The Journal of Systems and Software 213 (2024) 112038

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

CSVD-TF: Cross-project software vulnerability detection with TrAdaBoost by fusing expert metrics and semantic metrics☆

Zhilong Cai, Yongwei Cai, Xiang Chen *, Guilong Lu, Wenlong Pei, Junjie Zhao

School of Information Science and Technology, Nantong University, Nantong, China

ARTICLE INFO

Keywords:
Cross-project software vulnerability detection
Transfer learning
Expert metrics
Semantic metrics
Metric fusion

ABSTRACT

Recently, deep learning-based software vulnerability detection (SVD) approaches have achieved promising performance. However, the scarcity of high-quality labeled SVD data influences the practicality of these approaches. Therefore, cross-project software vulnerability detection (CSVD) has gradually attracted the attention of researchers since CSVD can utilize the labeled SVD data from the source project to construct an effective CSVD model for the target project via transfer learning. However, if a certain number of program modules in the target project can be labeled by security experts, it can help to improve CSVD model performance by effectively utilizing similar SVD data in the source project. For this more practical CSVD scenario, we propose a novel approach CSVD-TF via the transfer learning method TrAdaBoost. Moreover, we find expert metrics and semantic metrics extracted from the functions show a certain complementary in our investigated scenario. Therefore, we utilize a model-level metric fusion method to further improve the performance. We perform a comprehensive study to evaluate the effectiveness of CSVD-TF on four real-world projects. Our empirical results show that CSVD-TF can achieve performance improvements of 7.5% to 24.6% in terms of AUC when compared to five state-of-the-art baselines.

1. Introduction

A software vulnerability is a weakness or flaw in a computer program or system that can be exploited by malicious actors to compromise the security or functionality of the software. Software vulnerabilities have a wide range of impacts, such as data breaches, financial loss, data loss, and system disruption. Therefore, organizations and individuals must be proactive in identifying, mitigating, and patching software vulnerabilities to minimize these potential impacts.

In recent years, researchers have proposed different deep learning (DL)-based software vulnerability detection (SVD) approaches and these data-driven approaches have achieved promising results when compared to previous traditional approaches (such as static analyzers) (Steenhoek et al., 2023b; Lin et al., 2020; Harzevili et al., 2023; Ghaffarian and Shahriari, 2017; Chakraborty et al., 2021; Zheng et al., 2020).

For the target project, a significant challenge faced by the previous DL-based SVD studies lies in the scarcity of the high-quality labeled SVD dataset for training vulnerability detection models. This scarcity stems from the laborious, time-consuming, and error-prone nature of labeling vulnerable source code, even for experts in the security domain (Croft et al., 2023). One possible solution is to utilize the other project with the labeled SVD dataset (i.e. the source project). Due to the data distribution difference between the target project and the source project, we can resort to transfer learning (Pan and Yang, 2009), which can transfer the knowledge of the source project to build the SVD model for the target project. This research problem is called cross-project software vulnerability detection (CSVD) and attracts the attention of researchers. For example, Liu et al. (2020) was the first study to investigate this problem and proposed the approach CD-VulD based on metric transfer learning. Nguyen et al. (2019) and Nguyen et al. (2020) proposed the approaches SCDAN and Dual-GD-DDAN based on deep domain adaptation. Later, they Nguyen et al. (2022) further proposed the approach DAM2P, which learned domain-invariant features and used kernel methods with the max-margin principle. Recently, Zhang et al. (2023b) proposed the approach CPVD based on the graph attention network and domain adaptation representation learning.

However, the practicality of these previous CSVD studies (Liu et al., 2020; Nguyen et al., 2019, 2020, 2022; Zhang et al., 2023b) still has the following two limitations.

Limitation ❶: In previous CSVD studies, researchers mainly utilized the labeled SVD data in the source project and the unlabeled SVD data

☆ Editor: Dr. Hongyu Zhang.

* Corresponding author.
E-mail addresses: zhil.cai@outlook.com (Z. Cai), yongw.cai@outlook.com (Y. Cai), xchencs@ntu.edu.cn (X. Chen), guil.lu@outlook.com (G. Lu), wl.pei@outlook.com (W. Pei), zhaojunjie225@gmail.com (J. Zhao).

https://doi.org/10.1016/j.jss.2024.112038
Received 15 November 2023; Received in revised form 18 March 2024; Accepted 23 March 2024
Available online 25 March 2024

in the target project to construct the CSVD model. However, there still exists a high data distribution difference between the source project and the target project. If some program modules in the target project can be labeled by security experts, we can effectively utilize similar SVD data from the source project, which can help to improve the performance of the CSVD model and this new scenario is more practical when compared to previous CSVD studies.

Limitation ❷: In previous CSVD studies, researchers only considered semantic metrics, which were generated by code representation learning via deep learning. However, these studies ignored another important type of metrics (i.e., expert metrics), which were manually designed by domain experts. Whether these two different types of metrics have a certain complementary for vulnerability detection in our investigated scenario is unknown. Moreover, if the complementary exists, whether fusing these two different types of metrics can further improve the performance of CSVD is also unknown.

To alleviate the aforementioned two limitations, we propose a novel approach CSVD-TF in our study. Specifically, to alleviate limitation ❶, we mainly resort to the transfer learning method TrAdaBoost (Dai et al., 2007), which can effectively utilize the labeled SVD data in the source project and the limited number of labeled SVD data in the target project. Due to the small proportion of vulnerability program modules in the software projects, we use SMOTE (Synthetic Minority Over-sampling Technique) (Chawla et al., 2002) to alleviate the class imbalanced problem in the training data. Moreover, we use XGBoost (Chen et al., 2015b) as the base classifier. To alleviate limitation ❷, we train the CSVD model for each type of metric and analyze the complementary between these two types of metrics. Then we fuse these two CSVD models to generate the final detection result (i.e., vulnerable or non-vulnerable) for the new program modules in the target project.

To verify the effectiveness of our proposed approach CSVD-TF, we select four real-world software projects (i.e., FFmpeg, LibTIFF, LibPNG, and VLC) (Lin et al., 2018) as our experimental subjects and conduct an extensive evaluation. Final comparison results show that CSVD-TF achieves 80.8% in terms of AUC performance measure, which can improve the performance by 7.5% to 24.6% than state-of-the-art baselines. Our ablation study results also confirm the effectiveness of component settings (i.e., the usage of TrAdaBoost, metric fusion method, the use of SMOTE for alleviating the class imbalanced issue, and the use of XGBoost as the base classifier) in CSVD-TF.

Based on our study's findings, we find that labeling a certain number of program modules in the target project is a more practical scenario when compared to previous CSVD studies. For this scenario, designing more effective transfer learning methods is a potential way to improve the performance of CSVD. Moreover, these two different types of metrics should be both considered, and designing effective methods to fuse expert metrics and semantic metrics is also a promising way to improve the performance of CSVD. Therefore, we call on researchers to conduct more in-depth follow-up studies in our investigated scenario from these two directions.

The novelty and contributions of our study can be summarized as follows:

- **Scenario.** Compared to previous studies that only considered labeled SVD data of the target project, we consider a more practical scenario for CSVD. In this scenario, we aim to label a certain number of program modules in the target project. Then we construct the CSVD model by further considering the labeled SVD data in the source project.
- **Approach.** For this new CSVD scenario, we propose CSVD-TF based on the transfer learning method TrAdaBoost (Dai et al., 2007), which can effectively utilize similar SVD data in the source project based on a certain number of labeled program modules in the target project. In CSVD-TF, we also utilize SMOTE to alleviate the class imbalanced problem in the training data and use XGBoost as the base classifier. Finally, we use a model-level method to effectively fuse expert metrics and semantic metrics.
- **Evaluation.** We perform a comprehensive study to evaluate the effectiveness of CSVD-TF on four SVD datasets gathered from real-world software projects. Comparison results demonstrate that CSVD-TF can achieve better CSVD performance compared to five selected baselines.

Open Science. We share our dataset and source code in the GitHub repository (https://github.com/CSVD/CSVD-TF).

Paper Organization. Section 2 introduces the background and research motivation of our study. Section 3 presents the overall framework of CSVD-TF and the details of each phase. Section 4 describes our experimental setup, including research questions, experimental subjects, baselines, performance measures, and experimental settings. Section 5 presents our result analysis for each research question. Section 6 discusses the influence of the selection ratio of the SVD data in the target project for labeling, the performance of CSVD-TF in terms of MCC and effort-aware performance measures, and analyzes potential threats to our study. Section 7 summarizes the related work and shows the novelty of our study. Finally, Section 8 summarizes our work and shows potential future directions.

2. Background

In this section, we briefly introduce the background of cross-project software vulnerability detection and then analyze the limitations of previous studies on CSVD.

2.1. Cross-project software vulnerability detection

The performance of the previous deep learning-based software vulnerability detection studies is limited to the scarcity of the high-quality labeled SVD dataset, since identifying vulnerable program modules is tedious, time-consuming, and error-prone, even for security domain experts (Croft et al., 2023).

Cross-project software vulnerability detection is an effective approach to alleviate the high cost of labeling program modules in the target project. It aims to leverage labeled SVD data from other projects (i.e., source projects) to build a vulnerability detection model, which can detect vulnerabilities in the target project. However, due to the potential large data distribution difference between the source project and the target project, the model trained only on the labeled SVD data from the source project may not achieve satisfactory performance on the target project. To alleviate this data distribution difference, recent studies (Liu et al., 2020; Nguyen et al., 2019, 2020; Zhang et al., 2023b) mainly resorted to transfer learning methods for improving the model's performance in detecting vulnerabilities for the target project.

2.2. Research motivation

Although previous CSVD studies (Liu et al., 2020; Nguyen et al., 2019, 2020; Zhang et al., 2023b) have achieved promising performance, there still exist the following two limitations.

Limitation ❶: In previous CSVD studies, researchers only used the labeled SVD data in the source project. However, there exists a large data distribution difference between the source project and the target project. Specifically, the distribution difference between the source project and the target project arises from the unique characteristics inherent in each project. These differences may be caused by code complexity, different development processes, developer experience, or distinct vulnerability types. A more practical way is to select a certain number of program modules from the target project and label these program modules as vulnerable or non-vulnerable by the security experts. Then we can use these labeled SVD data of the target project for CSVD, which can help to learn vulnerability detection knowledge from the source project that is related to the target project. Notice in real-world scenarios, it could indeed be challenging and costly to

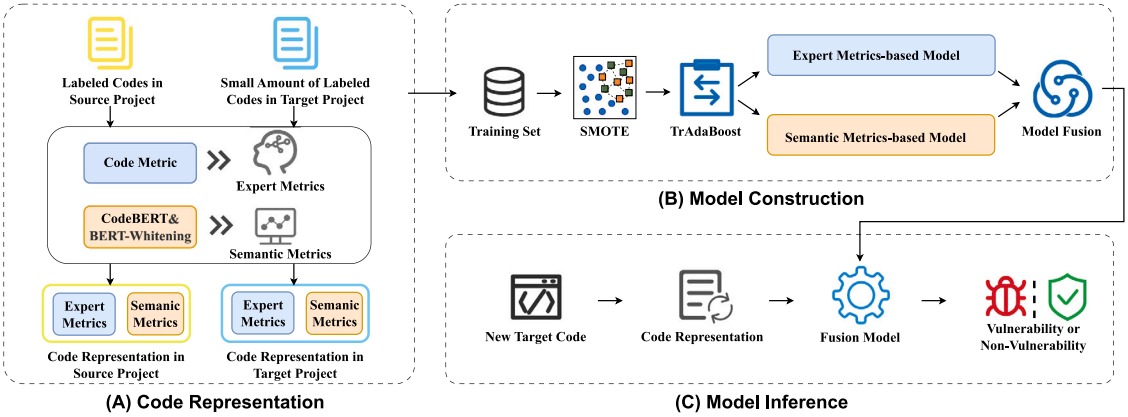

Fig. 1. Framework of our proposed approach CSVD-TF.

manually label software vulnerability. In our study, we suggest that this manual labeling may be performed by an in-house security expert or even automated to some extent by using some commercial tools (such as SAST[1]).

Limitation ❷: In previous CSVD studies, researchers mainly employed semantic metrics from code representation learning via deep learning. For example, CD-VulD (Liu et al., 2020) utilized a deep learning model to extract semantic metrics from code token sequences. However, these studies ignored another important type of metric (i.e., expert metric). Specifically, expert metrics and semantic metrics can differ due to the nature of the knowledge used to build them. Expert metrics often contain the experiential and domain-specific knowledge of security experts, while semantic metrics are largely derived from code information. This difference can lead to discrepancies between these two types of metrics due to the different knowledge bases they are centered on. If these two types of metrics have a certain complementary in detecting vulnerabilities, fusing these two types of metrics may help to improve the performance of the CSVD model. In the field of just-in-time defect prediction, Ni et al. (2022) showed the effectiveness of fusing these two types of metrics in defect prediction and localization. Then we Chen et al. (2023) also verified the effectiveness of the metric fusion for multi-objective just-in-time defect prediction. However, to the best of our knowledge, whether fusing these two types of metrics can improve the performance of CSVD has not been thoroughly investigated in previous studies.

3. Approach

In this section, we use Fig. 1 to show the framework of our proposed approach CSVD-TF, which is a cross-project software vulnerability detection approach with the transfer learning method TrAdaBoost (Dai et al., 2007) by fusing expert metrics and semantic metrics. In particular, in phase ❶, we extract expert metrics and semantic metrics for each function respectively. In phase ❷, for each type of metric, we construct a corresponding CSVD model via the transfer learning method TrAdaBoost, which can effectively use a certain number of program modules in the target project. Then we use Weighted Average as the calculation rule to fuse these two different CSVD models. In phase ❸, given a new function from the target project, we extract its expert metrics and semantic metrics respectively. Then we use the fusion model to predict whether this new function contains a vulnerability. In the rest of this section, we show the details of each phase.

3.1. Code representation extraction phase

In this phase, we aim to extract representative and useful metrics from raw data (i.e., function code), converting statistical data and code tokens into numerical representations for subsequent data analysis and model construction. In CSVD-TF, we extract the code representation from two different perspectives: expert metrics and semantic metrics.

3.1.1. Expert metrics

In our study, we consider 39 code metrics as expert metrics, which are collected by the commercial tool *Understand*.[2] *Understand* can compute both classical and object-oriented source code metrics for Java and C/C++ software projects. In a previous study, Zagane et al. (2020) used 18 code metrics for SVD. Therefore, we first selected these similar code metrics by using *Understand*. Then, we further considered more related metrics provided by *Understand*, which finally resulted in 39 code metrics. These 39 expert metrics can be roughly classified into five dimensions, such as code size, complexity, readability, maintainability, and performance. We show the details of these expert metrics in Table 1. In this table, we give the metric name, metric description, and corresponding dimension. Notice that each of the 39 expert metrics could provide unique and valuable insights into the vulnerabilities. There may exist redundancy concerns among those expert metrics. This can be addressed by adopting dimensionality reduction methods (such as principal component analysis (Wold et al., 1987)) or feature selection methods (Li et al., 2017). However, after adopting dimensionality reduction, we did not find an obvious performance improvement in our investigated task, the detailed results of using principal component analysis (PCA) for CSVD-TF in terms of AUC can be found in our GitHub repository.[3]

3.1.2. Semantic metrics

With the development of deep learning technology, current SVD studies (Li et al., 2018; Zou et al., 2019; Li et al., 2021b) came to use deep learning to extract code semantic features. In this study, we use the pre-trained model CodeBERT (Feng et al., 2020) to extract semantic metrics from code functions by following previous studies (Ni et al., 2022; Chen et al., 2023). This pre-trained model learns contextual word embeddings and has been widely utilized in different software engineering tasks (Gao et al., 2021; Liu et al., 2022; Yang et al., 2023b; Yu et al., 2022). In particular, for the given function-level code, we first split it according to the camel casing naming convention to get the input sequence $\{x_i\}_{i=1}^{M}$, where M denotes the length of the code sequence. Then we put the sequence into CodeBERT, and extract the

[1] https://www.synopsys.com/software-integrity/static-analysis-tools-sast.html.

[2] http://www.scitools.com/.

[3] https://github.com/CSVD/CSVD-TF/blob/main/figs/PCA.png.

Table 1
Description of expert metrics used by CSVD-TF.

Dimension	Metric name	Metric description
Code size	CountDeclClass	Number of classes
	CountDeclFunction	Number of functions
	CountLine	Total number of lines
	CountLineBlank	Number of blank lines (also known as BLOC)
	CountLineCode	Number of lines of source code (also known as LOC)
	CountLineCodeDecl	Number of lines of declarative source code
	CountLineComment	Number of lines of comments (also known as CLOC)
	CountLineInactive	Number of inactive lines
	CountLinePreprocessor	Number of lines containing preprocessor directives
Complexity	AvgCyclomatic	Average cyclomatic complexity of all nested functions or methods
	AvgCyclomaticModified	Average modified cyclomatic complexity of all nested functions or methods
	AvgCyclomaticStrict	Average strict cyclomatic complexity of all nested functions or methods
	AvgEssential	Average essential complexity of all nested functions or methods
	MaxCyclomatic	Maximum cyclomatic complexity of all nested functions or methods
	MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods
	MaxCyclomaticStrict	Maximum modified cyclomatic complexity of nested functions or methods
	MaxEssential	Maximum essential complexity of all nested functions or methods
	MaxNesting	Maximum nesting level of control constructs
	SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods
	SumCyclomaticModified	Sum of modified cyclomatic complexity of all nested functions or methods
	SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods
	SumEssential	Sum of essential complexity of all nested functions or methods
Readability	AvgLine	Average number of lines in all nested functions or methods
	AvgLineBlank	Average number of blank lines in all nested functions or methods
	AvgLineCode	Average number of lines of source code in all nested functions or methods
	AvgLineComment	Average number of lines of comments in all nested functions or methods
	AltAvgLineBlank	Average number of blank lines in all nested functions or methods (including inactive regions)
	AltAvgLineCode	Average number of lines of source code in all nested functions or methods (including inactive regions)
	AltAvgLineComment	Average number of lines of comments in all nested functions or methods (including inactive regions)
	AltCountLineBlank	Number of blank lines, including inactive regions
	AltCountLineCode	Number of lines of code, including inactive regions
	AltCountLineComment	Number of lines of comments, including inactive regions
	RatioCommentToCode	Ratio of comment lines to code lines
Maintainability	CountStmt	Number of statements
	CountStmtDecl	Number of declarative statements
	CountStmtEmpty	Number of empty statements
	CountStmtExe	Number of executable statements
Performance	CountLineCodeExe	Number of lines of executable source code
	CountSemicolon	Number of semicolons

hidden states of the first and last layers in the output. After averaging them, we get the semantic metric vector of the code $X \in \mathbb{R}^H$, in which H denotes the hidden size of CodeBERT.

After the above semantic metric extraction process, we convert function-level code snippets into a set of row vectors $\{X_i\}_{i=1}^{N}$, where N denotes the training set size. To improve training speed and achieve better performance, it is necessary to reduce metric redundancy and ensure independence and uniform distribution among the data metrics. Therefore, we further employ the BERT-Whitening operation (Su et al., 2021), which utilizes a simple linear transformation to enhance the isotropy of sentence representations, to reduce the dimensionality of previously obtained semantic metric vectors.

3.2. Model construction phase

After extracting expert metrics and semantic metrics for labeled functions in the source project and a certain number of labeled functions in the target project, we can construct the training set for model construction. Then in this phase, we first use SMOTE to alleviate the class imbalanced problem in the training set. Then we construct two CSVD models via TrAdaBoost for each type of metric. Finally, we use a model-level method for fusing these two CSVD models.

3.2.1. Data preprocessing via class imbalanced learning

In our training set, the number of vulnerable functions is much lower than the number of non-vulnerable functions (as discussed in Section 4.2). Therefore, there exists a class imbalanced problem in our training set, which can result in the low performance of the model in predicting vulnerabilities (Yang et al., 2023a). To alleviate the class imbalanced problem in the training data, we utilize SMOTE (Chawla et al., 2002) to preprocess the training set. Specifically, we first treat the vulnerable functions as the minority class and treat the non-vulnerable functions as the majority class. Then for a vulnerable function vf, we select the k nearest neighbors by calculating the distances to other vulnerable functions. Later, we randomly select a neighbor nf and generate a new synthetic sample by interpolating between the vulnerable function vf and the neighbor nf. We repeat this process until a specified number of synthetic samples are generated. Finally, we add these synthetic samples to the original vulnerable functions, which can eventually create a class-balanced training set for CSVD model training. Notice that we apply SMOTE to the labeled SVD data in the source project and the target project respectively in our study.

3.2.2. Model construction via transfer learning

After performing data preprocessing for the training set, we resort to the transfer learning method TrAdaBoost (Dai et al., 2007) for CSVD model training. In our investigated scenario, the training set includes a large amount of labeled SVD data in the source project and a certain number of labeled SVD data in the target project. Therefore, TrAdaBoost is specifically designed for our investigated scenario, intended to reduce the discrepancy between the source project and the target project. Specifically, due to the data distribution difference between the source project and the target project, some labeled SVD data from the source project may have a positive effect on vulnerability detection in the target project, while others may have a negative effect. The fundamental concept behind TrAdaBoost is to reduce the influence of

"inferior" data from the source project while improving the significance of "superior" data from the source project during each model training iteration. This approach guides the model to evolve in a manner that favors the labeled SVD data in the target project, ultimately enhancing the model's capacity to identify vulnerabilities in the target project. For the base classifier, we consider XGBoost (Chen et al., 2015b). XGBoost's success in classification tasks can be attributed to its robustness, efficiency, regularization techniques, and the flexibility it offers for customization and optimization.

Algorithm 1 CSVD Model Training via TrAdaBoost

Input: Labeled SVD data in the source project L_s; Labeled SVD data in the target project L_t; Unlabeled SVD data in the target project U_t; base classifier **Learner**; the maximum number of iterations N;

Initialize

1: $n \leftarrow$ the size of L_s, $m \leftarrow$ the size of L_t

2: $L \leftarrow L_s \cup L_t$

3: The initial weight vector $w^1 = \left(\omega_1^1, \omega_2^1, \ldots, \omega_{n+m}^1\right)$, where $\omega_i^1 = \begin{cases} 1/n, & 1 \le i \le n \\ 1/m, & n+1 \le i \le n+m \end{cases}$.

For $t = 1, \cdots, N$

4: Normalize the weight of each data, set $p^t = w^t / \left(\sum_{i=1}^{n+m} w_i^t\right)$

5: Call **Learner**, according to the merged training data L, the weight distribution p^t on L, and the unlabeled data U_t, obtain a classifier $h_t : V \to T$.

6: Calculate the error of h_t on L_t:

$$\epsilon_t = \sum_{i=n+1}^{n+m} \frac{w_i^t \cdot \left|h_t\left(x_i\right) - c\left(x_i\right)\right|}{\sum_{i=n+1}^{n+m} w_i^t}$$

7: Set $\beta_t = \epsilon_t / \left(1 - \epsilon_t\right)$ and $\beta = 1 / \left(1 + \sqrt{2ln\ n/N}\right)$

8: **if** $\epsilon_t > 0.5$ **then**

9: $\beta_t \leftarrow 1$

10: **end if**

11: **if** $\epsilon_t == 0$ **then**

12: **break**

13: **end if**

14: Update the new weight vector:

$$w_i^{t+1} = \begin{cases} w_i^t \ \beta^{|h_t(x_i) - c(x_i)|}, & 1 \le i \le n \\ w_i^t \ \beta_t^{-|h_t(x_i) - c(x_i)|}, & n+1 \le i \le n+m \end{cases}$$

Output: The final classifier:

$$h_f(x) = \begin{cases} 1, & \prod_{t=\lceil N/2 \rceil}^{N} \beta_t^{-h_t(x)} \ge \prod_{t=\lceil N/2 \rceil}^{N} \beta_t^{-\frac{1}{2}} \\ 0, & \text{otherwise} \end{cases}$$

The pseudo-code of CSVD model training via TrAdaBoost for expert metrics or semantic metrics is shown in Algorithm 1. Specifically, for the experimental subjects (introduced in Section 4.2) used by our study, we assume that the project "FFmpeg" is treated as the source project and the project "LibPNG" is treated as the target project as an illustration example. We use V_s and V_t to represent the sample spaces of the source project and the target project, respectively. The label set $T = \{0, 1\}$ denotes the set of labels, representing the non-vulnerable functions or vulnerable functions, respectively. The boolean function h_t used by the classifier (i.e., the vulnerability detection function) denotes the prediction function mapping from the sample space V ($V = V_s \cup V_t$) to the vulnerability label set T. Training data $L \in \{V \times T\}$ contains two parts of labeled data $L_s = \left\{\left(x_i^s, c\left(x_i^s\right)\right)\right\}$ and $L_t = \left\{\left(x_j^t, c\left(x_j^t\right)\right)\right\}$, where $x_i^s \in V_s\,(i = 1, \ldots, n)$ and $x_j^t \in V_t\,(j = 1, \ldots, m)$. $c(x)$ represents the label of sample data x. The unlabeled test data is denoted by $U_t = \left\{\left(x_i^t\right)\right\}$, where $x_i^t \in V_t\,(i = 1, \ldots, k)$.

For each iteration (Lines 4–14), if the training sample from "FFmpeg" is misclassified, it may contradict the training data from "LibPNG". In such cases, we multiply $\beta^{|h_t(x_i) - c(x_i)|}$ to decrease its weight (Line 14). Therefore, in the next iteration, the misclassified samples will have less influence on the CSVD model compared to the previous iteration. If the sample from "LibPNG" is misclassified, we multiply $\beta_t^{-|h_t(x_i) - c(x_i)|}$ to increase its training weight, in which $h_t\left(x_i\right)$ is the detection result and $c\left(x_i\right)$ is the ground truth. After a certain number of iterations, the SVD data in the source project that aligns with the SVD data in the target project can be assigned higher weights, otherwise can be assigned lower weights.

By utilizing expert metrics or semantic metrics as the input for CSVD model training, we can train two different CSVD models. For the convenience of the model fusion description, we denote the expert metric-based CSVD model as M_e and the semantic metric-based CSVD model as M_s.

3.2.3. Expert metrics and semantic metrics fusion method

There are two different types of methods for fusing expert metrics and semantic metrics. The most straightforward method is metric-level fusion. Specifically, we can directly concatenate the expert metric vector and the semantic metric vector into a final vector. In our study, the final vector is 295-dimensional (i.e., 39-dimensional expert metric vector and 256-dimensional semantic metric vector). Except for this type of method, we can consider model-level fusion. Specifically, model-level fusion can be performed by using different calculation rules to compute the final prediction value from the CSVD model M_e and the CSVD model M_s. In our study, we mainly consider the weighted average as the calculation rule of these two models. In weighted averaging, the output of each CSVD model (i.e., prediction value) is multiplied by the corresponding weight, and these weighted values are summed to obtain the final ensemble prediction value. This can be represented by the following formula:

$$P_{final} = P_{Me} \times W_{Me} + P_{Ms} \times W_{Ms} \tag{1}$$

where P_{final} represents the prediction value of the fusion model, P_{Me} and P_{Ms} represent the prediction value generated by the CSVD models M_e and M_s respectively, W_{Me} and W_{Ms} represent the weights assigned to the CSVD models M_e and M_s. Therefore, this method can create a final composite model where the contribution of each model is determined by its weight. In practice, it is necessary to select the optimal fusion method based on the actual scenario.

3.3. Model inference phase

In our study, we find that using the model-level fusion by weighting the prediction value for the model M_e and the model M_s in the 0.4:0.6 ratio can achieve the best performance for CSVD-TF (detailed result can be found in Section 5.4). Therefore, in our model inference phase, when given a new function, we first extract expert metrics and semantic metrics for this function. Then we input these two different types of metrics into the model M_e and the model M_s, respectively. Finally, we can obtain the final prediction value by taking a weighted sum of these two prediction values in the 0.4:0.6 ratio. Using this final prediction value, we can determine whether this function contains vulnerability (i.e., vulnerable when the final prediction value is larger than 0.5, otherwise non-vulnerable).

4. Experimental setup

In this section, we first introduce our designed research questions and their design motivation. Then, we provide detailed information about the experimental subjects, baselines, performance measures, and experimental settings.

4.1. Research questions

To show the competitiveness of CSVD-TF and the rationale of the component settings in CSVD-TF, we designed the following five research questions (RQs) in our empirical study.

RQ1: How effective is CSVD-TF compared to the baselines?

Motivation. Since we are the first to consider this new scenario for CSVD, there are no direct CSVD baselines for comparison. However, to assess the effectiveness of CSVD-TF, we consider baselines related to our study (introduced in Section 4.3) and design this RQ to investigate whether CSVD-TF can outperform these baselines.

RQ2: Whether using TrAdaBoost can further boost the performance of CSVD-TF?

Motivation. As discussed in Section 2.2, we assume there exists a certain number of program modules in the target project. Therefore, we design this RQ to investigate whether using TrAdaBoost can help to achieve better performance than other control approaches in our investigated scenario.

RQ3: Whether fusing both two types of metrics can further boost the performance of CSVD-TF?

Motivation. Previous studies (Liu et al., 2020; Nguyen et al., 2019, 2020; Zhang et al., 2023b) mainly focused on a single type of metric. However, these two types of metrics are designed from different perspectives. Therefore, we design this RQ to investigate whether these two types of metrics complement each other in detecting vulnerabilities for CSVD-TF. Moreover, we want to investigate whether fusing these two types of metrics can lead to performance improvement for CSVD.

RQ4: How different fusion methods influence the performance of CSVD-TF?

Motivation. As discussed in Section 3.2.3, there are two different types of methods for fusing expert metrics and semantic metrics (i.e., metric-level fusion and model-level fusion). Therefore, we design this RQ to analyze the influence of different fusion methods for CSVD-TF and aim to select the best fusion method that can achieve the best performance for CSVD in our investigated scenario.

RQ5: How does the class imbalance method SMOTE influence the performance of CSVD-TF?

Motivation. In the SVD task, the number of vulnerable functions is far less than the number of non-vulnerable functions in most of the software projects. This can cause the gathered SVD data to have a class imbalanced problem, which can make the trained model tend to predict the new functions as non-vulnerable. To alleviate the class imbalanced problem, different class imbalanced methods (He and Garcia, 2009) have been proposed. In our model, we choose the over-sampling method SMOTE to alleviate the class imbalanced problem. The purpose of designing this RQ is to analyze whether using SMOTE can improve the performance of CSVD-TF.

RQ6: How do different base classifiers influence the performance of CSVD-TF?

Motivation. The choice of base classifier is a crucial step in CSVD model construction. Common classifiers include eXtreme Gradient Boosting (XGBoost) (Chen et al., 2015b), Support Vector Machine (SVM) (Cortes and Vapnik, 1995), Random Forest (RF) (Breiman, 2001) and Decision Trees (DT) (Quinlan, 2014). Therefore, we designed this RQ to investigate the influence of different base classifiers on the performance of CSVD-TF and want to investigate whether using XGBoost as the base classifier can achieve the best performance for CSVD-TF.

Table 2
Statistical information of experimental subjects.

Project	Vulnerable functions	Non-vulnerable functions
FFmpeg	213	5552
LibTIFF	96	731
LibPNG	44	577
VLC	42	6320

4.2. Experimental subjects

In our empirical study, we use four real-world large-scale projects (i.e., FFmpeg, LibTIFF, LibPNG, and VLC) as our experimental subjects. Specifically, FFmpeg is an open-source cross-platform multimedia processing toolset. LibTIFF is an open-source C library for handling and manipulating TIFF image files. LibPNG is an open-source C library for handling and manipulating PNG image files. VLC is an open-source cross-platform multimedia player and framework. The vulnerability data were labeled from two prominent sources — the National Vulnerability Database (NVD) and Common Vulnerability and Exposures (CVE). Both these repositories use CVE IDs as unique identifiers, streamlining the process of distinguishing each vulnerability. CVE IDs play a crucial role in cybersecurity. They are assigned to each vulnerability, enabling security experts to readily access the technical details of known vulnerabilities from multiple CVE-compatible sources. The NVD provides a straightforward procedure for searching known vulnerabilities corresponding to a specific software project. By leveraging the detailed descriptions provided by the NVD, the respective version of a software project can be downloaded, and then each vulnerable function within the project can be labeled. These experimental subjects have been used in previous CSVD studies (Nguyen et al., 2019, 2020), which can ensure the representatives of our experimental conclusion.

Table 2 provides statistical information on our experimental subjects. For example, the FFmpeg project contains 213 vulnerable functions and 5552 non-vulnerable functions.

4.3. Baselines

To our best knowledge, we are the first to construct the CSVD model by considering the labeled SVD data in the source project and a certain number of labeled program modules in the target project. There are no previous CSVD studies investigating this new scenario. However, to show the competitiveness of CSVD-TF, we consider the following baselines, which are related to our study. For the description convenience for the baseline setting, we denote the labeled SVD data in the source project as L_s and the SVD data without labels in the target project as t. For the dataset t, we label a certain number of the SVD data as L_t and denote the remaining SVD data without labels as U_t. For CSVD-TF, we use L_s and L_t to train the model. Then we use U_t to evaluate the performance of the trained model.

Vuldeepecker. Li et al. (2018) utilized a BiLSTM network to extract code semantic metrics for vulnerability detection. Notice in our study, we use the dataset L_s as the training set and U_t as the test set for Vuldeepecker.

LineVul. Fu and Tantithamthavorn (2022) proposed a Transformer-based fine-grained vulnerability detection approach. We select LineVul as our baseline since this approach has shown high performance in a recent empirical study for deep learning-based vulnerability detection (Steenhoek et al., 2023b). Notice the experimental setting is the same as the baseline Vuldeepecker.

Dual-GD-DDAN. Nguyen et al. (2020) used a bidirectional recurrent neural network to extract semantic metrics. Dual-GD-DDAN has two different generators to obtain the code sequences for the source project and the target project. In addition, Dual-GD-DDAN also has two discriminators to discriminate these two different projects respectively. Notice for this baseline, we utilize the dataset L_s and the dataset t to

train the model. For a fair comparison, we use U_t as the test set to evaluate the model performance.

DAM2P. Nguyen et al. (2022) learned domain-invariant features and used kernel methods with the max-margin principle to bridge the gap between the source and target projects on a joint space. Since DAM2P investigates the same scenario for CSVD as Dual-GD-DDAN, we keep the same dataset setting.

DTB. Chen et al. (2015a) combined the data gravitation method with the transfer learning method TrAdaBoost for cross-project defect prediction. Since the scenario investigated by DTB for cross-project defect prediction is similar to our investigated scenario for CSVD, we consider the same dataset setting for CSVD-TF.

In Summary, TrAdaBoost has demonstrated effective results in similar tasks, such as cross-project software defect prediction (CPDP). Therefore, we consider the classical CPDP approach DTB as our baseline. Moreover, we consider state-of-the-art CSVD baselines (such as Dual-GD-DDAN and DAM2P), which were designed based on domain adaptation. Notice these CSVD baselines only consider the labeled data in the source project and do not consider any labeled modules in the target project. Finally, we also consider state-of-the-art within-project approaches (such as Vuldeepecker and LineVul). Therefore, our baselines can cover different types of transfer learning methods.

4.4. Performance measure

Previous research (Tantithamthavorn et al., 2018) has indicated that threshold-dependent performance measures (i.e., precision, recall, and F1 score) may not accurately measure the discrimination ability of a model in certain scenarios. The reasons can be summarized as follows: (1) the value computation of these performance measures relies on the chosen threshold for determining classification results and (2) the performance measures are sensitive to the training data with the class imbalanced problem. By following the suggestion of Tantithamthavorn et al. (2018), we use the Area Under the receiver operator characteristic Curve (AUC) as the measure to evaluate the performance of the trained CSVD models. Specifically, AUC is a threshold-independent measure, it is computed by measuring the area under the curve that plots the true positive rate against the false positive rate. The values of AUC range from 0 to 1, where 0 represents the poorest detection ability of the CSVD model, 0.5 represents the classifier's ability to predict vulnerabilities equivalent to random guessing, and 1 represents the perfect detection ability of the CSVD model.

4.5. Experimental settings

In our experimental study, we aim to demonstrate the effectiveness of our proposed approach CSVD-TF in our investigated scenario. Specifically, to simulate our investigated scenario for CSVD, we can select one project as the source project and three other projects as target projects (e.g. if the source project is set as FFmpeg, the target projects can be set as LibTIFF, LibPNG, or VLC). We calculate the performance for each CSVD project combination, such as FFmpeg→LibTIFF, FFmpeg→LibPNG, or FFmpeg→VLC. This process is repeated for all four projects, which can result in a total of 12 CSVD project combinations.

Moreover, during the training process, we utilize both the labeled SVD data in the source project and 30% of the labeled SVD data in the target project as the training set via Stratified Sampling. Then we treat the remaining 70% of the SVD data in the target project as the test set. Due to the randomness in the SVD data selection in the target project for labeling, we conducted independent experiments 10 times for this data selection process and selected different SVD data in the target project for labeling with different random seeds for each experiment.

In our hyperparameter setting, for BERT-Whitening operation (Su et al., 2021) (in semantic metrics extraction), we set the semantic metric dimension after BERT-Whitening to 256 (our investigated value range is {128, 256, 512}). For the data preprocessing via class imbalanced learning, we set the number of the nearest neighbors (i.e., the hyperparameter k) of SMOTE to 5 (our investigated value range is {1,3,5,10}). For the model construction via transfer learning, we set the number of iterations (N) for TrAdaBoost to 200 (our investigated value range is {50, 100, 200, 500}). The base classifier is set as XGBoost and we use the default parameter setting for this base classifier.

All the experiments are conducted on a computer (CPU 3.50 GHz) with a GeForce RTX4090 GPU (24 GB graphic memory). The running operating system is Windows 10.

5. Experimental results

5.1. RQ1: How effective is CSVD-TF compared to the baselines?

Approach: To answer RQ1, we use AUC to compare the performance of our proposed CSVD-TF with the five selected baselines. By comparing with the baselines Vuldeepecker (Li et al., 2018), LineVul (Fu and Tantithamthavorn, 2022), Dual-GD-DDAN (Nguyen et al., 2020) and DAM2P (Nguyen et al., 2022), we aim to investigate whether selecting a subset of labeled SVD data from the target project could significantly improve the CSVD model's ability to detect vulnerabilities in the target project. By comparing the baseline DTB (Chen et al., 2015a), we aim to investigate whether CSVD-TF outperforms this baseline for CSVD by considering a similar scenario in cross-project defect prediction. Notice, for the baselines Vuldeepecker, LineVul (Fu and Tantithamthavorn, 2022), Dual-GD-DDAN, and DAM2P, we only consider semantic metrics based on the setting of their original studies since these baselines are proposed for cross-project software vulnerability detection. For the baseline DTB, we consider both semantic metrics and expert metrics for a fair comparison since this baseline is proposed for a different task (i.e., cross-project defect prediction) and only considers traditional expert metrics in their original study. Following the aforementioned experimental setup, we randomly select 30% of the SVD data from the target project for labeling, while the remaining 70% of the SVD data is used for testing. We independently repeat this sampling process ten times and conduct experiments using the data obtained from each of the ten independent samplings. Notice to ensure a fair comparison, we apply the trained model for CSVD-TF and five baselines to the same testing set (i.e., the remaining unlabeled SVD data in the target project).

Results: In Table 3, we present comparison results between our approach CSVD-TF and the five baselines for 12 CSVD project combinations. In this table, we find that CSVD-TF can achieve an average of 80.8% in terms of AUC. Compared to Vuldeepecker, LineVul (Fu and Tantithamthavorn, 2022), Dual-GD-DDAN, DAM2P and DTB, CSVD-TF achieves performance improvements of 24.6%, 23.8%, 23.2%, 17.5% and 7.5% respectively. Notice that the baseline DTB also utilizes a portion of labeled SVD data from the target project and DTB achieves a performance improvement of 17.1%, 16.3%, 15.7%, and 10% in AUC compared to Vuldeepecker, LineVul, Dual-GD-DDAN, and DAM2P respectively. These results indicate that utilizing a portion of labeled SVD data from the target project can help to effectively use the labeled SVD data in the source project, which can help to learn vulnerability detection knowledge relevant to the target project.

To further examine the significance of performance differences between CSVD-TF and the five baselines, we conducted the Wilcoxon signed-rank test (Wilcoxon, 1992) at a confidence level of 95%. The Wilcoxon statistical test is often used to check if the difference between two means is statistically significant (i.e., the computed p-value is less than 0.05). In our study, the computed p-values are all less than 0.01, indicating that the performance improvement of CSVD-TF compared to the baselines is statistically significant.

Table 3
The comparison results between CSVD-TF and five baselines in terms of AUC.

Source→Target	CSVD-TF	Vuldeepecker	LineVul	Dual-GD-DDAN	DAM2P	DTB
FFmpeg→LibPNG	**0.933**	0.536	0.605	0.713	0.764	0.769
FFmpeg→LibTIFF	**0.772**	0.766	0.550	0.608	0.743	0.734
FFmpeg→VLC	**0.658**	0.534	0.546	0.502	0.568	0.517
LibPNG→FFmpeg	**0.883**	0.557	0.711	0.592	0.647	0.853
LibPNG→LibTIFF	**0.768**	0.500	0.573	0.594	0.630	0.727
LibPNG→VLC	**0.610**	0.500	0.606	0.515	0.610	0.582
LibTIFF→FFmpeg	**0.881**	0.555	0.537	0.701	0.747	0.843
LibTIFF→LibPNG	**0.945**	0.500	0.587	0.656	0.671	0.800
LibTIFF→VLC	**0.646**	0.604	0.540	0.533	0.541	0.495
VLC→FFmpeg	**0.881**	0.512	0.500	0.500	0.572	0.865
VLC→LibPNG	**0.937**	0.546	0.582	0.500	0.539	0.875
VLC→LibTIFF	**0.782**	0.633	0.500	0.500	0.567	0.738
Average	**0.808**	0.562	0.570	0.576	0.633	0.733
p-value	–	**	**	**	**	**

Notes: *** means p-value < 0.001, ** means p-value < 0.01, * means p-value < 0.05.

Summary for RQ1: CSVD-TF can significantly outperform the five baselines, which achieves performance improvements of 7.5% to 24.6% in terms of AUC. This shows that considering a small portion of labeled SVD data in the target project is an effective way to improve the performance of CSVD.

5.2. RQ2: Whether using TrAdaBoost can further boost the performance of CSVD-TF?

Approach: To investigate the effectiveness of using the transfer learning method TrAdaBoost for CSVD-TF, we design three control approaches based on CSVD-TF. For a fair comparison, we keep other component settings in CSVD-TF the same. That is, we also use XGBoost as the base classifier for cross-project vulnerability detection. For the first control approach, we construct the CSVD model only on the labeled SVD data in the source project (i.e., L_s) and denote this baseline as CSVD_S. For the second control approach, we construct the CSVD model only on a certain number of labeled SVD data in the target project (i.e., L_t) and denote this baseline as CSVD_T. For the third control approach, we simply merged the labeled SVD data in the source project and a certain number of labeled SVD data in the target project (i.e., $L_s \cup L_t$) as the training set. Then we use this training set for CSVD model training without considering transfer learning. We denote this baseline as CSVD_ST.

Results: In Table 4, we present comparison results between our approach CSVD-TF and three control approaches based on CSVD-TF. In this table, we find that compared to CSVD_S, CSVD_T, and CSVD_ST, CSVD-TF demonstrate performance improvement of 30.4%, 18%, and 19.3% respectively. Moreover, the computed p-values are all less than 0.01, indicating that the performance improvement of CSVD-TF compared to the three control approaches is statistically significant. This shows that by using the transfer learning method TrAdaBoost, CSVD-TF can outperform corresponding control approaches (i.e., utilizing only labeled SVD data from the source project, only a portion of labeled SVD data from the target project, or simply merging these two data). The comparison results demonstrate the effectiveness of considering the transfer learning method TrAdaBoost in CSVD-TF.

Summary for RQ2: CSVD-TF can significantly outperform corresponding control approaches, which achieves performance improvements of 18% to 30.4% in terms of AUC. This shows the effectiveness of considering the transfer learning method TrAdaBoost in CSVD-TF.

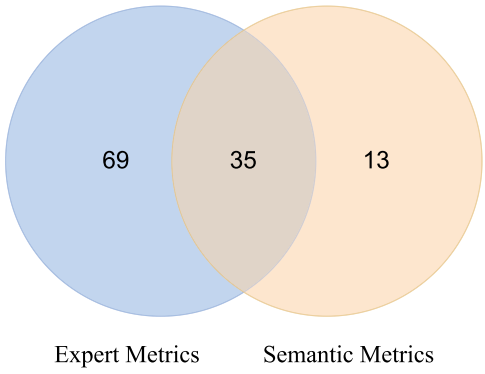

Fig. 2. The complementary analysis of the vulnerabilities correctly detected by CSVD-TF using only expert metrics or semantic metrics via the Venn diagram (the source project is set as LibTIFF and the target project is set as FFmpeg).

5.3. RQ3: Whether fusing both two types of metrics can further boost the performance of CSVD-TF?

Approach: To answer RQ3, we first aim to investigate whether these two types of metrics complement each other for CSVD. To achieve this goal, we first separately identify the vulnerabilities correctly detected by the CSVD model only using expert metrics or semantic metrics. Then we use a Venn diagram to visualize the complementary analysis results. If these two types of metrics show a certain degree of complementary, we further analyze whether fusing them can improve the performance of the CSVD model. Notice we employ the optimal model-level fusion in this RQ (which is discussed in Section 3.2.3).

Results: The complementary analysis result for a CSVD project combination can be found in Fig. 2. In this figure, we can observe that when the source project is set as LibTIFF and the target project is set as FFmpeg, only 35 vulnerable functions can be correctly identified by both types of metrics. Moreover, 69 vulnerable functions can be only correctly identified by CSVD-TF with expert metrics, and 13 vulnerable functions can be only correctly identified by CSVD-TF with semantic metrics. For other CSVD project combinations, we can achieve similar findings. These findings show that expert metrics or semantic metrics can capture different partial code information from different perspectives, which motivates us to design effective methods to fuse these two types of metrics.

Finally, we analyze the performance of CSVD-TF when only using one type of metric. Specifically, we use CSVD-TF-EM to denote the control approach, which only considers expert metrics. Then we use CSVD-TF-SM to denote the control approach, which only considers semantic metrics. The comparison results are shown in Table 5. This

Table 4
The influence of whether using TrAdaBoost on the performance of CSVD-TF in terms of AUC.

Source→Target	CSVD-TF	$CSVD_S$	$CSVD_T$	$CSVD_{ST}$
FFmpeg→LibPNG	**0.933**	0.499	0.778	0.605
FFmpeg→LibTIFF	**0.772**	0.507	0.610	0.577
FFmpeg→VLC	**0.658**	0.484	0.512	0.531
LibPNG→FFmpeg	**0.883**	0.498	0.619	0.647
LibPNG→LibTIFF	**0.768**	0.520	0.609	0.624
LibPNG→VLC	**0.610**	0.500	0.510	0.528
LibTIFF→FFmpeg	**0.881**	0.543	0.611	0.668
LibTIFF→LibPNG	**0.945**	0.503	0.807	0.789
LibTIFF→VLC	**0.646**	0.500	0.507	0.522
VLC→FFmpeg	**0.881**	0.500	0.618	0.604
VLC→LibPNG	**0.937**	0.500	0.763	0.700
VLC→LibTIFF	**0.782**	0.500	0.597	0.583
Average	**0.808**	0.504	0.628	0.615
p-value	–	**	**	**

Notes: *** means *p*-value < 0.001, ** means *p*-value < 0.01, * means *p*-value < 0.05.

Table 5
Comparison results between CSVD-TF only considering one type of metrics and CSVD-TF considering both types of metrics in terms of AUC.

Source→Target	CSVD-TF-EM	CSVD-TF-SM	CSVD-TF
FFmpeg→LibPNG	0.829	0.849	**0.933**
FFmpeg→LibTIFF	0.703	0.622	**0.772**
FFmpeg→VLC	0.525	0.591	**0.658**
LibPNG→FFmpeg	0.853	0.678	**0.883**
LibPNG→LibTIFF	0.672	0.618	**0.768**
LibPNG→VLC	0.539	0.577	**0.610**
LibTIFF→FFmpeg	0.860	0.685	**0.881**
LibTIFF→LibPNG	0.815	0.835	**0.945**
LibTIFF→VLC	0.528	0.606	**0.646**
VLC→FFmpeg	0.855	0.682	**0.881**
VLC→LibPNG	0.801	0.828	**0.937**
VLC→LibTIFF	0.721	0.598	**0.782**
Average	0.725	0.681	**0.808**
p-value	**	**	–

Notes: *** means *p*-value < 0.001, ** means *p*-value < 0.01, * means *p*-value < 0.05.

table shows that CSVD-TF with metric fusion achieves an average performance of 80.8%, which shows a performance improvement of 8.3% and 12.7%. The above results indicate that considering metric fusion is an effective way to improve CSVD model performance.

Summary for RQ3: CSVD-TF using expert metrics or semantic metrics shows a certain complementary for vulnerability detection. Therefore, by fusing both types of metrics, CSVD-TF can further boost the performance of CSVD.

5.4. RQ4: How different fusion methods influence the performance of CSVD-TF?

Approach: To answer RQ4, we compare the performance of CSVD-TF based on model-level fusion or metric-level fusion methods. Specifically, for metric-level fusion, we simply concatenate the expert metric vectors and semantic metric vectors, and input the concatenated vector for CSVD model training. For model-level fusion, we consider five different weight setting strategies for the CSVD model M_e and the CSVD model M_s (i.e., 0.3:0.7, 0.4:0.6, 0.5:0.5, 0.6:0.4, 0.7:0.3) to determine the contribution of each metric type in the metric fusion.

Results: Fig. 3 uses boxplot to show the performance of CSVD-TF using different fusion methods for all CSVD project combinations. In this figure, we can find that the model-level fusion outperforms

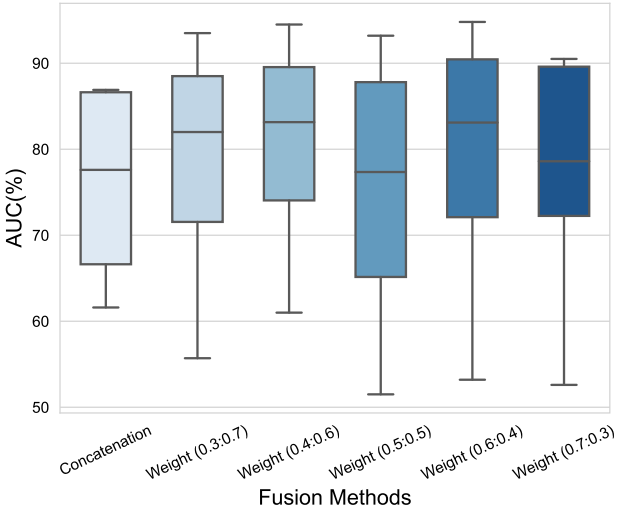

Fig. 3. The influence of using different fusion methods on the performance of CSVD-TF in terms of AUC.

the metric-level fusion. Among model-level fusion methods, when the weight ratio between the CSVD model M_e and the CSVD model M_s is 0.4:0.6, CSVD-TF achieves the best performance (i.e., 0.808) in terms of AUC. Moreover, according to the performance distribution of 10 independent runs, using this weight setting can achieve a relatively stable performance.

Summary for RQ4: Among different fusion methods, the model-level fusion with a weight of 0.4:0.6 between the CSVD model M_e and the CSVD model M_s can achieve the best and relatively stable performance for CSVD-TF.

5.5. RQ5: How does the class imbalance method SMOTE influence the performance of CSVD-TF?

Approach: To answer RQ5, we evaluate the contribution of the class imbalanced method SMOTE for CSVD-TF. Specifically, we consider CSVD-TF without using the class imbalanced method SMOTE as the control approach and other components remain unchanged for a fair comparison.

Results: The impact of using the class imbalanced method SMOTE on the performance of CSVD-TF is shown in Fig. 4. In this figure, we can

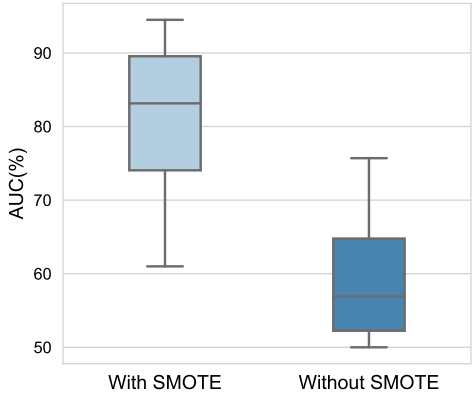

Fig. 4. The influence of using the class imbalanced method SMOTE on the performance of CSVD-TF in terms of AUC.

find that CSVD-TF with SMOTE can achieve better performance than CSVD-TF without SMOTE. Specifically, CSVD-TF with SMOTE achieves performance improvement ranging from 11% to 35.3% in terms of AUC. These results show that using SMOTE can effectively alleviate the class imbalanced problem in the SVD dataset and then improve the performance of CSVD-TF.

Summary for RQ5: The use of SMOTE for CSVD-TF can effectively alleviate the class imbalanced problem in CSVD and achieve a positive impact on the performance of CSVD-TF.

5.6. RQ6: How do different base classifiers influence the performance of CSVD-TF?

Approach: To answer RQ6, we select four popular machine learning classifiers to investigate the influence of base classifier selection on the performance of CSVD-TF, including eXtreme Gradient Boosting (XGBoost), Support Vector Machine (SVM), Decision Tree (DT), and Random Forest (RF). Specifically, XGBoost (Chen et al., 2015b) implements the gradient boosting framework and offers several useful features, like handling missing values, parallel processing, and regularization to avoid overfitting. SVM (Hearst et al., 1998) is a robust classifier that relies on finding the hyperplane that maximally separates different classes in the data. DT (Safavian and Landgrebe, 1991) is easy to understand, interpret, and visualize. RF (Breiman, 2001) is an ensemble classifier comprising multiple decision trees. These base classifiers have been widely used in previous vulnerability detection studies (Chernis and Verma, 2018; Chen et al., 2019, 2021; Zheng et al., 2020) and can cover different types of classifiers. In this RQ, we only modify the base classifier for CSVD-TF and other components remain unchanged for a fair comparison.

Results: Fig. 5 shows the performance of CSVD-TF using different base classifiers in terms of AUC. In this figure, we can find that across 12 CSVD project combinations, CSVD-TF with XGBoost can outperform CSVD-TF using other base classifiers (i.e., SVM, DT, or RF). Specifically, CSVD-TF with XGBoost achieves an average performance of 0.81 in terms of AUC, which outperforms CSVD-TF based on SVM, DT, and RF by 12.4%, 11.7%, and 13.8%, respectively.

Summary for RQ6: Selecting XGBoost as the base classifier of CSVD-TF can achieve the best performance for CSVD.

6. Discussion

6.1. The impact of selecting SVD data from the target project for labeling with different proportions on CSVD-TF

In this subsection, we aim to investigate the performance impact on CSVD-TF when selecting SVD data from the target project for labeling with different proportions. Notice we selected 50% of the SVD data from the target project as the test set in this study, which is different from the experimental setup in RQ1. Then we gradually increased the selection of the remaining SVD data for labeling (i.e., the proportions of all the SVD data in the target project for labeling are gradually selected as 20%, 30%, and 40% respectively). The detailed results are shown in Fig. 6. From this figure, we can find that when the proportion of the labeled SVD data in the target project increases from 20% to 30%, there is a significant improvement in the model's performance, with an average increase of 9.1%. However, when the proportion of labeled SVD data increases from 30% to 40%, the improvement in the model's performance is relatively small (i.e., only 2.3%). Considering the high costs for security experts to label these SVD data, selecting a 30% proportion of labeled SVD data in the target projects is a reasonable choice for this study and can achieve a good compromise. Moreover, by following this setting, CSVD-TF can achieve better performance than our considered baselines.

For some projects (such as FFmpeg and VLC) of our experimental subjects shown in Table 2, selecting 30% SVD data for labeling still needs a huge labeling effort. For example, for VLC → FFmpeg, we only consider 10% SVD data from VLC for labeling. Based on the comparison result shown in Fig. 7, we find that the performance of CSVD-TF still outperforms our considered five baselines (i.e., with the performance improvement from 3.9% to 31.9%). We can achieve the same finding for FFmpeg → VLC. These results show When the source project contains a large number of program modules, selecting 10% of the SVD data for labeling can also achieve good vulnerability detection performance.

6.2. Evaluation on MCC performance measure

In addition to AUC, we also consider the Matthews Correlation Coefficient (MCC) (Yao and Shepperd, 2020). MCC is a performance measure used to evaluate the performance of binary classification problems. It combines the quantities of true positives, true negatives, false positives, and false negatives. Similar to AUC, MCC is also suitable for assessing the performance of binary classification models in the presence of class imbalance.

The MCC ranges from −1 to 1, where 1 represents perfect prediction, 0 represents random prediction, and −1 represents completely opposite prediction. The formula for calculating MCC is as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (2)$$

where TP represents the number of true positives, TN represents the number of true negatives, FP represents the number of false positives, and FN represents the number of false negatives.

Fig. 8 shows the performance comparison between CSVD-TF and five baselines in terms of MCC. In the figure, we can find that our proposed approach CSVD-TF can outperform the baselines by 9% to 31.3% in terms of MCC. Moreover, the computed p-values are all less than 0.05, indicating that the performance improvement of CSVD-TF compared to the baselines is statistically significant.

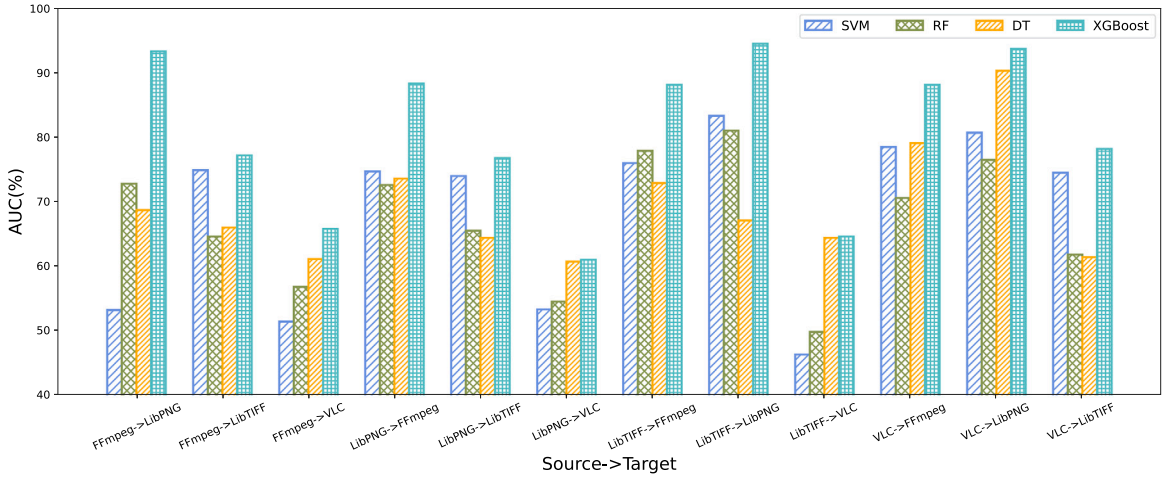

Fig. 5. The influence of using different base classifiers on the performance of CSVD-TF in terms of AUC.

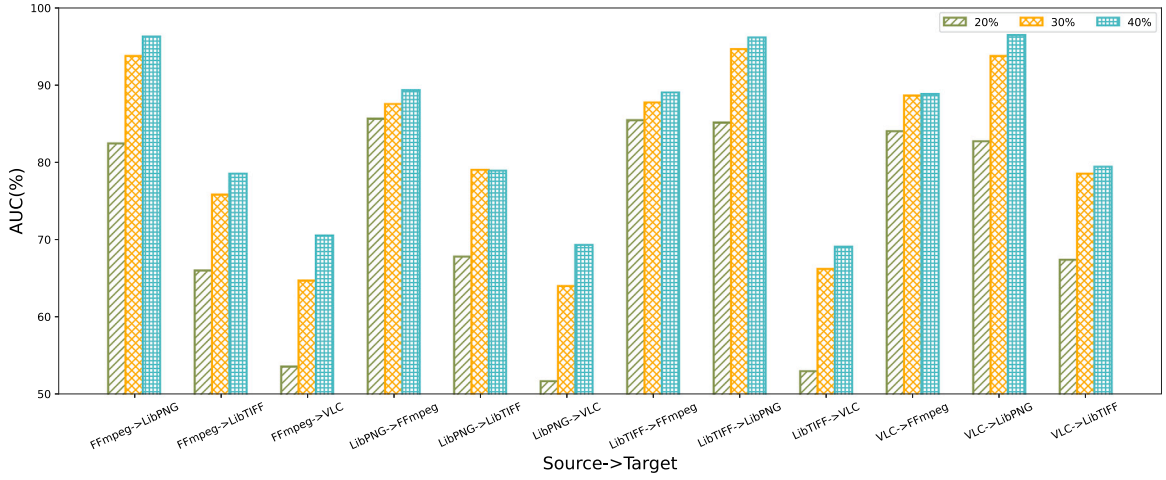

Fig. 6. The impact of selecting SVD data for labeling from the target project with different proportions on the performance of CSVD-TF.

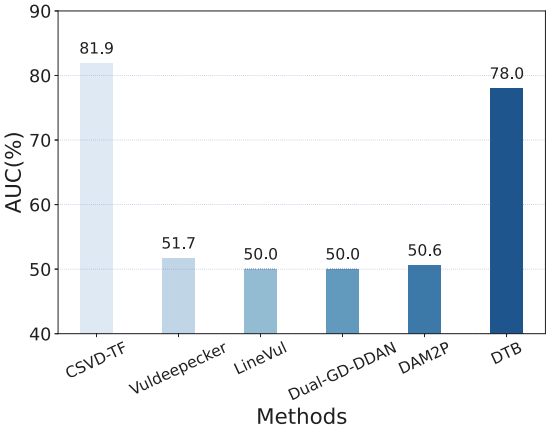

Fig. 7. The comparison results between CSVD-TF and five baselines when considering 10% SVD data for labeling, notice the source project is VLC and the target project is FFmpeg.

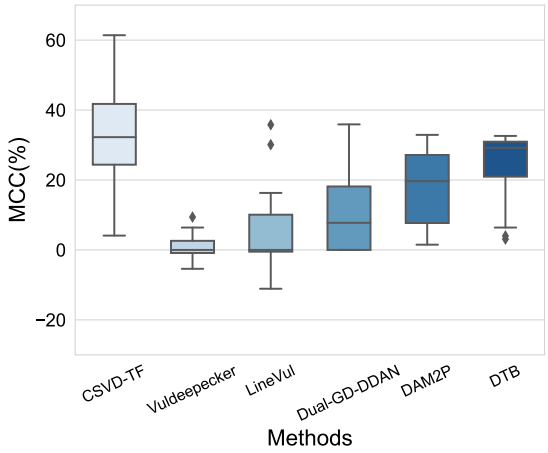

Fig. 8. The comparison results between CSVD-TF and five baselines in terms of MCC.

6.3. Evaluation on effort-aware performance measures

In addition to traditional performance measures, we also consider two effort-aware performance measures to evaluate the performance of CSVD models. Effort-aware performance measures evaluate the performance of vulnerability detection models by considering code inspection costs. Similar to previous software defect prediction studies (Ni et al., 2022; Chen et al., 2023), we consider LOC (lines of code) as the code inspection cost. Specifically, we consider two effort-aware measures:

Table 6
The comparison results between CSVD-TF and five baselines in terms of effort-aware performance measures.

Approach	R@20%E	P_{opt}
CSVD-TF	**0.440**	**0.650**
Vuldeepecker	0.210	0.481
LineVul	0.223	0.511
Dual-GD-DDAN	0.338	0.578
DAM2P	0.406	0.627
DTB	0.377	0.607

Recall@20%Effort (R@20%E) and P_{opt}. R@20%E measures the ratio between the number of vulnerabilities discovered at 20% of the code inspection cost and the total number of actual vulnerabilities. A higher R@20%E value indicates the ability to identify more vulnerabilities when spending 20% of the code inspection cost. P_{opt} is based on the Alberg plot and represents the relationship between the recall rate obtained by the vulnerability detection model and the code inspection cost for a specific vulnerability detection model. For this performance measure, the higher the value of P_{opt}, the better the trained vulnerability detection model. Table 6 shows the performance comparison between CSVD-TF and five baselines in terms of these two effort-aware measures. In the table, we can find that our proposed approach CSVD-TF can still outperform the baselines by 3.4% to 23% in terms of R@20%E, and by 2.3% to 16.9% in terms of P_{opt}. This indicates that CSVD-TF performs better in detecting more vulnerabilities when given the limited code inspection cost.

6.4. Threats to validity

Internal Threats. The first internal threat is related to the hyperparameter setting of our proposed approach CSVD-TF and baselines. To alleviate this threat, we adopted the hyperparameter setting of the baselines according to the setting in their responding studies. For our proposed approach CSVD-TF, we mainly followed the suggestions from the previous SVD studies (Li et al., 2020) and performed the hyperparameter optimization according to our empirical results. Notice we have not exhausted all possible hyperparameter values, but the hyperparameter values we use still outperform the baselines, and through hyperparameter optimization by considering more hyperparameter values, it is still possible to find better hyperparameter values. The second internal threat is that our selected experimental subjects have a class imbalanced problem. For these experimental subjects, the number of vulnerable functions is significantly smaller than the number of non-vulnerable functions. To alleviate this problem, we employ the class imbalanced method SMOTE and the effectiveness of using SMOTE can be verified in our ablation study (discussed in Section 5.5). The third internal threat is that we only selected a subset of SVD data from the target project for labeling and the performance improvement of CSVD-TF may be influenced by data selection. To alleviate this threat, we conduct ten independent SVD data selections from the target project with different random seeds and report the results of these ten independent runs. The final internal threat is the implementation faults in CSVD-TF and baselines. To alleviate this threat, we implemented the SVD baselines by their shared scripts. For CSVD-TF, we checked the code implementation by code inspection and used mature libraries.

External Threats. The threat to external validity is related to the experimental subjects we utilize. To alleviate this threat, we selected four real-world software projects as our experimental subjects. These experimental subjects have been considered in previous CSVD research (Nguyen et al., 2022, 2019), which can guarantee the confidence level of our empirical findings.

Construct Threats: The threat to construct validity is related to the performance measures. In our study, we mainly use AUC to evaluate the performance of our proposed approach CSVD-TF. The main reason is that this performance measure does not rely on arbitrarily selected thresholds and is not sensitive to the class imbalanced problem (Tantithamthavorn et al., 2018). Moreover, we also consider effort-aware performance measures (Yang et al., 2016; Chen et al., 2018, 2023), which consider the code inspection costs. In terms of these two effort-aware measures, CSVD-TF also shows competitiveness when compared to baselines.

Conclusion Threats: The first conclusion threat is the selection of baselines. To alleviate this threat, we first aim to select state-of-the-art CSVD baselines. In our study, we considered Dual-GD-DDAN (Nguyen et al., 2020) and DAM2P (Nguyen et al., 2022), which are recent CSVD approaches based on domain adaptation, as our baselines. We did not consider two baselines CD-VulD (Liu et al., 2020) and CPVD (Zhang et al., 2023b) due to the lack of the original data and completed reproducing source code from the authors. Moreover, we also choose DTB (Chen et al., 2015a) as our baseline since this approach investigates a similar scenario for cross-project defect prediction. The second conclusion threat is the hypothesis testing methods used to validate the statistical significance of performance differences among different approaches. To alleviate this threat, we used the Wilcoxon signed-rank test (Wilcoxon, 1992), which has also been widely used in previous similar studies (such as software defect prediction (Moussa and Sarro, 2022; Yuan et al., 2020)).

7. Related work

In this section, we analyze related studies from two perspectives: code representation learning in software vulnerability detection and cross-project software vulnerability detection. After analyzing these related studies, we emphasize the novelty of our study.

7.1. Code representation learning in software vulnerability detection

In the early studies in software vulnerability detection (SVD), researchers mainly used expert metrics (Neuhaus et al., 2007; Shin et al., 2010; Grieco et al., 2016; Walden et al., 2014; Chen et al., 2019, 2021), which were manually designed by security experts. For example, Shin et al. (2010) utilized expert metrics by considering code complexity, code churn, and developer activity. Walden et al. (2014) considered traditional software metrics inspired by traditional software defect prediction and text mining-based metrics for the modules of Web applications. However, these expert metrics may have the disadvantages of outdated experience, underlying biases, and low generalization. Therefore, expert metrics that work well in a certain software project may not perform well in other software projects. To alleviate these disadvantages of the expert metrics, the use of automatic semantic metrics (Dam et al., 2018; Ban et al., 2019; Li et al., 2018; Steenhoek et al., 2023b; Lin et al., 2020; Harzevili et al., 2023) has been studied recently. For example, Dam et al. (2018) employed RNN (Recurrent Neutral Network) to transform code token sequences to vector representations, which are further fed to traditional classifiers (such as support vector machine, or random forest) to construct SVD models. Recently, Steenhoek et al. (2023b) utilized the code pre-trained models (such as Code2Vec Alon et al., 2019, or CodeBERT Feng et al., 2020) to learn semantic metrics when conducting an empirical study of deep learning models for SVD. Except for treating the code as the token sequence, recent studies (Li et al., 2021a; Zhou et al., 2019; Cao et al., 2021; Cheng et al., 2021; Wen et al., 2023) came to treat the code as the graph. For example, Zhou et al. (2019) encoded a function into a joint graph structure from multiple syntax and semantic representations. Then they leveraged the composite graph representation for SVD. Wen et al. (2023) performed graph implications, which can reduce the node distance by shrinking the node sizes of code structure graphs.

Based on the above analysis, we can find researchers only consider one type of metric for SVD. In previous studies on software defect prediction, researchers (Ni et al., 2022; Chen et al., 2023) have confirmed

the effectiveness of fusing these two types of metrics in software defect prediction and localization, since these semantic metrics or expert metrics can only measure the program modules from different perspectives. However, to our best knowledge, whether fusing these two types of metrics can improve the performance for cross-project vulnerability detection has not been thoroughly investigated in previous studies.

7.2. Cross-project software vulnerability detection

One of the major challenges in software security vulnerability research is the scarcity of high-quality labeled SVD data for training models, since the process of labeling vulnerable source code is tedious, time-consuming, and error-prone even for security experts. However, some studies Lin et al. (2018) have accumulated some projects with the labeled SVD data. Therefore, a feasible solution is to perform transfer learning from the source project with the labeled SVD data to the target project and this problem is called cross-project vulnerability detection (CSVD). Liu et al. (2020) is the first study for CSVD and proposed the approach CD-VulD. CD-VulD utilized the Bi-LSTM network to map code sequences into a high-dimensional feature space. Then CD-VulD resorted to metric transfer learning to reduce the distribution difference between the source project and the target project, Nguyen et al. (2019) leveraged deep domain adaptation with automatic semantic metric learning and proposed the approach SCDAN. Then Nguyen et al. (2020) further proposed the approach Dual-GD-DDAN. This approach used a bidirectional recurrent neural network to extract semantic metrics. It has two different generators to obtain the code sequences for the source project and the target project. In addition, it also has two discriminators to discriminate these two different projects respectively. Later they Nguyen et al. (2022) proposed the approach DAM2P, which learns domain-invariant metrics and kernel methods with the max-margin principle. This approach can alleviate the gap between the source project and the target project on a joint space. Recently, Zhang et al. (2023b) proposed the approach CPVD. CPVD first uses the code property graph to represent the code. Then CPVD uses the graph attention network and convolution pooling network to extract the code vector representation. Finally, CPVD reduces the data distribution difference via domain adaptation representation learning.

However, in the previous CSVD studies, researchers only focused on semantic metrics while ignoring another crucial type of metrics (i.e., expert metrics). Moreover, they constructed CSVD models by only considering labeled SVD data in the source project. To alleviate these two limitations, we first investigate a more practical CSVD scenario. In this scenario, we select some program modules in the target project for labeling, which can improve CSVD model performance by effectively utilizing similar SVD data in the source project. We second confirm a certain complementary in vulnerability detection for these two types of metrics in our investigated scenario and utilize an effective model-level method to fuse these two types of metrics, which can further improve the performance of CSVD-TF.

8. Conclusion and future work

In previous cross-project vulnerability detection studies, researchers mainly utilized the labeled SVD data in the source project to construct the CSVD model. In this study, we aim to select a certain number of the program modules in the target project for labeling and investigate a more practical CSVD scenario. In this scenario, we propose a CSVD approach CSVD-TF, which can effectively utilize the labeled SVD data in the source project and a certain number of the labeled SVD data in the target project via TrAdaBoost. Moreover, to further improve the performance of CSVD-TF, we consider two types of metrics (i.e., expert metrics and semantic metrics) and perform metric fusion for CSVD-TF. To show the competitiveness of CSVD-TF, we consider four real-world projects as our experimental subjects. Final experimental results show that CSVD-TF can achieve better performance than our considered five baselines. A set of ablation studies also confirm the rationality of the component settings in CSVD-TF.

In the future, we first want to investigate the effectiveness of CSVD-TF for other programming languages (such as Java, and Python) by gathering datasets from corresponding large-scale real-world software projects. We second want to consider more advanced transfer learning methods for our investigated scenario and more advanced metric fusion methods. Third, we want to consider more advanced code representing learning methods (Steenhoek et al., 2023a; Zhang et al., 2023a; Cheng et al., 2022) to further improve the performance of CSVD-TF. Finally, we acknowledge our approach still requires data sharing among projects to train the CSVD models and does not fully consider protecting project confidentiality. To address this privacy issue, a possible solution is to employ federated learning (FL) (Yang et al., 2019; Yamamoto et al., 2023), a distributed machine learning approach that does not require data sharing.

CRediT authorship contribution statement

Zhilong Cai: Writing – review & editing, Validation, Software, Methodology, Data curation, Conceptualization. **Yongwei Cai:** Software, Methodology, Data curation, Conceptualization, Validation, Writing – review & editing. **Xiang Chen:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Guilong Lu:** Conceptualization, Data curation, Software. **Wenlong Pei:** Conceptualization, Data curation, Software. **Junjie Zhao:** Software, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. Zhilong Cai and Yongwei Cai have contributed equally to this work and they are co-first authors. Xiang Chen is the corresponding author. This work is supported in part by the National Natural Science Foundation of China (Grant no 62202419).

References

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), 1–29.

Ban, X., Liu, S., Chen, C., Chua, C., 2019. A performance evaluation of deep-learnt features for software vulnerability detection. Concurr. Comput.: Pract. Exper. 31 (19), e5103.

Breiman, L., 2001. Random forests. Mach. Learn. 45, 5–32.

Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136, 106576.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng..

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.

Chen, L., Fang, B., Shang, Z., Tang, Y., 2015a. Negative samples reduction in cross-company software defects prediction. Inf. Softw. Technol. 62, 67–77.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., et al., 2015b. Xgboost: extreme gradient boosting. pp. 1–4, R Package Version 0.4-2 1 (4).

Chen, X., Xia, H., Pei, W., Ni, C., Liu, K., 2023. Boosting multi-objective just-in-time software defect prediction by fusing expert metrics and semantic metrics. J. Syst. Softw. 206, 111853.

Chen, X., Yuan, Z., Cui, Z., Zhang, D., Ju, X., 2021. Empirical studies on the impact of filter-based ranking feature selection on security vulnerability prediction. IET Softw. 15 (1), 75–89.
Chen, X., Zhao, Y., Cui, Z., Meng, G., Liu, Y., Wang, Z., 2019. Large-scale empirical studies on effort-aware security vulnerability prediction methods. IEEE Trans. Reliab. 69 (1), 70–87.
Chen, X., Zhao, Y., Wang, Q., Yuan, Z., 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. Inf. Softw. Technol. 93, 1–13.
Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–33.
Cheng, X., Zhang, G., Wang, H., Sui, Y., 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. pp. 519–531.
Chernis, B., Verma, R., 2018. Machine learning methods for software vulnerability detection. In: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics. pp. 31–39.
Cortes, C., Vapnik, V., 1995. Support vector machine. Mach. Learn. 20 (3), 273–297.
Croft, R., Babar, M.A., Kholoosi, M.M., 2023. Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 121–133.
Dai, W., Yang, Q., Xue, G.-R., Yu, Y., 2007. Boosting for transfer learning. In: Proceedings of the 24th International Conference on Machine Learning. pp. 193–200.
Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2018. Automatic feature learning for predicting vulnerable software components. IEEE Trans. Softw. Eng. 47 (1), 67–85.
Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.
Gao, Z., Xia, X., Lo, D., Grundy, J., Zimmermann, T., 2021. Automating the removal of obsolete TODO comments. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 218–229.
Ghaffarian, S.M., Shahriari, H.R., 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. 50 (4), 1–36.
Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L., 2016. Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. pp. 85–96.
Harzevili, N.S., Belle, A.B., Wang, J., Wang, S., Ming, Z., Nagappan, N., et al., 2023. A survey on automated software vulnerability detection using machine learning and deep learning. arXiv preprint arXiv:2306.11673.
He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. 21 (9), 1263–1284.
Hearst, M.A., Dumais, S.T., Osuna, E., Platt, J., Scholkopf, B., 1998. Support vector machines. IEEE Intell. Syst. Appl. 13 (4), 18–28.
Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R.P., Tang, J., Liu, H., 2017. Feature selection: A data perspective. ACM Comput. Surv. (CSUR) 50 (6), 1–45.
Li, Y., Wang, S., Nguyen, T.N., 2021a. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303.
Li, K., Xiang, Z., Chen, T., Wang, S., Tan, K.C., 2020. Understanding the automated parameter optimization on transfer learning for cross-project defect prediction: an empirical study. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 566–577.
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. 19 (4), 2244–2258.
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.
Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: A survey. Proc. IEEE 108 (10), 1825–1848.
Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O., Montague, P., 2018. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inform. 14 (7), 3289–3297.
Liu, S., Lin, G., Qu, L., Zhang, J., De Vel, O., Montague, P., Xiang, Y., 2020. CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. IEEE Trans. Dependable Secure Comput. 19 (1), 438–451.
Liu, K., Yang, G., Chen, X., Zhou, Y., 2022. EL-CodeBert: Better exploiting CodeBert to support source code-related classification tasks. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware. pp. 147–155.
Moussa, R., Sarro, F., 2022. On the use of evaluation measures for defect prediction studies. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 101–113.
Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A., 2007. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 529–540.
Nguyen, V., Le, T., de Vel, O., Montague, P., Grundy, J., Phung, D., 2020. Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection. In: Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11–14, 2020, Proceedings, Part I 24. Springer, pp. 699–711.
Nguyen, V., Le, T., Le, T., Nguyen, K., DeVel, O., Montague, P., Qu, L., Phung, D., 2019. Deep domain adaptation for vulnerable code function identification. In: 2019 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–8.
Nguyen, V., Le, T., Tantithamthavorn, C., Grundy, J., Nguyen, H., Phung, D., 2022. Cross project software vulnerability detection via domain adaptation and max-margin principle. arXiv preprint arXiv:2209.10406.
Ni, C., Wang, W., Yang, K., Xia, X., Liu, K., Lo, D., 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. pp. 672–683.
Pan, S.J., Yang, Q., 2009. A survey on transfer learning. IEEE Trans. Knowl. Data Eng. 22 (10), 1345–1359.
Quinlan, J.R., 2014. C4. 5: Programs for Machine Learning. Elsevier.
Safavian, S.R., Landgrebe, D., 1991. A survey of decision tree classifier methodology. IEEE Trans. Syst. Man Cybern. 21 (3), 660–674.
Shin, Y., Meneely, A., Williams, L., Osborne, J.A., 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 37 (6), 772–787.
Steenhoek, B., Gao, H., Le, W., 2023a. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In: 2024 IEEE/ACM 46th International Conference on Software Engineering. ICSE, IEEE Computer Society, pp. 166–178.
Steenhoek, B., Rahman, M.M., Jiles, R., Le, W., 2023b. An empirical study of deep learning models for vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2237–2248.
Su, J., Cao, J., Liu, W., Ou, Y., 2021. Whitening sentence representations for better semantics and faster retrieval. arXiv preprint arXiv:2103.15316.
Tantithamthavorn, C., Hassan, A.E., Matsumoto, K., 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Softw. Eng. 46 (11), 1200–1219.
Walden, J., Stuckman, J., Scandariato, R., 2014. Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 23–33.
Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2275–2286.
Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.
Wold, S., Esbensen, K., Geladi, P., 1987. Principal component analysis. Chemometr. Intell. Lab. Syst. 2 (1–3), 37–52.
Yamamoto, H., Wang, D., Rajbahadur, G.K., Kondo, M., Kamei, Y., Ubayashi, N., 2023. Towards privacy preserving cross project defect prediction with federated learning. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 485–496.
Yang, Q., Liu, Y., Chen, T., Tong, Y., 2019. Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol. 10 (2), 1–19.
Yang, X., Wang, S., Li, Y., Wang, S., 2023a. Does data sampling improve deep learning-based vulnerability detection? yeas! and nays!. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2287–2298.
Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2023b. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 197, 111577.
Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 157–168.
Yao, J., Shepperd, M., 2020. Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters. In: Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering. pp. 120–129.
Yu, C., Yang, G., Chen, X., Liu, K., Zhou, Y., 2022. Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In: 2022 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 82–93.
Yuan, Z., Chen, X., Cui, Z., Mu, Y., 2020. ALTRA: Cross-project software defect prediction via active learning and tradaboost. IEEE Access 8, 30037–30049.
Zagane, M., Abdi, M.K., Alenezi, M., 2020. Deep learning for software vulnerabilities detection using code metrics. IEEE Access 8, 74562–74570.
Zhang, J., Liu, Z., Hu, X., Xia, X., Li, S., 2023a. Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans. Softw. Eng..

Zhang, C., Liu, B., Xin, Y., Yao, L., 2023b. CPVD: Cross project vulnerability detection based on graph attention network and domain adaptation. IEEE Trans. Softw. Eng..
Zheng, W., Gao, J., Wu, X., Liu, F., Xun, Y., Liu, G., Chen, X., 2020. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. J. Syst. Softw. 168, 110659.
Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.
Zou, D., Wang, S., Xu, S., Li, Z., Jin, H., 2019. MuVulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput. 18 (5), 2224–2236.

Zhilong Cai is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include automatic vulnerability detection.

Yongwei Cai is currently pursuing the Bachelor degree at the School of Information Science and Technology, Nantong University. His research interests include automatic vulnerability detection.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the Department of Information Science and Technology, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Software Testing, Verification and Reliability, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software - Practice and Experience, Automated Software Engineering, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER). His research interests include software engineering, in particular software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information about him can be found at: https://xchencs.github.io/index.html.

Guilong Lu is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include automatic vulnerability detection.

Wenlong Pei is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include automatic vulnerability detection.

Junjie Zhao is currently pursuing the Master degree at the School of Information Science and Technology, Nantong University. His research interests include software repository mining.