Less is More: Unlocking Semi-Supervised Deep Learning for Vulnerability Detection

XIAO YU, Huawei, China
GUANCHENG LIN, School of Cyber Science and Engineering, Wuhan University, China
XING HU*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JACKY WAI KEUNG, Department of Computer Science, City University of Hong Kong, China
XIN XIA, Huawei, China

Deep learning has demonstrated its effectiveness in software vulnerability detection, but acquiring a large number of labeled code snippets for training deep learning models is challenging due to labor-intensive annotation. With limited labeled data, complex deep learning models often suffer from overfitting and poor performance. To address this limitation, semi-supervised deep learning offers a promising approach by annotating unlabeled code snippets with pseudo-labels and utilizing limited labeled data together as training sets to train vulnerability detection models. However, applying semi-supervised deep learning for accurate vulnerability detection comes with several challenges. One challenge lies in how to select correctly pseudo-labeled code snippets as training data, while another involves mitigating the impact of potentially incorrectly pseudo-labeled training code snippets during model training. To address these challenges, we propose the Semi-Supervised Vulnerability Detection (SSVD) approach. SSVD leverages the information gain of model parameters as the certainty of the correctness of pseudo-labels and prioritizes high-certainty pseudo-labeled code snippets as training data. Additionally, it incorporates the proposed noise-robust triplet loss to maximize the separation between vulnerable and non-vulnerable code snippets to better propagate labels from labeled code snippets to nearby unlabeled snippets, and utilizes the proposed noise-robust cross-entropy loss for gradient clipping to mitigate the error accumulation caused by incorrect pseudo-labels. We evaluate SSVD with nine semi-supervised approaches on four widely-used public vulnerability datasets. The results demonstrate that SSVD outperforms the baselines with an average of 29.82% improvement in terms of F1-score and 56.72% in terms of MCC. In addition, SSVD trained on a certain proportion of labeled data can outperform or closely match the performance of fully supervised LineVul and ReVeal vulnerability detection models trained on 100% labeled data in most scenarios. This indicates that SSVD can effectively learn from limited labeled data to enhance vulnerability detection performance, thereby reducing the effort required for labeling a large number of code snippets.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software verification and validation**; • **Computing methodologies → Semi-supervised learning settings**.

Additional Key Words and Phrases: Vulnerability Detection, Semi-Supervised Learning, Information Gain

*Corresponding Author: Xing Hu.

Authors' Contact Information: Xiao Yu, Huawei, Hangzhou, China, yuxiao25@huawei.com; Guancheng Lin, School of Cyber Science and Engineering, Wuhan University, Wuhan, China, guanchenglin@whu.edu.cn; Xing Hu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Ningbo, China, xinghu@zju.edu.cn; Jacky Wai Keung, Department of Computer Science, City University of Hong Kong, Hong Kong, China, jacky.keung@cityu.edu.hk; Xin Xia, Huawei, Hangzhou, China, xin.xia@acm.org.

ACM 1557-7392/2024/10-ART
https://doi.org/10.1145/3699602

1 INTRODUCTION

Software vulnerability detection is a fundamental and crucial task in the fields of software engineering and information security [7, 14]. Recently, researchers [27, 58, 100, 104, 123, 125] have employed deep learning techniques to automatically learn and extract vulnerability features from code snippets for vulnerability detection. Although the deep learning-based vulnerability detection methods have achieved good performance, they require a large amount of correctly labeled code snippets as training data [32, 51, 65, 81, 122]. When only a limited number of labeled data is available, complex deep neural networks often suffer from over-fitting and perform poorly [108]. For example, as shown in Section 4.1, training with only 10% of the dataset results in the vulnerability detection models (LineVul [24] and ReVeal [7]) performing at less than half the performance of those trained with the full dataset. To address the challenge of limited labeled data, researchers typically adopt two primary approaches to acquire a substantial number of labeled code snippets. One approach is to automatically collect sufficient labeled code snippets through static analysis tools [130], synthetic creation [4], or from the Common Vulnerabilities and Exposures (CVE) system [20]. However, Croft et al. [14] discovered that a considerable percentage of vulnerability labels of the collected code snippets were inaccurate, leading to vulnerability detection models incorrectly inferring patterns between vulnerable and non-vulnerable code snippets, consequently resulting in poor model performance [14]. Another approach entails enlisting domain experts to label a large number of unlabeled code snippets. For example, Zhou et al. [132] developed the Devign dataset by collecting vulnerability-fixing commits from GitHub. To ensure the accuracy of vulnerability labels, a team of four professional security researchers spent about 600 man-hours labeling and cross-verifying around 27,000 vulnerability-fixing commits. In essence, obtaining a sufficient amount of correctly labeled data requires significant time and effort from domain experts for annotation [123].

1.1 Motivations

This situation highlights the difficulty of obtaining extensive and accurately labeled data in real-world scenarios, which has been the primary obstacle in the development of supervised deep learning-based vulnerability detection methods [51]. Fortunately, in practical scenarios, there is often an abundance of available unlabeled code snippets, which can be easily accessed through open-source websites, e.g., the CVE system [20]. By leveraging this abundance of unlabeled data alongside a small amount of labeled data, semi-supervised deep learning can be effectively applied. It provides a more practical solution, as it achieves comparable performance to supervised deep learning methods while reducing reliance on extensive and accurately labeled data. In general, the semi-supervised deep learning approach initiates by training a teacher model using a limited amount of labeled data through supervised deep learning techniques [8, 42, 66, 93, 111]. This trained teacher model is then utilized to generate pseudo-labels for all unlabeled code snippets. Subsequently, a sampling strategy is employed to select the correctly pseudo-labeled code snippets from the entire pool of pseudo-labeled code snippets, which are then used for training the student model. The trained student model is then utilized as the new teacher model in the subsequent iteration, generating pseudo-labels for the next round of training. There are two major challenges when applying semi-supervised deep learning to vulnerability detection.

(1) How to select correctly pseudo-labeled code snippets by the teacher model as the training data? The selection of correctly pseudo-labeled code snippets by the teacher model is crucial for training the student model effectively. Since the vulnerability detection model is a probabilistic classifier, some previous studies in software engineering [41, 67, 105, 113] regarded outputted classification probability [1] as the corresponding prediction's confidence and chose pseudo-labeled samples with higher classification probability as training

[1]The classification probability represents the likelihood of a code snippet belonging to a specific class (vulnerable or non-vulnerable) as predicted by the vulnerability detection model. In simpler terms, if the probability of a code snippet being vulnerable is p, and p is greater than or equal to 0.5 (indicating the code snippet is predicted to be vulnerable), its classification probability cp equals p. Conversely, if p is less

data. However, recent studies [26, 33, 68, 72, 78] show that interpreting classification probabilities as confidence scores is inappropriate, because even when the classification probability is high, the detection model can still be uncertain in its predictions. The classification probability only reflects how well the detection model's prediction matches the training data, but it does not reflect the model's certainty in its prediction [30]. Additionally, the deep learning-based detection model is highly sensitive to input variations, where even small perturbations can lead to significant changes in the classification probability output [68]. Consequently, selecting pseudo-labeled samples with higher classification probabilities does not guarantee the model's certainty about those predicted pseudo-labels. This increases the risk of selecting incorrectly pseudo-labeled code snippets as training data. To mitigate this issue, it is crucial to consider alternative measures of model certainty when selecting pseudo-labeled samples for training.

(2) How to address the smoothness assumption and mitigate the impact of incorrectly pseudo-labeled training code snippets to enhance the predictive capability of the student model? As the student model acts as the new teacher model in subsequent iterations to generate pseudo-labels for unlabeled code snippets, it becomes crucial to ensure that the student model demonstrates strong predictive capability by effectively considering the smoothness assumption and the presence of incorrect pseudo-labeled code snippets as the training data of student models. (a) Semi-supervised learning propagates labels from labeled code snippets to nearby unlabeled snippets, so the smoothness assumption should be considered: a vulnerable code snippet should be closer to another vulnerable one rather than a non-vulnerable one in the feature space, and vice versa [41, 93]. (b) Despite employing carefully designed sampling methods to select correctly pseudo-labeled code snippets during self-training, it is inevitable to include incorrectly pseudo-labeled code snippets (i.e., noise) in the student model's training data. Updating the student model based on the noise data can lead to error accumulation [85], potentially decreasing detection performance over time. However, previous semi-supervised learning approaches in software engineering [41, 67, 113] did not fully follow the smoothness assumption, failing to explicitly focus on learning a better feature representation of code snippets to effectively separate vulnerable and non-vulnerable ones. Moreover, they also did not adequately mitigate the impact of incorrect pseudo-labeled training code snippets.

1.2 Our Works and Contributions

To address the aforementioned challenges, we propose a **S**emi-**S**upervised **V**ulnerability **D**etection (SSVD) approach, which effectively utilizes a large volume of unlabeled code snippets to enhance detection performance with limited labeled code snippets. SSVD begins by training a teacher model using limited labeled code snippets. It then performs teacher-student self-training iterations. During these iterations, the teacher model generates pseudo-labels for all unlabeled code snippets. Then, SSVD employs the information gain [83] of model parameters as the certainty of the correctness of these pseudo-labels. The certainty reflects the model's confidence in the correctness of the pseudo-label, indicating that the pseudo-labels of code snippets with high certainty are more likely to be correct. This approach is commonly used in the field of uncertainty quantification [1] and provides a better reflection of the model's confidence than classification probabilities. During the training of the student model in each iteration, SSVD utilizes the proposed noise-robust triplet loss to maximize the separation between vulnerable and non-vulnerable code snippets in the latent space, facilitating accurate label propagation from labeled to nearby unlabeled snippets. Additionally, the noise-robust cross-entropy loss is proposed for gradient clipping to mitigate the error accumulation caused by incorrect pseudo-labels. The trained student model is then utilized as the new teacher model in the next iteration to generate pseudo-labels for the next round of training.

than 0.5 (indicating the code snippet is predicted to be non-vulnerable), its classification probability *cp* equals 1-*p*. In essence, *cp* is a decimal value between 0.5 and 1.

This teacher-student training process continues iteratively until further iterations do not lead to significant performance improvements.

We choose two types of vulnerability detection methods to serve as the teacher/ student models integrated into SSVD: token-based (LineVul [24]) and graph-based (ReVeal [7]). We compare SSVD and nine other semi-supervised learning approaches. The experimental results on four widely used vulnerability datasets demonstrate that SSVD outperforms all the baseline approaches. In particular, SSVD achieves the average F1-score improvement of 36.21% and 23.42%, as well as the average Matthews Correlation Coefficient (MCC) improvement of 74.27% and 39.17% across the four vulnerability datasets when using LineVul and ReVeal as the base detection model, respectively. Moreover, SSVD trained on a certain proportion of labeled data can surpass or closely match the performance of fully supervised LineVul and ReVeal models trained on 100% labeled data in most scenarios. This underscores SSVD's effectiveness in learning from both limited labeled data and a vast number of unlabeled code snippets, resulting in superior vulnerability detection performance and reducing the effort needed for labeling a large number of code snippets.

The main contributions of our work can be summarized as follows:

(1) We propose a novel semi-supervised deep learning approach for vulnerability detection, named SSVD, which effectively selects correctly pseudo-labeled code snippets as training data by using the information gain of model parameters to assess the certainty of the pseudo-labels' correctness. It learns a better feature representation through noise-robust triplet loss and mitigates error accumulation through noise-robust cross-entropy loss to enhance vulnerability detection.

(2) We conduct a comprehensive experiment to compare SSVD with nine semi-supervised learning approaches on four vulnerability datasets, and the results show the effectiveness of SSVD for vulnerability detection with limited labeled data.

1.3 Organizations

Section 2 describes the SSVD approach. Sections 3 and 4 present the experimental setup and results, respectively. Section 5 discusses why the SSVD approach works and addresses the threats to validity. Section 6 introduces the related work. Finally, Section 7 concludes the paper and discusses future work.

2 APPROACH

The overall procedure of the SSVD approach, depicted in Figure 1, is based on the self-training framework. Initially, SSVD trains a teacher model f^{θ_t} using limited labeled code snippets $D_L = \{x_i, y_i\}_{i=1}^{N_L}$, where x_i represents the i-th code snippet, y_i represents the corresponding label (i.e., vulnerable or non-vulnerable), and N_L represents the number of labeled code snippets in the dataset. In the subsequent self-training iterations, the teacher model generates pseudo-labels for all unlabeled code snippets $D_U = \{x_u\}_{u=1}^{N_U}$. SSVD employs the information gain [83] of model parameters as the certainty of the correctness of these pseudo-labels, and utilizes the Bayesian Neural Network (BNN) with the Monte Carlo (MC) dropout technique to estimate the certainty. Based on these certainties, sampling weights are calculated for the pseudo-labeled snippets. Next, SSVD samples the pseudo-labeled code snippets based on the sampling weights to form the training dataset D_S for the student model f^{θ_s}. During the training of the student model, we incorporate our proposed noise-robust triplet loss, which aims at maximizing the separation between vulnerable and non-vulnerable code snippets in the feature space, facilitating accurate label propagation from labeled to nearby unlabeled snippets. Additionally, we propose the noise-robust cross-entropy loss to mitigate the issue of error accumulation via gradient clipping during the training process. In subsequent iterations, we assess the performance of the student model compared to the teacher model on the validation set. If the student model outperforms the teacher model, it becomes the new teacher model and generates new pseudo-labels for unlabeled data. Otherwise, we continue using the previous teacher model. This teacher-student training process continues iteratively until further iterations do not lead to significant performance improvement.

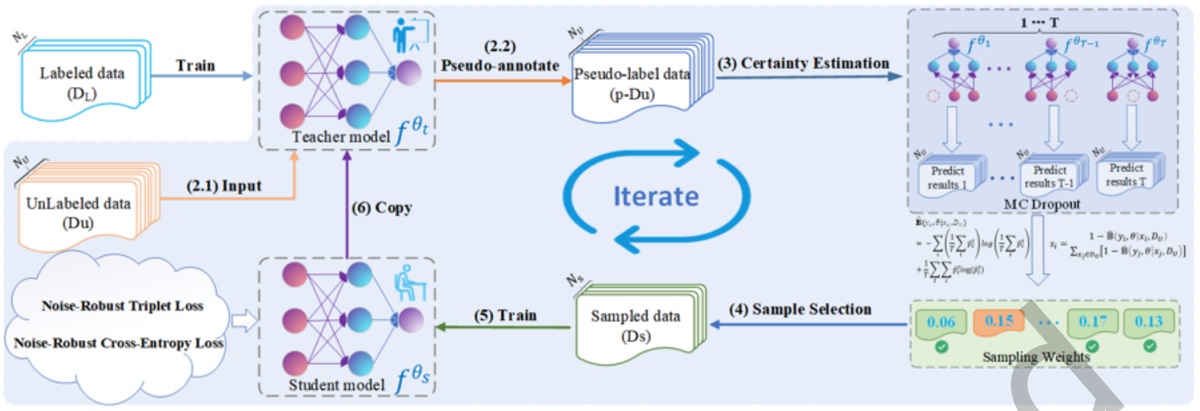


Fig. 1. The overall framework of the SSVD approach.

2.1 Certainty-Aware Sample Selection

Previous studies in software engineering [41, 67, 105, 113] rely on pseudo-labeled samples with high classification probability, which increases the risk of selecting incorrectly pseudo-labeled samples as training data. Therefore, we propose a certainty-aware sample selection strategy to address the problem. According to the information theory proposed by Shannon [83], we utilize the information gain of model parameters to estimate the model's certainty regarding the correctness of the pseudo-labels assigned to the code snippets, following previous studies in the field of artificial intelligence [34, 66, 93] (Section 2.1.1). Then, we utilize the MC dropout technique to approximate a BNN to calculate the information gain (Section 2.1.2). The code snippets with high certainty are more likely to have correct pseudo-labels. Subsequently, we compute sampling weights for each pseudo-labeled code snippet based on its certainty and sample from them accordingly (Section 2.1.3). Specifically, code snippets with higher certainties are assigned a higher probability of being sampled, while those with lower certainties have a lower probability of being selected. By selecting code snippets with high certainty rather than high classification probability, the model can select more correctly pseudo-labeled code snippets as the training data.

2.1.1 Obtaining Approximate Value of $p(y=c|x)$ via BNN and MC Dropout. The BNN is a type of neural network model in which the parameters are not fixed values but instead follow a probability distribution [25]. During the training process, the optimization objective of the BNN model, denoted as f^{θ}, is to obtain a posterior distribution $p(\theta|D_L)$ on the training dataset D_L for the model parameters θ. During the inference process, predictions are made based on the posterior distribution. Specifically, for an input code snippet x, the probability of the output vulnerability label c is represented as the expectation value and can be calculated through integral [2]:

$$p(y=c|x) = \mathbb{E}_{p(\theta|D_L)}\left[p\left(y=c\middle|f^{\theta}(x)\right)\right] = \int_{\theta} p(y=c|f^{\theta}(x))p(\theta|D_L)d\theta. \tag{1}$$

However, computing this expectation value by considering all possible parameters θ is computationally impractical [93]. We have to find a surrogate distribution $q(\theta)$ in a tractable family of distributions to replace the true model posterior distribution $p(\theta|D_L)$. Gal et al. [26] introduced the Monte-Carlo dropout technique, which can be incorporated into the BNN model. They showed that the prediction probability $p(y=c|x)$ for vulnerability label c can be approximated by considering $q(\theta)$ as a dropout distribution [88]. During inference, dropout is

[2]For a continuous random variable X with a probability density function $f(x)$, its expectation $\mathbb{E}(X)$ can be calculated using $\mathbb{E}(X) = \int(xf(x))dx$, where the integral covers the range of possible values of the random variable.

applied by randomly masking certain model parameters T times, enabling outputs to be obtained from different sets of "virtual" model parameters. In practical applications, we only need to ensure that the dropout layer is enabled, and then perform inference T times. Formally, we sample T masked model parameters $\left\{\tilde{\theta}_t\right\}_{t=1}^{T} \sim q(\theta)$, and through the Monte-Carlo dropout technique, we can obtain an approximate value of $p(y = c|x)$:

$$p\left(y = c|x\right) \approx \frac{1}{T}\sum_{t=1}^{T} p\left(y = c\middle|f^{\tilde{\theta}_t}(x)\right). \tag{2}$$

As T becomes sufficiently large, $p(y = c|x)$ and $\frac{1}{T}\sum_{t=1}^{T} p\left(y = c\middle|f^{\tilde{\theta}_t}(x)\right)$ become increasingly similar.

2.1.2 Certainty Estimation. Following previous studies in the field of artificial intelligence [34, 66, 93], we utilize the information gain of model parameters as the certainty measure. We use entropy $\mathbb{H}(\cdot)$ to measure the information we have, where a higher entropy implies greater information:

$$\mathbb{H} = -\sum_{c} p\left(y = c|x\right) log\left(p\left(y = c|x\right)\right). \tag{3}$$

The information gain $\mathbb{B}$ is defined as the difference between the entropy $\mathbb{H}\left(y_u|x_u, D_U\right)$ of the model after seeing information from the entire unlabeled dataset D_U and the entropy $\mathbb{E}_{p(\theta|D_U)}\left[\mathbb{H}\left(y_u|x_u, \theta\right)\right]$ [3] of the model given model parameters θ. Formally, for the code snippet x_u, the information gain $\mathbb{B}$ with respect to its label y_u can be formulated as:

$$\mathbb{B}\left(y_u, \theta|x_u, D_U\right) = \mathbb{H}\left(y_u|x_u, D_U\right) - \mathbb{E}_{p(\theta|D_U)}\left[\mathbb{H}\left(y_u|x_u, \theta\right)\right], \tag{4}$$

where $p\left(\theta|D_U\right)$ represents the posterior distribution of model parameters on the unlabeled dataset D_U. However, computing this information gain value by considering all possible parameters θ is challenging to compute and can be approximated using the MC dropout technique. Therefore, by substituting Equation 2 and Equation 3 into Equation 4, we obtain the following approximate equation:

$$\mathbb{B}\left(y_u, \theta|x_u, D_U\right) \approx -\sum_{c}\left(\frac{1}{T}\sum_{t}\hat{p}_c^t\right) log\left(\frac{1}{T}\sum_{t}\hat{p}_c^t\right) + \frac{1}{T}\sum_{t}\sum_{c}\hat{p}_c^t log\left(\hat{p}_c^t\right) \tag{5}$$

where $\hat{p}_c^t = p\left(y_u = c\middle|f^{\tilde{\theta}_t}\left(x_u\right)\right)$ represents the predicted probability of class c given the Monte-Carlo dropout model parameters $\tilde{\theta}_t \sim q(\theta)$ in the t-th time. When the value of $\mathbb{B}\left(y_u, \theta|x_u, D_U\right)$ is low, it indicates that the model is highly certain about its predicted pseudo-label y_u for the code snippet x_u. This is because no additional information can be gained even after seeing the whole unlabeled code snippets in D_U. Therefore, the information gain $\mathbb{B}\left(y_u, \theta|x_u, D_U\right)$ is inversely proportional to the certainty, and we define the certainty as $1 - \mathbb{B}\left(y_u, \theta|x_u, D_U\right)$. When the information gain is low, the certainty is high.

2.1.3 Sample Selection. Given the certainty, we can sample the unlabeled code snippets accordingly. The certainty reflects the model's confidence in the correctness of the pseudo-label, indicating that the pseudo-labels of the code snippets with high certainty are more likely to be correct. One possible sampling strategy is to prioritize code snippets with high certainties. However, only relying on high-certainty data, which typically have lower information gain, the student model learns little from the data and may even result in potential overfitting. Therefore, we introduce sampling weights and perform weighted sampling. Formally, for $x_u \in D_U$, its sampling weight s_u is:

$$s_u = \frac{1 - \mathbb{B}\left(y_u, \theta|x_u, D_U\right)}{\sum_{x_u \in D_U}\left[1 - \mathbb{B}\left(y_u, \theta|x_u, D_U\right)\right]}. \tag{6}$$

[3]Since the parameters θ follow a probability distribution, the entropy of the model given parameters θ is the expectation.

Specifically, code snippets with high certainty are assigned higher sampling weights, increasing their chances of being selected during sampling. Conversely, those with low certainty receive lower sampling weights and are less likely to be chosen. By employing this approach, we increase the likelihood of obtaining correct pseudo-labels for the student model training, while also incorporating data with higher information gain to prevent overfitting. For example, as shown in Figure 1, we sample the code snippets with the sampling weights of 0.06, ..., 0.17, and 0.13 as the training dataset for the student model.

2.2 Noise-Robust Triplet Loss

2.2.1 The Original Triplet Loss Function. In semi-supervised learning, labels spread from labeled code snippets to nearby unlabeled ones, adhering to the smoothness assumption: a vulnerable code snippet should be closer to another vulnerable one rather than a non-vulnerable one in the feature space, and vice versa [41, 93]. To integrate this smoothness assumption and enhance accurate label propagation from labeled to nearby unlabeled code snippets, we propose a noise-robust triplet loss function that aims to maximize the separation between vulnerable and non-vulnerable code snippets in the feature space of D_S (sampled code snippets) in a semi-supervised setting. The original triplet loss function operates on triplet combination (x_a, x_p, x_n), where x_a represents an anchor code snippet from D_S, x_p represents the positive code snippet from D_S that shares the same class label as x_a (known as matched pairs), and x_n represents the negative code snippet from D_S with the different class label from x_a (known as unmatched pairs). The objective of the triplet loss function is to learn a feature embedding space in which the distance between code snippets with the same class labels is minimized, while maximizing the distance between code snippets with different class labels, as shown in Figure 2. To achieve this, each code snippet $x_{a_{(i)}}$ in D_S is treated as an anchor, and its negative code snippet $x_{n_{(i)}}$ as well as a positive code snippet $x_{p_{(i)}}$ are identified. Hence, the formalization of the original triplet loss function can be expressed as follows:

$$L_{triplet} = \sum_{i}^{N_s} \left[d\left(x_{a_{(i)}},\ x_{p_{(i)}}\right) - d\left(x_{a_{(i)}},\ x_{n_{(i)}}\right) + m \right]_+, \tag{7}$$

where $d\left(x_{a_{(i)}},\ x_{p_{(i)}}\right) = \left\| h(x_{a_{(i)}}) - h\left(x_{p_{(i)}}\right) \right\|_2^2$ represents the distance between matched pairs, $d\left(x_{a_{(i)}},\ x_{n_{(i)}}\right) = \left\| h(x_{a_{(i)}}) - h\left(x_{n_{(i)}}\right) \right\|_2^2$ represents the distance between unmatched pairs, $h(\cdot)$ denotes the embedding function, m is a margin parameter, and the notation $[a]_+$ denotes the positive part of a, i.e., $[a]_+ = max(0,a)$. The goal of the loss function is to make that $d\left(x_{a_{(i)}},\ x_{n_{(i)}}\right)$ is larger than $d\left(x_{a_{(i)}},\ x_{p_{(i)}}\right)$ plus m.

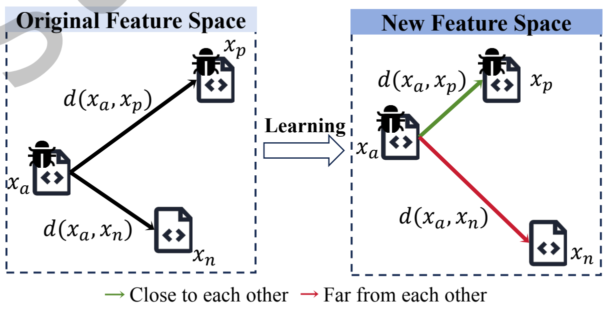
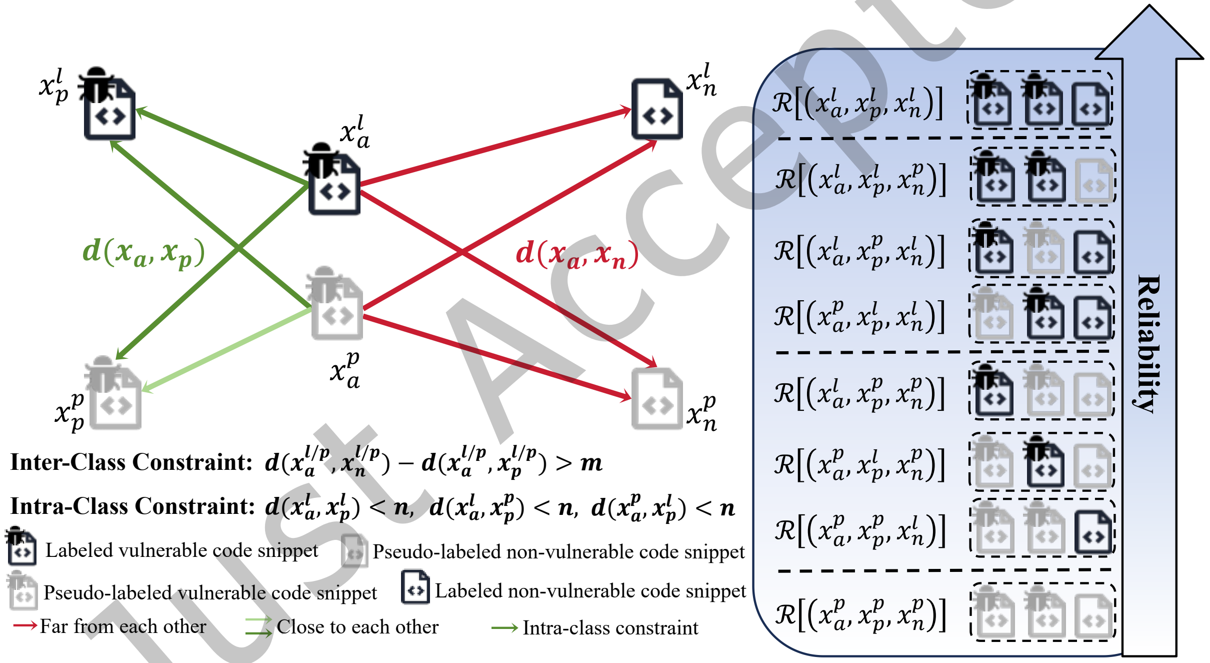


Fig. 2. An example of the original triplet loss function.

However, in the semi-supervised setting, we cannot directly use the original triplet loss function due to the potential inclusion of incorrectly pseudo-labeled code snippets (i.e., noise) in the training data, although we propose to employ certainty-aware sample selection to identify high-certainty code snippets. The presence of noise data may lead to the positive code snippet x_p belonging to a different class than x_a in the triplet loss, while the negative code snippet x_n may be from the same class as x_a. Consequently, the vulnerability detection model may become confused, as it encourages code snippets from different classes to be closer in the feature space. As shown in Figure 3, in the semi-supervised scenario, the feature space contains labeled vulnerable data x_a^l and x_p^l (for the purpose of clarity, we assume that the current anchor class is vulnerable in the figure), pseudo-labeled vulnerable data x_a^p and x_p^p, labeled non-vulnerable data x_n^l, and pseudo-labeled non-vulnerable data x_n^p. These six types of code snippets can form a total of eight possible triplet combinations, i.e., $\left(x_a^l, x_p^l, x_n^l\right)$, $\left(x_a^l, x_p^l, x_n^p\right)$, $\left(x_a^l, x_p^p, x_n^l\right)$, $\left(x_a^p, x_p^l, x_n^l\right)$, $\left(x_a^l, x_p^p, x_n^p\right)$, $\left(x_a^p, x_p^l, x_n^p\right)$, $\left(x_a^p, x_p^p, x_n^l\right)$, and $\left(x_a^p, x_p^p, x_n^p\right)$, where the superscript l (or p) indicates that the corresponding code snippet is from the labeled dataset (or pseudo-labeled dataset). The original triplet loss treats all of the triplet combinations equally, assigning them the same weight when calculating the triplet loss. However, triplet combinations with a higher proportion of pseudo-labeled code snippets should be given lower weights when calculating the triplet loss, as pseudo-labeled code snippets may contain erroneous vulnerability labels.



x_p^l
x_a^l
x_n^l
$d(x_a, x_p)$
$d(x_a, x_n)$
x_a^p
x_p^p
x_n^p
Inter-Class Constraint: $d(x_a^{l/p}, x_n^{l/p}) - d(x_a^{l/p}, x_p^{l/p}) > m$
Intra-Class Constraint: $d(x_a^l, x_p^l) < n$, $d(x_a^l, x_p^p) < n$, $d(x_a^p, x_p^l) < n$
Labeled vulnerable code snippet
Pseudo-labeled non-vulnerable code snippet
Pseudo-labeled vulnerable code snippet
Labeled non-vulnerable code snippet
Far from each other
Close to each other
Intra-class constraint
$\mathcal{R}[(x_a^l, x_p^l, x_n^l)]$
$\mathcal{R}[(x_a^l, x_p^l, x_n^p)]$
$\mathcal{R}[(x_a^l, x_p^p, x_n^l)]$
$\mathcal{R}[(x_a^p, x_p^l, x_n^l)]$
$\mathcal{R}[(x_a^l, x_p^p, x_n^p)]$
$\mathcal{R}[(x_a^p, x_p^l, x_n^p)]$
$\mathcal{R}[(x_a^p, x_p^p, x_n^l)]$
$\mathcal{R}[(x_a^p, x_p^p, x_n^p)]$
Reliability

Fig. 3. An example of the noise-robust triplet loss function.

2.2.2 *The Noise-Robust Triplet Loss Function.* To address the issue, we propose an enhanced version of the original triplet loss function, called the noise-robust triplet loss function. In this approach, we calculate a reliability score $\mathcal{R}$ for each triplet combination. Triplet combinations with lower reliability scores are given lower weights during loss computation, thereby minimizing the potential impact of incorrectly pseudo-labeled data on the training process. Specifically, $\mathcal{R}$ is calculated based on the variance of predictions, which reflects the degree of uncertainty in the model's predictions for a given code snippet. A higher variance indicates greater uncertainty, while a lower variance suggests higher certainty in the predictions. For the pseudo-labeled code snippet x^p within the triplet combination, its variance of predictions is computed over T rounds of MC dropout iterations:

$$Var\left(x^p\right) = \frac{1}{T}\sum_{t=1}^{T}\left(p_t - \frac{1}{T}\sum_{t=1}^{T}p_t\right)^2, \tag{8}$$

where p_t is the pseudo-label prediction probability for x^p in the t-th iteration. For the labeled code snippet x^l, we define $Var\left(x^l\right) = 0$. Therefore, the reliability of a triplet combination $\mathcal{R}\left[\left(x_a, x_p, x_n\right)\right]$ is defined as follows:

$$\mathcal{R}\left[\left(x_a, x_p, x_n\right)\right] = -log\left[\frac{Var\left(x_a\right) + Var\left(x_p\right) + Var\left(x_n\right)}{3}\right]. \tag{9}$$

Higher reliability scores for a triplet combination indicate that the code snippets within the triplet are more likely to have correct pseudo-labels.

In addition, the original triplet loss function only requires that the distance between unmatched pairs (x_a, x_n) should be greater than the distance between matched pairs (x_a, x_p) by a margin distance m (i.e., inter-class constraint). However, it does not specify how close the distance within the matched pairs (x_a, x_p) should be (i.e., intra-class constraint). This lack of specification may lead to a sparse embedding space, where code snippets of the same class are widely scattered, failing to fulfill the smoothness assumption. To address the issue, we introduce an additional term to the noise-robust triplet loss, which further constrains the distance between the matched pair (x_a, x_p) to be smaller than a distance n. Here, n is defined as a value less than m. The final noise-robust triplet loss, incorporating the intra-class constraint, is as follows:

$$L_{NRTL} = \frac{1}{N_s}\sum_{i}^{N_s}\left\{\mathcal{R}\left[\left(x_{a_{(i)}}, x_{p_{(i)}}, x_{n_{(i)}}\right)\right]\left[d\left(x_{a_{(i)}}, x_{p_{(i)}}\right) - d\left(x_{a_{(i)}}, x_{n_{(i)}}\right) + m\right]_+ + \gamma\left[d\left(x_{a_{(i)}}, x_{p_{(i)}}\right) - n\right]_+\right\}, \tag{10}$$

where γ is used to adjust the proportion of the intra-class constraint. Moreover, we calculate the distance only between the matched pairs $\left(x_a^l, x_p^l\right)$, $\left(x_a^l, x_p^p\right)$, $\left(x_a^p, x_p^l\right)$, excluding the matched pair $\left(x_a^p, x_p^p\right)$ containing both pseudo-labeled code snippets. By excluding the matched pair $\left(x_a^p, x_p^p\right)$ from the calculation of the intra-class constraint, we reduce the influence of potentially incorrect pseudo-labels on the optimization process. This exclusion ensures that only pairs containing at least one true labeled code snippet contribute to the intra-class constraint, thereby enhancing the robustness and reliability of the training process.

2.3 Noise-Robust Cross-Entropy Loss

Researchers [24, 107, 123, 125, 132] commonly use cross-entropy loss as a measure of the performance of vulnerability detection models and to guide the training process:

$$L_{CE} = -\left(ylog(p) + (1-y)\log\left(1-p\right)\right), \tag{11}$$

where y is the true label of the code snippet x (1 for vulnerable and 0 for non-vulnerable), and p is the probability of the code snippet being vulnerable. During the backpropagation process in model optimization, we need to calculate the gradient of the model parameters θ based on the L_{CE}. According to the chain rule, we can obtain:

$$\frac{\partial L_{CE}}{\partial \theta} = \frac{\partial L_{CE}}{\partial p}\frac{\partial p}{\partial \theta} = \left(-\frac{y}{p} + \frac{1-y}{1-p}\right)\frac{\partial p}{\partial \theta} = -\frac{1}{cp}\frac{\partial cp}{\partial \theta}, \tag{12}$$

where $\frac{\partial L_{CE}}{\partial p}$ is the gradient of the loss function with respect to p, $\frac{\partial p}{\partial \theta}$ is the gradient of the model output p with respect to the model parameters θ, and cp is the classification probability. When the classification probability cp is smaller, the gradients for the model parameters θ (i.e., $\frac{\partial L_{CE}}{\partial \theta}$) will have larger values [4]. In a fully supervised scenario with correct labels, this characteristic can accelerate the convergence speed of the model during the optimization process. However, pseudo-labels may be incorrect in the semi-supervised scenario. If these incorrectly pseudo-labeled data also have a low classification probability cp, learning from them would result in large gradients for the model parameters θ. The model will aggressively memorize the wrong information through optimization, finally leading to the accumulation of errors.

To address the challenge of incorrect pseudo-labels in semi-supervised scenarios, we propose the noise-robust cross-entropy loss function, aiming to enhance the original cross-entropy loss's robustness against incorrect pseudo-labels (i.e., noise). The noise-robust cross-entropy loss is defined as follows:

$$L_{NRCE} = \begin{cases} -\left(yp + (1-y)(1-p)\right), \ if\ cp \le cp_k \\ -\left(y\log(p) + (1-y)\log(1-p)\right), \ otherwise \end{cases}, \tag{13}$$

where pseudo-labeled code snippets with classification probability cp less than cp_k are involved in gradient clipping. Specifically, L_{NRCE} refrains from applying the logarithmic function to p and $1-p$, in order to prevent the gradients from becoming excessively large during the gradient computation (i.e., we avoid multiplying $\frac{1}{cp}$ in Equation 12 when cp is less than cp_k). During the training process, we set the value of cp_k as the maximum classification probability within the bottom $k\%$ range of classification probabilities. This gradient clipping strategy effectively prevents the model from making excessive updates based on incorrectly pseudo-labeled code snippets, thereby avoiding error accumulation during learning. Finally, the loss function for SSVD is obtained by combining the noise-robust triplet loss and the noise-robust cross-entropy loss:

$$L_{SSVD} = L_{NRTL} + L_{NRCE}. \tag{14}$$

3 EXPERIMENT SETUP

3.1 Datasets

Existing open-source software vulnerability datasets can be categorized into four types based on how the vulnerability labels of code snippets are obtained: developer-provided, security vendor-provided, tool-created, and synthetically created [7, 14]. Our preference is to primarily utilize the two developer-provided datasets, namely Devign [132] and ReVeal$_D$[5] [7], as the core experiment datasets, since they undergo better quality assurance due to manual annotation [14]. To enhance the overall generality of SSVD, we also incorporate the security vendor-provided Big-Vul [20] dataset and synthetically created Juliet [4] dataset. However, we choose not to include the tool-created D2A dataset [130] due to its reliance on static analysis tools for labeling vulnerability-related fixes. These tools often introduce a significant number of false positive vulnerability warnings, leading to inaccuracies in the vulnerability labels of D2A [14]. Although these four chosen datasets are widely utilized in previous vulnerability detection studies, Roland et al. [14] identified some data quality issues within them, including the presence of duplicate and conflicting code snippets, as well as code snippets with inaccurate vulnerability labels. Such issues can adversely affect the model training [14]. To address these concerns, we utilize the cleaned versions of Devign, Big-Vul, and Juliet provided by Roland et al. [14], which eliminate the majority of duplicate and conflicting code snippets. However, due to the lack of appropriate metadata, the ReVeal$_D$ dataset could not

[4]In Equation 12, the negative sign in the gradient do not represent the gradient is negative, but rather indicate the direction of the gradient. It ensures that the parameters are updated in the direction of the loss function descent during the parameter update process.
[5]To differentiate between the vulnerability dataset named Reveal and the base vulnerability detection model also called Reveal, we refer to the dataset as ReVeal$_D$ in this paper.

be cleaned by Roland et al. [14], and we use the original ReVeal$_D$ dataset in our experiments. We summarize the statistical information of the used datasets in Table 1.

Table 1. The statistical information of our experimental datasets.

Dataset	# Code Snippets	Source of Annotation	Vulnerable Ratio
Devign	24,491	developer provided	45.72%
ReVeal$_D$	22,734	developer provided	9.85%
Big-Vul	156,632	security vendor-provided	6.17%
Juliet	38,234	synthetically created	40.50%

3.2 Base Vulnerability Detection Models

It is worth noting that our SSVD approach, unlike the PILOT semi-supervised vulnerability detection approach [105], is model-agnostic, meaning SSVD can be integrated with any underlying vulnerability detection model as the teacher and student models for semi-supervised vulnerability detection. Given the rapidly evolving landscape of deep learning-based vulnerability detection research, it is challenging to single out one or even a few vulnerability detection techniques as representatives of the entire community. To enhance the generalizability and applicability of our SSVD approach, similar to Yang et al. [115], we select two families of vulnerability detection methods: token-based (LineVul [24]) and graph-based (ReVeal [7]) as the base vulnerability detection models integrated into SSVD. Additionally, these two models serve as common baselines for evaluating proposed methods within the vulnerability detection community [6, 103, 104], showcasing their representativeness.

LineVul [24] converts code snippets into code tokens and generates embedding vectors. It then utilizes a BERT architecture with a stack of 12 Transformer encoder blocks for vulnerability detection. ReVeal [7] transforms code snippets into code property graphs. Then, it employs a graph-gated neural network that takes feature vectors of all nodes and edges as input to detect code vulnerability. During the initialization of the teacher model, we train the two base detection models using cross-entropy loss. In the subsequent self-training iterations, we utilize the noise-robust triplet loss and noise-robust cross-entropy loss as the loss function. We utilize the same hyperparameter settings described in LineVul [24] and ReVeal [7], including learning rate and batch size.

3.3 Baselines

We compare SSVD with six semi-supervised deep learning methods and three semi-supervised machine learning methods:

(1) Standard Self-Training (SST) [42] is a classic self-training method that does not consider sample selection and smoothness assumptions. It iteratively trains a classifier on the labeled dataset and then utilizes that classifier to predict labels for all unlabeled samples. These pseudo-labeled samples are subsequently added to the labeled dataset, and the process continues until convergence.

(2) Uncertainty-aware Self-Training (UST) [66] utilizes bayesian neural network to calculate information gain for selecting difficult or easy pseudo-labeled samples for model training, and computes classification loss weights based on the predicted variance of the samples, allows the model to focus more on low-variance samples.

(3) Debiased Self-Training (DST) [8] introduces three different classification heads to address biases during the self-training process. Specifically, it decouples the generation and utilization of pseudo-labels using two parameter-independent classifier heads. Furthermore, it incorporates an additional classification head to consider worst-case classification scenarios and performs adversarial training to avoid the worst-case.

(4) PositIve and unlabeled Learning mOdel for vulnerability deTection (PILOT) [105] first labels unlabeled code snippets as non-vulnerable based on the maximum distance difference from labeled vulnerable code snippets. Next,

a detection model is trained using CodeBERT [22] on both labeled vulnerable code snippets and pseudo-labeled non-vulnerable code snippets. This model is then utilized to classify unlabeled code snippets. Those code snippets with high classification probabilities are selected as pseudo-labeled data, which are then incorporated into the training set for fine-tuning of the model.

(5) HADES [41] is a semi-supervised approach for identifying system anomalies using heterogeneous data. It selects pseudo-labeled samples with high classification probability from the teacher model and trains the student model using both the labeled samples and the pseudo-labeled samples, adjusting hyperparameters to assign a lower weight to the loss function of the pseudo-labeled data.

(6) SSVD (prob) is a variant of SSVD that selects pseudo-labeled samples with high classification probability instead of using certainty-aware sample selection to choose high-certainty samples.

(7) Label Propagation (LP) [76] uses the relationships between labeled and unlabeled samples to assign labels to the unlabeled samples based on the labels of their neighboring instances.

(8) Co-Training (CT) [3] trains two classifiers on different subsets of features or views of the labeled dataset, exchanging predictions on unlabeled samples each iteration. Samples, where the two classifiers agree, are added to the labeled dataset, and the classifiers are iteratively retrained until convergence. The final trained classifiers can then be used to predict new samples.

(9) Tri-Training (TT) [133] extends the concept of co-training by utilizing three classifiers instead of two. Similarly, each classifier is trained on a different subset of features or views of the labeled dataset. However, in tri-training, during each iteration, each classifier labels the unlabeled samples, and a sample is added to the labeled set only if at least two out of the three classifiers agree on its label.

The reasons for choosing these baselines are as follows. SST, UST, and DST are recognized as state-of-the-art semi-supervised deep learning methods in the field of artificial intelligence. PILOT and HADES, on the other hand, are state-of-the-art semi-supervised deep learning methods proposed in the domain of software engineering. Additionally, label propagation, co-training, and tri-training are widely used semi-supervised machine learning methods in software engineering [43, 46, 63, 129]. For the semi-supervised deep learning methods except PILOT, we utilize LineVul and ReVeal as the base detection models. As for the semi-supervised machine learning methods, we employ random forest as the base detection model, following the setup of most software engineering studies [43, 44, 46, 56, 57, 94]. This choice is attributed to findings from these studies indicating that using random forest as the base detection model yields superior performance compared to other basic machine learning classification models such as naive Bayes, decision trees, and k-nearest neighbors. We leverage CodeBERT to extract feature vectors from code snippets as the input of random forest, as it shows excellent performance in extracting semantic features from code in the field of software engineering [69, 105]. However, we refrain from selecting active learning as a baseline method due to its reliance on manual annotation by experts. Including active learning in the comparison would introduce bias and be unfair in this particular context.

3.4 Evaluation

We employ three commonly used evaluation metrics Precision, Recall, and F1 in previous vulnerability detection studies [12, 24, 70, 100, 101, 103, 104, 106, 115, 123, 125]. In addition, recognizing that MCC is recommended for assessing software engineering tasks with class imbalance [6, 92], we also incorporate MCC into our evaluation to provide a comprehensive assessment of vulnerability detection performance. These metrics can be derived from a confusion matrix, as depicted in Table 2. In this matrix, TP denotes the number of vulnerable code snippets correctly identified as vulnerable, FP represents the number of non-vulnerable code snippets incorrectly classified as vulnerable, FN refers to the number of vulnerable code snippets incorrectly predicted as non-vulnerable, and TN corresponds to the number of non-vulnerable code snippets correctly predicted as such. The total number of code snippets in the testing dataset is represented by TP+TN+FN+FP.

Table 2. The confusion matrix

	Truly vulnerable	Truly non-vulnerable
Predicted vulnerable	TP	FP
Predicted non-vulnerable	FN	TN

(1) Precision is the percentage of the truly vulnerable code snippets to all the predicted vulnerable code snippets:

$$Precision = \frac{TP}{TP + FP}. \tag{1}$$

A higher precision means the vulnerability detection model has a lower rate of false alarms.

(2) Recall is the percentage of the correctly predicted vulnerable code snippets to all the truly vulnerable ones in the testing dataset:

$$Recall = \frac{TP}{TP + FN}. \tag{2}$$

A higher recall indicates that the vulnerability detection model can find more truly vulnerable code snippets effectively and reduce the likelihood of missing any potential vulnerabilities.

(3) F1 is considered the harmonic mean between precision and recall, and it provides a balance between the two metrics:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \tag{3}$$

(4) MCC is a fully symmetric metric that considers all four values (TP, TN, FP, and FN) in the confusion matrix when calculating the correlation between ground truth and predicted values. It provides a balanced measure of classification performance, particularly in scenarios where class imbalances exist. MCC values range from -1 to 1, and MCC is defined as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{4}$$

The Wilcoxon signed-rank test [79] and Cliff's δ [61] are widely used in the field of software engineering [10, 21, 47, 116, 120], we also follow the settings of previous works and utilize them to assess the significance of the difference between our SSVD approach and other semi-supervised vulnerability detection approaches. Since multiple comparisons with SSVD are performed, we use the Benjamini-Hochberg (BH) [23] procedure to adjust p-values. The null hypothesis of the Wilcoxon signed-rank test assumes no significant difference between two semi-supervised vulnerability detection approaches, with a predefined significance level of 0.05 (α = 0.05). Selecting a significance threshold of 0.05 is a common practice in statistical hypothesis testing. If the p-value after BH correction is less than 0.05, we reject the null hypothesis, indicating a statistically significant difference between the two approaches. Otherwise, we accept the null hypothesis. If the Wilcoxon signed-rank test reveals a significant difference, we then employ Cliff's δ to determine the magnitude of the difference. The effect size is categorized as negligible (0 <|Cliff's δ| <0.147), small (0.147 ≤ |Cliff's δ| <0.33), medium (0.33 ≤ |Cliff's δ| <0.474), or large (|Cliff's δ| ≥ 0.474). In summary, SSVD is considered to perform significantly better or worse than the compared semi-supervised vulnerability detection approaches, if the p-value is less than 0.05 and the effect size is not negligible. If the p-value is not less than 0.05 or the p-value is less than 0.05 but the effect size is negligible (less than 0.147), the difference is not significant [47].

3.5 Implementation Details

The experiments are conducted on a server equipped with a high-performance 24GB GPU and 90 GB of RAM. With the default SSVD settings, LineVul as the base detection model requires approximately 22GB of GPU memory and less than 10GB of RAM, while ReVeal requires around 10GB of GPU memory and less than 30GB of RAM. Memory and GPU usage can be adjusted by tuning the batch size and the dataset size loaded into memory. Since there is currently no open-source code available for semi-supervised methods that can be directly applied to software vulnerability datasets, we adapt the chosen semi-supervised methods by modifying or reproducing them to suit our dataset and base detection models. In order to maintain fairness, we ensure that the compared semi-supervised approach and our SSVD approach have consistent hyperparameters in the overlapping experimental settings, including the sampling ratio for unlabeled code snippets of 25% and the minimum teacher-student self-training iterations of 25. We set the minimum epochs of training for the teacher model to 35, and stop training if the model performance on the validation set does not increase. Based on the experimental results in Section 4.4, we set k to 10%, and we set γ to 0.15. Following Mukherjee et al. [66], we set T to 30. Furthermore, our preliminary experiments confirm that setting T to 30 yields the best results, and increasing T further does not lead to a performance improvement in SSVD. Following Chakraborty et al. [7], we set m to 0.5. As n is a value smaller than m, we set it to 0.1.

For the Big-Vul dataset, we partition the dataset into train, validation, and test sets with an 8:1:1 ratio based on the "update date" of the vulnerability data, maintaining chronological order. The training set is further divided into labeled and unlabeled data. Then, we run these semi-supervised vulnerability detection approaches on these train, validation, and test sets ten times. However, for the Devign, ReVeal$_D$, and Juliet datasets, as they lack time information, we employ five-fold cross-validation to evaluate the models and mitigate the impact of data partitioning. In five-fold cross-validation, the dataset is divided into five equally sized subsets, where four subsets are used as the training set, and the remaining subset is equally split into the test set and the validation set. This process is repeated five times, each time using a different subset as the test and validation set. Subsequently, the five-fold cross-validation procedure is repeated twice, each time with a different random seed for dataset partitioning. Consequently, for each dataset, we obtain ten results. These ten results are utilized for statistical significance analysis using the Wilcoxon signed-rank test and Cliff's δ. Additionally, the averages of the results from the ten runs are presented in Section 4.

4 EXPERIMENT RESULTS

We organize the experiment results by addressing the Research Questions (RQs), which are presented as the titles for each subsection.

4.1 RQ1: How does SSVD perform in comparison to existing semi-supervised learning approaches in vulnerability detection?

We compare SSVD with six semi-supervised deep learning approaches and three semi-supervised machine learning approaches. Table 3 shows the performance of the semi-supervised deep learning approaches using LineVul as the base detection model and the semi-supervised machine learning approaches, while Table 4 presents the performance of the semi-supervised deep learning approaches using the ReVeal detection model and the semi-supervised machine learning approaches. Gray-shaded cells in the table highlight significant differences between SSVD and other approaches in terms of this metric, and each bolded value indicates that the semi-supervised approach achieves the highest performance.

SSVD outperforms the three semi-supervised deep learning approaches in the field of artificial intelligence, including SST, UST, and DST. Regardless of whether ReVeal or LineVul is used as the base detection model, SSVD consistently surpasses these approaches in terms of F1-score and Recall across all datasets, and in terms of MCC across all datasets except UST on the ReVeal$_D$ dataset. When LineVul is used as the base detection model, SSVD

Table 3. The Precision, Recall, F1-score, and MCC values of SSVD and the compared approaches when using LineVul as the base detection model.

Dataset		Full Train	10% Labeled Data										
		LineVul	LineVul	SST	UST	DST	PILOT	HADES	SSVD (prob)	LP	CT	TT	SSVD
Devign	P	0.6100	**0.7284**	0.5911	0.5481	0.5556	0.5461	0.5756	0.5484	0.4871	0.4833	0.6000	0.5426
	R	0.5665	0.1993	0.3297	0.4524	0.4615	0.4283	0.4469	0.5214	0.4605	**0.6340**	0.2093	0.5992
	F	0.5676	0.3074	0.4070	0.4888	0.4964	0.4765	0.4972	0.5253	0.4734	0.5430	0.3035	**0.5622**
	M	0.2522	**0.1930**	0.1456	0.1366	0.1491	0.1325	0.1666	0.1574	0.0500	0.0576	0.1170	0.1723
ReVeal$_D$	P	0.4727	0.3578	0.3649	0.3924	0.3651	0.3780	0.3562	0.3548	0.2897	0.1805	**0.4195**	0.3305
	R	0.3734	0.2241	0.2563	0.2741	0.2946	0.2634	0.2634	0.3019	0.1643	0.2378	0.0475	**0.3438**
	F	0.4073	0.2724	0.2942	0.3187	0.3243	0.3079	0.3008	0.3160	0.2096	0.2036	0.0847	**0.3370**
	M	0.3579	0.2212	0.2406	**0.2660**	0.2617	0.2537	0.2416	0.2546	0.1564	0.1059	0.1135	0.2630
Big-Vul	P	0.8713	0.8333	0.8225	0.8624	0.8496	0.8531	0.8515	0.8553	0.1975	0.2251	0.1229	**0.8732**
	R	0.6898	0.6213	0.6263	0.6103	0.6184	0.6168	0.6217	0.6130	0.1451	0.0111	0.0014	**0.6267**
	F	0.7689	0.7117	0.7110	0.7146	0.7156	0.7157	0.7186	0.7139	0.1672	0.0211	0.0028	**0.7297**
	M	0.7619	0.7041	0.7020	0.7110	0.7100	0.7106	0.7129	0.7093	0.1221	0.0377	0.0079	**0.7259**
Juliet	P	0.7432	**0.6411**	0.6367	0.6179	0.6155	0.6268	0.6297	0.6248	0.5696	0.5114	0.5968	0.6368
	R	0.7994	0.6181	0.6309	0.6960	0.7123	0.6814	0.7017	0.7109	0.5083	0.7117	0.4725	**0.7153**
	F	0.7701	0.6241	0.6288	0.6512	0.6585	0.6513	0.6609	0.6651	0.5371	0.5950	0.5274	**0.6738**
	M	0.6052	0.3817	0.3839	0.3961	0.4013	0.4004	0.4135	0.4078	0.2561	0.2521	0.2719	**0.4312**

Table 4. The Precision, Recall, F1-score, and MCC values of SSVD and the compared approaches when using ReVeal as the base detection model.

Dataset		Full Train	10% Labeled Data										
		ReVeal	ReVeal	SST	UST	DST	PILOT	HADES	SSVD (prob)	LP	CT	TT	SSVD
Devign	P	0.5529	0.5171	0.4944	0.4880	0.4929	0.5150	0.5003	0.4958	0.4871	0.4833	**0.6000**	0.5101
	R	0.4399	0.2686	0.5199	0.5596	0.4930	0.3785	0.5037	0.5960	0.4605	0.6340	0.2093	**0.6509**
	F	0.4893	0.3516	0.5059	0.5207	0.4887	0.4323	0.4997	0.5403	0.4734	0.5430	0.3035	**0.5720**
	M	0.1444	0.0667	0.0723	0.0661	0.0672	0.0819	0.0810	0.0863	0.0500	0.0576	0.1170	**0.1258**
ReVeal$_D$	P	0.4281	0.2808	0.2862	0.3199	0.3118	0.3273	0.3133	0.3030	0.2897	0.1805	**0.4195**	0.3125
	R	0.4315	0.2848	0.3125	0.3179	0.3196	0.2777	0.3098	0.3286	0.1643	0.2378	0.0475	**0.4018**
	F	0.4185	0.2744	0.2930	0.3160	0.3119	0.2981	0.3088	0.3095	0.2096	0.2036	0.0847	**0.3516**
	M	0.3619	0.1977	0.2143	0.2425	0.2374	0.2301	0.2347	0.2338	0.1564	0.1059	0.1135	**0.2734**
Big-Vul	P	0.2768	0.1593	0.1614	0.1662	0.1598	0.1724	0.1679	0.1606	0.1975	**0.2251**	0.1229	0.1798
	R	0.1531	0.1226	0.1365	0.1559	0.1322	0.1229	0.1373	0.1408	0.1451	0.0111	0.0014	**0.1690**
	F	0.1951	0.1361	0.1457	0.1588	0.1432	0.1421	0.1495	0.1482	0.1672	0.0211	0.0028	**0.1729**
	M	0.1668	0.0888	0.0954	0.1060	0.0932	0.0977	0.1007	0.0967	**0.1221**	0.0377	0.0079	0.1206
Juliet	P	0.7044	**0.5970**	0.5871	0.5807	0.5784	0.5854	0.5965	0.5893	0.5696	0.5114	0.5968	0.5904
	R	0.7192	0.6962	0.7482	0.7696	0.7692	0.7585	0.7351	0.7835	0.5083	0.7117	0.4725	**0.7936**
	F	0.7111	0.6424	0.6574	0.6618	0.6598	0.6601	0.6581	0.6721	0.5371	0.5950	0.5274	**0.6765**
	M	0.5116	0.3693	0.3822	0.3847	0.3814	0.3855	0.3890	0.4050	0.2561	0.2521	0.2719	**0.4123**

demonstrates superior performance in terms of F1-score, with an average improvement ranging from 4.91% to 12.82%, and it also achieves a higher MCC with an average improvement ranging from 4.62% to 8.17% across the datasets. On the other hand, when ReVeal is employed as the base detection model, SSVD demonstrates an average improvement in F1-score ranging from 6.98% to 10.68%, an improvement in MCC ranging from 16.62% to

22%, and outperforms these three baseline approaches in Precision in most cases. Moreover, the results of the Wilcoxon signed-rank test and Cliff's δ indicate that SSVD significantly outperforms these three approaches in most cases in terms of F1-score and MCC. Among the three methods, SST shows the worst performance due to its lack of a sampling strategy for selecting suitable pseudo-labeled code snippets. Additionally, the direct use of the original cross-entropy loss exacerbates error accumulation. The superior performance of SSVD can be attributed to its consideration of the smoothness assumption and incorrectly pseudo-labeled training code snippets, which are not taken into account by the three compared approaches.

SSVD surpasses PILOT, HADES, and SSVD (prob), proposed in the field of software engineering. When employing LineVul as the base detection model, SSVD achieves an average F1-score improvement of 7.03%, 5.75%, and 3.71% compared to PILOT, HADES, and SSVD (prob), respectively. When using ReVeal, SSVD demonstrates an average F1-score improvement of 15.69%, 9.72%, and 6.17% respectively. Additionally, when using LineVul as the base detection model, SSVD achieves average MCC improvements of 6.36%, 3.78%, and 4.15%. When using ReVeal as the base detection model, SSVD achieves average MCC improvements of 17.23%, 15.74%, and 13.45% respectively. Moreover, the results of the Wilcoxon signed-rank test and Cliff's δ indicate that when LineVul is used as the base detection model, SSVD significantly outperforms these three approaches in terms of the F1-score. Similarly, when ReVeal is used as the base detection model, SSVD significantly outperforms these three approaches in terms of Recall, F1-score, and MCC in the majority of cases. The primary reason for the improved performance of SSVD over the three approaches lies in its focus on selecting pseudo-labeled code snippets with high certainty rather than high classification probability. High classification probability in pseudo-labels does not necessarily indicate the model's certainty in its predictions and increases the risk of choosing incorrectly pseudo-labeled code snippets as training data. Additionally, SSVD considers the smoothness assumption and the issue of incorrectly pseudo-labeled training code snippets compared to PILOT and HADES, which contributes to its superior performance.

The three machine learning approaches, label propagation, co-training, and tri-training, perform the worst among all semi-supervised approaches. Specifically, when using LineVul as the base detection model, SSVD achieves an average enhancement of 65.97%, 68.99%, and 150.73% in the F1-score compared to the three approaches. Similarly, with ReVeal as the base model, SSVD showcases average improvements of 27.81%, 30.11%, and 93.06%, respectively. Additionally, when LineVul is used as the base detection model, SSVD exhibits significant improvements of 172.4%, 251.37%, and 212.09% in terms of MCC. Similarly, when ReVeal serves as the base detection model, SSVD demonstrates notable improvements of 59.47%, 105.66%, and 82.68% in terms of MCC. Furthermore, the results of the Wilcoxon signed-rank test and Cliff's δ indicate that when LineVul is used as the base detection model, SSVD significantly outperforms these three approaches on all datasets. Similarly, when ReVeal is used as the base detection model, SSVD demonstrates significant superiority over these three approaches in terms of Recall, F1-score, and MCC in the majority of cases. Although tri-training achieves a high Precision value on several datasets, it also exhibits very low Recall on these datasets. As a result, testers may be hesitant to use it due to the high ratio of false negatives, leading to the possibility of numerous undetected potential vulnerabilities [35, 39]. The primary reason for the poor performance of these three machine learning approaches is their reliance on code features extracted using CodeBERT and their utilization of the random forest classifier. In contrast, SSVD leverages LineVul and ReVeal to automatically extract vulnerability-related features, which can be more effective. Additionally, these machine learning approaches do not consider the smoothness assumption and the issue of incorrectly pseudo-labeled training code snippets, further contributing to their inferior performance.

When using LineVul as the base detection model, SSVD significantly outperforms LineVul trained on 10% labeled data across all datasets and metrics, except on the Devign dataset in terms of Precision. Specifically, SSVD outperforms LineVul by 2.53% to 82.93% in terms of F1-score and 3.1% to 18.9% in terms of MCC. When using Reveal as the base detection model, SSVD significantly outperforms Reveal trained on 10% labeled data across all datasets and metrics, except on the Juliet dataset in terms of Precision. When trained on 10% labeled data, LineVul and Reveal are unable to adequately learn vulnerability patterns due to the limited labeled vulnerability data. As a result, compared to SSVD, they are more likely to predict new code snippets as non-vulnerable, leading to lower precision for SSVD compared to LineVul and Reveal. Specifically, SSVD demonstrates superiority over ReVeal by margins ranging from 5.32% to 62.67% in F1-score and 11.64% to 88.86% in MCC across the four datasets. These results demonstrate that SSVD effectively leverages 90% unlabeled data to significantly enhance vulnerability detection capabilities.

When employing LineVul as the base detection model, SSVD demonstrates a higher F1-score and MCC on the Big-Vul dataset compared to its performance with ReVeal. The result is consistent with the findings of previous research conducted by Fu et al. [24], which proposed the LineVul approach and conducted a comparative analysis with Devign—a graph-based approach similar to ReVeal. The experiment conducted by Fu et al. [24] revealed a significantly higher F1-score for LineVul in comparison to Devign on the Big-Vul dataset. The potential reason could be that the smaller graph-based model Reveal struggles to capture the complex structural information of data flow graphs and control flow graphs in extremely imbalanced large datasets [17, 105], while the larger BERT-based model (LineVul) has a greater capacity to capture complex underlying semantic features.

Answer to RQ1:

SSVD surpasses existing semi-supervised learning approaches in terms of Recall, F1-score, and MCC, exhibiting an average improvement of 45.91%, 36.21%, and 74.27% when employing LineVul as the base detection model, and an average improvement of 40.69%, 23.42%, and 39.17% when utilizing ReVeal as the base detection model across the four vulnerability datasets.

4.2 RQ2: How does the proportion of labeled training data impact SSVD's performance?

In RQ1, we assume that 10% of the data in the training set is labeled, while 90% is unlabeled. To illustrate the impact of the proportion of labeled training data, we compare the performance of SSVD across different proportions of labeled data, ranging from 10% to 100% (10%, 30%, 50%, 70%, 90%, and 100%). Figures 4-7 demonstrate how the proportion of labeled training data affects the performance of SSVD and the supervised learning approaches LineVul and ReVeal. In these figures, the notations SSVD (LineVul) and SSVD (ReVeal) indicate that SSVD uses LineVul and ReVeal as the base detection models, respectively.

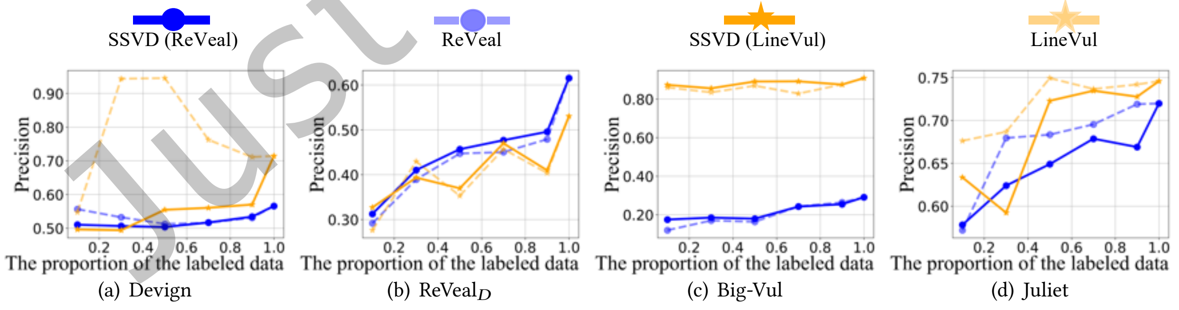


Fig. 4. The Precision value of SSVD (LineVul), SSVD (ReVeal), LineVul, and ReVeal when trained with different proportions of labeled training data on the four datasets.

In terms of all metrics, there is a consistent trend of improved performance for SSVD as the proportion of labeled training data increases. However, there are instances where the performance fluctuates as we increase the proportion of labeled training data. For example, the F1-score and MCC values of SSVD using LineVul as the

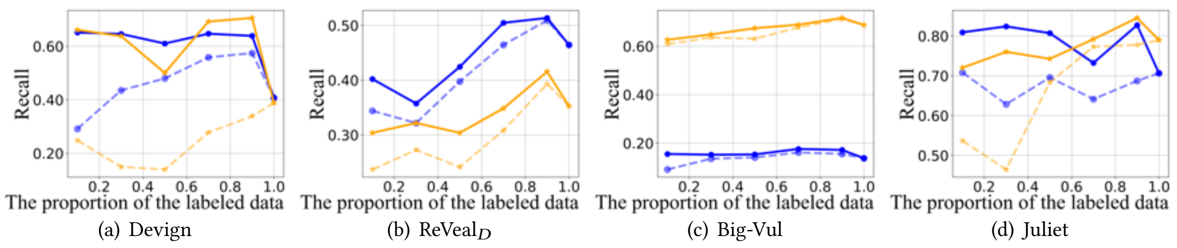


Fig. 5. The Recall value of SSVD (LineVul), SSVD (ReVeal), LineVul, and ReVeal when trained with different proportions of labeled training data on the four datasets.

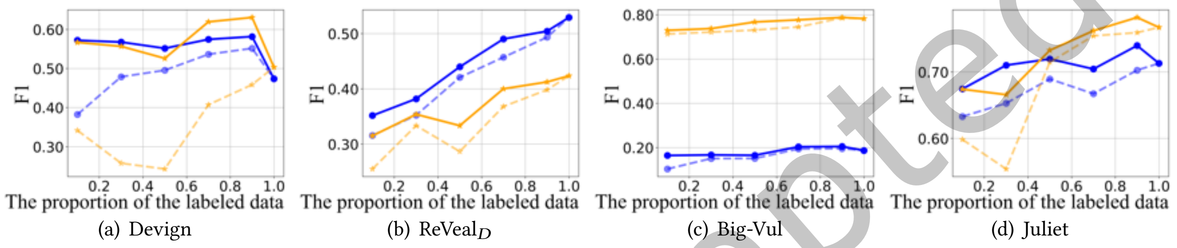


Fig. 6. The F1-score of SSVD (LineVul), SSVD (ReVeal), LineVul, and ReVeal when trained with different proportions of labeled training data on the four datasets.

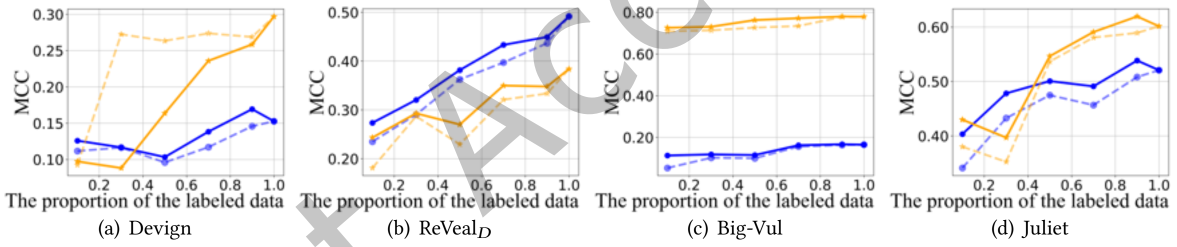


Fig. 7. The MCC value of SSVD (LineVul), SSVD (ReVeal), LineVul, and ReVeal when trained with different proportions of labeled training data on the four datasets.

base detection model exhibit a slight decrease when trained with 30% labeled data compared to when trained with 10% labeled data in both the Devign and Juliet datasets. When analyzing the performance on the Big-Vul dataset, SSVD shows the slight improvement trend as the proportion of labeled data increases. One potential reason for this could be that the Big-Vul dataset contains a substantial amount of code snippets compared to the other three datasets. Hence, using only 10% of labeled code snippets may already provide sufficient information for training vulnerability detection models effectively. As the proportion of labeled data increases, the additional labeled data may not contribute to further improving the performance of SSVD on the Big-Vul dataset.

In the vast majority of datasets and evaluation metrics, there is at least one scenario where SSVD trained on a certain proportion of labeled data outperforms or closely matches the performance of the fully supervised LineVul or ReVeal approaches (trained on 100% labeled data). For example, concerning F1-score and Recall, SSVD (LineVul) and SSVD (ReVeal) trained on only 10% labeled data have exhibited better performance compared to

LineVul and ReVeal trained on 100% labeled data on the Devign dataset, respectively. These findings indicate that by leveraging the unlabeled data, SSVD can overcome the limitations associated with having a limited amount of labeled data, achieving competitive or even superior performance in comparison to fully supervised approaches.

Regardless of the proportion of labeled training data used, SSVD (LineVul) consistently outperforms LineVul, while SSVD (ReVeal) surpasses ReVeal in terms of Recall, F1-score, and MCC in most scenarios. Although LineVul demonstrates higher Precision and MCC values on the Devign dataset than SSVD (LineVul), it exhibits very low Recall and F1-score values, potentially resulting in a large number of actually vulnerable code snippets remaining undetected. The primary reason for the superior performance of SSVD is attributed to the ability of SSVD to effectively leverage the additional unlabeled data during the training process compared to the supervised LineVul and ReVeal approaches. By incorporating the unlabeled data into the training process, SSVD can capture more information from the data, leading to improved performance in terms of Recall, F1-score, and MCC.

Answer to RQ2:

Increasing the proportion of labeled training data typically leads to improved performance for SSVD. SSVD trained on a certain proportion of labeled data can outperform or closely match the performance of LineVul and ReVeal trained on 100% labeled data in most scenarios.

4.3 RQ3: What is the impact of the three modules (i.e., certainty-aware sample selection, noise-robust triplet loss, and noise-robust cross-entropy loss) for SSVD?

We conduct ablation studies to assess the effectiveness of each module in SSVD, namely the Certainty-Aware Sample Selection module (CASS), the Noise-Robust Triplet Loss module (NRTL), and the Noise-Robust Cross-Entropy Loss module (NRCE). Table 5 presents the performance comparison of SSVD and its variants on the four vulnerability datasets.

To explore the contribution of the CASS module, we create a variant of SSVD without CASS (referred to as SSVD$_{w/oCASS}$) that selects all pseudo-labeled code snippets to train the student model. The inclusion of CASS in SSVD demonstrates an improvement in the performance of SSVD across all datasets in terms of F1-score (1.58% in Devign, 23.85% in ReVeal$_D$, 2.08% in Big-Vul, and 1.13% in Juliet), Precision (7.72% in Devign, 19.53% in ReVeal$_D$, 0.96% in Big-Vul, and 0.74% in Juliet), MCC (64.34% in Devign, 35.57% in ReVeal$_D$, 2.04% in Big-Vul, and 2.82% in Juliet), and on three out of the four datasets in terms of Recall (28.33% in ReVeal$_D$, 2.89% in Big-Vul, and 1.56% in Juliet) using the LineVul detection method. When utilizing ReVeal as the base detection model, the CASS module enhances the performance of SSVD in terms of Recall (0.56% in Devign, 3.45% in ReVeal$_D$, and 35.07% in Big-Vul), F1-score (1.19% in Devign, 1.44% in ReVeal$_D$, 28.2% in Big-Vul, and 1.78% in Juliet), MCC (17.29% in Devign, 1.82% in ReVeal$_D$, 45.38% in Big-Vul, and 7.41% in Juliet), and Precision (1.67% in Devign, 22.89% in Big-Vul, and 3.65% in Juliet). The results indicate that the CASS module focusing on selecting high-certainty pseudo-labeled code snippets can benefit the performance of vulnerability detection.

To investigate the effectiveness of the NRTL module, we deploy a variant of SSVD without the NRTL module (referred to as SSVD$_{w/oNRTL}$), where we utilize the noise-robust cross-entropy loss as the loss function. When using LineVul as the base detection model, the inclusion of NRTL in SSVD results in improved performance on two out of the four datasets in terms of Recall (29.06% in Devign, and 4.33% in Juliet), as well as across all datasets in terms of F1-score (16.19% in Devign, 2.18% in ReVeal$_D$, 1.59% in Big-Vul, and 2.05% in Juliet), Precision (7.38% in Devign, 4.29% in ReVeal$_D$, 4% in Big-Vul, and 0.02% in Juliet), and MCC (109.21% in Devign, 3.76% in ReVeal$_D$, 2.11% in Big-Vul, and 3.93% in Juliet). Similarly, when employing the ReVeal detection method, the NRTL module enhances the performance of SSVD on all datasets in terms of Recall (3.25% in Devign, 11.12% in ReVeal$_D$, 21.04% in Big-Vul, and 1.51% in Juliet), F1-score (2.2% in Devign, 2.45% in ReVeal$_D$, 19.02% in Big-Vul, and 3.13% in Juliet), and MCC (17.6% in Devign, 2.11% in ReVeal$_D$, 31.66% in Big-Vul, and 12.02% in Juliet), as well as on three out of

Table 5. The performance comparison of SSVD and its variants on the four datasets.

Datasets	approaches	LineVul				ReVeal			
		Precision	Recall	F1-score	MCC	Precision	Recall	F1-score	MCC
Devign	$\text{SSVD}_{w/oCASS}$	0.5037	**0.6143**	0.5535	0.1048	0.5017	0.6473	0.5653	0.1073
	$\text{SSVD}_{w/oNRTL}$	0.5053	0.4643	0.4839	0.0823	0.5032	0.6304	0.5597	0.1070
	$\text{SSVD}_{w/oNRCE}$	0.5015	0.6054	0.5485	0.0988	**0.5144**	0.6223	0.5632	**0.1278**
	SSVD	**0.5425**	0.5992	**0.5622**	**0.1722**	0.5101	**0.6509**	**0.5720**	0.1259
ReVeal_D	$\text{SSVD}_{w/oCASS}$	0.2765	0.2679	0.2721	0.1940	0.3129	0.3884	0.3466	0.2686
	$\text{SSVD}_{w/oNRTL}$	0.3169	0.3435	0.3298	0.2535	**0.3266**	0.3616	0.3432	0.2678
	$\text{SSVD}_{w/oNRCE}$	0.3039	0.2768	0.2897	0.2164	0.2813	0.3616	0.3164	0.2335
	SSVD	**0.3305**	**0.3438**	**0.3370**	**0.2630**	0.3125	**0.4018**	**0.3516**	**0.2735**
Big-Vul	$\text{SSVD}_{w/oCASS}$	0.8649	0.6091	0.7148	0.7115	0.1463	0.1251	0.1349	0.0830
	$\text{SSVD}_{w/oNRTL}$	0.8396	**0.6277**	0.7183	0.7109	0.1515	0.1396	0.1453	0.0916
	$\text{SSVD}_{w/oNRCE}$	0.8480	0.6174	0.7145	0.7087	0.1393	0.1282	0.1335	0.0791
	SSVD	**0.8732**	0.6267	**0.7297**	**0.7260**	**0.1797**	**0.1689**	**0.1729**	**0.1206**
Juliet	$\text{SSVD}_{w/oCASS}$	0.6321	0.7043	0.6663	0.4195	0.5696	**0.7979**	0.6647	0.3838
	$\text{SSVD}_{w/oNRTL}$	0.6367	0.6856	0.6602	0.4150	0.5651	0.7818	0.6560	0.3681
	$\text{SSVD}_{w/oNRCE}$	0.6261	0.7082	0.6646	0.4141	0.5731	0.7870	0.6632	0.3832
	SSVD	**0.6368**	**0.7153**	**0.6738**	**0.4313**	**0.5903**	0.7936	**0.6765**	**0.4123**

the four datasets in terms of Precision (1.37% in Devign, 18.68% in Big-Vul, and 4.47% in Juliet). These findings highlight the performance improvement in vulnerability detection achieved by incorporating the NRTL module into SSVD, which maximizes the separation between vulnerable and non-vulnerable code snippets in the latent space, facilitating more accurate label propagation from labeled to nearby unlabeled snippets.

We evaluate the impact of the NRCE module by deploying a variant of SSVD without the NRCE module (referred to as $\text{SSVD}_{w/oNRCE}$). In this variant, the noise-robust triplet loss is used in conjunction with the original cross-entropy loss as the loss function. The inclusion of NRCE in SSVD enhances the performance across all datasets, as evidenced by improved F1-score (2.5% in Devign, 16.33% in ReVeal_D, 2.13% in Big-Vul, and 1.38% in Juliet), MCC (74.43% in Devign, 21.54% in ReVeal_D, 2.43% in Big-Vul, and 4.15% in Juliet), Precision (8.19% in Devign, 8.75% in ReVeal_D, 2.97% in Big-Vul, and 1.71% in Juliet), and three out of four datasets in terms of Recall (24.21% in ReVeal_D, 1.51% in Big-Vul, and 1.01% in Juliet) under the LineVul detection method. Furthermore, the NRCE module also improves SSVD's performance on all dataset under the ReVeal detection method, resulting in improved Recall (4.60% in Devign, 11.12% in ReVeal_D, 31.81% in Big-Vul, and 0.84% in Juliet), F1-score (1.56% in Devign, 11.13% in ReVeal_D, 29.54% in Big-Vul, and 2.01% in Juliet), and three out of four datasets in terms of MCC (17.1% in ReVeal_D, 52.44% in Big-Vul, and 7.58% in Juliet) and Precision (11.09% in ReVeal_D, 29.07% in Big-Vul, and 3.01% in Juliet). These results show that the NRCE module, aimed at mitigating the issue of error accumulation during the training process, is also beneficial for improving vulnerability detection performance.

Answer to RQ3:

The certainty-aware sample selection, noise-robust triplet loss, and noise-robust cross-entropy loss modules demonstrate their effectiveness in enhancing SSVD's performance.

4.4 RQ4: How do hyper-parameters affect the performance of SSVD?

We explore the impact of two key hyper-parameters, including γ which represents the weight assigned to the intra-class constraint term, and the bottom $k\%$ data for gradient clipping.

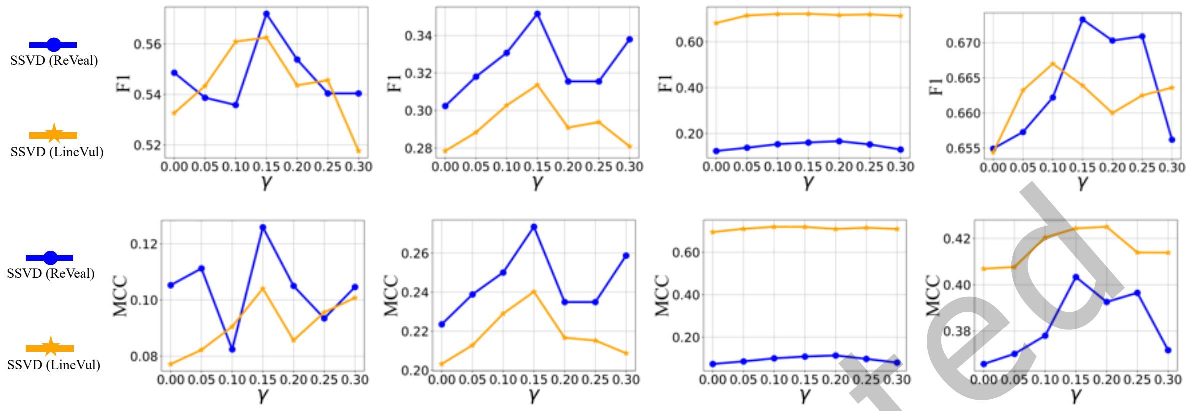


Fig. 8. The effect of different γ on the performance of the SSVD (From left to right, the columns represent Devign, ReVeal$_D$, Big-Vul, and Juliet datasets, respectively).

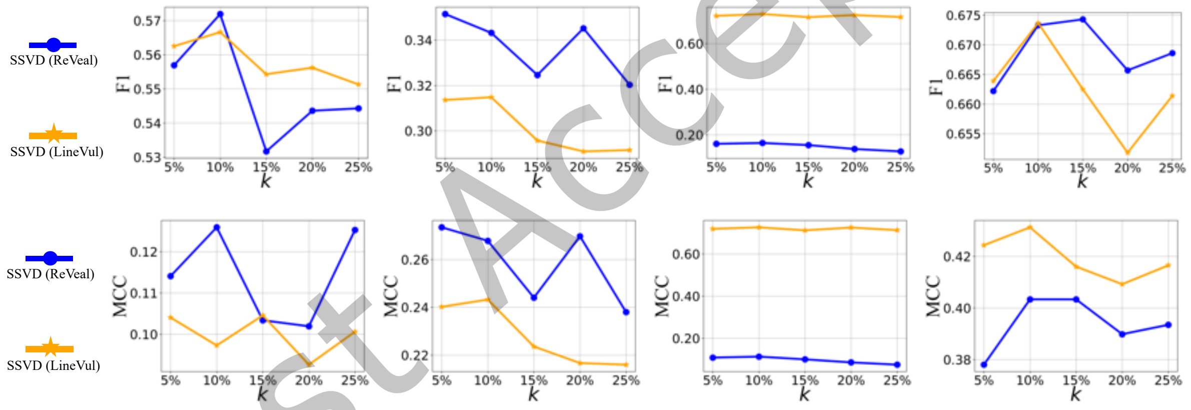


Fig. 9. The effect of different k on the performance of the SSVD (From left to right, the columns represent Devign, ReVeal$_D$, Big-Vul, and Juliet datasets, respectively).

The weight of the intra-class constraint term. Figure 8 shows the performance of SSVD on F1-score and MCC with different weights of the intra-class constraint term γ using the LineVul and ReVeal base detection models, respectively. A higher value of γ indicates that the intra-class constraint term has a higher weight in the noise-robust triplet loss function. In both the LineVul and ReVeal models, an upward trend followed by a subsequent decline in the F1-score and MCC can be observed across the majority of datasets. When employing the LineVul as the detection model, a γ setting of 0.15 yields the highest F1-scores and MCC on Devign, ReVeal$_D$, and Big-Vul datasets. When employing the ReVeal as the detection model, a γ setting of 0.15 achieves the highest results in terms of F1-score and MCC across all datasets. When the intra-class constraint term is not used, i.e., by setting γ to 0, the lowest F1-score and MCC are achieved on the ReVeal$_D$, Big-Vul, and Juliet datasets. This demonstrates the effectiveness of the intra-class constraint term. In most cases, setting γ>0.15 can result in a decline in model performance. This is because we calculate the distance for the matched pairs $\left(x_a^l, x_p^l\right)$, $\left(x_a^l, x_p^p\right)$, and $\left(x_a^p, x_p^l\right)$ in the intra-class constraint term. Since pseudo-labeled code snippets x_a^p and x_p^p may have incorrect pseudo-labels, an increase in the proportion of the intra-class constraint term may cause these incorrectly labeled snippets and genuinely labeled ones to appear closer, even though they have different true labels.

Bottom k% data for gradient clipping. We also explore the effect of the bottom k% data for gradient clipping. A larger value of k indicates that more code snippets are involved in gradient clipping. Figure 9 illustrates the impact of the proportion of data involved in gradient clipping on the performance of SSVD. When utilizing the LineVul detection model, a k setting of 10 yields the highest F1-score across all four datasets and the highest MCC on ReVeal$_D$, Big-Vul, and Juliet. On the other hand, when employing the ReVeal detection model, a k setting of 10 results in the highest F1-score and MCC on Devign and Big-Vul. Setting k>10 generally leads to a decrease in F1 and MCC values in most cases. This decline may be attributed to more correctly pseudo-labeled code snippets being clipped, causing the potential loss of some critical gradient information and consequently reducing performance.

Aswer to RQ4:

When the hyper-parameters are set to $\gamma = 0.15$ and $k = 10$, SSVD demonstrates superior performance on the majority of datasets.

4.5 RQ5: What are the training and testing time costs of SSVD?

Table 6. The average time costs on four datasets with and without using SSVD.

Approaches	Training time	Test time
LineVul	5h 34m 59s	0.21s
SSVD (LineVul)	14h 28m 47s	0.23s
ReVeal	2h 12m 36s	0.09s
SSVD (ReVeal)	7h 26m 16s	0.08s

Table 6 shows the average training and testing times across four datasets. We observe that LineVul's training time is around 5 hours and 34 minutes, which increases to 14 hours and 28 minutes with SSVD. Similarly, ReVeal's training time extends from 2 hours and 12 minutes to 7 hours and 26 minutes when SSVD is applied. The extended training time with SSVD is mainly due to the certainty-aware sample selection module, which requires multiple rounds of inference on the unlabeled dataset to calculate information gain. However, SSVD only impacts the training phase and does not affect inference time. As shown in Table 6, inference on a single code snippet takes 0.21 seconds with LineVul and 0.23 seconds with SSVD (LineVul), and 0.09 seconds with ReVeal and 0.08 seconds with SSVD (Reveal). Thus, SSVD does not introduce any additional time costs during the testing phase, making it a practical option for efficient vulnerability detection once training is complete.

Aswer to RQ5:

SSVD increases the time cost during the training phase but does not affect the testing phase.

5 DISCUSSION

5.1 Why does SSVD work?

We identify three advantages of SSVD that can explain its effectiveness in vulnerability detection.

(1) The ability to select correctly pseudo-labeled code snippets as the training data. Figure 10 and Figure 11 present the F1-score of pseudo-labeled code snippets sampled and not sampled by SSVD and SSVD (prob), which samples data based on classification probability, across ten iterations of self-training, utilizing LineVul and ReVeal as the base detection models. The F1-score of the sampled pseudo-labeled code snippets by SSVD is consistently higher than those of the unsampled data, indicating that SSVD successfully identifies and selects correctly pseudo-labeled code snippets as training data. Conversely, SSVD (prob) fails to sample more correctly pseudo-labeled code snippets in numerous self-training iterations, resulting in a lower F1-score for the sampled data compared to the unsampled data. Moreover, the F1-score of pseudo-labeled code snippets sampled by SSVD is generally higher than those sampled by SSVD (prob) in most cases. This highlights the effectiveness of SSVD's certainty-aware sample selection module, which evaluates the certainty of pseudo-labels and makes accurate choices of correctly pseudo-labeled code snippets. However, SSVD (prob), which selects samples based on classification probability, is more prone to selecting incorrect pseudo-labeled code snippets, thus reducing the quality of the training dataset. By prioritizing high-certainty samples, SSVD improves the overall quality of the selected pseudo-labeled data, leading to better vulnerability detection performance.

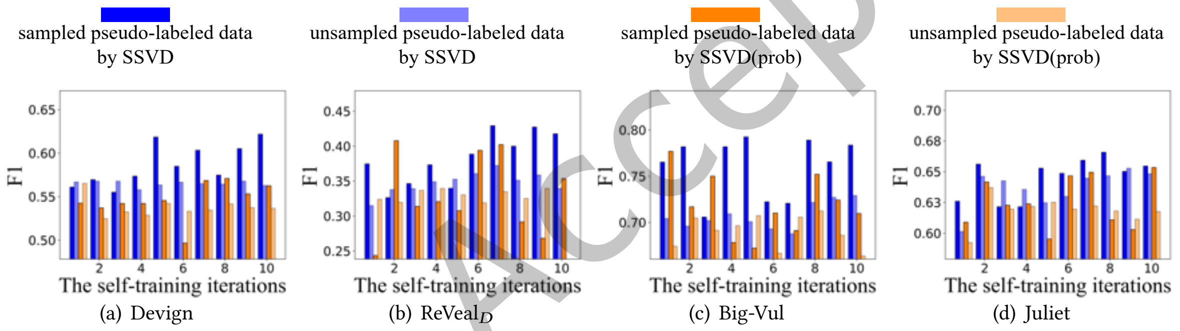


Fig. 10. The F1-score of the sampled pseudo-labeled data by SSVD, unsampled pseudo-labeled data by SSVD, sampled pseudo-labeled data by SSVD (prob), and unsampled pseudo-labeled data by SSVD (prob) in ten iterations of self-training using LineVul as base detection model.

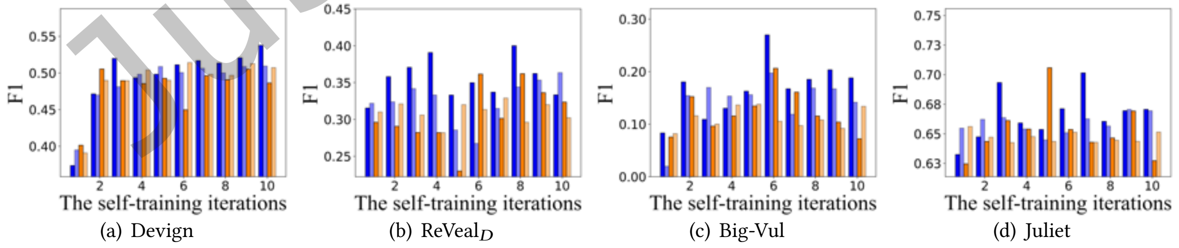


Fig. 11. The F1-score of the sampled pseudo-labeled data by SSVD, unsampled pseudo-labeled data by SSVD, sampled pseudo-labeled data by SSVD (prob), and unsampled pseudo-labeled data by SSVD (prob) in ten iterations of self-training using ReVeal as base detection model.

(2) The ability to maximize the separation between vulnerable and non-vulnerable code snippets, as well as to cluster code snippets of the same class in the feature space. Figures 12 and 13 display the original feature space generated by SSVD$_{w/oNRTL}$ and the new feature space generated by SSVD using LineVul and ReVeal as the detection models, respectively, visualized using the t-SNE plot [98]. In the original feature space, we observe a significant overlap between vulnerable code snippets (represented by red points) and non-vulnerable code snippets (represented by black points), making it challenging to establish clear boundaries between the two classes. However, when incorporating the noise-robust triplet loss module, we observe a noticeable separation between vulnerable and non-vulnerable code snippets in the new feature space. We calculate the inter-class distance and intra-class distance to quantify the degree of separation among different classes and the degree of clustering within the same class, respectively. For instance, when LineVul is employed as the base detection model on the Big-Vul dataset, NRTL increases the inter-class distance from 0.5871 to 0.6715 and decreases the intra-class distance from 0.2946 to 0.2645. This result indicates that SSVD effectively considers the smoothness assumption, leading to stronger clustering within the same class and pushing different classes further apart from each other.

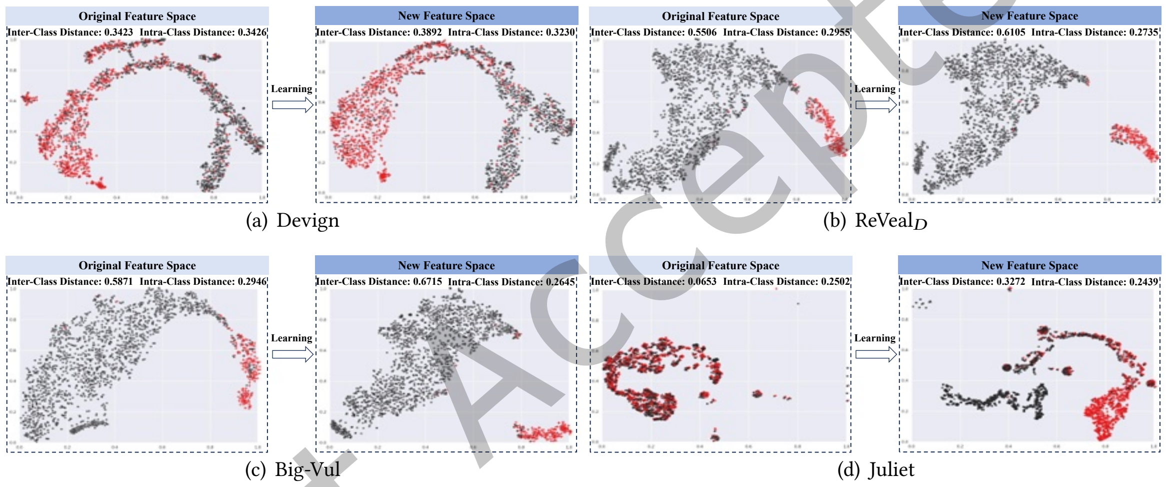
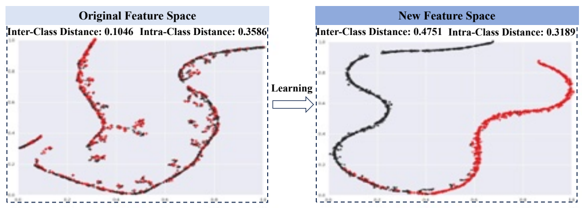


Fig. 12. The original and new feature spaces generated by SSVD$_{w/oNRTL}$ (LineVul) and SSVD (LineVul). t-SNE plots illustrating the separation between vulnerable (red points) and non-vulnerable (black points) code snippets.

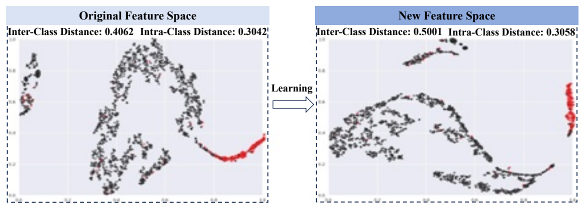
(3) The ability to alleviate accumulated errors. Figure 14 illustrates the variation trend of F1-scores for SSVD and SSVD$_{w/oNRCE}$ over the iterations of self-training across the four datasets. Initially, both SSVD and SSVD$_{w/oNRCE}$ exhibit similar performance levels, indicating comparable error rates at the beginning of the self-training process. However, as the iterations progress, the difference in F1-scores between SSVD and SSVD$_{w/oNRCE}$ gradually widens, ultimately resulting in SSVD achieving higher F1-scores in the final stages of self-training. The superior performance of SSVD compared to SSVD$_{w/oNRCE}$ is attributed to its utilization of the noise-robust cross-entropy loss function. By mitigating the adverse effects of incorrect pseudo-labels, SSVD prevents the model from memorizing erroneous information and accumulating errors over time. As a result, SSVD consistently outperforms SSVD$_{w/oNRCE}$ in the later stages of the self-training process.

5.2 Case Study

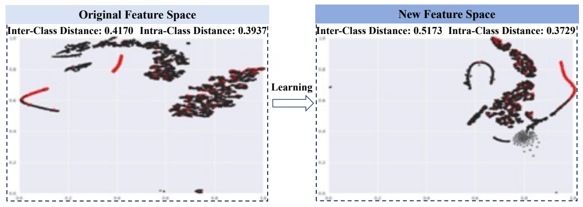
We use three examples shown in Figure 15 and Figure 16 to further demonstrate the effectiveness of SSVD.



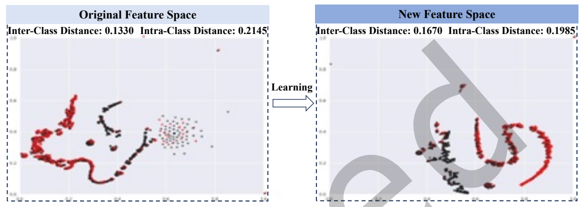
(a) Devign



(b) ReVeal$_D$

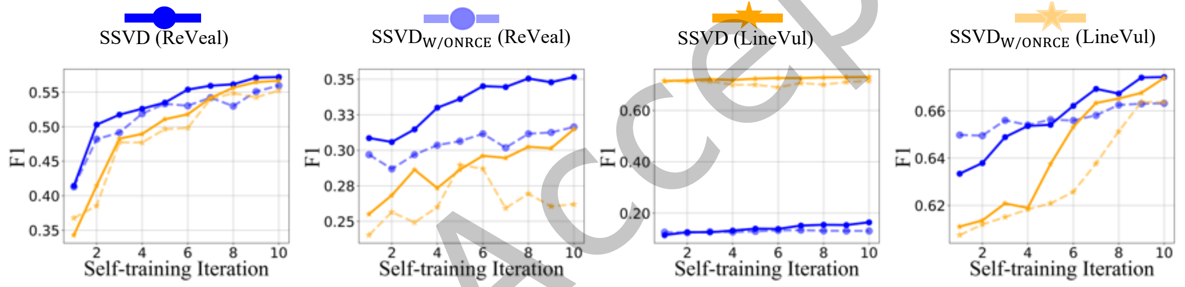


(c) Big-Vul



(d) Juliet

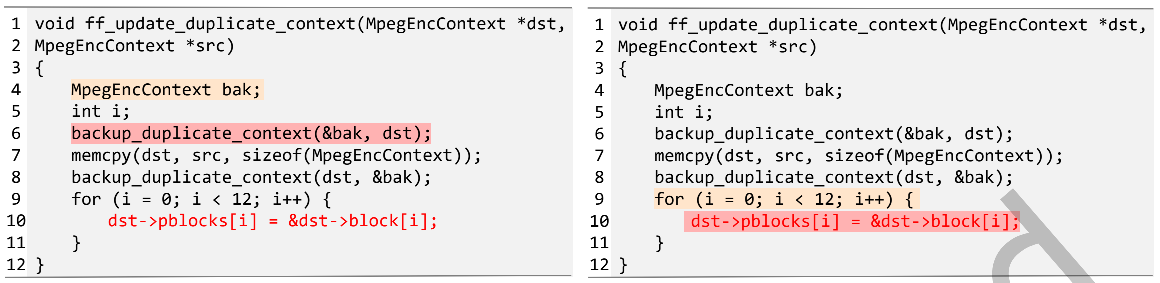
Fig. 13. The original and new feature spaces generated by SSVD$_{w/oNRTL}$ (ReVeal) and SSVD (ReVeal). t-SNE plots illustrating the separation between vulnerable (red points) and non-vulnerable (black points) code snippets.



(a) Devign (b) ReVeal$_D$ (c) Big-Vul (d) Juliet

Fig. 14. The F1-score over self-training iterations. The color blue represents the SSVD (ReVeal) model, while the light blue color indicates the SSVD$_{w/oNRCE}$ (ReVeal) variant with class balanced term. The orange color represents the SSVD (LineVul) model, and the light orange color represents the SSVD$_{w/oNRCE}$ (LineVul) variant with class balanced term.

Figure 15 presents a vulnerable code snippet exhibiting an *out-of-bounds read* vulnerability. In this code, at line 9, the integer *12* is used as the condition for terminating the *for* loop without validating that the integer accurately reflects the length of *pblocks* and *block* at line 10. This can result in an *out-of-bounds read* vulnerability by reading from memory beyond the bounds of the buffer, if the integer *12* indicates a length that is longer than the size of *pblocks* and *block*. To understand what features are utilized by the vulnerability detection model for making predictions, following [7], we employ Lemna [31] to assess the importance of features assigned to the predicted code. Lemna assigns a value to each code token in the input, representing its contribution to the prediction. A higher value indicates a larger impact of the code token on the prediction, while a lower value suggests less contribution. Figure 15(a) and 15(b) illustrate the contribution of different code components in the prediction results by LineVul trained on 10% labeled data and SSVD using the LineVul base model, respectively. When trained with only 10% labeled data, LineVul overlooks features related to the actual vulnerability that appears in line 10, leading to an incorrect classification as non-vulnerable. However, after training with the remaining 90% unlabeled data using SSVD, the detection model can prioritize features related to vulnerabilities and correctly classify the code snippet as vulnerable. This demonstrates that SSVD is capable of leveraging unlabeled data to enhance the model's detection ability by focusing on code components related to vulnerabilities.



```c
void ff_update_duplicate_context(MpegEncContext *dst,
MpegEncContext *src)
{
    MpegEncContext bak;
    int i;
    backup_duplicate_context(&bak, dst);
    memcpy(dst, src, sizeof(MpegEncContext));
    backup_duplicate_context(dst, &bak);
    for (i = 0; i < 12; i++) {
        dst->pblocks[i] = &dst->block[i];
    }
}
```

(a) The code snippet that is incorrectly predicted as non-vulnerable by the LineVul model trained only on 10% labeled data.

```c
void ff_update_duplicate_context(MpegEncContext *dst,
MpegEncContext *src)
{
    MpegEncContext bak;
    int i;
    backup_duplicate_context(&bak, dst);
    memcpy(dst, src, sizeof(MpegEncContext));
    backup_duplicate_context(dst, &bak);
    for (i = 0; i < 12; i++) {
        dst->pblocks[i] = &dst->block[i];
    }
}
```

(b) The code snippet that is correctly predicted as vulnerable by the SSVD approach using the LineVul as the base model.

Fig. 15. The contribution of different code components in the prediction results by LineVul trained on 10% labeled data and SSVD using LineVul as the base model, respectively. Red-shaded code elements are the most contributing, while yellow-shaded elements are the second most contributing. Red-colored code represents the source of vulnerabilities.

We select two cases, as shown in Figure 16(a) and Figure 16(b), to visually demonstrate the superiority of our certainty-aware sample selection method over the classification probability-based approach. Figure 16(a) displays an unlabeled code snippet from the Devign dataset, whose actual label is vulnerable. The code snippet utilizes the pointers *chr* and *s*. If the *opaque* pointer is *NULL*, then *NULL* is assigned to the *chr*. Subsequently, accessing *chr* in the line 9 may result in a *null pointer dereference*. Likewise, if the *opaque* member pointed to by the *chr* pointer is *NULL*, accessing *s* in the line 5 may also lead to a *null pointer dereference*. The SSVD approach predicts this code snippet to be vulnerable and assigns it the vulnerable pseudo-label accordingly. Additionally, SSVD outputs the classification probability and the certainty of the pseudo-label. This code snippet ranks 28th in certainty among 17,633 unlabeled code snippets in the Devign dataset, while its classification probability ranks at 17,625. Figure 16(b) showcases an unlabeled code snippet from the Devign dataset, whose actual label is non-vulnerable. The SSVD approach predicts this code snippet to be non-vulnerable and assigns the non-vulnerable pseudo-label accordingly. It ranks 165th in certainty among unlabeled code snippets, while its classification probability ranks at 16,632. The two examples illustrate that even correctly pseudo-labeled code snippets can have lower classification probabilities. Therefore, if we were to choose code snippets with high classification probability as training data, as done in previous software engineering research [41, 67, 105, 113], these two correctly pseudo-labeled code snippets would not be selected. However, utilizing certainty-aware sample selection in the SSVD approach would include them due to their higher certainty, resulting in a higher-quality pseudo-labeled dataset.

5.3 The Impact of Initial Training Samples

In this section, we explore the impact of initial training samples on SSVD's performance, as the initial training samples influence the initial teacher model, thereby affecting SSVD's effectiveness. As shown in Figure 17 and Figure 18, we keep the test and validation sets consistent and divide the training data into ten equal parts. For each iteration, one part is used as the labeled dataset, while the other nine serve as the unlabeled dataset. We repeat this process ten times, selecting different labeled data. We can observe that using different initial training data affects SSVD's performance on the test set, causing the model's performance to fluctuate within a certain range. However, this fluctuation is not substantial. For example, when LineVul is used as the base model on the ReVeal$_D$ dataset, the F1-score fluctuates between 0.33 and 0.35, and MCC between 0.26 and 0.28. Similarly, with

Certainty ranking: 28/17633
Classification probability ranking: 17625/17633

```
static gboolean pty_chr_timer(gpointer opaque)
{
    struct CharDriverState *chr = opaque;
    PtyCharDriver *s = chr->opaque;
    if (s->connected) {
        goto out;
    }
    pty_chr_update_read_handler(chr);
out:
    s->timer_tag = 0;
    return FALSE;
}
```

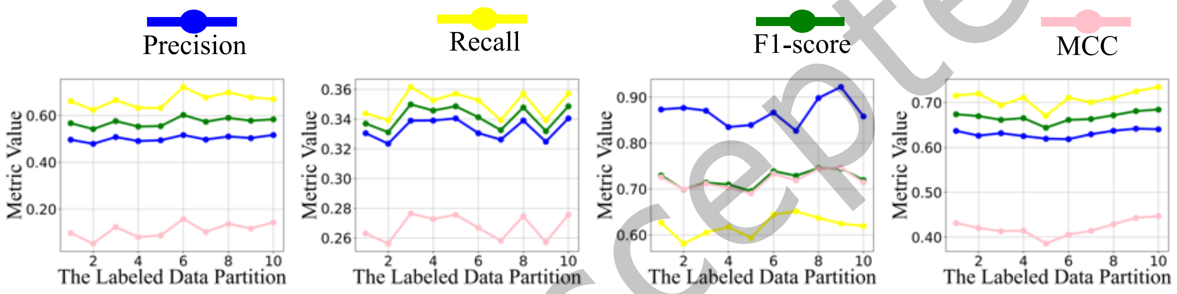
(a) The unlabeled code snippet with an actual vulnerable label.

Certainty ranking: 165/17633
Classification probability ranking: 16632/17633

```
void virtio_queue_notify(VirtIODevice *vdev, int n)
{
  if (n < VIRTIO_PCI_QUEUE_MAX && vdev->vq[n].vring.desc){
      trace_virtio_queue_notify(vdev, n, &vdev->vq[n]);
      vdev->vq[n].handle_output(vdev, &vdev->vq[n]);
  }
}
```

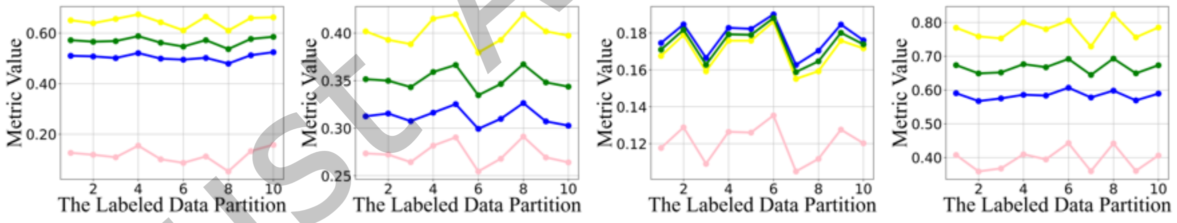
(b) The unlabeled code snippet with an actual non-vulnerable label.

Fig. 16. The unlabeled code snippets from the Devign dataset, which have been assigned correct pseudo-labels.



(a) Devign (b) ReVeal$_D$ (c) Big-Vul (d) Juliet

Fig. 17. The metric values with LineVul as the base model across four datasets using different initial training data.



(a) Devign (b) ReVeal$_D$ (c) Big-Vul (d) Juliet

Fig. 18. The metric values with ReVeal as the base model across four datasets using different initial training data.

ReVeal as the base model on the Big-Vul dataset, the F1-score varies between 0.16 and 0.19, and MCC between 0.11 and 0.14. This suggests that SSVD does not experience substantial performance improvement or degradation due to the choice of initial data. The underlying reason could be that, although different labeled data are used in the initial iteration, SSVD ultimately trains based on the same data (i.e., labeled data plus unlabeled data), so the knowledge learned by the model is based on the same data distribution, which helps reduce performance fluctuations. Additionally, the validation set remains consistent during training, and SSVD only retains the models that show improvement on the validation set, helping to gradually correct the initial teacher model's biases during self-training iterations, while also reducing the impact of initial training data selection on SSVD.

5.4 Threats of Validity

We utilize four vulnerability datasets containing C/C++ code snippets, while not including datasets from other programming languages. Our main focus is on detecting the presence of vulnerabilities in code snippets, similar to the majority of vulnerability detection studies, without explicitly distinguishing between different types of vulnerabilities. In the future, we plan to experiment with more additional programming language datasets to further evaluate the effectiveness of SSVD and explore multiple types of vulnerability detection tasks. In real-world scenarios, the proportion of vulnerable code is relatively low. However, we have selected the Devign and Juliet datasets, which have higher proportions of vulnerable code. The reason is that these datasets are widely used public benchmarks in the field of vulnerability detection [7, 24, 48, 55, 104], thus facilitating comparisons between our work and other studies utilizing the same datasets. In addition, we have also tested our SSVD approach on ReVeal$_D$ and Big-Vul, which have lower proportions of vulnerable code. SSVD demonstrates strong performance on these datasets as well.

Another threat of validity is that we are unable to explore the impact of all hyperparameters due to hardware limitations. To mitigate this, we specifically examine the impact of two crucial hyperparameters: γ and k. γ determines the weight assigned to the intra-class constraint term in the noise-robust triplet loss, while k determines the proportion of code snippets involved in gradient clipping within the noise-robust cross-entropy loss. As for the hyperparameters T and m, we opt for the best-performing values reported in their respective papers. Regarding the weights of L_{NRTL} and L_{NRCE}, we follow the practices outlined in Xu et al.'s study [112] in software engineering and several studies [54, 121, 124] in artificial intelligence. These studies use both cross-entropy loss and triplet loss, assigning them equal weights. Therefore, we adopt the same setting in our work to reduce the complexity of hyperparameter tuning. In the absence of experimental evidence favoring one loss function over the other, equal weights provide a neutral starting point, ensuring that both loss functions contribute equally during training. In terms of general parameters seen in deep learning models, such as learning rate and batch size, we maintain consistency with the LineVul and ReVeal models. We also take great care to replicate the baselines to ensure reasonable experimental settings and align them as closely as possible with the descriptions provided in the relevant papers and replication packages. Moreover, we ensure that the general parameters remain consistent between our method and the baselines to maintain experimental fairness.

6 RELATED WORK

6.1 Deep Learning-based Vulnerability Detection

Existing methods can be categorized into two types: token-based and graph-based methods. The former treats code as a sequence of tokens and leverages different neural networks to detect vulnerabilities [11, 24, 50, 81]. For example, Russell et al. [81] utilized Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) to capture vulnerability features from code token sequences. VulDeePecker [50] and SySeVR [49] first extracted code slices from specific "interesting points" in the code (e.g., API calls, array indexing, and pointer usage), since not every line of code holds equal importance for vulnerability detection. By focusing only on these code slices, the models utilized LSTM and GRU to enhance vulnerability detection performance without considering the remaining code. Furthermore, Fu et al. [24] proposed LineVul, a Transformer-based line-level vulnerability detection approach, which can capture the long-dependency relationship of code token sequences.

Graph-based methods [2, 7, 18, 27, 48, 49, 58, 86, 100, 104, 106, 107, 123, 125] first transform code snippets into various graphs and then employ graph neural networks to learn the structural features of the code for vulnerability detection. For example, Devign [132], Reveal [7], and IVDetect [48] employed gated graph recurrent network, graph neural network, or graph convolution network to capture structural features from abstract syntax tree, control flow graph, data flow graph, code property graph, or program dependency graph. Wu et al. [107] transformed the program dependence graph of code snippets into images to improve both the scalability and accuracy of vulnerability detection. Cao et al. [6] employed a flow-sensitive graph neural network to jointly learn semantic and structural information of the code to detect statement-level memory-related vulnerability. Wang et al. [101] proposed a graph-based neural network vulnerability detection model that focused on extracting class-separation features between vulnerable and non-vulnerable code. Wen et al. [103] proposed graph simplification and enhanced graph representation learning to effectively capture global information of code. Zhang et al. [125] utilized the CodeBERT and convolutional neural network to learn path representations from syntax-based control flow graph. Wang et al. [100] proposed to use graph embedding and bidirectional gated convolutional neural network to effectively addresses the issue of limited representation of nonlinear information within the code graph structure. Wen et al. [104] proposed a meta-path based attentional graph learning model, which constructed a multi-granularity meta-path graph for each code snippet.

However, these deep learning-based methods face challenges due to the requirement of a large amount of accurately labeled data [32, 51, 65, 81, 122]. Notably, Zhang et al. [123] proposed a cross-domain vulnerability detection method, CPVD, to address the issue of limited labeled code snippets in new projects. However, CPVD still necessitated lots of labeled code snippets from other projects for effective cross-domain training. In addition, some recent studies [13, 55, 64, 89, 99, 131] have investigated the application of large language models, like ChatGPT, for vulnerability detection. Some investigations [13, 89] have indicated that ChatGPT's vulnerability detection capabilities are not yet optimal. Another crucial aspect to consider is that sharing code with public large language models like ChatGPT for vulnerability detection can pose security risks of code leakage, as developers are required to upload their code to ChatGPT for analysis. However, the SSVD approach enables developers to train their local models independently. Therefore, we recommend using SSVD to construct one's own vulnerability detection model after manually annotating a small amount of code snippets, thus mitigating the risk of code leakage.

6.2 Semi-Supervised Learning for Software Engineering

The semi-supervised learning paradigm is associated with constructing models that use both labeled and unlabeled data [114]. Currently, the mainstream methods for deep semi-supervised learning can be broadly classified into deep generative methods, consistency regularization methods, graph-based methods, and pseudo-labeling methods. Deep generative methods [15, 19, 60, 87, 102] typically use generative models, such as Variational Auto-Encoders (VAEs) [37] and Generative Adversarial Networks (GANs) [29], to explore the distribution of the training dataset and generate new training samples. Consistency regularization methods [40, 77, 91, 126] are based on the smoothness assumption that small perturbations to input data should not alter its class. These methods design a consistency regularization term that requires the model to maintain consistent output results when small perturbations are applied to the input data. Graph-based methods [5, 38, 71] construct a similarity graph where nodes represent training data and edges represent the similarity between data points. This graph structure is used to propagate label information through the connections. Pseudo-labeling methods [42, 73, 109], also known as self-training, are based on the idea of using the model's own predictions to extend the labeled dataset, thereby enhancing the model's performance.

Some studies have explored applying semi-supervised learning in software engineering. There are three studies (Shar et al. [84], Meng et al. [65], and Yu et al. [120]) applied semi-supervised machine learning techniques to software vulnerability detection. They utilized co-training [3], label propagation [76], and active learning [75] techniques with manually extracted vulnerability features. Compared to these studies, our semi-supervised deep learning-based SSVD method can automatically discover intricate patterns and vulnerability features within the data, leading to improved performance. In recent work, Wen et al. [105] proposed a semi-supervised deep learning vulnerability detection method called PILOT. It utilizes the positive and unlabeled learning technique [128], which assumes that the training set only contains positive (vulnerable) code snippets and unlabeled code snippets.

Initially, PILOT identifies unlabeled samples with the maximum distance difference from all vulnerable code snippets as non-vulnerable. The initial detection model is then trained using both actually vulnerable code snippets and pseudo-labeled non-vulnerable code snippets. This model is employed to classify the unlabeled samples. The samples with high classification probabilities are selected as pseudo-labeled data and incorporated into the training set for further fine-tuning of the model. However, there are several potential issues with this method. First, the chosen distance metric in PILOT may not accurately capture the similarity or dissimilarity between code snippets, largely due to the class overlap problem, where vulnerable and non-vulnerable code snippets may share similar characteristics or have overlapping features. Consequently, PILOT may mislabel actually vulnerable code snippets as non-vulnerable. Second, in practical scenarios, it is typically possible to have access to both vulnerable and non-vulnerable code snippets. It is recommended to include actual non-vulnerable code snippets in the training set instead of relying on pseudo-labeled non-vulnerable code snippets as done in PILOT. The quality of pseudo non-vulnerable labels assigned to unlabeled code snippets based solely on distance may not be as reliable as manual labeling. Third, the selection of samples based on high classification probabilities may not guarantee the correctness of those pseudo-labels.

Besides software vulnerability detection, researchers also applied semi-supervised learning to other areas related to software vulnerability. For instance, Sun et al. [90] employed active learning to detect vulnerabilities in smart contracts. Riom et al. [80] and Sawadogo et al. [82] introduced the co-training method to automatically detect if an incoming commit will introduce vulnerabilities and automatically identify whether source code changes serve as security patches. In addition, semi-supervised learning algorithms have also been successfully applied in various other areas of software engineering over the years, such as defect prediction [36, 46, 57, 63, 97, 118, 129], log anomaly detection [41, 52, 59, 113, 127], technical debt detection [74, 95, 119], tag recommendation [9], malware classification [28, 45, 62], API recognition [117], learning-based traceability [16], test case generation [53], static code warning recognizer [94, 96], issue closed time predictor [94, 96], and license incompatibility detection [110]. These studies have leveraged various semi-supervised learning techniques to achieve advancements in their respective domains, including self-training [41, 57, 74, 110], co-training [63], tri-training [46], label propagation [129], active learning [90, 94, 95, 97, 119], and positive unlabeled learning [105, 113]. However, these semi-supervised methods tend to choose pseudo-labeled samples with higher classification probability as training data and do not fully consider the smoothness assumption.

Although there are many semi-supervised learning methods, most are unsuitable for vulnerability detection. Deep generative methods struggle to generate syntactically correct vulnerable and non-vulnerable samples from limited labeled data. Consistency regularization methods require perturbations that do not alter the class of code snippets, but minor changes can inadvertently introduce or remove vulnerabilities. Graph-based methods require constructing a graph on the training data, which involves high storage and computational costs, making them difficult to apply to large-scale vulnerability datasets. Therefore, SSVD is based on self-training, which is suitable for vulnerability detection tasks and has demonstrated feasibility in other areas of software engineering.

7 CONCLUSION AND FUTURE WORK

This paper proposes SSVD, a novel semi-supervised vulnerability detection method that leverages a small amount of labeled code snippets and a large volume of unlabeled code snippets to enhance detection performance. SSVD evaluates the certainty of pseudo labels using the information gain of model parameters, prioritizing high-certainty pseudo-labeled code snippets for training. Additionally, it addresses the smoothness assumption and the problem of incorrect pseudo-label occurrences in the student model's training data by integrating the proposed noise-robust triplet loss function and noise-robust cross-entropy loss function. These components aim to maximize the separation between vulnerable and non-vulnerable code snippets while mitigating the accumulation of errors caused by incorrect pseudo-labels. Experimental results on four vulnerability datasets demonstrate the effectiveness of SSVD. In future work, we plan to consistently collect more datasets spanning a wider array of projects across different programming languages to further validate the effectiveness of our proposed SSVD approach. Additionally, we will extend SSVD to other software engineering tasks, such as log anomaly detection, technical debt detection, and defect prediction, which also face challenges with labeling data.

8 ACKNOWLEDGMENTS

This research was supported by the advanced computing resources provided by the Supercomputing Center of Hangzhou City University, Ningbo Natural Science Foundation (No. 2023J292), and the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong (6000796, 9229109, 9229098, 9220103, 9229029).

REFERENCES

[1] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U Rajendra Acharya, et al. 2021. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion* 76 (2021), 243–297.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[3] Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*. 92–100.

[4] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[5] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep neural networks for learning graph representations. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[6] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, Xiaoxue Wu, Chuanqi Tao, Tao Zhang, and Wei Liu. 2023. Learning to Detect Memory-related Vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–35.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296.

[8] Baixu Chen, Junguang Jiang, Ximei Wang, Pengfei Wan, Jianmin Wang, and Mingsheng Long. 2022. Debiased self-training for semi-supervised learning. *Advances in Neural Information Processing Systems* 35 (2022), 32424–32437.

[9] Wei Chen, Jia-Hong Zhou, Jia-Xin Zhu, Guo-Quan Wu, and Jun Wei. 2019. Semi-supervised learning based tag recommendation for docker repositories. *Journal of Computer Science and Technology* 34 (2019), 957–971.

[10] Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, and Chao Ni. 2019. Software defect number prediction: Unsupervised vs supervised methods. *Information and Software Technology* 106 (2019), 161–181.

[11] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.

[12] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 519–531.

[13] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232* (2023).

[14] Roland Croft, M Ali Babar, and Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. *arXiv preprint arXiv:2301.05456* (2023).

[15] Zihang Dai, Zhilin Yang, Fan Yang, William W Cohen, and Russ R Salakhutdinov. 2017. Good semi-supervised learning that requires a bad gan. *Advances in neural information processing systems* 30 (2017).

[16] Liming Dong, He Zhang, Wei Liu, Zhiluo Weng, and Hongyu Kuang. 2022. Semi-supervised pre-processing for learning-based traceability framework on real-world software projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 570–582.

[17] Guodong Du, Jia Zhang, Min Jiang, Jinyi Long, Yaojin Lin, Shaozi Li, and Kay Chen Tan. 2021. Graph-based class-imbalance learning with label enhancement. *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[18] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities.. In *IJCAI*. 4665–4671.

[19] M Ehsan Abbasnejad, Anthony Dick, and Anton van den Hengel. 2017. Infinite variational autoencoder for semi-supervised learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5888–5897.

[20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[21] Shuo Feng, Jacky Keung, Xiao Yu, Yan Xiao, Kwabena Ebo Bennin, Md Alamgir Kabir, and Miao Zhang. 2021. COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction. *Information and Software Technology* 129 (2021), 106432.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[23] JA Ferreira and AH Zwinderman. 2006. On the benjamini–hochberg method. (2006).

[24] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[25] Yarin Gal and Zoubin Ghahramani. 2015. Bayesian convolutional neural networks with Bernoulli approximate variational inference. *arXiv preprint arXiv:1506.02158* (2015).

[26] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. PMLR, 1050–1059.

[27] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code Structure–Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[28] Xianwei Gao, Changzhen Hu, Chun Shan, Baoxu Liu, Zequn Niu, and Hui Xie. 2020. Malware classification for the cloud via semi-supervised transfer learning. *Journal of Information Security and Applications* 55 (2020), 102661.

[29] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems* 27 (2014).

[30] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International conference on machine learning*. PMLR, 1321–1330.

[31] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 364–379.

[32] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* 179 (2021), 103009.

[33] Tao He, Xiaoming Jin, Guiguang Ding, Lan Yi, and Chenggang Yan. 2019. Towards better uncertainty sampling: Active learning with multiple views for deep convolutional neural network. In *2019 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 1360–1365.

[34] Neil Houlsby, Ferenc Huszár, Zoubin Ghahramani, and Máté Lengyel. 2011. Bayesian active learning for classification and preference learning. *arXiv preprint arXiv:1112.5745* (2011).

[35] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* 24, 5 (2019), 2823–2862.

[36] Xiao-Yuan Jing, Fei Wu, Xiwei Dong, and Baowen Xu. 2016. An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering* 43, 4 (2016), 321–339.

[37] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[38] Thomas N Kipf and Max Welling. 2016. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* (2016).

[39] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[40] Samuli Laine and Timo Aila. 2016. Temporal ensembling for semi-supervised learning. *arXiv preprint arXiv:1610.02242* (2016).

[41] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, Yongqiang Yang, and Michael R. Lyu. 2023. Heterogeneous Anomaly Detection for Software Systems via Semi-supervised Cross-modal Attention. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1724–1736.

[42] Dong-Hyun Lee et al. 2013. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, ICML*, Vol. 3. Atlanta, 896.

[43] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. 2012. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering* 19 (2012), 201–230.

[44] Ming Li and Zhi-Hua Zhou. 2007. Improve computer-aided diagnosis with machine learning techniques using undiagnosed samples. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 37, 6 (2007), 1088–1098.

[45] Qian Li, Qingyuan Hu, Yong Qi, Saiyu Qi, Xinxing Liu, and Pengfei Gao. 2021. Semi-supervised two-phase familial analysis of Android malware with normalized graph embedding. *Knowledge-Based Systems* 218 (2021), 106802.

[46] Weiwei Li, Wenzhou Zhang, Xiuyi Jia, and Zhiqiu Huang. 2020. Effort-aware semi-supervised just-in-time defect prediction. *Information and Software Technology* 126 (2020), 106364.

[47] Yanhui Li, Linghan Meng, Lin Chen, Li Yu, Di Wu, Yuming Zhou, and Baowen Xu. 2022. Training data debugging for the fairness of machine learning software. In *Proceedings of the 44th International Conference on Software Engineering*. 2215–2227.

[48] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. 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*. 292–303.

[49] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[50] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[51] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: a survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.

[52] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 102–111.

[53] Xiao Ling and Tim Menzies. 2023. On the Benefits of Semi-Supervised Test Case Generation for Cyber-Physical Systems. *arXiv preprint arXiv:2305.03714* (2023).

[54] Haijun Liu, Xiaoheng Tan, and Xichuan Zhou. 2020. Parameter sharing exploration and hetero-center triplet loss for visible-thermal person re-identification. *IEEE Transactions on Multimedia* 23 (2020), 4414–4425.

[55] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* (2024), 112031.

[56] Huihua Lu, Bojan Cukic, and Mark Culp. 2011. An iterative semi-supervised approach to software fault prediction. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. 1–10.

[57] Huihua Lu, Bojan Cukic, and Mark Culp. 2012. Software defect prediction using semi-supervised learning with dimension reduction. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 314–317.

[58] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. Graphcode2vec: Generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 524–536.

[59] Xiaoxue Ma, Jacky Keung, Pinjia He, Yan Xiao, Xiao Yu, and Yishu Li. 2023. A Semi-supervised Approach for Industrial Anomaly Detection via Self-adaptive Clustering. *IEEE Transactions on Industrial Informatics* (2023).

[60] Lars Maaløe, Casper Kaae Sønderby, Søren Kaae Sønderby, and Ole Winther. 2016. Auxiliary deep generative models. In *International conference on machine learning*. PMLR, 1445–1453.

[61] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[62] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A Ghorbani. 2020. Dynamic android malware category classification using semi-supervised deep learning. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 515–522.

[63] Suvodeep Majumder, Joymallya Chakraborty, and Tim Menzies. 2022. When Less is More: On the Value of" Co-training" for Semi-Supervised Software Defect Predictors. *arXiv preprint arXiv:2211.05920* (2022).

[64] Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Mei Nagappan, and Shane McIntosh. 2024. LLbezpeky: Leveraging Large Language Models for Vulnerability Detection. *arXiv preprint arXiv:2401.01269* (2024).

[65] Qingkun Meng, Shameng Wen, Chao Feng, and Chaojing Tang. 2016. Predicting buffer overflow using semi-supervised learning. In *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. IEEE, 1959–1963.

[66] Subhabrata Mukherjee and Ahmed Awadallah. 2020. Uncertainty-aware self-training for few-shot text classification. *Advances in Neural Information Processing Systems* 33 (2020), 21199–21212.

[67] Pradeep K Murukannaiah and Munindar P Singh. 2015. Platys: An active learning framework for place-aware application development and its evaluation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–32.

[68] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 427–436.

[69] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. 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*. 672–683.

[70] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In *IEEE/ACM 45th International Conference on Software Engineering(ICSE) https://www. software-lab. org/publications/icse2023_VulGen. pdf*.

[71] Shirui Pan, Ruiqi Hu, Sai-fu Fung, Guodong Long, Jing Jiang, and Chengqi Zhang. 2019. Learning graph embedding with adversarial training methods. *IEEE transactions on cybernetics* 50, 6 (2019), 2475–2487.

[72] Tim Pearce, Alexandra Brintrup, and Jun Zhu. 2021. Understanding softmax confidence and uncertainty. *arXiv preprint arXiv:2106.04972* (2021).

[73] Hieu Pham, Zihang Dai, Qizhe Xie, and Quoc V Le. 2021. Meta pseudo labels. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 11557–11568.

[74] Julian Aron Prenner and Romain Robbes. 2021. Making the most of small Software Engineering datasets with modern machine learning. *IEEE Transactions on Software Engineering* 48, 12 (2021), 5050–5067.

[75] Michael Prince. 2004. Does active learning work? A review of the research. *Journal of engineering education* 93, 3 (2004), 223–231.

[76] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[77] Antti Rasmus, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko. 2015. Semi-supervised learning with ladder networks. *Advances in neural information processing systems* 28 (2015).

[78] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. *ACM computing surveys (CSUR)* 54, 9 (2021), 1–40.

[79] Denise Rey and Markus Neuhäuser. 2011. *Wilcoxon-Signed-Rank Test*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1658–1659. https://doi.org/10.1007/978-3-642-04898-2_616

[80] Timothé Riom, Arthur Sawadogo, Kevin Allix, Tegawendé F Bissyandé, Naouel Moha, and Jacques Klein. 2021. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. *Empirical Software Engineering* 26 (2021), 1–30.

[81] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[82] Arthur D Sawadogo, Tegawendé F Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. 2022. SSPCatcher: Learning to catch security patches. *Empirical Software Engineering* 27, 6 (2022), 151.

[83] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[84] Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. 2014. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on dependable and secure computing* 12, 6 (2014), 688–707.

[85] Hwanjun Song, Minseok Kim, Dongmin Park, Yooju Shin, and Jae-Gil Lee. 2022. Learning from noisy labels with deep neural networks: A survey. *IEEE Transactions on neural networks and learning systems* (2022).

[86] Zihua Song, Junfeng Wang, Shengli Liu, Zhiyang Fang, Kaiyuan Yang, et al. 2022. HGVul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. *Security and Communication Networks* 2022 (2022).

[87] Jost Tobias Springenberg. 2015. Unsupervised and semi-supervised learning with categorical generative adversarial networks. *arXiv preprint arXiv:1511.06390* (2015).

[88] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[89] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. *arXiv preprint arXiv:2403.17218* (2024).

[90] Xiaobing Sun, Liangqiong Tu, Jiale Zhang, Jie Cai, Bin Li, and Yu Wang. 2023. ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. *Journal of Information Security and Applications* 73 (2023), 103423.

[91] Antti Tarvainen and Harri Valpola. 2017. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. *Advances in neural information processing systems* 30 (2017).

[92] Haonan Tong, Dalin Zhang, Jiqiang Liu, Weiwei Xing, Lingyun Lu, Wei Lu, and Yumei Wu. 2024. MASTER: Multi-Source Transfer Weighted Ensemble Learning for Multiple Sources Cross-Project Defect Prediction. *IEEE Transactions on Software Engineering* (2024).

[93] Austin Cheng-Yun Tsai, Sheng-Ya Lin, and Li-Chen Fu. 2022. Contrast-Enhanced Semi-supervised Text Classification with Few Labels. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11394–11402.

[94] Huy Tu and Tim Menzies. 2021. FRUGAL: unlocking semi-supervised learning for software analytics. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 394–406.

[95] Huy Tu and Tim Menzies. 2022. DebtFree: minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning. *Empirical Software Engineering* 27, 4 (2022), 80.

[96] Huy Tu and Tim Menzies. 2023. Less, but Stronger: On the Value of Strong Heuristics in Semi-supervised Learning for Software Analytics. *arXiv preprint arXiv:2302.01997* (2023).

[97] Huy Tu, Zhe Yu, and Tim Menzies. 2020. Better data labelling with emblem (and how that impacts defect prediction). *IEEE Transactions on Software Engineering* 48, 1 (2020), 278–294.

[98] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[99] Jin Wang, Zishan Huang, Hengli Liu, Nianyi Yang, and Yinhao Xiao. 2023. Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism. *arXiv preprint arXiv:2309.15324* (2023).

[100] Sixuan Wang, Chen Huang, Dongjin Yu, and Xin Chen. 2023. VulGraB: Graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. *Software: Practice and Experience* (2023).

[101] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2249–2261.

[102] Xiang Wei, Boqing Gong, Zixia Liu, Wei Lu, and Liqiang Wang. 2018. Improving the improved training of wasserstein gans: A consistency term and its dual effect. *arXiv preprint arXiv:1803.01541* (2018).

[103] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. 2023. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. (2023), 2275–2286.

[104] Xin-Cheng Wen, Cuiyun Gao, Jiaxin Ye, Yichen Li, Zhihong Tian, Yan Jia, and Xuan Wang. 2024. Meta-path based attentional graph learning model for vulnerability detection. *IEEE Transactions on Software Engineering* (2024).

[105] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When Less is Enough: Positive and Unlabeled Learning Model for Vulnerability Detection. (2023), 345–357.

[106] Tongshuai Wu, Liwei Chen, Gewangzi Du, Dan Meng, and Gang Shi. 2024. UltraVCS: Ultra-fine-grained Variable-based Code Slicing for Automated Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* (2024).

[107] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*. 2365–2376.

[108] Qizhe Xie, Zihang Dai, Eduard Hovy, Thang Luong, and Quoc Le. 2020. Unsupervised data augmentation for consistency training. *Advances in neural information processing systems* 33 (2020), 6256–6268.

[109] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. 2020. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10687–10698.

[110] Sihan Xu, Ya Gao, Lingling Fan, Zheli Liu, Yang Liu, and Hua Ji. 2023. Lidetector: License incompatibility detection for open source software. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–28.

[111] Yi Xu, Lei Shang, Jinxing Ye, Qi Qian, Yu-Feng Li, Baigui Sun, Hao Li, and Rong Jin. 2021. Dash: Semi-supervised learning with dynamic thresholding. In *International Conference on Machine Learning*. PMLR, 11525–11536.

[112] Zhou Xu, Shuai Li, Jun Xu, Jin Liu, Xiapu Luo, Yifeng Zhang, Tao Zhang, Jacky Keung, and Yutian Tang. 2019. LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software* 158 (2019), 110402.

[113] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1448–1460.

[114] Xiangli Yang, Zixing Song, Irwin King, and Zenglin Xu. 2022. A survey on deep semi-supervised learning. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[115] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2287–2298.

[116] Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. 2023. On the significance of category prediction for code-comment synchronization. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–41.

[117] Deheng Ye, Lingfeng Bao, Zhenchang Xing, and Shang-Wei Lin. 2018. APIReal: an API recognition and linking approach for online developer forums. *Empirical Software Engineering* 23 (2018), 3129–3160.

[118] Xiao Yu, Man Wu, Yiheng Jian, Kwabena Ebo Bennin, Mandi Fu, and Chuanxiang Ma. 2018. Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning. *Soft Computing* 22 (2018), 3461–3472.

[119] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. 2020. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1676–1691.

[120] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. 2019. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2401–2420.

[121] Ye Yuan, Wuyang Chen, Yang Yang, and Zhangyang Wang. 2020. In defense of the triplet loss again: Learning robust person re-identification with fast approximated triplet loss and label distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 354–355.

[122] Peng Zeng, Guanjun Lin, Lei Pan, Yonghang Tai, and Jun Zhang. 2020. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access* 8 (2020), 197158–197172.

[123] Chunyong Zhang, Bin Liu, Yang Xin, and Liangwei Yao. 2023. CPVD: Cross Project Vulnerability Detection Based On Graph Attention Network And Domain Adaptation. *IEEE Transactions on Software Engineering* (2023).

[124] Fandong Zhang, Shiyuan Xin, and Jufu Feng. 2019. Combining global and minutia deep features for partial high-resolution fingerprint matching. *Pattern Recognition Letters* 119 (2019), 139–147.

[125] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability Detection by Learning from Syntax-Based Execution Paths of Code. *IEEE Transactions on Software Engineering* (2023).

[126] Liheng Zhang and Guo-Jun Qi. 2020. Wcp: Worst-case perturbations for semi-supervised deep learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3912–3921.

[127] Mingyang Zhang, Jianfei Chen, Jianyi Liu, Jingchu Wang, Rui Shi, and Hua Sheng. 2022. LogST: Log Semi-supervised Anomaly Detection Based on Sentence-BERT. In *2022 7th International Conference on Signal and Image Processing (ICSIP)*. IEEE, 356–361.

[128] Ya-Lin Zhang, Longfei Li, Jun Zhou, Xiaolong Li, Yujiang Liu, Yuanchao Zhang, and Zhi-Hua Zhou. 2017. POSTER: A PU learning based system for potential malicious URL detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2599–2601.

[129] Zhi-Wu Zhang, Xiao-Yuan Jing, and Tie-Jian Wang. 2017. Label propagation based semi-supervised learning for software defect prediction. *Automated Software Engineering* 24 (2017), 47–69.

[130] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.

[131] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. *arXiv preprint arXiv:2401.15468* (2024).

[132] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019), 10197–10207.

[133] Zhi-Hua Zhou and Ming Li. 2005. Tri-training: Exploiting unlabeled data using three classifiers. *IEEE Transactions on knowledge and Data Engineering* 17, 11 (2005), 1529–1541.

Received 13 May 2024; revised 15 August 2024; accepted 14 September 2024