Automated Software Engineering (2025) 32:57
https://doi.org/10.1007/s10515-025-00526-4

Semi-supervised software vulnerability assessment via code lexical and structural information fusion

Wenlong Pei[1] · Yilin Huang[1] · Xiang Chen[1,2] · Guilong Lu[1] · Yong Liu[3] · Chao Ni[4]

Received: 12 August 2024 / Accepted: 17 May 2025

Abstract

In recent years, data-driven approaches have become popular for software vulnerability assessment (SVA). However, these approaches need a large amount of labeled SVA data to construct effective SVA models. This process demands security expertise for accurate labeling, incurring significant costs and introducing potential errors. Therefore, collecting the training datasets for SVA can be a challenging task. To effectively alleviate the SVA data labeling cost, we propose an approach SURF, which makes full use of a limited amount of labeled SVA data combined with a large amount of unlabeled SVA data to train the SVA model via semi-supervised learning. Furthermore, SURF incorporates lexical information (i.e., treat the code as plain text) and structural information (i.e., treat the code as the code property graph) as bimodal inputs for the SVA model training, which can further improve the performance of SURF. Through extensive experiments, we evaluated the effectiveness of SURF on a dataset that contains C/C++ vulnerable functions from real-world software projects. The results show that only by labeling 30% of the SVA data, SURF can reach or even exceed the performance of state-of-the-art SVA baselines (such as DeepCVA and Func), even if these supervised baselines use 100% of the labeled SVA data. Furthermore, SURF can also exceed the performance of the state-of-the-art Positive-unlabeled learning baseline PILOT when both are trained on 30% of the labeled SVA data.

Keywords Software vulnerability assessment · Semi-supervised learning · Information fusion · Deep learning

Extended author information available on the last page of the article

Published online: 03 June 2025

1 Introduction

Attackers exploit software vulnerabilities to gain unauthorized access, modify data, execute malicious code, or cause system crashes, which can result in significant economic losses. However, due to the complexity of the software and the limitation of software quality assurance resources (Khan and Parkinson 2018; Iannone et al. 2022), it is impossible to fix all the vulnerabilities in the software. Therefore, it is necessary to identify high-risk vulnerabilities and repair them with high priority. For example, CWE-79 (XSS vulnerability) is an example of such a high-risk vulnerability.[1] XSS vulnerabilities can allow attackers to execute malicious scripts in victims' browsers, potentially leading to the theft of sensitive information (such as login credentials and session tokens or the manipulation of web page content), and then resulting in severe economic losses. Therefore, such vulnerabilities should be fixed with higher priority.

To allocate vulnerability repair resources more effectively, it is important to determine the repair priority based on the severity and potential risks of vulnerabilities. Common Vulnerability Scoring System (CVSS) (Feutrill et al. 2018) is one of the most widely used frameworks for software vulnerability assessment (SVA). CVSS considers multiple aspects of vulnerabilities and generates a score ranging from 0 to 10, indicating the severity of the vulnerability. Therefore, these metric scores can be used to determine the priority of vulnerability fixing. As a result, practitioners started to retrieve vulnerability-related information from the National Vulnerability Database (NVD) and used CVSS metrics to predict and assess the severity of vulnerabilities[2].

Recently, researchers have proposed different data-driven approaches (Yamamoto et al. 2015; Le et al. 2019; Spanos and Angelis 2018; Elbaz et al. 2020; Gong et al. 2019) that automatically assess the CVSS metrics of software vulnerabilities using vulnerability descriptions. However, Li and Paxson (2017) found that organizations often need a substantial amount of time (with a median delay of 438 days) to publicly share vulnerability descriptions after fixing the vulnerabilities. This delay is intentional, which aims to prevent malicious actors from exploiting unpatched vulnerabilities to attack software systems (Zhou et al. 2021). However, employing vulnerability descriptions in vulnerability assessments in this setting can raise timeliness concerns.

To address this issue, researchers resorted to code-related information to construct SVA models. Le et al. (2021) first proposed the SVA approach at the commit level. This approach uses features of the commit code and its surrounding context to assess seven CVSS metrics. Later, they (Le and Babar 2022) presented a framework that integrates vulnerability statements in the function granularity and different types of code context (such as vulnerability-related/surrounding code) for constructing the SVA model. This SVA model utilizes machine learning (ML) classifiers to assess seven CVSS metrics for vulnerabilities. However, there are still two limitations to these SVA approaches based on code-related information (Le et al. 2021; Le and Babar 2022).

Limitation ❶: The performance of these data-driven approaches is heavily dependent on the quality and size of the training datasets for SVA. However, there may still

[1] http://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[2] https://nvd.nist.gov/general

be errors and significant costs when assessing CVSS metrics for vulnerabilities, even for security experts. Therefore, collecting high-quality training datasets for SVA is a challenging task.

Limitation ❷: Previous vulnerability assessment approaches rely only on lexical data from the code without incorporating structural information from the code, potentially hurting the performance of the SVA model.

To alleviate these two limitations, we propose a novel approach **SURF** (Semi-Supervised Vulnerability Assessment via Code Lexical and Structural Information Fusion). Specifically, for a large amount of unlabeled SVA data (i.e., vulnerable codes), we can label a limited number of SVA data. Then we can employ semi-supervised learning (Zhu and Goldberg 2009) to train the SVA model by considering these labeled SVA data and the remaining unlabeled SVA data. During SVA model construction, we simultaneously incorporate both the lexical information (i.e., treat the code as plain text) and structural information (i.e., treat the code as the code property graph) as the bimodal input, which can help the SVA model perform a more comprehensive understanding of the code.

To evaluate the effectiveness of SURF, we conduct extensive comparative experiments on 4,467 C/C++ vulnerable functions gathered from Big-Vul (Fan et al. 2020), which is a large-scale dataset for previous vulnerability detection studies (Li et al. 2021). Based on our empirical results, we find that only using 30% of the labeled SVA data, SURF can outperform state-of-the-art supervised baselines (i.e., DeepCVA (Le et al. 2021) and Func (Le and Babar 2022), which are trained on 100% of the labeled SVA data) as well as state-of-the-art Positive-unlabeled (PU) learning baselines like PILOT (Wen et al. 2023), in terms of F1 score and MCC performance measures. Specifically, SURF obtains 0.690 of the F1 score and 0.591 of MCC on average, which outperforms the baselines by 11%∼24% and 22%∼39% respectively. Ablation studies also verify the effectiveness of component settings (such as input modal setting and code representation learning through GraphCodeBERT (Guo et al. 2020)) in SURF.

Our study shows that the use of semi-supervised learning is an effective way to alleviate the challenge of labeling SVA data. Moreover, instead of only considering lexical information, incorporating more code structural information is a promising research direction for further improving the performance of SVA models.

To the best of our knowledge, the main contributions of our study can be summarized as follows.

- **Direction**. We are the first to focus on the high cost of the SVA data labeling issue and resort to semi-supervised learning to alleviate this issue, which only needs a limited amount of the labeled SVA data.
- **Approach**. We propose a semi-supervised vulnerability assessment approach SURF by fusing lexical and structural information from the code, which can comprehensively capture the code's semantic information.
- **Study**. We compare SURF with state-of-the-art SVA baselines on 4,467 C/C++ vulnerable functions and investigate the rationality of the component setting (such as the input modal setting and representation learning approaches) in SURF.

Open Science To support the open science community, we share our studied experimental subjects, code scripts, and detailed experimental results in our GitHub repository (https://github.com/SVA-CVSS/SURF001).

Paper Organization Section 2 provides the background of our study and highlights the limitations of previous studies. Section 3 presents the details of our proposed approach SURF. Section 4 describes our experimental setup. Section 5 shows our experimental results. Section 6 conducts additional discussions. Section 7 analyzes the potential threats to the validity. Section 8 summarizes the related work and shows the novelty of our study. Section 9 concludes our study and discusses future work.

2 Background

2.1 Vulnerability assessment with CVSS

Software vulnerability assessment is a crucial step in vulnerability detection and fixing (Smyth 2017; Wen et al. 2023). Due to limited resources, it is essential to assess the risk of vulnerabilities since high-risk vulnerabilities may pose higher threats to the software. Therefore, SVA can help developers prioritize their efforts on high-risk vulnerabilities. The CVSS is one of the most commonly used frameworks for SVA.[3] Specifically, CVSS provides a standardized approach to assess the severity and impact of vulnerabilities. It considers different aspects of vulnerabilities (such as exploitability, impact, and severity) and assigns corresponding weights to each aspect. By synthesizing these aspects, CVSS generates a score ranging from 0 to 10, indicating the severity of the vulnerability, with 10 representing the most severe vulnerability.

The calculation of the CVSS score is based on a set of metrics, including base metrics, temporal metrics, and environmental metrics. Specifically, the base metrics consider the characteristics of the vulnerability itself. The temporal metrics take into account changes in the vulnerability over time (such as its disclosure and exploitability). The environmental metrics allow evaluators to personalize vulnerability assessments based on organizational or specific system context. However, obtaining temporal and environmental metrics directly from source code or project descriptions can be challenging. Therefore, previous studies (Le et al. 2019; Spanos and Angelis 2018; Yamamoto et al. 2015; Han et al. 2017; Le et al. 2021; Le and Babar 2022) mainly calculate the CVSS score focusing only on base metrics.

Base metrics for CVSS version 2 have been widely used and consist of seven metrics. Specifically, ❶ **Access Vector** describes the attacker's contact with the affected system, considering the distinction between local and remote attacks. ❷ **Access Complexity** measures the attacker's conditions required to exploit the vulnerability, ranging from low to high difficulty. ❸ **Authentication** indicates the authentication the attacker needs before exploiting the vulnerability, covering scenarios of no authentication, single authentication, and multiple authentications. ❹ **Confiden-**

[3] https://www.first.org/cvss/

tiality Impact describes the extent to which successful exploitation of vulnerability affects confidentiality. ❺ **Integrity Impact** measures the impact on integrity caused by the exploitation of the vulnerability. ❻ **Availability Impact** indicates the extent to which the successful exploitation of the vulnerability affects system availability. Finally, ❼ **Severity** indicates the difficulty of exploiting the vulnerability. For this metric, the first six metrics are combined to calculate a score ranging from 0 to 10, which can then be classified into three categories: low (0–3.9), medium (4–6.9), and high (7–10). Therefore, we incorporate all seven of these metrics into the SVA task, which is consistent with previous SVA studies (Le et al. 2021; Le and Babar 2022).

2.2 Semi-supervised learning

Semi-supervised learning is a machine learning approach aimed at bridging the gap between supervised learning and unsupervised learning (Chapelle et al. 2006; Zhu 2005). In semi-supervised learning, only a small subset of samples is labeled, while the majority of samples remain unlabeled. This situation is common in practical applications because obtaining labeled data is typically time-consuming, expensive, and error-prone (Zhu 2005). Through semi-supervised learning, models can effectively leverage the combination of limited labeled data and a vast amount of unlabeled data for training. This helps reveal hidden information within the unlabeled data and assists the model in gaining a better understanding of the data distribution and structure, thus improving the performance and robustness of models while reducing the costs of labeling data. Until now, researchers have developed various semi-supervised approaches, such as self-training (Triguero et al. 2015), co-training (Blum et al. 2004), and label propagation (Iscen et al. 2019).

2.3 Research motivation

In early studies, researchers developed different data-driven approaches (Costa et al. 2022; Shahid and Debar 2021; Le et al. 2019; Yamamoto et al. 2015; Spanos and Angelis 2018), using vulnerability descriptions to automatically assess the CVSS metric of software vulnerabilities. However, there can be significant delays in acquiring vulnerability description information to deter attackers from exploiting unpatched vulnerabilities, which can lead to timeliness concerns. Therefore, recent studies (Le et al. 2021; Le and Babar 2022) came to utilize code information for the SVA task. However, there still exist two limitations to these SVA approaches based on code information.

Limitation ❶: In a recent survey, Le et al. (2022) mentioned that the labeled data for vulnerability assessment and prioritization tasks are quite limited. Gathering vulnerable code in different projects is relatively straightforward. However, assigning CVSS metrics to these vulnerable codes is susceptible to errors, even for security experts. Moreover, small and medium-sized enterprises often lack the necessary resources (such as sufficient security experts or labeling budgets) to construct large-scale, high-quality vulnerability datasets. This limitation poses significant challenges for training fully supervised models in practical settings. Therefore, a possible scenario is that we can label a limited amount of vulnerable code and then resort to semi-

supervised learning to train the SVA model based on a limited amount of labeled SVA data and a large amount of unlabeled SVA data. To the best of our knowledge, this scenario has not been thoroughly investigated in previous SVA studies.

Limitation ❷: Previous vulnerability assessment approaches only utilize code lexical information, such as vulnerability statements and their partial context (Le and Babar 2022), or lines of code modified within functions (Le et al. 2021). However, these previous SVA approaches ignore the structural information (such as control flow information and data flow information) of functions. Therefore, we consider enhancing code representations by combining lexical information from the code with structural information to achieve a more comprehensive understanding of the code.

3 Approach

To alleviate the aforementioned two limitations, we propose SURF, a semi-supervised vulnerability assessment approach by considering the lexical and structural information of the code. As shown in Fig. 1, SURF mainly consists of three phases: ❶ Data Pre-processing phase, ❷ Graph Construction and Representation Learning phase, and ❸ Model Training and Vulnerability Assessment phase.

3.1 Data pre-processing phase

In this phase, we perform data preprocessing to acquire a dataset suitable for software vulnerability assessment. Specifically, we first select vulnerable functions from real-world software projects. In our study, we select the dataset Big-Vul shared by Fan et al. (2020) as our experimental subject and only select C/C++ vulnerable functions from different projects. Second, following the perspective of McIntosh and Kamei (2018), we remove functions with code exceeding 200 lines to avoid the negative impact of large functions on the SVA task. Third, based on our observation, we find that there exists a significant difference in the number of vulnerabilities for different projects in Big-Vul. Specifically, the top 10 projects account for 79.47% of the total vulnerable functions in Big-Vul (Fan et al. 2020). Therefore, we only consider the top 10 projects to ensure the quality of our experimental subjects. Finally, we remove

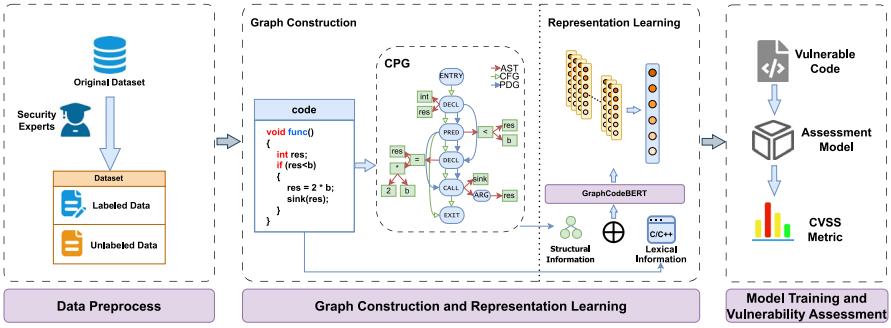

Fig. 1 Framework of our proposed approach SURF

empty lines and comments in all the selected functions, since they do not impact the functionality of the code. By following the settings of previous semi-supervised learning studies (Zhang et al. 2021; Berthelot et al. 2020), in our study, we select 30% of the vulnerable functions as labeled SVA data, and the remaining functions as unlabeled SVA data to simulate our investigated scenario.

3.2 Graph construction and representation learning phase

The primary objective of this phase is to map code lexical information and its associated structural information into a continuous vector space, facilitating effective analysis and comparison within this space. Specifically, in the graph construction step, we use the Code Property Graph (CPG) (Yamaguchi et al. 2014) to model the code. In the representation learning step, we feed both the code lexical information (treated as plain text) and structural information (treated as graph representation via CPG) as the bimodal input to the pre-trained model GraphCodeBERT (Guo et al. 2020) and generate the final vector representation of the functions.

3.2.1 Graph construction step

In this step, we model the function as the CPG, which is composed of the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). By integrating AST, CFG, and PDG into a unified graph representation, we can capture key features (such as syntax, control flow, and data dependencies) of the code, which can facilitate a more comprehensive understanding and analysis of the functions. Formally, we show the definition of CPG as follows:

Definition 1 A CPG $G = (N, E, \lambda, V)$ is constructed from the AST, CFG, and PDG of source code with

- $N = N_A$,
- $E = E_A \cup E_C \cup E_P$,
- $\lambda = \lambda_A \cup \lambda_C \cup \lambda_P$ and
- $V = V_A \cup V_E$

Where N_A denotes a set of the nodes in CPG, E_A, E_C, and E_P denote the corresponding sets of directed edges labeled by the labeling functions λ_A, λ_C, and λ_P for AST, CFG, and PDG, respectively. The properties of edges and nodes can be determined through functions $V : (N \cup E) \times K \rightarrow S$ where K is the set of property keys, and S is the set of property values. Since the CPG contains multiple graphs, two nodes can be connected by multiple edges.
In our study, we extracted the CPG from the function by using the code analysis platform Joern[4].

[4] https://joern.io/

3.2.2 Representation learning step

After obtaining the graph representation of the function, we perform representation learning of the function. Recently, pre-trained models for code (such as CodeBERT (Feng et al. 2020), GraphCodeBERT (Guo et al. 2020) and CodeT5 (Wang et al. 2021)) have shown promising performance in various software engineering tasks (such as automatic program repair (Xia and Zhang 2022; Zhang et al. 2022), automatic code generation (Paolone et al. 2020; Le et al. 2022), and source code summarization (Zhou et al. 2022; Gong et al. 2022)). Considering the different structural information of the code, we chose to use GraphCodeBERT to learn the representation of the code. GraphCodeBERT was introduced by Guo et al. (2020), which leverages the foundation of the pre-trained BERT (Devlin et al. 2018) model. This pre-trained model not only incorporates the underlying structure of code but also introduces tasks that emphasize structural awareness. Previous studies (Li et al. 2021; Hin et al. 2022) found that treating code as plain text or treating code as a graph has its advantages, so considering an effective fusion approach would be a promising way to utilize two types of code information effectively.

In our study, we fuse both the information as plain text of the code X_{code} and the graph information of the code X_{graph} as the bimodal input for the training of the SVA model. To clearly distinguish these two types of information, we utilize special identifiers. Specifically, we add a special identifier <code> in front of the plain text representation and another special identifier <graph> to distinguish between the plain text representation and the graph representation. These special identifiers play a crucial role in guiding the SVA model to distinguish between plain text representation and graph representation.

The format of the bimodal input is shown as follows:

$$X =< code > \oplus X_{code} \oplus < graph > \oplus X_{graph} \tag{1}$$

After obtaining the source code and constructing its CPG, we first perform code chunking based on the structural boundaries of the CFG. Using control flow nodes such as if/else and for/while as splitting points, we segment the code into blocks while ensuring the syntactic integrity of each sub-block and the closure of variable scopes. During the chunking process, the CPG undergoes topological adjustments, where the original AST nodes and CFG edges are divided into subgraphs. Additionally, the representations of each graph are simplified into symbolic formats.

Subsequently, each sub-code block and its corresponding CPG subgraph are encoded separately using GraphCodeBERT. Since our focus is only on the overall semantics of the code blocks, we employ a straightforward average pooling strategy during the hierarchical block fusion stage. This approach aggregates the representations of all blocks to generate the global vector representation of the code. Figure 2 uses a simple code snippet to illustrate our bimodal representation learning process.

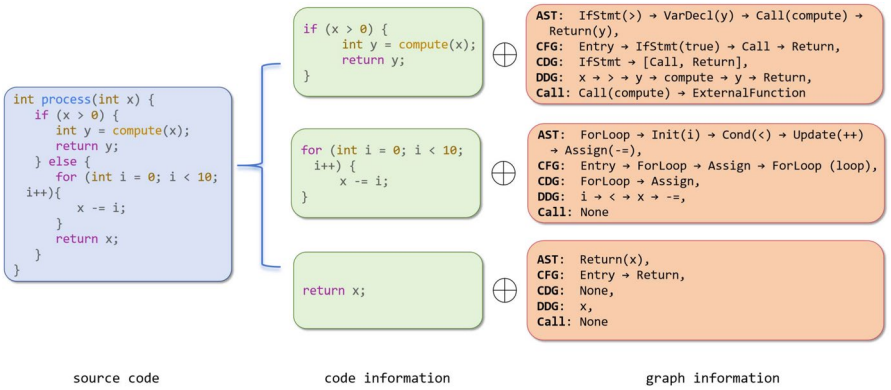

Fig. 2 The example of bimodal representation learning

3.3 Model training and vulnerability assessment phase

3.3.1 Model training phase

With the rise of deep learning (DL), researchers have begun to apply DL techniques to the field of semi-supervised learning. The advantages of DL models in the context of large-scale datasets, particularly their ability to learn feature representations, have brought new opportunities to semi-supervised learning. A range of DL-based semi-supervised approaches have been proposed (such as GANs (Goodfellow et al. 2014), Autoencoders (Kingma and Welling 2013)). These approaches improve model performance by incorporating unlabeled data into the network and have shown promising results.

However, in the field of SVA, the diversity of vulnerabilities and the complexity of the SVA task make the training of deep learning models more complex, which also brings challenges to semi-supervised learning approaches. Therefore, we follow the ideas of Sohn et al. (2020), which is a self-training approach, and use two different data enhancement operators to improve the stability of SVA. The detailed process is shown in Algorithm 1:

Algorithm 1 Construction of a vulnerability assessment model

Input: Labeled Data D_l, Unlabeled Data D_u
Initialize Initialize assessment model with random weights M
$t \leftarrow 0$ ▷ Initialize the iteration counter
while $t < T$ **do** ▷ Run for T iterations
  Generate pseudo-labels for D_u using the current model M
  $D_{lp} \leftarrow D_l \cup D_u$
  $\ell_s \leftarrow \text{ComputeSupervisedLoss}(M, D_l)$
  $\ell_u \leftarrow \text{ComputeUnsupervisedLoss}(M, D_{lp})$
  $\ell_{total} \leftarrow \ell_s + \ell_u$
  $M \leftarrow \text{Optimize}(M, \ell_{total})$
  $t \leftarrow t + 1$ ▷ Increment the iteration counter
end while
Output: Assessment Model M

First, we use random weights to initialize the SVA model (Initialize). Subsequently, we use the current SVA model to generate pseudo-labels for unlabeled SVA data (Step 3), and then we use a supervised loss function to train weakly-augmented samples (Step 5) with the following formula:

$$\ell_s = \frac{1}{B}\sum_{b=1}^{B} \mathrm{H}\left(p_b, p_{\mathrm{m}}\left(y \mid \alpha\left(x_b\right)\right)\right) \tag{2}$$

where B represents the number of samples in a batch, p_b represents the predicted label for the samples x_b obtained using a weak augmentation approach $\alpha\left(x_b\right)$, $\mathrm{H}\left(p_b, p_{\mathrm{m}}\left(y \mid \alpha\left(x_b\right)\right)\right)$ represents the cross-entropy loss between the predicted label p_b and the true label y for the sample x_b when using the weak augmentation approach $alpha\left(x_b\right)$. Then, we use an unsupervised loss function to train strongly augmented samples (Step 6) with the following formula:

$$\ell_u = \frac{1}{\mu B}\sum_{b=1}^{\mu B} \left(\max\left(q_b\right) \geq \tau\right) \mathrm{H}\left(\hat{q}_b, p_{\mathrm{m}}\left(y \mid \mathcal{A}\left(u_b\right)\right)\right) \tag{3}$$

Where B represents the number of samples in a batch, $\max\left(q_b\right)$ represents the maximum value of the predicted results obtained for the unlabeled sample μb using strong augmentation approach $\mathcal{A}\left(u_b\right)$. $\mathrm{H}\left(\hat{q}_b, p_{\mathrm{m}}\left(y \mid \mathcal{A}\left(u_b\right)\right)\right)$ represents the cross-entropy loss between the predicted label distribution and the true label y for the sample μb when using the strong augmentation approach $\mathcal{A}\left(u_b\right)$. Then we perform optimization by using the Adam algorithm (Kingma and Ba 2014). By comparing the assessment differences between these two approaches, we can reduce the assessment noise of weakly augmented samples and improve the performance of the SVA model.

Furthermore, since the Wide ResNet model considered by the original study of Sohn et al. (2020) is utilized for processing image data, for the SVA task, we employ FT_Transformer (Gorishniy et al. 2021) as a substitute, which serves as the backbone network for SVA model training. Specifically, FT_Transformer is a simple adaptation of the Transformer architecture (Vaswani et al. 2017). It introduces a feature tokenizer before the features are fed into the Transformer. This tokenizer converts all features (including categorical and numerical ones) into embeddings and applies the Transformer layers to these embeddings. Therefore, each Transformer layer of the FT_Transformer operates at the feature level, making it highly effective at capturing relationships and features within code representation data.

3.3.2 Vulnerability assessment phase

After training the SVA model, given a vulnerable function that needs to be assessed, SURF will construct the corresponding CPG, input the code lexical information and structural information together into the trained model, and finally get the assessment score.

4 Experimental evaluation

In this section, we first show our designed research questions and their motivation for design. Then we introduce the experimental subjects, baselines, performance evaluation measures, and the details of our experimental setup.

4.1 Research questions

In our empirical study, we design the following three research questions.

RQ1: How effective is SURF compared to the state-of-the-art baseline?

Motivation Since we are the first to consider the scenario of constructing the SVA model with a limited number of labeled data through semi-supervised learning, no baseline can be directly compared with our proposed approach SURF. In the previous studies, Le et al. proposed supervised SVA approaches (i.e., DeepCVA (Le et al. 2021) and Func (Le and Babar 2022)). Therefore, we consider comparing the scenario where SURF only uses the SVA data labeled with 30% with the scenario where these two baselines use the SVA data labeled with 100%. In addition to supervised learning baselines, we also consider PILOT (Wen et al. 2023), a state-of-the-art PU learning-based semi-supervised approach, as a relevant baseline for comparison. To show the competitiveness of SURF, we consider five automatic evaluation measures (i.e., Accuracy, Precision, Recall, F1 score, and MCC) in this RQ.

RQ2: How do the different input modal settings affect the performance of SURF?

Motivation Since our approach SURF involves capturing the code's structural information by generating its CPG, this structural information is then combined with the lexical information to create the bimodal input for the SVA model training. To investigate whether this bimodal input setting can achieve the best performance for SVA, we designed this RQ to investigate the impact of other different input modal settings (such as only considering the code as plain text, only considering the code as CPG or bimodal input by further considering the code as abstract syntax tree) on the performance of SURF.

RQ3: Can using GraphCodeBERT for code representation learning achieve the best performance of SURF?

Motivation Since the bimodal input contains graph representation information from the code, we chose GraphCodeBERT (Guo et al. 2020) for code representation learning. In addition to GraphCodeBERT, several different representation learning methods have also emerged, such as the traditional graph embedding methods (Grover and Leskovec 2016; Narayanan et al. 2017), the pre-training methods (Feng et al. 2020; Wang et al. 2021), and deep neural network-based methods (Kipf and Welling 2016; Velickovic et al. 2018). To consider different types of representation learning methods, we select the most representative methods for each category, including Node2

Vec (Grover and Leskovec 2016), CodeBERT (Feng et al. 2020), and GCN (Kipf and Welling 2016). Then we design this RQ to analyze whether the use of GraphCodeBERT can achieve the best performance for SURF.

4.2 Experimental subject

To evaluate the effectiveness of SURF, we select the Big-Vul dataset (Fan et al. 2020) as our experimental subject. This dataset contains real-world C/C++ code vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database and covers 91 different types of vulnerabilities[5]. The Big-Vul dataset has been publicly released on GitHub and has been widely used in previous software vulnerability detection studies (Li et al. 2021) due to its high quality[6]. To adapt the Big-Vul dataset for the SVA task, we perform the following preprocessing steps. First, we use only vulnerable code, since our goal is to assess vulnerabilities, not to detect whether a function is vulnerable. For each vulnerability, we collect its CVSS metrics, the corresponding vulnerable code, and the function implementations before and after the fix. Second, following our data preprocessing steps (in Section 3.1), we remove all non-vulnerable functions and functions with more than 200 lines of code. Then we only keep the vulnerable functions from the top 10 projects. Since DeepCVA (Le et al. 2021) requires constructing datasets by deleting codes from code commits, we exclude functions that repair vulnerabilities by only adding code. Note that repairing a vulnerable function may result in one or more vulnerability-contributing functions. We examine the signature of each function, removing any functions that do not match the signature of the fixing function. Finally, we remove functions that cannot be successfully analyzed by Joern. Therefore, our construct dataset finally comprises 4,467 C/C++ vulnerable functions for the SVA task.

We show the distribution of CVSS metric categories for our experimental subject in Fig. 3. Taking the metric Access Vector as an example, this metric contains three categories (i.e., Local, Adjacent Network, and Network). In our dataset, the functions with the category Local account for 1.6%, the functions with the category Adjacent Network account for 21.6%, and the functions with the category Network account for 76.8%. Note that the categories of other CVSS metrics and their meaning can be found on the official website of NVD for more detailed information.[7]

4.3 Baselines

Since we are the first to consider the scenario of constructing the SVA model with a limited number of labeled SVA data, no baseline can be directly compared with SURF. Therefore, we consider the three state-of-the-art approaches for SVA (i.e., DeepCVA (Le et al. 2021), Func (Le and Babar 2022) and PILOT (Wen et al. 2023)) as our baselines. We briefly introduce these three baselines as follows.

[5] https://cve.mitre.org/

[6] https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

[7] https://nvd.nist.gov/

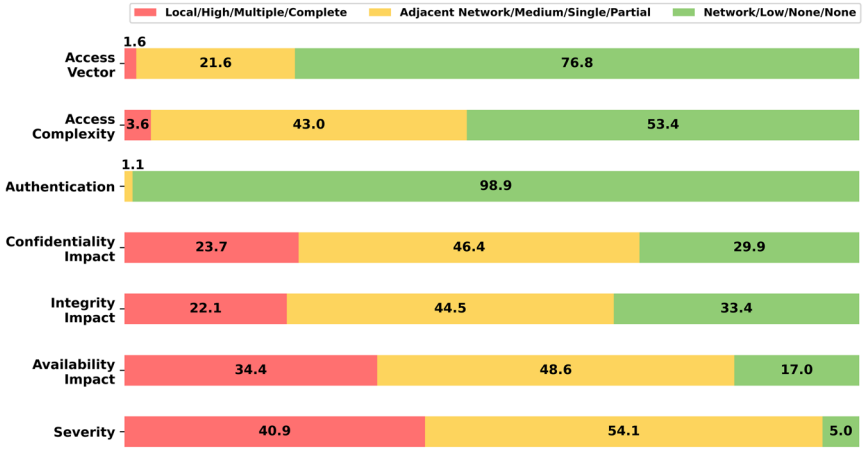

Fig. 3 Category distribution of the seven CVSS metrics of our dataset

DeepCVA Le et al. (2021) proposed a novel deep multi-task learning model called DeepCVA, which utilizes deep multi-task learning for commit-level vulnerability assessment. DeepCVA is the first effective and efficient solution for SVA. In our experiment, we only adopted the methodological components of DeepCVA and utilized function-level vulnerable code as the input to train the SVA model.

Func Le and Babar (2022) proposed a machine learning model for function-level vulnerability assessment, utilizing vulnerable and non-vulnerable statements and extracting context from vulnerable statements to develop a function-level vulnerability assessment model. To train different data-driven models to assess seven common CVSS metrics, they considered Logistic regression (LR), support vector machine (SVM), random forest (RF), and light gradient boosting machine (LGBM). Since their empirical results showed that using LGBM and RF can achieve the best performance for Func, we considered the following two classifiers for Func as our baselines: Func_{LGBM} and Func_{RF}.

PILOT Wen et al. (2023) introduced a novel Positive-unlabeled (PU) learning model named PILOT, specifically designed for vulnerability detection in software systems. PILOT addresses the challenge of limited labeled data by leveraging a small set of positive samples (known vulnerable code) and a large amount of unlabeled data, which may contain both vulnerable and non-vulnerable instances. In our experimental setup, PILOT is adapted for the software vulnerability assessment task by modifying the classifier component of the PILOT model to handle multi-class classification. We adopt PILOT as a baseline to evaluate its effectiveness in scenarios with scarce labeled data.

To guarantee a fair comparison, we use the same experimental subject and the dataset split method. Note that DeepCVA and Func are compared using 100% of

the SVA-labeled data, while PILOT and SURF are evaluated using only 30% of the SVA-labeled data.

Notice that we did not select vulnerability description-based approaches (Sun et al. 2023; Ni et al. 2022) as our baselines since these approaches only rely on vulnerability descriptions to conduct vulnerability assessment, making them unsuitable for comparison with our proposed approach only based on vulnerability code information.

4.4 Evaluation measures

Following previous studies (Le et al. 2021; Le and Babar 2022), we also consider five performance measures: Accuracy, Precision, Recall, F1 score, and Matthews Correlation Coefficient (MCC), which can provide a comprehensive performance evaluation.

We define our model's performance metrics using true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Specifically, TP represents instances where the model correctly identifies vulnerabilities as belonging to the considered vulnerability severity category. TN indicates the model's accurate identification of all other categories that do not fall under the considered vulnerability severity category. FP refers to cases where the model incorrectly classifies vulnerabilities from other vulnerability severity categories into the considered vulnerability severity category. FN denotes instances where the model fails to identify vulnerabilities within the considered vulnerability severity category.

Accuracy: Accuracy in the SVA task usually measures the ratio of correct vulnerability severity assessments by the model.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4}$$

Precision: Precision reflects the model's accuracy when predicting that a vulnerability belongs to the considered severity category.

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

Recall: Recall refers to the proportion of TP samples correctly identified by the model.

$$Recall = \frac{TP + TN}{TP + FN} \tag{6}$$

F1 score: F1 score is a classification model performance measure that combines precision and recall, representing their harmonic mean.

$$F1score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{7}$$

MCC: MCC is a widely used performance measure for the dataset with the class imbalanced problem.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{8}$$

Since the CVSS metrics we need to assess are all three-category metrics, we follow the previous work (Novielli et al. 2018) and use macro-averaged metrics to present the final results. For example, the formula for macro-averaged precision is shown as follows:

$$Precision_{\text{ macro}} = \frac{\sum_{i=1}^{k} Precision_i}{k} \tag{9}$$

where P_i represent the precision for the i-th class (i.e., category) respectively, k denotes the number of categories for each CVSS metric. Similarly, we can calculate the macro-averaged values for other performance measures.

The score of the first four performance measures ranges from 0 to 1 (1 indicates the best and 0 indicates the worst), while the MCC ranges from −1 to 1 (1 indicates perfect, 0 indicates random, and −1 indicates complete inconsistency). The higher the values of these performance measures, the better their model performance.

4.5 Implementation details

During the dataset splitting phase, we first select 10% of the SVA data as test data. For the remaining SVA data, we select 90% of the SVA data as training data, and the rest of the SVA data as validation data. To simulate our investigated scenario, we randomly selected 30% of the training data as the labeled SVA data and the remaining as unlabeled SVA data.

To implement our proposed approach, we use GraphCodeBERT, which is available in the Hugging Face[8]. To avoid the overfitting issue, we stop the model training when the SVA model's performance on the validation set starts to degrade. Regarding the representation learning phase, we use the default hyperparameter settings of GraphCodeBERT to obtain a 768-dimensional vector representation. Specifically, we set the learning rate to 1e-4, employ a batch size of 32, limit the maximum sequence length to 256, and limit the maximum number of nodes to 64. This vector is then directly fed into the model training phase via semi-supervised learning. Regarding the model training phase in Algorithm 1, we used Adam (Kingma and Ba 2014) as the optimizer for the model training and configured the iteration number (i.e., the parameter T in Algorithm 1) to 5,000 with a learning rate of 1e-4.

[8] https://huggingface.co/microsoft/graphCodeBERT-base

All experiments were performed on a computer with a GeForce RTX 4090 GPU with 24 GB of graphics memory.

5 Experimental results

5.1 RQ1: comparison with baselines

Approach In this RQ, our primary focus is on comparing the performance of our proposed approach SURF with the state-of-the-art SVA baselines (i.e., DeepCVA (Le et al. 2021), Func$_{RF}$(Le and Babar 2022), Func$_{LGBM}$(Le and Babar 2022), PILOT (Wen et al. 2023)). To show the competitiveness of SURF when compared to the state-of-the-art SVA baselines, we employ widely used performance measures, including Accuracy, Precision, Recall, F1 score, and MCC. Following the data partition strategy mentioned above, we use only 30% of the training set as labeled SVA data, while using the remaining data as unlabeled SVA data. Notably, since PILOT is specifically designed for PU learning, it also uses only 30% of the training data as labeled data, similar to our approach. Then, for the remaining supervised baselines, we assume all the training data is labeled. Through a comprehensive analysis of the performance difference between SURF and these baselines, we aim to analyze whether SURF, which only uses 30% labeled data, can achieve similar or even better performance compared to the baselines using all the labeled data.

Results The comparison results can be found in Table 1, with the best results highlighted in bold. Based on these results, we can observe that even with only 30% of labeled SVA data, SURF can achieve better performance than the baselines on average. Specifically, SURF achieves promising performance in terms of Accuracy (0.799), Precision (0.785), Recall (0.664), F1 score (0.690), and MCC (0.591). For example, in the case of Integrity, SURF achieves an F1 score of 0.729 and an MCC score of 0.601, which shows the performance improvement of 3% to 14% and 5% to 21% compared to baselines. In the case of Access Complexity, SURF achieves an F1 score of 0.609 and an MCC score of 0.456, showing a performance improvement of 10% to 26% and 1% to 11% compared to baselines. For some metrics (such as Confidentiality and Access Vector), SURF can achieve similar performance in terms of F1 score and MCC. Since only 30% of the labeled data was used, this result still indicates the competitiveness of our proposed approach.

Notice that due to the extreme imbalance of Authentication shown in Fig. 3, its category distribution exhibits a ratio of 0:1.1:98.9, leading to a significant performance decrease for all baselines. However, SURF can effectively alleviate this class imbalance issue and still achieve promising performance. This highlights the advantages of using semi-supervised learning for SVA, which improves the SVA model's generalization ability and robustness by learning shared features and patterns from unlabeled SVA data.

Finally, we analyze the training costs of the models. Baseline Func$_{RF}$ and Func$_{LGBM}$ are based on traditional machine learning techniques. Therefore, these

Table 1 Comparison results between SURF and SVA baselines

Metrics	Approach	Accuracy	Precision	Recall	F1 score	MCC
Confidentiality	DeepCVA	0.673	0.669	0.664	0.666	0.532
	Func_{RF}	0.662	0.711	0.617	0.632	0.479
	Func_{LGBM}	0.707	**0.732**	0.676	**0.688**	0.549
	PILOT	0.669	0.706	0.611	0.623	0.478
	SURF	**0.718**	0.702	**0.680**	0.686	**0.553**
Integrity	DeepCVA	0.706	0.701	0.691	0.691	0.575
	Func_{RF}	0.685	**0.752**	0.641	0.656	0.511
	Func_{LGBM}	0.720	0.749	0.693	0.707	0.566
	PILOT	0.678	0.736	0.625	0.637	0.494
	SURF	**0.745**	0.736	**0.724**	**0.729**	**0.601**
Availability	DeepCVA	0.733	0.705	0.696	0.700	**0.602**
	Func_{RF}	0.700	0.781	0.590	0.582	0.513
	Func_{LGBM}	0.734	0.772	0.657	0.674	0.567
	PILOT	0.716	**0.807**	0.593	0.596	0.525
	SURF	**0.758**	0.742	**0.698**	**0.712**	0.601
Access Vector	DeepCVA	0.869	0.650	0.603	0.621	0.724
	Func_{RF}	0.879	0.561	0.547	0.553	0.652
	Func_{LGBM}	**0.924**	**0.924**	0.658	**0.684**	**0.794**
	PILOT	0.887	0.574	0.557	0.564	0.681
	SURF	**0.924**	0.922	**0.660**	0.683	0.793
Access Complexity	DeepCVA	0.672	0.505	0.485	0.485	0.422
	Func_{RF}	0.698	0.800	0.494	0.506	0.409
	Func_{LGBM}	**0.718**	**0.812**	0.528	0.555	0.450
	PILOT	0.650	0.770	0.545	0.541	0.448
	SURF	**0.718**	0.743	**0.572**	**0.609**	**0.456**
Authentication	DeepCVA	0.991	0.498	0.498	0.498	0.000
	Func_{RF}	**0.996**	0.498	0.500	0.499	0.000
	Func_{LGBM}	**0.996**	0.498	0.500	0.499	0.000
	PILOT	0.985	0.492	0.500	0.496	0.000
	SURF	0.993	**0.997**	**0.700**	**0.784**	**0.630**
Severity	DeepCVA	0.689	0.575	0.554	0.562	0.453
	Func_{RF}	0.700	0.482	0.465	0.458	0.403
	Func_{LGBM}	0.729	0.707	0.526	0.541	0.464
	PILOT	0.690	**0.810**	0.561	0.557	0.388
	SURF	**0.736**	0.655	**0.615**	**0.629**	**0.502**
Average	DeepCVA	0.762	0.615	0.599	0.603	0.473
	Func_{RF}	0.760	0.655	0.551	0.555	0.424
	Func_{LGBM}	0.790	0.742	0.605	0.621	0.484
	PILOT	0.754	0.699	0.570	0.573	0.431
	SURF	**0.799**	**0.785**	**0.664**	**0.690**	**0.591**

baselines only require a short time to complete the training process. In contrast, our proposed approach, along with other deep learning-based baselines, demands significantly more training time due to the complexity of the underlying models (i.e., SURF requires 4.09 hours, DeepCVA requires 1.13 hours, and PILOT requires 2.12 hours). Despite the higher training cost, deep learning approaches require only a single training phase when building the model and can show superior performance in the vulner-

ability assessment task. Thus, the additional training time is a worthwhile investment considering the improved assessment performance.

Answer to RQ1 SURF demonstrates promising performance for software vulnerability assessment. Specifically, with only SVA data labeled with 30%, SURF has an average of 11%∼24% and 22%∼39% performance improvement in terms of F1 score and MCC than state-of-the-art SVA baselines, even these baselines using SVA data labeled with 100%.

5.2 RQ2: effectiveness of our input modal setting

Approach To enhance the input for the SVA model training, we aim to capture the code's structural information through the code property graph, which contains different types of code dependencies (such as control dependencies and data flow dependencies). To show the effectiveness of our customized input bimodal setting, we also consider three control input modal settings. In the first setting, we only treat the code as plain text (denoted as **PT**). In the second setting, we only treat the code as the graph representation via CPG (denoted as **GR**). In the third setting, we also construct a bimodal input using the abstract syntax tree (AST) and the plain text of the code (denoted as **PGA**). In this RQ, we compare the performance of SURF by considering different input modal settings.

Results We show the performance of SURF for using different input modal settings in Table 2, with the best results indicated in bold. When only treating the code as plain text, the performance of SURF decreased slightly. Specifically, the performance of SURF decreased by an average of 10% and 10% in terms of F1 and MCC. When

Table 2 The impact of different input modal settings on the performance of SURF

Metrics	Evaluation Measures	Input Modal Settings			
		PT	GR	PGA	SURF
Confidentiality	**F1 score**	0.650	0.635	0.669	**0.686**
	MCC	0.501	0.483	**0.553**	**0.553**
Integrity	**F1 score**	0.694	0.686	0.708	**0.729**
	MCC	0.555	0.560	0.567	**0.601**
Availability	**F1 score**	0.675	0.639	0.685	**0.712**
	MCC	0.570	0.515	0.570	**0.601**
Access Vector	**F1 score**	0.599	0.597	0.603	**0.683**
	MCC	0.787	0.779	**0.795**	0.793
Access Complexity	**F1 score**	0.496	0.480	0.604	**0.609**
	MCC	0.383	0.334	0.423	**0.456**
Authentication	**F1 score**	0.664	0.697	0.664	**0.784**
	MCC	0.445	0.393	0.445	**0.630**
Severity	**F1 score**	0.545	0.563	0.590	**0.629**
	MCC	0.481	0.454	0.473	**0.502**
Average	**F1 score**	0.618	0.614	0.646	**0.690**
	MCC	0.532	0.503	0.544	**0.591**

only treating the code as the graph representation via CPG, the performance of SURF will significantly decrease. Specifically, the performance of SURF decreased by an average of 11% and 15% in terms of F1 and MCC. Finally, when considering the AST and plain text of code as a bimodal input, SURF's performance improved when compared to only considering single input mode, but still did not outperform our input modal setting. Specifically, the performance of SURF decreased by an average of 6% and 8% in terms of F1 and MCC. These results indicate that there exists certain supplementary information in different modalities used in our input setting. Moreover, considering code structural information via CPG can help to achieve the best performance for SURF.

Answer to RQ2 By fusing code lexical information and structural information via the code property graph, the performance of SURF can be improved by at least 7% and 9% in terms of the F1 score and MCC.

5.3 RQ3: effectiveness of GraphCodeBERT for code representation learning

Approach In this RQ, we compare the performance differences between our used GraphCodeBERT and other representation learning methods. Specifically, we consider three representation learning methods, including the traditional graph embedding method Node2 Vec (Grover and Leskovec 2016), the pre-training method CodeBERT (Feng et al. 2020), and the deep neural network-based method GCN (Graph Convolutional Network) (Kipf and Welling 2016). Specifically, Node2 Vec (Grover and Leskovec 2016) is a graph embedding method employed to create feature vectors for nodes within a graph, enabling the capture of graph structures and relationships. CodeBERT (Feng et al. 2020) is a specialized method for extracting feature vectors from code. It leverages pre-trained models to generate contextual embeddings for code snippets. GCN (Kipf and Welling 2016) is a deep learning method explicitly designed for graph data. It extracts feature vectors for nodes by aggregating information from their neighboring nodes. We compared these methods with GraphCodeBERT and evaluated their performance using the F1 score and MCC. Given the different characteristics of these methods, we use their default settings to ensure a fair comparison.

Results The performance differences between different representation learning methods are shown in Fig. 4. We evaluated the performance of seven CVSS basic metrics respectively and obtained the score for different methods under different basic metrics. For example, when evaluating the **Integrity** metric, the F1 score and MCC scores of Node2 Vec, CodeBERT, GCN, and SURF are 0.577 and 0.385, 0.690 and 0.549, 0.682 and 0.538, and 0.729 and 0.601, respectively. Based on the comparison results, we can find that for each CVSS metric, the performance of SURF using GraphCodeBERT consistently outperforms other representation learning methods, such as GCN and Node2 Vec.

Answer to RQ3 Using GraphCodeBERT for code representation learning can achieve better performance than using other representation learning methods for SURF.

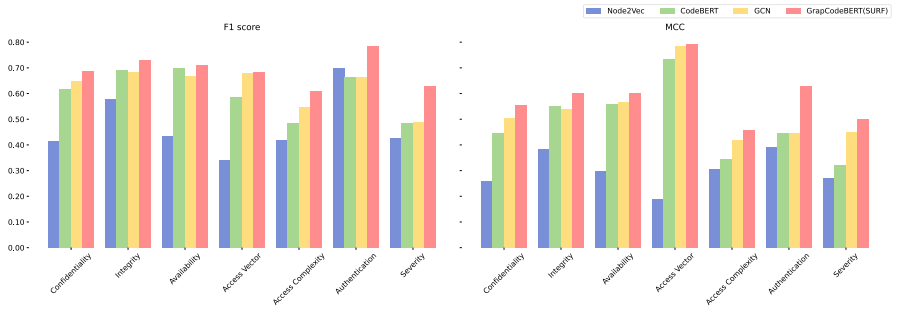

Fig. 4 The performance differences between GraphCodeBERT and other representation learning methods for SURF in terms of F1 score and MCC. Note: The *y*-axis scale of the left subfigure and the right subfigure is consistent

Table 3 The impact of training data size

Measure	Approach	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
F1 score	DeepCVA	0.383	0.429	0.520	0.545	0.578	0.609	0.613	0.617	0.620	0.621
	Func_{RF}	0.417	0.459	0.514	0.526	0.537	0.540	0.544	0.549	0.550	0.553
	Func_{LGBM}	0.389	0.447	0.599	0.618	0.649	0.658	0.661	0.668	0.671	0.684
MCC	DeepCVA	0.323	0.463	0.572	0.613	0.664	0.688	0.701	0.707	0.718	0.724
	Func_{RF}	0.479	0.501	0.532	0.605	0.628	0.644	0.649	0.652	0.655	0.652
	Func_{LGBM}	0.353	0.411	0.515	0.601	0.676	0.729	0.754	0.778	0.788	0.794

6 Discussion

6.1 The impact of training data size

To validate limitation 1, we conducted experiments by replicating supervised SVA approaches (DeepCVA, Func_{RF}, and Func_{LGBM}) to investigate the impact of the amount of training data on the performance of SVA. Specifically, we evaluated the performance of these supervised approaches under different proportions of labeled data (e.g., 10%, 20%, 30%, 40%, 50%, $\cdots$, 100%). Table 3 presents the experimental results, using the Access Vector as an example[9], which show that as the amount of labeled data increases, the performance of the supervised approaches improves significantly, but plateaus after reaching 50%. This indicates that while more labeled data generally enhances performance, the marginal benefits diminish beyond a certain threshold. The experimental results reveal that the supervised approaches require at least 50% of the labeled data to perform effectively, achieving performance close to their maximum potential. However, even with only 30% labeled data, the supervised approaches can still achieve reasonable performance, although it is slightly inferior compared to using more data. This finding supports the motivation behind our proposed semi-supervised approach SURF, which aims to achieve competitive performance with significantly less labeled data (e.g., 30%).

[9] Including every metric could result in an overly lengthy in Section 6. Therefore, we have shared the results for all available metrics in our GitHub repository

6.2 The impact of labeled data size

In this subsection, we investigate the impact of different ratios of labeled SVA data on the performance of SURF. Specifically, we aim to train the SVA models using 10%, 30%, 50%, 60%, 70%, and 80% of labeled SVA data, and compare the performance of these trained models. The detailed results are shown in Fig. 5. In this figure, we find that when the labeled SVA data ratio increases from 10% to 30%, the performance improvement trend is relatively high. For example, Authentication shows the most notable improvement, with its F1 score increasing from 0.664 to 0.784 (i.e., an increase of 18%) and MCC from 0.445 to 0.630 (i.e., an increase of 42%). However, as the labeled SVA data ratio increases from 30% to 50%, the performance improvement trend begins to decrease. Then, when using Authentication as an example, its F1 score increases from 0.784 to 0.832 (an increase of 6%) and MCC from 0.630 to 0.668 (an increase of 6%). This represents a significant slowdown in performance gains compared to the range 10%∼30%. When the labeled data ratio increases from 50% to 80%, the performance improvement becomes even more gradual, with F1 scores and MCC values rising by only 1%∼5% in most cases. Taking Authentication as an example, its F1 score increases from 0.832 to 0.834 (i.e., an increase of almost 0%) and MCC from 0.668 to 0.698 (i.e., an increase of 5%), indicating that it reaches the plateaus at 80%. This suggests that further increases in labeled data yield only marginal performance gains, making additional labeling efforts less cost-effective. Therefore, the results show that the performance improvement rate is highest when increasing the labeled data from 0% to 30%. Beyond 30%, the performance gains gradually diminish, and the curve begins to plateau between 50% and 60%. From a cost-effectiveness perspective, using 30% labeled data provides a favorable trade-off between model performance and labeling effort.

Therefore, increasing the labeled SVA data ratio can help to improve the performance of the trained SVA model. However, the performance improvement comes at the cost of increasing the costs of SVA data labeling. In our study, using 30% labeled SVA data is a cost-effective solution, which can achieve promising performance with a reasonable labeling cost as discussed in Section 5.1.

6.3 The impact of backbone network setting

After code representation learning, our lexical information and graph information have been represented as vector structures, and the obtained data can be regarded as tabular data. We compare three backbone networks: Vanilla Transformer (Vaswani et al. 2017), FT_Transformer (Gorishniy et al. 2021), and Wide ResNet (WRN) (Zagoruyko and Komodakis 2016). Although Vanilla Transformer excels at global dependency modeling, its standard self-attention mechanism does not perform well in processing table data. Additionally, its high computing cost may make it unsuitable for resource-constrained scenarios. Therefore, we consider using FT_Transformer (Gorishniy et al. 2021), which is customized for tabular data, as our backbone network to replace WRN used in the original semi-supervised learning method proposed by Sohn et al. (2020). In this subsection, we want to analyze whether our backbone network setting can achieve better performance for SVA model construction. To com-

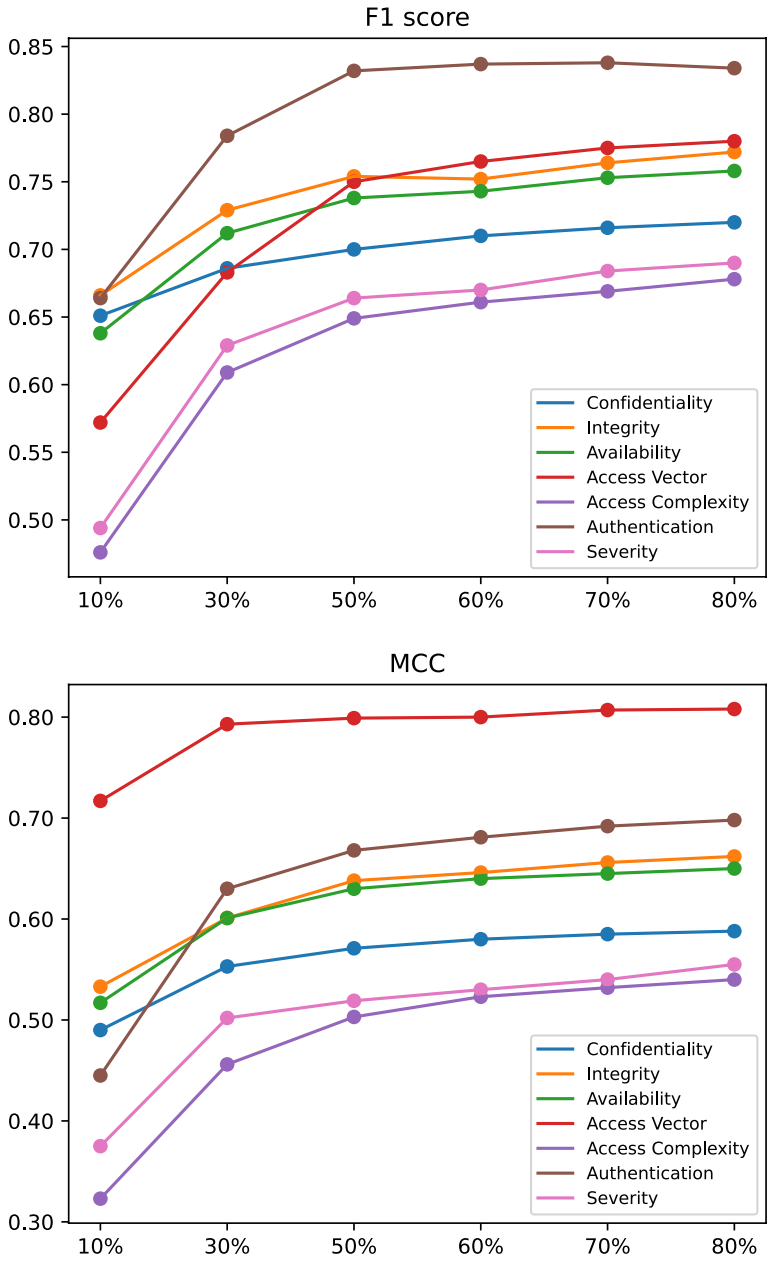

Fig. 5 Performance trends of SURF using different labeled SVA data ratios (The *x*-axis represents the proportion of data division, and the *y*-axis represents the metric score)

Table 4 The impacts of different backbone networks on the performance of SURF

Metrics	Backbone network	F1 score	MCC
Confidentiality	WRN	0.620	0.450
	Vanilla Transformer	0.605	0.433
	FT_Transformer	**0.686**	**0.553**
Integrity	WRN	0.628	0.466
	Vanilla Transformer	0.612	0.449
	FT_Transformer	**0.729**	**0.601**
Availability	WRN	0.594	0.479
	Vanilla Transformer	0.578	0.461
	FT_Transformer	**0.712**	**0.601**
Access Vector	WRN	0.570	0.698
	Vanilla Transformer	0.563	0.682
	FT_Transformer	**0.683**	**0.793**
Access Complexity	WRN	0.555	0.377
	Vanilla Transformer	0.541	0.361
	FT_Transformer	**0.609**	**0.456**
Authentication	WRN	0.499	0.000
	Vanilla Transformer	0.499	0.000
	FT_Transformer	**0.784**	**0.630**
Severity	WRN	0.429	0.308
	Vanilla Transformer	0.417	0.291
	FT_Transformer	**0.629**	**0.502**
Average	WRN	0.556	0.397
	Vanilla Transformer	0.524	0.367
	FT_Transformer	**0.690**	**0.591**

pare the performance difference using different backbone networks fairly, we keep other configurations of SURF unchanged. Table 4 shows the comparison results, with the best results in bold. In this table, we find that utilizing FT_Transformer as the backbone network achieves better performance compared to using either WRN or Vanilla Transformer. Specifically, for SURF, FT_Transformer outperforms WRN by an average of 24% in F1 score and 49% in MCC, and outperforms Vanilla Transformer by 32% in F1 score and 61% in MCC. These performance improvements verify the rationality of using FT_Transformer in SURF.

6.4 Evaluation with LLM-based approaches

To compare SURF with LLM-based approaches, we select the following LLMs, including CodeBert (Feng et al. 2020), CodeT5 (Wang et al. 2021), CodeLLaMa[10], and DeepSeek-Coder[11]. These models represent the current state-of-the-art in code understanding and vulnerability-related tasks.

The comparison results can be found in Table 5. In this table, we find that SURF consistently outperforms these LLM-based approaches across different key metrics, demonstrating its effectiveness for vulnerability assessment. This comparison further

[10] https://ai.meta.com/blog/code-llama-large-language-model-coding

[11] https://deepseekcoder.github.io

Table 5 Comparison of SURF with LLM-based Approaches for Vulnerability Assessment

Metrics	Approaches	F1 score	MCC
Confidentiality	CodeBERT	0.474	0.306
	CodeT5	0.505	0.420
	CodeLLaMa	0.581	0.465
	DeepSeek-Coder	0.665	0.517
	SURF	**0.686**	**0.553**
Integrity	CodeBERT	0.511	0.411
	CodeT5	0.680	0.543
	CodeLLaMa	0.588	0.427
	DeepSeek-Coder	0.625	0.506
	SURF	**0.729**	**0.601**
Availability	CodeBERT	0.404	0.248
	CodeT5	0.427	0.221
	CodeLLaMa	0.610	0.491
	DeepSeek-Coder	0.703	0.590
	SURF	**0.712**	**0.601**
Access Vector	CodeBERT	0.343	0.189
	CodeT5	0.613	0.368
	CodeLLaMa	0.509	0.300
	DeepSeek-Coder	0.528	0.321
	SURF	**0.683**	**0.793**
Access Complexity	CodeBERT	0.430	0.302
	CodeT5	0.472	0.410
	CodeLLaMa	0.433	0.329
	DeepSeek-Coder	0.435	0.320
	SURF	**0.609**	**0.456**
Authentication	CodeBERT	0.497	0.000
	CodeT5	0.498	0.000
	CodeLLaMa	0.480	0.000
	DeepSeek-Coder	0.498	0.000
	SURF	**0.784**	**0.630**
Severity	CodeBERT	0.373	0.290
	CodeT5	0.487	0.468
	CodeLLaMa	0.329	0.410
	DeepSeek-Coder	0.570	0.460
	SURF	**0.629**	**0.502**

supports the strength of our approach in scenarios where fine-grained code analysis about vulnerabilities is essential.

6.5 Evaluation on latest shared SVA datasets

In this subsection, we evaluate the effectiveness of SURF on recently shared SVA datasets: ReposVul (Wang et al. 2024), MegaVul (Ni et al. 2024), and REEF (Wang et al. 2023). Table 6 presents the detailed statistics of these SVA datasets. First, we provide the number of vulnerabilities included in these datasets. Then we categorize them into three severity levels: low (0∼3.9), medium (4∼6.9), and high (7∼10), and show the proportion of each type. Based on these datasets, we conducted com-

Table 6 Statistics of the latest shared datasets

Dataset	Vul.	Low	Medium	High
ReposVul (Wang et al. 2024)	233,092	5%	41%	54%
MegaVul (Ni et al. 2024)	17,380	10%	64%	26%
REEF (Wang et al. 2023)	9,201	1%	35%	64%

Table 7 Comparison results between SURF and the baselines on the three SVA datasets

Dataset						
Baseline	ReposVul (Wang et al. 2024)		MegaVul (Ni et al. 2024)		REEF (Wang et al. 2023)	
	F1 score	MCC	F1 score	MCC	F1 score	MCC
DeepCVA	0.695	0.563	0.616	0.478	0.627	0.511
Func_{RF}	0.682	0.617	0.547	0.465	0.594	0.482
Func_{LGBM}	0.704	0.577	0.652	0.503	0.658	0.523
PILOT	0.713	0.588	0.643	0.482	0.680	0.541
SURF	**0.733**	**0.609**	**0.667**	**0.508**	**0.689**	**0.554**

Notice we only show the F1 score and MCC for the severity metric

parative experiments between SURF and baseline methods by following the same experimental setup, with the results shown in Table 7. The comparison results for the severity metric demonstrate that SURF achieves the highest F1 scores (i.e., 0.733, 0.689, 0.667) and MCC values (i.e., 0.609, 0.554, 0.508) on ReposVul, MegaVul, and REEF, respectively. This represents at least a 2% improvement in F1 score and a 3% improvement in MCC over the best-performing baseline. For the remaining metrics (such as Confidentiality and integrity), we achieve the same findings. The above results indicate that our proposed approach SURF remains highly competitive on the latest shared SVA datasets.

7 Threats to validity

Threats to internal validity lies in the potential defects that arise when implementing SURF and baselines. To mitigate this threat, we implemented baselines by using the source code shared by the corresponding studies, and we used the same hyperparameter setting as described in the original studies for a fair comparison. To avoid implementation errors, we perform software testing and use mature frameworks to implement SURF. The second threat is the maximum input length of GraphCodeBERT. Initially, when directly feeding the code and structural information into GraphCodeBERT, approximately 56.84% of the code snippets and their corresponding structural information had token lengths within 512 tokens. To further optimize the input length, we split the longer inputs. Specifically, we divided the code snippets and structural information according to control flow nodes and aggregated their representations. After this operation, about 93.94% of the code snippets and structural information were successfully controlled within 512 tokens. For the remaining inputs that still exceeded 512 tokens, we applied a truncation strategy. This approach ensures that the model can efficiently process the majority of inputs while minimizing information loss. The final threat concerns the proportion of labeled data required

for training. To mitigate this, we evaluated model performance under different labeling ratios in our empirical study. The results show that performance improves most significantly when increasing the labeled data from 0% to 30%, with diminishing returns beyond that. Using 30% labeled data offers a favorable trade-off between assessment performance and labeling cost. Regarding the practical feasibility of obtaining 30% labeled data, we acknowledge that data availability may vary across different scenarios. Therefore, we propose 30% as a recommended value rather than a strict requirement.

Threats to external validity concerns the dataset used in our study. To mitigate this threat, we employed the Big-Vul dataset (Fan et al. 2020), which is a large-scale dataset collected from real-world open-source projects. Moreover, this dataset also provided rich information about CVSS scores and detailed information for the base metrics of CVSS. Finally, we select related information for the SVA task and propose a set of filtering rules to improve the dataset quality.

Threats to construct validity is related to the performance measures for evaluating the performance of our approach. To alleviate this issue, we considered widely used performance measures (such as F1 score and MCC) in the previous SVA studies (Le et al. 2021; Le and Babar 2022; Le et al. 2022).

8 Related work

Effective SVA approaches can help developers allocate repair resources more effectively for vulnerabilities with high priority. The Common Vulnerability Scoring System (CVSS) is a framework used to describe the exploitability, impact, and severity of vulnerabilities and is considered one of the most reliable metrics for SVA (Johnson et al. 2016).

To automate the assessment of software vulnerabilities, particularly in assessing CVSS metrics, various approaches have been proposed. For example, Yamamoto et al. (2015) first utilized vulnerability descriptions from the NVD database to assess CVSS metrics. Later, Wen et al. (2015) employed software vulnerability databases other than NVD (such as SecurityFocus[12], OSVDB[13], and IBM X-Force[14]) to assess the metrics. Then, researchers (Le et al. 2019; Spanos and Angelis 2018; Elbaz et al. 2020; Gong et al. 2019) utilized vulnerability description information to predict CVSS metrics. In addition, Ni et al. (2022) proposed a BERT-CNN model to predict the severity of software vulnerabilities by combining contextual embeddings from BERT with convolutional neural networks (CNNs), which effectively captures localized patterns in vulnerability descriptions to improve the accuracy of severity predictions. Recently, Sun et al. (2023) used BERT-MRC to extract vulnerability ele-

[12] http://www.securityfocus.com

[13] https://www.osvdb.org/.

[14] http://www.iss.net/xforce

ments from vulnerability descriptions and performed vulnerability assessment based on these extracted elements.

However, according to previous research (Le et al. 2021; Le and Babar 2022; Li and Paxson 2017), vulnerability reports are typically only published after the corresponding vulnerabilities have been fixed, resulting in a lack of timeliness in assessing CVSS metrics based on vulnerability descriptions. Specifically, 50% of vulnerabilities have a time interval exceeding 438 days from their discovery to their remediation (Li and Paxson 2017). Therefore, researchers came to focus on source code-based vulnerability assessment. For example, Le et al. (2021) initially proposed a multitask learning model based on CVSS metrics for the commit-level SVA task. Then they (Le and Babar 2022) studied the value of vulnerability statements in the assessment model and proposed a function-level vulnerability assessment approach.

However, these studies only considered the statement information of the code in assessing vulnerabilities, neglecting the structural information of the code. Recently, Nguyen et al. (2024) conducted an empirical evaluation of graph-based neural networks (GNNs) for vulnerability assessment in C/C++ code. Their study highlights the potential of GNNs in capturing structural dependencies and further demonstrates the importance of structural information in code for software vulnerability assessment tasks. However, their approach relies exclusively on structural features, overlooking the complementary value of lexical information. Hence, we use the lexical information (i.e., treat the code as plain text) and structural information (i.e., treat the code as code property graph) as the bimodal input to train the model for the vulnerability assessment task. Moreover, the high cost of data labeling is a challenging problem for the SVA task. To alleviate this problem, we study a more practical scenario (i.e., with a limited amount of the labeled SVA data and a large amount of unlabeled SVA data) and then propose a novel approach SURF with semi-supervised learning.

9 Conclusion and future research

To the best of our knowledge, we are the first to be concerned about the difficulty of the data labeling problem for the SVA task. To alleviate this problem, we consider a more practical scenario where we can label a small amount of SVA data. Then we propose a semi-supervised SVA approach SURF. To further improve the performance of SURF, we fuse code lexical and structural information as the bimodal input. Our empirical results show that when only utilizing 30% of labeled SVA data, SURF can outperform state-of-the-art SVA baselines using 100% labeled SVA data. Specifically, SURF can achieve a performance improvement of 11%∼24% in F1 score and an improvement of 22%∼39% in MCC measures. These findings emphasize the potential of applying semi-supervised learning in future SVA studies.

In the future, we want to further improve the performance of SURF by considering more advanced code representation approaches. We also want to evaluate the generalization of SURF by considering vulnerable codes in other programming languages.

Acknowledgements Xiang Chen is the corresponding author. This research was partially supported by the National Natural Science Foundation of China (Grant No. 61202006), the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University under (Grant No. KFKT2024B21) and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant No. SJCX24_2018).

Author Contributions Wenlong Pei: Data curation, Software, Validation, Conceptualization, Methodology, Writing-review & editing. Yiling Huang: Data curation, Software, Validation, Conceptualization, Methodology, Writing-review & editing. Xiang Chen: Conceptualization, Methodology, Writing-review & editing, Supervision. Guilong Lu: Conceptualization, Data curation, Software. Yong Liu: Conceptualization, Data curation, Writing-review & editing. Chao Ni: Conceptualization, Data curation, Writing-review & editing.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

Berthelot, D., Carlini, N., Cubuk, E.D., Kurakin, A., Sohn, K., Zhang, H., Raffel, C.: Remixmatch: Semi-supervised learning with distribution matching and augmentation anchoring. In: International Conference on Learning Representations (2020)

Blum, A., Lafferty, J., Rwebangira, M.R., Reddy, R.: Semi-supervised learning using randomized mincuts. In: Proceedings of the Twenty-first International Conference on Machine Learning, p. 13 (2004)

Chapelle, O., Chi, M., Zien, A.: A continuation method for semi-supervised svms. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 185–192 (2006)

Costa, J.C., Roxo, T., Sequeiros, J.B., Proenca, H., Inacio, P.R.: Predicting cvss metric via description interpretation. IEEE Access **10**, 59125–59134 (2022)

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. (2018). arXiv:1810.04805

Elbaz, C., Rilling, L., Morin, C.: Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–10 (2020)

Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 508–512 (2020)

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. (2020). arXiv:2002.08155

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

Gong, Z., Gao, C., Wang, Y., Gu, W., Peng, Y., Xu, Z.: Source code summarization with structural relative position guided transformer. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 13–24. IEEE (2022)

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

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. Adv. Neural. Inf. Process. Syst. **27** (2014)

Gorishniy, Y., Rubachev, I., Khrulkov, V., Babenko, A.: Revisiting deep learning models for tabular data. Adv. Neural. Inf. Process. Syst. **34**, 18932–18943 (2021)

Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864 (2016)
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al.: Graphcodebert: Pre-training code representations with data flow. (2020). arXiv:2009.08366
Han, Z., Li, X., Xing, Z., Liu, H., Feng, Z.: Learning to predict severity of software vulnerability using only vulnerability description. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 125–136. IEEE (2017)
Hin, D., Kan, A., Chen, H., Babar, M.A.: Linevd: Statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 596–607 (2022)
Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., Palomba, F.: The secret life of software vulnerabilities: A large-scale empirical study. IEEE Trans. Softw. Eng. **49**(1), 44–63 (2022)
Iscen, A., Tolias, G., Avrithis, Y., Chum, O.: Label propagation for deep semi-supervised learning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 5070–5079 (2019)
Johnson, P., Lagerström, R., Ekstedt, M., Franke, U.: Can the common vulnerability scoring system be trusted? a bayesian analysis. IEEE Trans. Dependable Secure Comput. **15**(6), 1002–1015 (2016)
Khan, S., Parkinson, S.: Review into state of the art of vulnerability assessment using artificial intelligence. Guide to Vulnerability Analysis for Computer Networks and Systems: An Artificial Intelligence Approach, 3–32 (2018)
Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. (2014). arXivpreprint arXiv:1412.6980
Kingma, D.P., Welling, M.: Auto-encoding variational bayes. (2013). arXivpreprint arXiv:1312.6114
Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. (2016). arXiv:1609.02907
Le, T.H.M., Babar, M.A.: On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 621–633 (2022)
Le, T.H.M., Hin, D., Croft, R., Babar, M.A.: Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 717–729. IEEE (2021)
Le, T.H.M., Sabir, B., Babar, M.A.: Automated software vulnerability assessment with concept drift. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 371–382. IEEE (2019)
Le, T.H., Chen, H., Babar, M.A.: A survey on data-driven software vulnerability assessment and prioritization. ACM Comput. Surv. **55**(5), 1–39 (2022)
Le, H., Wang, Y., Gotmare, A.D., Savarese, S., Hoi, S.C.H.: Coderl: Mastering code generation through pretrained models and deep reinforcement learning. Adv. Neural. Inf. Process. Syst. **35**, 21314–21328 (2022)
Li, F., Paxson, V.: A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2201–2215 (2017)
Li, Y., Wang, S., Nguyen, T.N.: 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 (2021)
McIntosh, S., Kamei, Y.: 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, pp. 560–560 (2018)
Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S.: graph2vec: Learning distributed representations of graphs. (2017). arXiv:1707.05005
Nguyen, A.T., Le, T.H.M., Babar, M.A.: Automated code-centric software vulnerability assessment: How far are we? an empirical study in c/c++. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 72–83 (2024)
Ni, C., Shen, L., Yang, X., Zhu, Y., Wang, S.: Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), pp. 738–742. IEEE (2024)
Ni, X., Zheng, J., Guo, Y., Jin, X., Li, L.: Predicting severity of software vulnerability based on bert-cnn. In: 2022 International Conference on Computer Engineering and Artificial Intelligence (ICCEAI), pp. 711–715. IEEE (2022)

Novielli, N., Girardi, D., Lanubile, F.: A benchmark study on sentiment analysis for software engineering research. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 364–375 (2018)

Paolone, G., Marinelli, M., Paesani, R., Di Felice, P.: Automatic code generation of mvc web applications. Computers **9**(3), 56 (2020)

Shahid, M.R., Debar, H.: Cvss-bert: Explainable natural language processing to determine the severity of a computer security vulnerability from its description. In: 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1600–1607. IEEE (2021)

Smyth, V.: Software vulnerability management: how intelligence helps reduce the risk. Netw. Secur. **2017**(3), 10–12 (2017)

Sohn, K., Berthelot, D., Carlini, N., Zhang, Z., Zhang, H., Raffel, C.A., Cubuk, E.D., Kurakin, A., Li, C.-L.: Fixmatch: Simplifying semi-supervised learning with consistency and confidence. Adv. Neural. Inf. Process. Syst. **33**, 596–608 (2020)

Spanos, G., Angelis, L.: A multi-target approach to estimate software vulnerability characteristics and severity scores. J. Syst. Softw. **146**, 152–166 (2018)

Sun, X., Ye, Z., Bo, L., Wu, X., Wei, Y., Zhang, T., Li, B.: Automatic software vulnerability assessment by extracting vulnerability elements. J. Syst. Softw. **111790** (2023)

Triguero, I., García, S., Herrera, F.: Self-labeled techniques for semi-supervised learning: taxonomy, software and empirical study. Knowl. Inf. Syst. **42**, 245–284 (2015)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł, Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30** (2017)

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. Stat. **1050**, 4 (2018)

Wang, X., Hu, R., Gao, C., Wen, X.-C., Chen, Y., Liao, Q.: Reposvul: A repository-level high-quality vulnerability dataset. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, pp. 472–483 (2024)

Wang, C., Li, Z., Pena, Y., Gao, S., Chen, S., Wang, S., Gao, C., Lyu, M.R.: Reef: A framework for collecting real-world vulnerabilities and fixes. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1952–1962. IEEE (2023)

Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. (2021). arXiv:2109.00859

Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q.: Vulnerability detection with graph simplification and enhanced graph representation learning. (2023). arXiv:2302.04675

Wen, X.-C., Wang, X., Gao, C., Wang, S., Liu, Y., Gu, Z.: When less is enough: Positive and unlabeled learning model for vulnerability detection. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 345–357. IEEE (2023)

Wen, T., Zhang, Y., Dong, Y., Yang, G.: A novel automatic severity vulnerability assessment framework. J. Commun. **10**(5), 320–329 (2015)

Xia, C.S., Zhang, L.: Less training, more repairing please: revisiting automated program repair via zero-shot learning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 959–971 (2022)

Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. IEEE (2014)

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

Zagoruyko, S., Komodakis, N.: Wide residual networks. (2016). arXiv preprint arXiv:1605.07146

Zhang, Y., Xiao, Y., Kabir, M.M.A., Yao, D., Meng, N.: Example-based vulnerability detection and repair in java code. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 190–201 (2022)

Zhang, B., Wang, Y., Hou, W., Wu, H., Wang, J., Okumura, M., Shinozaki, T.: Flexmatch: Boosting semi-supervised learning with curriculum pseudo labeling. Adv. Neural. Inf. Process. Syst. **34**, 18408–18419 (2021)

Zhou, J., Pacheco, M., Wan, Z., Xia, X., Lo, D., Wang, Y., Hassan, A.E.: Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 705–716. IEEE (2021)

Zhou, Y., Shen, J., Zhang, X., Yang, W., Han, T., Chen, T.: Automatic source code summarization with graph attention networks. J. Syst. Softw. **188**, 111257 (2022)

Zhu, X., Goldberg, A.: Introduction to Semi-supervised Learning. Morgan & Claypool Publishers,??? (2009)
Zhu, X.J.: Semi-supervised learning literature survey (2005)

Authors and Affiliations

Wenlong Pei[1] · Yilin Huang[1] · Xiang Chen[1,2] · Guilong Lu[1] · Yong Liu[3] · Chao Ni[4]

✉ Xiang Chen
xchencs@ntu.edu.cn

Wenlong Pei
wl.pei@outlook.com

Yilin Huang
ylhuangs@outlook.com

Guilong Lu
guil.lu@outlook.com

Yong Liu
lyong@mail.buct.edu.cn

Chao Ni
chaoni@zju.edu.cn

[1] School of Artificial Intelligence and Computer Science, Nantong University, Nantong 226019, Jiangsu, China

[2] State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China

[3] College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China

[4] School of Software Technology, Zhejiang University, Hangzhou 310058, Zhejiang, China