2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC) | 979-8-3503-3750-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPC58990.2023.00048

FVA: Assessing Function-Level Vulnerability by Integrating Flow-Sensitive Structure and Code Statement Semantic

Chao Ni[†*], Liyu Shen[†], Wei Wang[†], Xiang Chen[‡], Xin Yin[†] and Lexiao Zhang[†]
[†]Zhejiang University, China. Email: {chaoni,liyushen,wangw99,xyin,lexiaozhang}@zju.edu.cn
[‡] School of Information Science and Technology, Nantong University, China. Email: xchencs@ntu.edu.cn

***Abstract*—Previous studies have been conducted on software vulnerability (SV) assessment at the code-based level, especially the function level. However, a key limitation of these studies is that they do not consider the structure information (e.g., control dependency and data dependency) of a vulnerable function, which is crucial for understanding SVs and assigning priority for fixing. In this study, we propose a flow-sensitive, multi-task, and function-level vulnerability assessment method named FVA, which considers both global structure information and local semantic information. More specifically, FVA considers two types of flow information extracted from the control dependence graph and the data dependence graph. Meanwhile, FVA also considers the deep semantic information of the statement as well as its various types of contexts (i.e., surrounding context and program slicing context). We evaluate the effectiveness of FVA on the large-scale dataset (4,467 functions) by comparing it with four state-of-the-art baselines in terms of five performance measures. The experimental results indicate that FVA outperforms these baselines by a significant margin. More precisely, on average, FVA obtains 0.795 of F1-score and 0.727 of MCC, which improves baselines by 5%-14% and 8%-20%, respectively.**

***Index Terms*—Security Vulnerability, Vulnerability Assessment, Function Level, Deep Learning, Flow Analysis**

I. Introduction

Software Vulnerabilities (SVs) can expose software systems to risk situations and consequently make the software under cyber-attacks, eventually causing huge economic losses and even threatening people's lives. Therefore, detecting and fixing these SVs is an important task for software quality assurance. However, due to the limitation of software quality assurance resources [1], it is impossible to treat all SVs equally and fix all SVs simultaneously. Thus, it is necessary to prioritize these detected software vulnerabilities for better treatment. In practice, an effective solution to prioritize those SVs, which has imminent and serious threats to the systems of interest, is to use one of the most widely known SV assessment framework CVSS (Common Vulnerability Scoring System) [2], which characterizes SVs by considering three metric groups: Base, Temporal and Environmental. The metrics that are in the groups can be further used as the criterion for selecting serious SVs to fix early. For example, an SV with low attack complexity can be easily exploited and consequently has a critical impact on a system. Such an SV should be fixed first.

*Chao Ni is the corresponding author.

However, there usually has a delay in manually assigning CVSS metrics to the new SVs conducted by security experts [3]. Therefore, an automatic approach to assessing detected SVs is urgently needed. Different approaches [2], [4]–[7] have been proposed to automatically assess CVSS metrics of SVs by training data-driven models with SV reports. NVD (National Vulnerability Database [8]) is the most widely used source of the reports [2] since it provides more high-quality SV-specific information. Unfortunately, NVD reports are often provided after a period of time when the corresponding SVs have been fixed (i.e., on average four months delay) [9]–[12], which means that the CVSS metrics are unavailable at the fixing time and consequently limits the adoption of report-based automatic SV assessment methods.

Recently, a few straightforward approaches [9], [10] are proposed for assessing SVs instead of using SVs reports. These methods directly take the codes of function as the input once the functions are confirmed as vulnerable ones, which means these methods can assign them the CVSS metrics before the vulnerable functions are fixed and their corresponding report is available. These function-level methods confirmed the feasibility of SVs assessments with function only and indeed achieved promising performance. However, previous work only utilizes a portion of the statement information of the functions (e.g., the vulnerable statement as well as its limited context [9], modified lines of codes in functions [10]) which may ignore the structure information of the function. Intuitively, **the structure of the function provides the global information while the statement of the function provides the local information.** For example, the control dependence graph of the function records the dependencies of statements on predicates, while the data dependence graph of the function records the relatedness of variables. Therefore, combining both the global structure information and the local semantic information can represent complex, representative, and rich information about its functionality and vulnerabilities.

To fully utilize the implicit but rich information about the function, we propose a novel flow-sensitive vulnerability assessment method named FVA at the function level. This method considers both global structure information and local semantic information. More specifically, FVA totally considers two types of flow information extracted from the control dependence graph (CDG) and data dependence graph (DDG).

2643-7171/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPC58990.2023.00048

Since both types of information are usually represented as a graph, we embed them into a graph model [13] inspired by previous study [14] which shows that graph-based representation incorporating diverse program semantics can outperform other representation techniques (e.g., feature-based techniques [15] or sequence-based techniques [16]). Besides, FVA also utilizes the deep semantic information of the statement as well as its various types of contexts (i.e., surrounding context and program slicing context). Moreover, a previous study [10] also confirms that CVSS assessment has relatedness among different basic aspects and different work [10], [17], [18] has successfully utilized a shared model for predicting related tasks. Therefore, we model FVA as a multi-task one for better utilizing the shared information among various basic aspects of CVSS.

To verify the effectiveness of our method, we conduct a large-scale study on 4,467 functions by comparing it with four state-of-the-art baselines in terms of five performance measures. The experimental results indicate that our method outperforms these baselines by a significant margin. In particular, on average, FVA obtains 0.795 of F1-score and 0.727 of MCC, which improves baselines by 5%-14% and by 8%-20%, respectively.

This study makes the main contributions as follows:

- To the best of our knowledge, we are the first to utilize the flow-related information with the graph embedding model to conduct function-level automatic SV assessment.
- We conduct a large-scale experiment on 4,467 functions to show the effectiveness of FVA by comparing with state-of-the-art baselines.
- We study the impacts of different graph embedding models, different types of context as well as different statement embedding approaches on automatic SV assessment.

II. Background and Motivation

A. SV Assessment with CVSS

Software Vulnerability (SV) assessment plays a critical role in determining the various characteristics of detected SVs [19]. These characteristics help developers comprehensively understand the SVs and subsequently help to assign different priorities to SVs. For example, confidentiality is extremely important to a system, which means the attackers can access or steal sensitive information from the system. Therefore, if an SV is severely damaged, it should have a high fixing priority and a few fixing methods (e.g., checking privileges to access the influenced component) will be adopted to ensure confidentiality.

Common Vulnerability Scoring System (CVSS) [20] is one of the most commonly used frameworks to assess software vulnerability in both the academic community and the industry community since it is well maintained by experts. The CVSS consists of three metric groups: Base, Temporal, and Environmental. The base metrics are widely used to determine the attack vectors exploited by SVs, to assess their potential impacts on systems, and to help developers to better prioritize the alleviation of such SVs. The temporal metrics measure the current exploitability of a vulnerability, while the environmental metrics allow organizations to modify the base metrics based on security requirements. However, they are unlikely obtained from project artifacts (e.g., SV code/reports) alone. There are two main versions of CVSS, namely versions 2 and 3. In this study, we prefer to adopt the base metrics of CVSS version 2 to assess SVs rather than version 3 (introduced in 2015) since this version is still more predominantly adopted.

CVSS version 2 provides metrics to quantify the three main aspects of SVs (i.e., exploitability, impact, and severity) by considering seven dimensions: Availability, Access Vector, Access Complexity, Confidentiality, Authentication, Integrity, and Severity. More precisely, the base Exploitability metrics evaluate both the technique (Access Vector) and the complexity to initiate an exploit (Access Complexity) as well as the authentication requirement (Authentication). As for the Impact metrics of CVSS, it focuses on three dimensions of a system: Confidentiality, Integrity, and Availability. Meanwhile, both the Exploitation and Impact metrics can be used to evaluate the Severity metric. The severity is similar to the criticality of software vulnerabilities. However, it is not sufficient to rely on the Severity metric alone since medium-severity SVs may still generate high impacts when it is extremely complex to be exploited. Assigning a high fixing priority to such an SV seems significantly important to avoid a system from the serious risk of cyber-attack. Thus, in this study, we consider all the base metrics of CVSS version 2 to develop SV assessment models, which is consistent with previous studies [9], [10].

B. SV Assessment in Code Functions

Software vulnerability assessment helps to prioritize vulnerable functions when the resources are limited during the fixing phases. Many previous works [2]–[6] have been proposed to automatically assess CVSS metrics of SVs by developing data-driven models with SV reports. However, there usually has a delay in security experts manually assigning CVSS metrics to the newly detected SVs [3], and actually, the official reports (e.g., NVD reports) are usually provided long after the corresponding SVs have been fixed (i.e., on average four months delay) [9]–[12]. Meanwhile, SVs are mostly rooted in source code and it is natural for both researchers and practitioners to perform code-based SV assessments.

The statements represent the core parts of the SVs and recently a few function-level methods [9], [10] are proposed to confirm the feasibility of SVs assessments with function only. However, previous work only utilizes a portion of the statement information inside functions (e.g., the vulnerable statement as well as its limited context [9], modified lines of codes in functions [10]) which may ignore the structure information inside functions. Intuitively, **the structure of the function provides the global information while the statement of the function provides the local information.** For example, the control dependence graph of the function records the dependencies of statements on predicates, while the data flow graph of the function records the relatedness

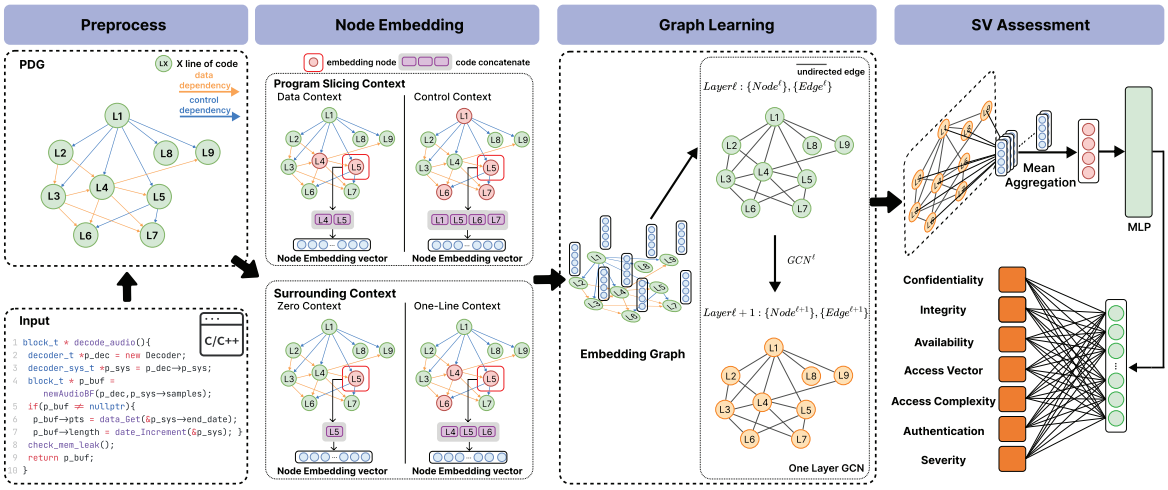

Fig. 1: The framework of our approach.

of variables. Therefore, we hypothesize that using the global structure information as well as the local semantic information can represent complex, representative, and rich information about its functionality and vulnerabilities and consequently helps with code-base SVs assessment.

III. Approach

To investigate the feasibility of our intuitive hypothesis, we propose a novel approach FVA, which fully considers the flow-related structure information as well as the context-related semantic information for automatic function-level software vulnerability assessment. As illustrated in Fig. 1, FVA contains four main steps: ❶ data pre-processing and graph constructing, ❷ node embedding, ❸ graph-based model learning, and ❹ software vulnerability assessing. Details of FVA are presented in the following subsections.

A. Data Pre-processing and Graph Constructing

To train FVA, we conduct a set of data pre-processing operations to obtain high-quality vulnerable functions and construct their corresponding structure information (e.g., CDG and DDG) for model building.

Data Pre-processing. Data pre-processing helps to remove noise from functions and improves the quality of the trained models. In particular, we remove the empty lines as well as the comments since they have no impact on code functionality. Besides, we also trim the space from both sides of each statement to better embed their semantic information. Then, we keep the code tokens as the original ones without any other transformation (such as uppercase, lowercase, or stems) since these operations may change the code functionality or variable meaning.

Graph Constructing. In this study, we extract the program dependence graph, which consists of both data dependency edges and the control dependency edges. Data dependency edges describe the dependency of the variable reference statement on the statement defining the variable, while control dependency edges describe the execution of a statement depending on another statement. Both of the two types of dependency are extracted with *Joern* program [21]. More precisely, we use *Joern* to parse the source code of a function, then the function will be modeled as a graph *G=(V, E)*, where *V* is the set of statements, represented as nodes (v) in the graph *G* and *E* is the set of flow-information (i.e., the control dependency flow and the data dependency flow) among statements, represented as the edges (e) in the graph *G*.

B. Node Embedding

The program dependence graph contains flow-sensitive structure information about a function, which can be treated as the global view of such a function. Apart from the global view of a function, we also need to obtain local semantic information about the statements in a function. Pre-trained models associated with programming languages [22], [23] are widely used in software engineering and indeed have achieved much success since they have a good understanding of tokens in the programming language domain. CodeBERT proposed by Feng et al. [24] is one of the most widely used ones and we adopt it as the semantic extractor in FVA.

Moreover, to better obtain the semantic information of a statement (represented as a node, $v \in V$), we also consider the surrounding context of each statement, which has been studied in many previous studies for software vulnerability-related tasks [9], [10], [25]. In summary, we consider four methods to extract their context: ① *zero context,* ② *one-line context,* ③ *data dependency context*, and ④ *control dependency context*, which can be grouped into two types: surrounding context and program slicing context. We introduce the details of these methods as follows.

Surrounding Context. The direct way to define the surrounding context is by utilizing a fixed number (n) of lines before and after a given statement. The surrounding statements may contain relevant information (e.g., values or usage of variables), which is also consistent with the observation that developers usually look at the nearby codes when they look at a given statement [26]. As introduced in pre-processing, we

remove the surrounding lines, which are code comments or blank lines, since considering these statements have no impacts on the functionality of a function [27]. In total, we consider two settings of n lines: zero and one, and refer to them as *zero context* and *one-line context* for the convenience of subsequent description. Notice, for the margin statements (e.g., the first and the last statement), we limit the surrounding lines to be within the function.

Program slicing context. Program slicing captures the relevant statements to a point (i.e., statement) in a function [28], which is also utilized for SV detection task [29], [30]. In the scenario of function-level SV assessment, we treat each statement as the starting point and consider the intra-procedural program slicing for every statement. That is, program slicing will find the relevant statements with the boundary of the studied function. Following previous studies [9], [30], [31], we adopt the program dependence graph (PDG) extracted from the source code to obtain the program slices for each statement in the function. We extract two types of slices (i.e., backward and forward) based on the data and control dependencies. Especially, the *backward slices* directly change or control the execution of statements, which affects the values of variables in the analyzed statements. As for *forward slices*, they are data-dependent or control-dependent on the analyzed statements [32]. For a given PDG, the backward slices are nodes that can reach the analyzed nodes through one or more directed edges. For the forward slices, they are the nodes that can be reached from the analyzed nodes by following one or more directed edges. In total, we consider another two types (data dependency and control dependency) of slice context and refer them to as *data context* and *control context* for the convenience of subsequent description.

Finally, to obtain the semantic embedding of each node v in the graph G, we directly treat the statement as well as its context in v as the input of CodeBERT. We also limit the token length in node v to 128, because there will be many statement nodes generated by a function (<= 200) and each node will require a certain amount of memory for CodeBERT embedding, so the whole graph model will consume a lot of memory, limiting the token length of individual nodes ensures that all nodes in the graph will be efficiently trained together. Meanwhile, CodeBERT has a special token [CLS] that can embed the semantics of the whole input. Therefore, similar to previous work [33], we utilize the embedding vector of [CLS] as the final node semantic, which is a 768-dimensional vector.

C. Graph Learning

The specific structure information of flow (such as data dependency and control dependency) is usually represented as a graph, allowing the graph model to better capture the deeper information inside the function. At present, many graph models have been proposed and indeed have many successful applications in different application domains (e.g., vulnerability detection [34], defect prediction [18]). Graph Convolutional Network (GCN) [35] is one of the most popular graph neural network (GNN) models that can effectively learn the fusion feature from graph structure and node features simultaneously.

The vulnerable function is firstly transformed into a graph ($\mathcal{G}$) and subsequently the nodes are embedded along with its context. Following that, the full graph is fitted into a one-layer GCN model. Meanwhile, in order to learn more rich information about the graph, we treat the graph as an undirected one and add a self-loop edge at each node. After that, the GCN model will aggregate information about each node from its neighboring nodes by using the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{1}$$

$$\tilde{A} = A + I_N \tag{2}$$

where A is the adjacency matrix of the undirected graph $\mathcal{G}$ and I_N is the identity matrix. We use symmetric normalization $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, where $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, to get rid of the problem that multiplying with $\tilde{A}$ directly will change the scale of the feature vectors. $H^l \in \mathbb{R}^{N\times D}$ is the matrix of activations in the l^{th} layer, with $H^{(0)} = X$ and $H^{(L)} = Z$ (Z for graph-level outputs). $W^{(l)}$ is a layer-specific trainable weight matrix, and $\sigma(\cdot)$ denotes a differentiable, non-linear activation function.

Eventually, each node obtains aggregated information from its neighbor nodes, and the global information of the function can flow along the edges. FVA extracts information from graph structure and node embedding for SVs assessment and we use the average information from all nodes to get the final feature representation $F_{\mathcal{G}}$ of this graph.

D. SV Assessment with Multi-task Learning

Considering the relatedness among several SV assessment tasks, we also build a multi-task and end-to-end model as suggested by previous work [10]. That is, FVA simultaneously gives the prediction results of the seven CVSS metrics for a specific vulnerable function. On top of the shared features $F_{\mathcal{G}}$ extracted by GCN, task-specific blocks are necessary to capture the differences among the seven tasks. Therefore, for each task-specific block, we implement it by adopting a fully connected layer whose output length is the number of classes for that task.

Then, each task-specific block can determine the output ($pred_i$) of each task with the highest probability after being processed by the $softmax$ layer, which is defined as follows.

$$pred_i = argmax\ softmax(W_i F_{\mathcal{G}} + b_i) \tag{3}$$

$$\mathrm{softmax}(\mathrm{z_j}) = \frac{\exp(\mathrm{z_j})}{\sum_{\mathrm{c}=1}^{\mathrm{nlabels_i}} \exp(\mathrm{z_c})} \tag{4}$$

where $softmax(W_i F_{\mathcal{G}} + b)$ contains the probabilites for each of the $nlabels_i$ possible classes of the task i; and $\mathbf{prob_i}$ is the probabilities of labels for $task_i$; W_i and b_i are the learnable weights and bias.

During the training phase, we calculate their cross-entropy losses for each task and define a multi-task loss function that averages the cross-entropy losses of all tasks as follows.

$$loss_{FVA} = -\sum_{i=1}^{7}\sum_{c=1}^{nlabels_i} \mathcal{I}(y_i = c) log(prob_i^c) \quad (5)$$

where y_i, $prob_i^c$, and $nlabels$ represent the predicted label, the predicted probability, and the number of labels of task i, respectively. $\mathcal{I}(\cdot)$ is a indicator function and its value equals 1 if $y_i = c$ is true else equals 0.

IV. Experimental Design and Settings

In this section, we first briefly introduce the dataset. Then, the state-of-the-art baselines are presented. Following that, the experimental settings and the evaluation performance are described in sequence.

A. Datasets

To develop SV assessment models, we need a large-scale dataset with vulnerable functions and their CVSS base metrics. Fan et al. [36] build a dataset named Big-Vul by collecting the vulnerability from 348 different open-source C/C++ projects, which are hosted in GitHub. Big-Vul owns the true source code vulnerabilities from 91 different vulnerability types since these vulnerabilities are collected from the public common vulnerabilities and exposures (CVE) database [37]. The dataset also provides enriched information in many aspects (e.g., CVE severity scores, code changes) for researchers to conduct a variety of tasks (e.g. vulnerability detection, vulnerability assessment, automatic vulnerability repair, etc). To ensure the quality of such a dataset, Fan et al. spend a substantial amount of manual resources to verify the correctness of the dataset and ended up with 188,636 functions.

TABLE I: The number of functions after each filtering step

No.	Filtering rules	# Functions
0	The origin dataset of Big-Vul	188,636
1	Removing non-vulnerable function	10,880
2	Removing the functions that have no CVSS base metrics	10,075
3	Removing functions without deleted lines of code	7,604
4	Removing the larges-scale functions (i.e., LOCs > 200)	7,089
5	Removing the functions that *Joern* fails to analyze	6,726
6	Selecting the functions from Top-10 projects	5,113
7	Removing the function that *git blame* fails to analyze	4,467

However, since DeepCVA, proposed at the commit level, is treated as the baseline in this study, we conduct a few filtering operations on the original dataset. There are two versions of a specific function included in the dataset: function before fix and function after the fix. The former represents the function before the vulnerability is fixed, while the latter represents the function after the vulnerability is fixed. First, we remove all the non-vulnerable functions since the SVs assessment research is conducted on the detected vulnerable functions. In addition, Big-Vul has provided rich information about the vulnerability (e.g., CVSS score), as well as contained detailed CVSS base metrics (e.g., availability, access complexity, etc.), which are important for precise SVs assessment. However, some functions are missing this important information and we filter them out. Then, we remove such functions in which their vulnerability are fixed by adding more lines of code rather than deleting some existing code. We remove them since DeepCVA builds its dataset by tracing the deleted lines in the fixed function. After that, we conduct an analysis on the size of vulnerable functions and find that functions with less than 200 lines of code (LOC) account for 93%. As for the remaining functions, they have varying function sizes, ranging from 201 LOCs to 2,697 LOCs. Therefore, to avoid the noise caused by these large-scale functions, we focus our research on functions with no more than 200 LOCs and filter out those large-scale ones. Following that, we obtain the program dependence graph by using the *Joern* tool and remove those functions, which cannot be successfully analyzed by *Joern*. Moreover, since there are many projects considered in Big-Vul and the number of vulnerabilities varies greatly in different projects. We only consider the Top-10 projects (e.g., Chrome, Linux, etc.), which have the most vulnerability following the previous practice [36]. Finally, to build the dataset for the DeepCVA approach, we adopt the *git blame* command to extract the corresponding vulnerability contributing version by tracing from the deleted lines in fixed functions. More precisely, we obtain the vulnerability contributing version with the help of PyDriller [38], which is a Python framework that helps developers on mining software repositories. Notice that, there may have a few deleted lines in fixed functions which also cause one or more vulnerability-contributing functions (VCF). To obtain the precise functions, we check each function's signature to make sure they are as same as the fixed function's signature. Finally, our studied dataset contains 4,467 functions. The details of the number of functions retained in each filtering step are listed in Table I and the distributions of curated CVSS metrics are illustrated in Fig. 2.

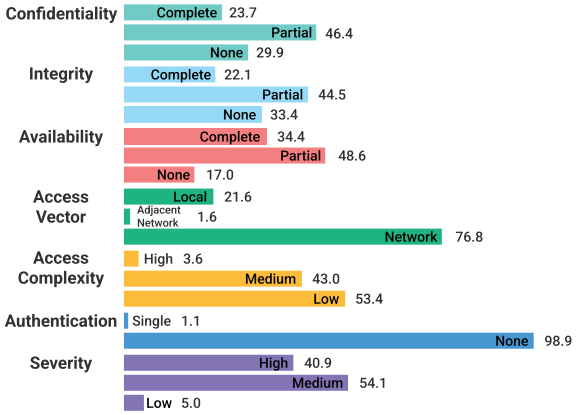

Fig. 2: Class distributions of the seven CVSS metrics.

B. Baselines

To comprehensively evaluate the performance of FVA with prior work, we totally consider four state-of-the-art approaches

(e.g., DeepCVA [10], Func [9] and CodeBERT [24]) in our study since there has limited function-level approaches are originally proposed for SVs assessment. To make our study self-contained, we briefly introduce these methods as follows.

DeepCVA. Le et al. [10] propose a novel approach named DeepCVA for software vulnerability assessment at the commit level. It extracts the closest enclosing scope of modified lines of code (i.e., *added or deleted lines*) to identify the context in the vulnerable code changes. Then, DeepCVA builds its model with a CNN (Convolutional Neural Network) with the input of three parts (i.e., added lines, deleted lines, and their contexts). Finally, DeepCVA is trained as a multi-task model for assessing the seven CVSS base metrics simultaneously.

Func. Le et al. [9] subsequently propose another framework to assess the software vulnerability at the function level. Their framework contains two major components: feature extraction and model building. As for feature extraction, they adopt some NLP methods (e.g., FastText [39], Word2Vec [40]) to extract the semantic and important information about vulnerability with the input of the vulnerable statements along with their related context (i.e., program slice context, surrounding context, and function context). As for the model building, they also consider many data-driven models (e.g., Random Forest (RF) [41], eXtreme Gradient Boosting [42], and Light Gradient Boosting Machine (LGBM) [42]) to build the SV classifiers. As stated in their study, the models built with LGBM and RF can achieve the best performance on average. Therefore, we consider the two types of models as baselines and refer them to as $\mathbf{Func_{RF}}$ and $\mathbf{Func_{LGBM}}$ separately for easier reference. To our best knowledge, this is the first study to evaluate function-level vulnerabilities.

CodeBERT. Pre-trained technologies have achieved great success in Natural Language Processing (NLP) [43], [44] and pre-trained models associated with programming languages [22], [23] also make a great process of code intelligence. Recently, these models are widely used in software engineering and indeed have achieved much success. CodeBERT proposed by Feng et al. [24] is one of the most widely used ones and is also adopted as a semantic extractor in our study. Therefore, we treat it as the baseline for the comparison.

C. Hyperparameters and Training Settings

Hyperparameter settings. We implement FVA in Python with the help of the widely used framework PyTorch [45] and run all experiments on the Nvidia GeForce RTX 3090 driven by CUDA 11.1 in Ubuntu 20.04. We use the average F1-Score performance on validating data to evaluate the merits of our model for optimizing hyperparameters. There may be too many nodes in the program dependence graph, so we limit the length of tokens to 128 for each node and set the batch size to 1 by considering the limited GPU memory. As for extracting the node embedding, we adopt the CodeBERT and obtain the 768-dimensional vector, which is then passed directly into a GCN without any other transformation. Meanwhile, we also set the dimension of the GCN's output to 300.

Training setting. For optimizing our model, we utilize the state-of-the-art stochastic gradient descent [46] algorithm to train FVA with the initialized learning rate of 0.0005. We also consider $StepLR$ to adjust the learning rate to get the optimal model and set the parameter γ in $StepLR$ to 0.9. We train FVA for at most 500 epochs since it will stop training if the best validation F1-Score does not increase in the last eight epochs. At the same time, $StepLR$ will update the learning rate if the best validation F1-Score does not increase in the last two epochs. To avoid over-fitting, we employ the Dropout strategy with a dropout rate of 0.1.

D. Evaluation Measures

To evaluate the effectiveness of FVA, we mainly consider two widely used performance measures: F1-score [33], [47] and Matthews Correlation Coefficient (MCC) [9], [10]. Both of them are suitable for datasets with class imbalance issues. The range of the F1-score is [0,1], while the range of MCC is [-1,1]. For both performance measures, the larger value a model obtains, the better performance a model has. Besides, for a comprehensive comparison among different models, we also consider another three performance measures: Accuracy, Precision, and Recall. Notice that, in our multi-classes scenario, we follow the same macro setting as prior work does for calculating these measures [10].

V. Experimental Results

To fully explore the effectiveness of FVA as well as its performance impact factors, our experiments focus on the following four research questions.

- *RQ-1: How does FVA perform compared to baseline models for function-level SV assessment?*
- *RQ-2: How does the context affect the performance of FVA?*
- *RQ-3: To what extent do different node embedding techniques contribute to SV assessment?*
- *RQ-4: How do different graph embedding techniques affect the performance of FVA?*

A. [RQ-1]: Effectiveness of Software Vulnerability Assessment

Objective: Software vulnerability assessment is vital for professionals to determine the characteristic of detected software vulnerability (SV) and consequently helps to prioritize the vulnerable functions for fixing when the resources are limited. Various effective approaches have been proposed for SV assessment, which is mostly proposed by using software vulnerability reports. Since there has been a delay when the vulnerability has been detected but their report is unavailable, recently, the researcher has proposed a few function-level SV assessment approaches which only take input as the detected vulnerable functions and output their several severity assessments. However, previous work only utilizes a portion of the statement information (e.g., a vulnerable statement as well as its limited context [9], modified lines of codes in functions [10]) which may ignore some important ones, e.g. the data or control dependency information about the function. In this study, we propose a novel approach named

FVA which considers not only the flow-sensitive information of the function but also the rich semantic information of each statement as well as its context. Therefore, we want to investigate to what extent the assessment performance can FVA achieve with comprehensive information.

Experiment Design: We totally consider four state-of-the-art baselines: DeepCVA [10], Func$_{RF}$ and Func$_{LGBM}$ [9], and CodeBERT [24]. Besides, to comprehensively compare the performance among baselines and FVA, we consider five widely used performance measures. Notice that, our approach FVA, DeepCVA as well as the CodeBERT are multi-task models, which means that they learn generalized representations of the input function and out multiple aspects of CVSS base metrics simultaneously. However, both Func$_{RF}$ and Func$_{LGBM}$ are single task models. That is, they will train separate models for each metric in the CVSS base (i.e., seven models for seven metrics).

In addition, DeepCVA is originally proposed for SV assessment at the commit level, which mainly considers the modified code in the function as well as its self-defined closest enclosing scope context. Therefore, DeepCVA is also applicable to the setting studied in this paper and we extend DeepCVA's application scope to the function level. Moreover, there has a few different types of components inside our model, e.g., the node context, node embedding, and graph model. We select the best components (e.g., one-line context, CodeBERT for node embedding, and GCN for graph model) to conduct this study and the detailed comparison will be presented in the following sections. We randomly split the dataset into three parts following previous works [33]: training data, validating data, and testing data, which account for 80%, 10%, and 10% of the original dataset, respectively.

Results: The comparison results are shown in Table II. The best performances are highlighted in bold. According to these results, we find that FVA almost outperforms all baselines in terms of five performance measures in assessing different CVSS metrics. On average, FVA obtains the best average performance in terms of Accuracy (0.869), Recall (0.813), F1-score (0.795), and MCC (0.727). More precisely, FVA performs best in most vulnerability assessment tasks except for both the $Access\ Vector$ task in terms of F1-score and the $Severity$ task in terms of MCC.

As for $Confidentiality$, FVA achieves 0.808 in terms of F1-score and 0.725 in terms of MCC, which improves baselines by 4%-11%, and by 2% - 20%, respectively. As for $Integrity$, FVA obtains 0.828 in terms of F1-score and 0.754 in terms of MCC, which improves baselines by 2%-11% and by 5%-15%, respectively. As for $Availability$, FVA obtains 0.832 in terms of F1-score and 0.753 in terms of MCC, which improves baselines by 5%-16% and by 2%-9%, respectively. As for $Access\ Complexity$, FVA achieves 0.762 in terms of F1-score and 0.649 in terms of MCC, which improves baselines by 3%-22% and by 12%-37%, respectively. As for $Authentication$, FVA achieves 0.832 in terms of F1-score and 0.668 in terms of MCC, which improves baselines by 6%-25% and by 6%-91%.

TABLE II: Software vulnerability assessment results of FVA compared against four baselines

Metrics	Model	Accuracy	Precision	Recall	F1-score	MCC
Confidentiality	DeepCVA	0.774	0.771	0.744	0.754	0.638
	CodeBERT	0.787	0.779	0.773	0.776	0.662
	Func$_{LGBM}$	0.814	0.811	**0.798**	0.804	0.709
	Func$_{RF}$	0.749	0.772	0.714	0.731	0.606
	FVA	**0.828**	**0.825**	**0.798**	**0.808**	**0.725**
Integrity	DeepCVA	0.821	0.813	0.807	0.807	0.721
	CodeBERT	0.796	0.790	0.782	0.785	0.680
	Func$_{LGBM}$	0.819	0.825	0.800	0.809	0.718
	Func$_{RF}$	0.776	0.820	0.731	0.747	0.654
	FVA	**0.843**	**0.839**	**0.821**	**0.828**	**0.754**
Availability	DeepCVA	0.834	0.840	0.766	0.787	0.729
	CodeBERT	0.823	0.810	0.776	0.789	0.710
	Func$_{LGBM}$	0.834	**0.860**	0.755	0.776	0.735
	Func$_{RF}$	0.801	0.842	0.702	0.717	0.682
	FVA	**0.848**	0.837	**0.830**	**0.832**	**0.753**
Access Vector	DeepCVA	0.933	0.828	0.746	**0.776**	0.820
	CodeBERT	0.944	0.607	0.645	0.623	0.860
	Func$_{LGBM}$	0.942	**0.866**	0.711	0.747	0.857
	Func$_{RF}$	0.933	0.776	0.650	0.664	0.834
	FVA	**0.949**	0.739	**0.830**	0.772	**0.866**
Access Complexity	DeepCVA	**0.814**	0.791	0.710	0.740	0.648
	CodeBERT	0.783	0.691	0.604	0.627	0.582
	Func$_{LGBM}$	0.772	**0.843**	0.652	0.706	0.550
	Func$_{RF}$	0.736	0.820	0.588	0.638	0.475
	FVA	**0.814**	0.769	**0.755**	**0.762**	**0.649**
Authentication	DeepCVA	0.991	0.496	0.500	0.498	0.000
	CodeBERT	0.991	0.747	0.624	0.664	0.350
	Func$_{LGBM}$	0.984	0.992	0.650	0.727	0.543
	Func$_{RF}$	0.987	**0.993**	0.700	0.782	0.628
	FVA	**0.993**	0.799	**0.873**	**0.832**	**0.668**
Severity	DeepCVA	**0.821**	0.773	0.686	0.712	**0.673**
	CodeBERT	0.792	0.741	0.642	0.669	0.616
	Func$_{LGBM}$	0.783	**0.812**	0.670	0.712	0.596
	Func$_{RF}$	0.756	0.771	0.582	0.612	0.547
	FVA	0.810	0.713	**0.785**	**0.734**	0.672
Avg.	DeepCVA	0.855	0.759	0.708	0.725	0.604
	CodeBERT	0.845	0.738	0.692	0.705	0.637
	Func$_{LGBM}$	0.850	**0.858**	0.719	0.754	0.673
	Func$_{RF}$	0.820	0.828	0.667	0.699	0.632
	FVA	**0.869**	0.789	**0.813**	**0.795**	**0.727**

As for both $Access\ Vector$ and $Severity$, our proposed method can also obtain comparable performance with the best-performing baselines: 0.772 *v.s.* 0.776 considering $Access\ Vector$ and 0.672 *v.s.* 0.673 considering $Severity$.

***Answer to RQ-1**: FVA outperforms the state-of-the-art baselines on software vulnerability assessment, especially achieving overwhelming results at both F1-score and MCC in almost all cases. It indicates that the graph integrating the global flow-sensitive information and the local semantic information can achieve better performance on software vulnerability assessment than the models considering local partial semantic information.*

B. [RQ-2]: Effectiveness of Node Context

Objective: In the constructed graph of a vulnerable function, the nodes represent the semantic information of statements. As confirmed in previous studies [9], [10], the contexts that are related to a specific statement will have a certain impact on its semantics and consequently may impact the performance

of downstream tasks based on statement understanding. Therefore, in this research question, we explore how the context of the statement affects the effectiveness of the SV assessment.

Experiment Design: We mainly consider two groups of context: surrounding context and program slicing context. The former group considers the natural statements around a specific statement. That is, for a given statement, the surrounding context only considers the statements before and after itself in order. To better compare the context's effect, we study two methods in this group: $zero\ context$ and $one-line\ context$. The latter group considers the flow-related statements around a specific statement. That is, for a given statement, the surrounding context only considers the control-dependent or data-dependent statements. More detailed information can be referred to Section III-B. Besides, we set the distance of context dependence to a specific statement as one hop since the graph model will cause a large consumption of memory. The experimental dataset is set the same as the experiment of RQ-1. Moreover, we also consider three widely used performance measures to demonstrate the difference.

TABLE III: The impacts of node context on the performance of FVA

Metric	Node Context	Accuracy	F1-Score	MCC
Confidentiality	Zero Context	**0.848**	**0.836**	**0.762**
	One-line Context	0.828	0.808	0.725
	Data Context	0.803	0.787	0.689
	Control Context	0.808	0.797	0.697
Integrity	Zero Context	**0.868**	**0.855**	**0.793**
	One-line Context	0.843	0.828	0.754
	Data Context	0.826	0.808	0.725
	Control Context	0.848	0.833	0.761
Availability	Zero Context	0.846	0.825	0.748
	One-line Context	**0.848**	**0.832**	**0.753**
	Data Context	0.839	0.816	0.737
	Control Context	0.843	0.822	0.744
Access Vector	Zero Context	**0.953**	0.752	**0.873**
	One-line Context	0.949	**0.772**	0.866
	Data Context	0.937	0.692	0.839
	Control Context	0.951	0.628	0.868
Access Complexity	Zero Context	0.801	0.731	0.631
	One-line Context	**0.814**	**0.762**	**0.649**
	Data Context	0.792	0.705	0.610
	Control Context	0.785	0.668	0.607
Authentication	Zero Context	0.991	0.748	0.495
	One-line Context	**0.993**	**0.832**	**0.668**
	Data Context	0.991	0.664	0.350
	Control Context	0.989	0.640	0.283
Severity	Zero Context	0.828	**0.763**	0.686
	One-line Context	0.810	0.734	0.672
	Data Context	**0.841**	0.758	**0.713**
	Control Context	0.821	0.706	0.674

Results: Table III shows the comparison results and the best performances are highlighted in bold for each node context setting. According to the results, we can obtain the following observations: 1) Different types of node contexts have varying impacts on the performance of software vulnerability assessment. 2) Program slicing context performs better in terms of *Severity* while surrounding contexts perform better in the rest metric of CVSS. 3) Among surrounding contexts, *zero context* performs best in terms of *Confidentiality*, *Integrity* and *Access Context*, while *one-line context* performs best in terms of *Availability*, *Access Complexity* and *Authentication*. 4) *Control context* has the least contribution among the four types of context in the SVs assessment task. 5) Overall, *one-line context* achieves the most counts of best performance.

By analyzing the comparison performance difference, we conjecture that the pre-trained model also has an influence on extracting the statements' semantics as well as their context. In this experiment, we adopt the CodeBERT to embed the statements' semantics and the CodeBERT is pre-trained on one or more successive statements, which leads to the two-fold explanations: contributing to successive statements and hurting to in-consecutive statements. The surrounding context (i.e., *one-line context* and *zero context*) has a similar usage setting with the pre-trained tasks of CodeBERT, while program slicing context (i.e., *data context* and *control context*) has no similarity since its pre-trained task does not consider the flow information.

> ***Answer to RQ-2****: Different types of contexts have varying impacts on models' performance in SV assessment tasks. The surrounding contexts perform better in almost all cases and one-line context has the overall best performance.*

C. [RQ-3]: Effectiveness of Node Embedding

Objective: The statements contain rich semantic information about the functionality of the target functions. Therefore, a good representation of statements can contribute a lot to the downstream task, such as the SV assessment in our study. In the previous studies, many approaches [24], [48], [49] have been proposed for learning the code representation and indeed achieved promising results, especially for the large-scale pre-trained models [24], [50]. Therefore, in this research question, we want to explore how different statement embedding methods affect the effectiveness of software vulnerability assessment.

Experiment Design: We consider five widely used word embedding models: LSTM [51], TextCNN [49], FastText [48], CodeBERT [24] and UniXcoder [50]. Specifically, LSTM [51] is a special Recurrent Neural Network and has a good ability to handle long-sequence training (e.g., code tokens). TextCNN [49] is a simple but well-perform convolutional neural network (CNN) with one layer of convolution on top of word vectors obtained from an unsupervised neural language model. FastText [48] is a lightweight library for efficient learning of word representations and sentence classification. CodeBERT [24] and UniXcoder [50] are pre-trained models on the basics of BERT for programming language. Therefore, these embedding methods can be grouped into two types: (1) programming language (PL) related ones and (2) non-programming language-related ones. Moreover, we also con-

sider the two comprehensive performance measures (i.e., F1-score and MCC) to demonstrate the performance difference.

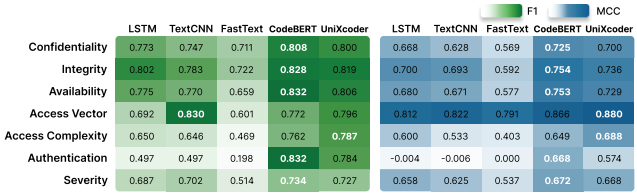

Fig. 3: The performance impact of different statement embedding methods

Results: The comparison results are shown in Fig. 3 and we use different colors to indicate the performance measure (i.e., F1-score in green and MCC in blue). Meanwhile, to better show the difference, different performance values are figured with different transparency colors and the best ones are shown in white bold. From the two figures, we obtain the following observations: 1) PL-related embedding methods (i.e., CodeBERT and UniXcoder) have a better understanding of code tokens than those (i.e., LSTM, TextCNN, and FastText) trained on none programming language documents and sentences. 2) CodeBERT performs best in most cases and FastText performs the worst in most cases. 3) The lasted state-of-the-art pre-trained model (i.e, UniXcoder trained on nine programming languages) cannot achieve good performance than the prior one (i.e., CodeBERT trained on six programming languages).

***Answer to RQ-3**: Node embedding methods have large impacts on SV assessment tasks and the embedding methods pre-trained on programming languages perform better, especially for CodeBERT.*

D. [RQ-4]: Effectiveness of Graph Embedding

Objective: The global flow-sensitive information (i.e, data and control dependence) is represented in the graph and many graph embedding methods [13], [35], [52] have been proposed and indeed achieve promising performance. Therefore, it is critically important to explore a suitable graph embedding model to assess SVs.

Experiment Design: We consider four state-of-the-art graph embedding models: GCN [35], GAT [52], GAT V2 [13], and GraphSAGE [53]. Specifically, GCN [35] is an efficient variant of convolutional neural networks by using an efficient layer-wise propagation rule, which is based on a first-order approximation of spectral convolutions on graphs. GAT [52] leverages the masked self-attentional layers to assign different weights to each neighbor node which ensures GAT focuses on the key information. Compared with GAT, GAT V22 [13] improves GAT by replacing the masked self-attention layer with the dynamic attention mechanism. GraphSAGE (Graph SAmple and aggreGatE) [53] firstly samples the neighbor nodes for each node and then aggregates the information from the selected neighbors. The experimental setting is set the same as the experiment of RQ-1. Meanwhile, we conduct the same optimization operations to the parameters of each model on the validating dataset and evaluate their performance with the two comprehensive performance measures (i.e., F1-score and MCC).

Results: The performance differences among different graph embedding models are shown in Fig. 4. To improve readability, we first calculate the average performance of all models in terms of each base metric in CVSS. For example, all models achieve an average performance of 0.815 (0.808 of GCN, 0.822 of GAT, 0.800 of GAT V2, and 0.828 of GraphSAGE) in assessing the *Confidentiality* task. Then, we draw the relative difference between each model and the average value. Meanwhile, we enlarge the performance gap by 100 times for better readability. According to the results, we can achieve the following observations: 1) There has little performance difference among all graph embedding models in most tasks. 2) Different graph embedding models perform differently in assessing the *Authentication* task. 3) Considering the extreme imbalance (1.1:98.9) of *Authentication* (cf. Section 2) as well as the good performance of GCN in *Authentication*, overall, GCN seems to be the better one.

***Answer to RQ-4**: Graph embedding models do not have obvious impacts on SVs assessment and overall GCN model is the better one.*

VI. Threat to Validity

Threats to Internal Validity mainly correspond to the potential mistakes in our implementation of both our approach and the baselines. To alleviate this threat, we not only implement these approaches by pair programming but also directly use the original source code from the GitHub repositories shared by corresponding authors. Meanwhile, we adopt the same hyperparameters as the author stated in their original papers. The authors also carefully review the experimental scripts to ensure their correctness.

Threats to External Validity mainly correspond to the studied dataset. To alleviate this threat, we directly adopt the true vulnerabilities in the widely used dataset Big-Vul, which are collected from the public common vulnerabilities and exposures (CVE). Therefore, this type of threat can be minimized.

Threats to Construct Validity mainly correspond to the performance metrics in our evaluations. To alleviate this threat, we consider a few widely used performance measures (i.e., F1-score and MCC).

VII. Related Work

A. Data-driven SV Detection

SV detection has attracted much attention from researchers and many approaches have been proposed to automate this task [54]. The rule-based approaches were proposed by leveraging known vulnerability patterns to discover potentially vulnerable code (e.g., RATS [55] and Coverity [56]). Usually,

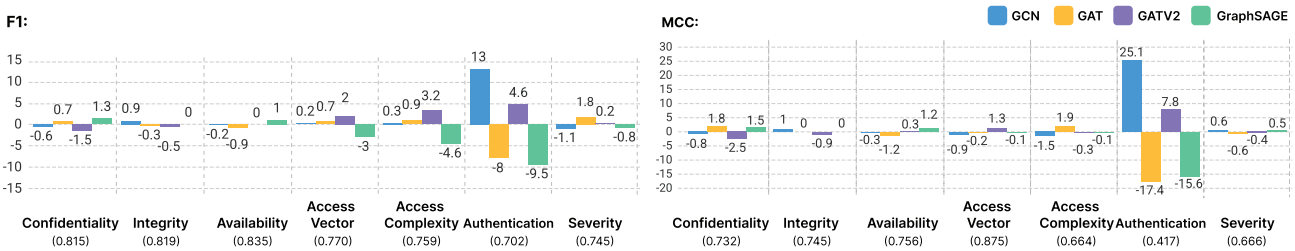

Fig. 4: The performance differences of both F1-score and MCC (multiplied by 100 for readability) of the graph embedding model variants compared to the proposed FVA in Section III. Note: The average of both F1-score and MCC values (without multiplying by 100) of the model variants are in parentheses.

the patterns are manually defined by experts and the state-of-the-art tools adopt static analysis to provide the different types of rules for varying vulnerability types.

Then, machine learning (ML)-based vulnerability detection approaches [57]–[61] become popular, which usually requires the human-crafted metrics (features) to characterize vulnerabilities and train the machine learning models on the defined features to predict whether the given code has a vulnerability or not. Following that, deep learning (DL) has been applied to detect vulnerabilities [16], [62]–[64] since these techniques can automatically learn a good representation of the vulnerability in codes. Meanwhile, many follow-up studies [65]–[67] on different granularity (e.g., component-level or file level) have been inspired and function-level SV detection tasks [34], [68], [69] have become more popular as functions are usually much smaller than files, significantly reducing inspection effort for developers.

More recently, researchers focus on the finer prediction of vulnerability in functions, such as predicting exact vulnerable statements/lines in functions [34], [70]–[72]. These approaches are proposed on the basis of the important observation that only a small number of lines is the root cause of software vulnerability in functions. Instead of detecting SVs and inspired by previous work [9], we also focus on software vulnerability assessment tasks at the function level to support software vulnerability understanding before fixing.

B. Data-driven SV Assessment

SV assessment has been an extremely important approach for addressing SVs. Meanwhile, public security databases (e.g., NVD) can provide large-scale data to determine the characteristics of SV, and expert-based SV scoring framework (e.g., CVSS) has been treated as one of the most reliable metrics for SV assessment [73]. Therefore, many approaches are proposed to automatically assess SVs [2], especially predicting the CVSS metrics (e.g., confidentiality impact, integrity impact, and availability impact) for characterizing SVs. Bozorgi et al. [74] was the first one to predict when SVs would be exploited with a Support Vector Machine model. Following that, more information about SV on NVD is explored to analyze the fine-grained information of SVs including the types [75], severity level [76], and exploitability [77]. Most of the existing works [5], [6], [78]–[80] predict the CVSS metrics by utilizing the descriptions in the bug/SV reports/databases (e.g., NVD). There also have some studies [81], [82] which leverage code patterns in fixing commits of third-party libraries to assess SVs in such libraries.

However, according to previous researches [9], [10], vulnerability reports can only be available after the corresponding vulnerabilities have been fixed, which may result in a large delay and potentially untimely for SV fixing at the check-in time. Therefore, researchers recently pay much attention to source code level vulnerability assessment. Le et al. [10] first proposed a multitask model for assessing several characteristics of vulnerability at the commit level. Following that, they proposed another approach for automating the function-level SV assessment [9]. However, these studies do not consider the structure of function (e.g., control-related or data-related) when assessing SV. In this paper, we fully utilize the structure information of source code and assess SV at the function level.

VIII. Conclusion and Future Work

We propose a flow-sensitive, multi-task learning model named FVA to tackle the function-level software vulnerability assessment. FVA can provide seven important base metrics (i.e., Confidentiality, Integrity, etc) of CVSS to practitioners promptly when a function is detected as a vulnerable one. The experimental results indicate that FVA outperforms four state-of-the-art baselines on the seven SV assessment tasks. Especially, on average, multi-task learning utilizing the relationship of assessment tasks help FVA improve baselines by 5%-14% in terms of F1-score and by 8%-20% in terms of MCC, respectively. We also release our dataset and model for public verification and further research [83].

Our future work involves extending our evaluation by considering more programming languages. We also plan to implement FVA into a tool (e.g., a GitHub plugin) to assess its usefulness in practice.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (No. 62202419), the Ningbo Natural Science Foundation (No. 2022J184) and Alibaba-Zhejiang University Joint Institute of Frontier Technologies and Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] S. Khan and S. Parkinson, "Review into state of the art of vulnerability assessment using artificial intelligence," *Guide to Vulnerability Analysis for Computer Networks and Systems*, pp. 3–32, 2018.

[2] T. H. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," *ACM Computing Surveys (CSUR)*, 2021.

[3] A. Feutrill, D. Ranathunga, Y. Yarom, and M. Roughan, "The effect of common vulnerability scoring system metrics on vulnerability exploit delay," in *2018 Sixth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2018, pp. 1–10.

[4] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 125–136.

[5] T. H. M. Le, B. Sabir, and M. A. Babar, "Automated software vulnerability assessment with concept drift," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.

[6] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems and Software*, vol. 146, pp. 152–166, 2018.

[7] Y. Yamamoto, D. Miyamoto, and M. Nakayama, "Text-mining approach for estimating vulnerability score," in *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2015, pp. 67–73.

[8] "National Vulnerability Database." [Online]. Available: https://nvd.nist.gov/

[9] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022, pp. 621–633.

[10] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[11] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[12] V. Piantadosi, S. Scalabrino, and R. Oliveto, "Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat," in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 68–78.

[13] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[14] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," *arXiv preprint arXiv:2203.11790*, 2022.

[15] D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. IEEE, 2014, pp. 294–299.

[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[17] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[18] C. Ni, K. Yang, X. Xia, D. Lo, X. Chen, and X. Yang, "Defect identification, categorization, and repair: Better together," *arXiv preprint arXiv:2204.04856*, 2022.

[19] V. Smyth, "Software vulnerability management: how intelligence helps reduce the risk," *Network Security*, vol. 2017, no. 3, pp. 10–12, 2017.

[20] "Common Vulnerability Scoring System." [Online]. Available: https://www.first.org/cvss/

[21] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[22] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[23] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547.

[25] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," *arXiv preprint arXiv:2203.02660*, 2022.

[26] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 981–992.

[27] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 560–560.

[28] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[29] W. Zheng, Y. Jiang, and X. Su, "Vu1spg: Vulnerability detection based on slice property graph representation learning," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 457–467.

[30] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[31] S. Salimi, M. Ebrahimzadeh, and M. Kharrazi, "Improving real-world vulnerability characterization with vulnerable slices," in *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, 2020, pp. 11–20.

[32] S. Dashevskyi, A. D. Brucker, and F. Massacci, "A screening test for disclosed vulnerabilities in foss components," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 945–966, 2018.

[33] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: integrating semantic features with expert features for defect prediction and localization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 672–683.

[34] Y. Li, S. Wang, and T. N. Nguyen, "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*, 2021, pp. 292–303.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[36] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[37] "Common vulnerabilities and exposures." [Online]. Available: https://cve.mitre.org/

[38] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[39] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[41] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[42] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[44] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[46] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[47] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, "Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction," *IEEE Transactions on Software Engineering*, 2020.

[48] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[49] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. [Online]. Available: https://aclanthology.org/D14-1181

[50] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

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

[52] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[53] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[54] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[55] S. S. Inc., "Rough auditing tool for security (rats)." [Online]. Available: https://code.google.com/p/rough-auditing-tool-for-security/

[56] Synopsys, "Coverity scan." [Online]. Available: https://scan.coverity.com/

[57] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[58] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.

[59] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 359–368.

[60] X. Chen, Y. Zhao, Z. Cui, G. Meng, Y. Liu, and Z. Wang, "Large-scale empirical studies on effort-aware security vulnerability prediction methods," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 70–87, 2019.

[61] X. Chen, Z. Yuan, Z. Cui, D. Zhang, and X. Ju, "Empirical studies on the impact of filter-based ranking feature selection on security vulnerability prediction," *IET Software*, vol. 15, no. 1, pp. 75–89, 2021.

[62] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[63] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[64] W. Zheng, J. Gao, X. Wu, F. Liu, Y. Xun, G. Liu, and X. Chen, "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study," *Journal of Systems and Software*, vol. 168, p. 110659, 2020.

[65] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[66] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[67] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "Predicting vulnerable components via text mining or software metrics? an effort-aware perspective," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 27–36.

[68] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[69] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[70] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 959–970.

[71] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[72] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Information-theoretic source code vulnerability highlighting," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[73] P. Johnson, R. Lagerström, M. Ekstedt, and U. Franke, "Can the common vulnerability scoring system be trusted? a bayesian analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 1002–1015, 2016.

[74] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.

[75] S. Neuhaus and T. Zimmermann, "Security trend analysis with cve topic models," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 111–120.

[76] G. Spanos, L. Angelis, and D. Toloudis, "Assessment of vulnerability severity using text mining," in *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, 2017, pp. 1–6.

[77] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin, "Predicting exploitation of disclosed software vulnerabilities using open-source data," in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, 2017, pp. 45–53.

[78] X. Duan, M. Ge, T. H. M. Le, F. Ullah, S. Gao, X. Lu, and M. A. Babar, "Automated security assessment for the internet of things," in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2021, pp. 47–56.

[79] C. Elbaz, L. Rilling, and C. Morin, "Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[80] X. Gong, Z. Xing, X. Li, Z. Feng, and Z. Han, "Joint prediction of multiple vulnerability characteristics through multi-task learning," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 31–40.

[81] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[82] ——, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[83] "FVA." [Online]. Available: https://github.com/Icyrockton/FVA