability weakness; as a result the associated vulnerability functions exhibit large differences in structure and semantics, thereby compromising the assignment.

Table 10 Performance with 50 CVEs per CWE for different configurations

Configuration	P	R	F1	A
2-CWE	0.61	0.63	0.62	0.67
5-CWE	0.68	0.62	0.65	0.68
8-CWE	0.62	0.65	0.63	0.69

Note that the number of CVEs for CWE-476 is the smallest, i.e., it takes only 8.7% in our dataset, as shown in Table 2. However, the performance on CWE-476 is good, i.e., 0.886, as shown in Table 8, the second best among all CWE types. CWE-476 denotes NULL pointer dereference. As explained before, we have a syntax feature specific to pointer usage upon code characterization, which helps express this type of vulnerability snippet.

From these results, it can be concluded that our method has good assignment ability for relatively type-focused vulnerabilities, e.g., memory related and pointer-related vulnerabilities. In addition, although it performs relatively poorly on CWE types that are more generic, it still performs better than other existing methods.

The effects of CWEs and CVEs on performance

The results in "Performance on CWE Assignment" Section show that the performance decrease with the number of CWEs, i.e., the performance is poor in 5-CWE compared to that in 8-CWE. To explore the reasons for the decrease, we perform two experiments by varying the number of CWEs and the number of CVEs, respectively.

Varying number of CWEs

We first measure the performance with varying number of CWEs, each has equal number of CVEs. For the 2-CWE, 5-CWE and 8-CWE configurations, instead of using all the CVEs of each CWE types, we randomly select 50 CVEs for each CWE type and perform training and testing using these CVEs for each configuration. As can be seen in Table 10, the precision, recall and f1-score are comparable for the three configurations, e.g., the f1-score is 0.62, 0.65, and 0.63 for 2-CWE, 5-CWE, and 8-CWE, respectively. In other words, the performance doesn't decrease significantly as the increase of the number of CWE types.

Varying number of CVEs

We then measure the performance with increasing number of CVEs per CWE. We use 50, 100 and all CVEs per CWE for the 5-CWE configuration and perform training

Table 11 Performance with increasing number of CVEs per CWE

Number of CVEs	P	R	F1	A
50	0.68	0.62	0.65	0.68
100	0.75	0.71	0.81	0.79
All	0.83	0.83	0.83	0.885

and testing on these CVEs. As can be seen in Table 11, the precision, recall and f1-score increases gradually as the number of CVEs increases, e.g., f1 increases from 0.65 for 50 CWEs to 0.81 for 100 CVEs. This is because when the number of vulnerabilities in the dataset becomes larger, the model is better able to learn features implied in each category of vulnerabilities and is better able to perform the task of vulnerability classification. Although letting data pairing in Siamese networks requires less data than a general deep learning model, a certain amount of data is needed to ensure the effectiveness of the model. Due to heavy manual effects, our dataset currently is relatively small, and it only contains 8 CWEs of vulnerabilities whose number is more than 190 for each CWE.

From the two experiments, we can see that the number of CVEs per CWE is one key factor that affects the performance in our prepared dataset. From the results from natural language processing (NLP) or computer vision (CV), more classes would decrease the classification performance. Therefore, it can be concluded that with more CWE types, the performance would decrease as well. To this end, we plan to extend our work in two directions. First, we plan to collect more CVEs spanning more CWEs to enrich our dataset. Second, we plan to employ the popular large language models (LLM), e.g., Codex, to improve our method especially in learning from few samples.

Performance of assigning newly reported CVEs

In practical usage, we train the model on existing vulnerabilities and then use the CWE for newly reported vulnerabilities. The new vulnerabilities (or CVEs) fall into two types within the paper, i) it belongs to one CWE type in our dataset, ii) it doesn't belong to any CWE type in our dataset (i.e., unknown CWE).

New CVEs with labeled CWEs

To verify whether our approach can perform the classification of newly discovered vulnerabilities correctly, we divide the dataset into two parts according to the reported time of vulnerabilities. Specifically, we use 706 CVEs discovered from 2013 to 2017 as the training dataset and 71 CVEs reported from 2018 and 2019 as the testing dataset. For the model learned on the training dataset, the vulnerabilities in the testing dataset are viewed as newly discovered vulnerabilities. Note that one CVE ID contains the time when the vulnerability was discovered, which makes it easier for us to prepare the experiment.

Table 12 reports the results of this experiment. Similar to our previous experiments, we also conducted experiments in 2CWEs, 5CWEs, and 8CWEs and computed the accuracy, precision, recall, and F1-score. As we can see, the model works well for these newly discovered vulnerabilities. For example, 2CWEs, 5CWEs, and 8CWEs achieve F1-scores of 0.934, 0.81 and 0.803 respectively; note that these results are slightly lower than the previous results, which are 0.953, 0.885 and 0.818, respectively. This is reasonable since these new vulnerabilities are not seen in the training dataset, which makes the model compromised when predicting the new vulnerabilities. However, we believe the results are still promising for assigning future vulnerabilities.

New CVEs with unknown CWEs

Since the performance of the model degrades when CWEs become more numerous, we added 300 CVEs that are not part of the dataset (the CWEs of these vulnerabilities are also not in the dataset) to be paired with the CVEs in the original test set and input to the trained model, and if these vulnerability pairs are identified as different categories, it proves that the model is effective for the assignment of CWEs. Table 13 shows the results of the experiments, and it can be seen that the performance of the classification is decreased compared to the previous experiments, which is understandable, firstly, the accuracy of the deep learning model decreases when there are too many categories, which is a problem that also needs to be solved in the field of deep learning.

Table 12 Performance on classifying newly reported CVEs

CWE type	P	R	F1	A
2CWE	0.79	0.86	0.82	0.934
5CWE	0.83	0.81	0.81	0.810
8CWE	0.74	0.72	0.72	0.803

Table 13 Performance on classifying unknown CVEs

CWE Type	P	R	F1	A
2CWE	0.77	0.90	0.83	0.92
5CWE	0.81	0.81	0.81	0.78
8CWE	0.71	0.69	0.7	0.75

Second, some of the additional data we added are vulnerabilities that have not yet been assigned CWEs, which may belong to CWEs in the dataset, and some of the CWEs are vaguely defined or similar, such as CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-120 (Buffer Copy without Checking Size of Input) can both indicate Buffer Overflow, which causes Buffer Overflow to arise for different reasons distinguishing the categories, which leads to a decrease in model accuracy.

Time overhead

In this experiment, we measure whether our method is efficient in CWE assignment. The time overhead mainly lies in two aspects: code characterization and training on a dataset.

Code characterization involves transforming a function in the form of a source into several graphs. On our dataset, the average time to process vulnerability is 6 s. Although the overhead is slightly higher, a function is processed only once and can be stored in the database for future usage. Therefore, the overhead is affordable.

For the time overhead upon training, the results are reported in Table 14. Note that the inputs of these methods are embedded vectors; therefore, the training is fast, i.e., dozens of seconds. Our method takes more time than the other methods upon training, e.g., for the dataset across 8 CWE types, it uses the longest time to train the model, i.e., 98 s. This is mainly because the Siamese network needs pairs of vulnerability functions; therefore, the actual samples upon training are multiplied. Although it consumes more time, the pairing helps mitigate the overfitting problem when the amount of data is too small and thus improves the performance of the model.

Conclusion

CWE assignment is useful for categorizing newly discovered vulnerabilities. In this paper, we propose an automatic CWE assignment method with deep neural networks. We prepare a dataset that contains 3394 real world vulnerabilities. Then, we extract statements with vulnerability syntax features from these vulnerabilities and use program slicing to slice them according to the categories of syntax features. On top of slices, we represent these slices with graphs that characterize the data dependency and control dependency between statements. We employ the graph neural network to learn the hidden information from these graphs and leverage the Siamese network to compute the similarity between vulnerability functions, thereby assigning CWE IDs for these vulnerabilities. The experimental results show that the proposed method is effective compared to existing methods. In the future, we plan to improve our method along two directions. First, we plan to extend our dataset to cover more CWE types and release it publicly to foster further research. Second, we plan to design more types of vulnerability syntax features to cover more types of weakness so that the performance on CWE type that is generic can be improved.

Table 14 Time overhead (seconds) upon training the model

Model	2CWE	5CWE	8CWE
SVM	35	72	98
Naïve Bayes	23	57	74
CNN	28	57	76
LSTM	21	45	63
GCN	12	25	64
Our Method	16	54	98

Acknowledgements
Not applicable

Author contributions
PL: Conceptualization, Methodology, Resources, Writing-review& editing, Supervision. WY: Investigation, Resources, Writing-original draft. HD: Investigation. XL: Supervision. SZ: Investigation. CY: Resources. YL: Resources. All authors read and approved the final manuscript.

Funding
The research was supported in part by the National Natural Science Foundation of China (Nos. 62166004, U21A20474), the Guangxi Science and Technology Major Project (No. AA22068070), the Guangxi Natural Science Foundation (No. 2020GXNSFAA297075), the Center for Applied Mathematics of Guangxi, the Guangxi "Bagui Scholar" Teams for Innovation and Research Project, the Guangxi Talent Highland Project of Big Data Intelligence and Application, the Guangxi Collaborative Center of Multisource Information Integration and Intelligent Processing and Fundamental Research Funds for the Central Universities (No. 2021JKF06).

Availability of data and materials
Data sharing is not applicable to this article.

Declarations

Competing interests
The authors declare no competing interests.

Received: 31 January 2023 Accepted: 20 April 2023
Published online: 02 November 2023

References

Aivatoglou G, Anastasiadis M, Spanos G, Voulgaridis A, Votis K, Tzovaras D (2021) A tree-based machine learning methodology to automatically classify software vulnerabilities. In: 2021 IEEE International Conference on Cyber Security and Resilience (CSR), pp 312–317
Common Vulnerabilities and Exposures (2023) https://cve.mitre.org/. Accessed on 15 Jan 2023
Common Weakness Enumeration (2023) https://cwe.mitre.org/. Accessed on 15 Jan 2023
Cui L, Hao Z, Jiao Y, Fei H, Yun X (2020) Vuldetector: detecting vulnerabilities using weighted feature graph comparison. IEEE Trans Inf Forensics Secur 16:2004–2017

CVE-2016-2842 (2023) https://www.cvedetails.com/cve/CVE-2016-2842/. Accessed on 15 Jan 2023
CVE-2022-32552 (2023) https://www.cvedetails.com/cve/CVE-2022-32552/. Accessed on 15 Jan 2023
CVE-2022-33936 (2023) https://www.cvedetails.com/cve/CVE-2022-33936/. Accessed on 15 Jan 2023
CVEDetails (2023) https://www.cvedetails.com/. Accessed on 15 Jan 2023
Dam HK, Pham T, Ng SW, Tran T, Grundy J, Ghose A, Kim CJ (2018) A deep tree-based model for software defect prediction. arXiv preprint arXiv:1802.00921
Das SS, Serra E, Halappanavar M, Pothen A, Al-Shaer E (2021) V2w-bert: a framework for effective hierarchical multiclass classification of software vulnerabilities. In: 2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA), pp 1–12
DeLooze LL (2004) Classification of computer attacks using a self-organizing map. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, pp 365–369
Fukushima K (1980) A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biol Cybern 36:193–202
Joern tool (2023) https://joern.io/. Accessed on 15 Jan 2023
Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907
Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO, pp 75–86
Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: International conference on machine learning, pp 1188–1196
Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021a) Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans Dependable Secure Comput 19(4):2244–2258
Li Y, Wang S, Nguyen TN (2021b) Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 292–303
Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781
Na S, Kim T, Kim H (2017) A study on the classification of common vulnerabilities and exposures using naïve bayes. In: Advances on Broad-Band Wireless Computing, Communication and Applications: Proceedings of the 11th International Conference On Broad-Band Wireless Computing, Communication and Applications (BWCCA–2016) November 5–7, 2016, Korea, Springer International Publishing, pp 657–662
Neculoiu P, Versteegh M, Rotaru M (2016) Learning text similarity with siamese recurrent networks. In: Proceedings of the 1st Workshop on Representation Learning for NLP, pp 148–157
Neuhaus S, Zimmermann T (2010) Security trend analysis with cve topic models. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering, pp 111–120
Rahman MM, Yeasmin S (2013) Adaptive bug classification for cve list using bayesian probabilistic approach. USask, Saskatoon
Russell SJ (2010) Artificial intelligence a modern approach. Pearson Education, Inc
Shi X, Chen Z, Wang H, Yeung DY, Wong WK, Woo WC (2015) Convolutional LSTM network: A machine learning approach for precipitation nowcasting. Adv Neural Inf Process Syst 28
Sun H, Cui L, Li L, Ding Z, Hao Z, Cui J, Liu P (2021) VDSimilar: vulnerability detection based on code similarity of vulnerabilities and patches. Comput Secur 110:102417
The code static analysis tool Checkmarx (2023) https://checkmarx.com/. Accessed on 15 Jan 2023
Vulncode-db (2023) https://www.vulncode-db.com/. Accessed on 15 Jan 2023
Wang Q, Li Y, Wang Y, Ren J (2022) An automatic algorithm for software vulnerability classification based on CNN and GRU. Multim Tools Appl 81(5):7103–7124
Wita R, Teng-Amnuay Y (2005) Vulnerability profile for linux. In: 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers) Vol 1, pp 953–958
Xiao Y, Chen B, Yu C, Xu Z, Yuan Z, Li F, Shi W (2020) MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In: USENIX Security Symposium, pp 1165–1182
Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv Neural Inf Process Syst 32

Publisher's Note