The Journal of Systems and Software 166 (2020) 110585

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Automated defect identification via path analysis-based features with transfer learning

Yuwei Zhang[a,*], Dahai Jin[a], Ying Xing[b], Yunzhan Gong[a]

[a] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China
[b] Automation School, Beijing University of Posts and Telecommunications, China

ARTICLE INFO

Article history:
Received 9 September 2019
Revised 16 February 2020
Accepted 19 March 2020
Available online 5 April 2020

Keywords:
Machine learning
Automated defect identification
Path analysis
Transfer learning
Model evaluation

ABSTRACT

Recently, artificial intelligence techniques have been widely applied to address various specialized tasks in software engineering, such as code generation, defect identification, and bug repair. Despite the diffuse usage of static analysis tools in automatically detecting potential software defects, developers consider the large number of reported alarms and the expensive cost of manual inspection to be a key barrier to using them in practice. To automate the process of defect identification, researchers utilize machine learning algorithms with a set of hand-engineered features to build classification models for identifying alarms as actionable or unactionable. However, traditional features often fail to represent the deep syntactic structure of alarms. To bridge the gap between programs' syntactic structure and defect identification features, this paper first extracts a set of novel fine-grained features at variable-level, called path-variable characteristic, by applying path analysis techniques in the feature extraction process. We then raise a two-stage transfer learning approach based on our proposed features, called feature ranking-matching based transfer learning, to increase the performance of cross-project defect identification. Our experimental results for eight open-source projects show that the proposed features at variable-level are promising and can yield significant improvement on both within-project and cross-project defect identification.

1. Introduction

Software has been integrated into our everyday lives, which makes improving software quality an increasingly critical challenge for software developers. Recently, research at the intersection of software engineering and artificial intelligence has emerged as an important means to address these challenges. In intelligent software engineering, machine learning techniques have been frequently applied to propose learnable probabilistic models of source code that mine the patterns of code. And these techniques have been widely adopted in practice and proven to help resolve various software engineering problems (Wang et al., 2016; Hu et al., 2018a; 2018b). Thus, instilling intelligence in solutions for software engineering problems has attracted a lot of attention.

Static analysis (SA) tools (e.g., Defect Testing System (DTS) (Yang et al., 2008)) are designed to automatically detect source code defects that might jeopardize the security and performance of software systems. The diffuse usage of SA tools provides solid evidence that SA techniques are of great significance to aid developers. However, the large number of alarms reported and cost incurred in their manual inspection may be regarded as a key barrier to using SA tools in practice (Muske and Serebrenik, 2016; Koc et al., 2019). Since the program under analysis is not executed, SA tools are required to speculate on how the program will behave actually (Ruthruff et al., 2008). That is, the output of SA tools becomes imprecise. As a consequence, developers need to manually sift through a plethora of reported alarms to partition them into true defects and false positives.

To automate the process of defect identification, numerous approaches have been proposed for handling SA alarms (Heckman and Williams, 2011; Muske and Serebrenik, 2016). Recently, most research has applied machine learning techniques in this area to automatically identify SA alarms by using classification algorithms. Aiming at learning patterns of false positives that are difficult to observe by traditional SA approaches, machine learning techniques thereby can greatly reduce the cost of manual inspection and improve the benefit ratio of using SA tools in practice (Koc et al., 2017; Raghothaman et al., 2018).

In general, automated identification of SA alarms can be framed as a standard binary classification problem, which shares a common procedure. First and foremost, these approaches come up

[*] Corresponding author.
E-mail address: hyun@bupt.edu.cn (Y. Zhang).

https://doi.org/10.1016/j.jss.2020.110585

with a set of hand-engineered features that are based on static code metrics, churn data and defect information. Then, each alarm is transformed into one vector with designed features via a mapping function. Vectors with labels (true defect or false positive) are used to train machine learning classifiers. Finally, trained models are utilized to predict new reported SA alarms as actionable (true defects) or unactionable (false positives). Existing features at either file-level or method-level mainly focus on the statistical characteristics of the source code under analysis and presume that actionable and unactionable alarms have distinguishable statistical characteristics. However, empirical results (Wang et al., 2016; Zhang et al., 2020; Koc et al., 2019) indicate that these features lack precision in representing the deep syntactic structure of alarms and cannot distinguish alarms with different semantics. To exemplify, the feature vectors of alarms with opposite manual inspection results can be similar or even identical when using traditional features, inevitably leading to a loss of accuracy.

To bridge the gap between the reported alarms syntactic structure and features used for defect identification, Zhang et al. (2020) proposed a set of novel features at variable-level, named variable characteristic (VC). These features were extracted from the source code of variables that cause potential defects. The experimental results demonstrated that features at variable-level improve the performance significantly in automatically identifying true defects from the reported SA alarms. Given that only the variables that cause potential defects are considered in the feature extraction process, however, further study is called for.

Different from traditional hand-engineered feature extraction approaches, we utilize path analysis techniques in this paper during the feature extraction process. Existing studies (Le and Soffa, 2007; Zhao et al., 2011; Fan et al., 2019) have shown that path-sensitivity analysis is a commonly used technique for false positives elimination when detecting defects. Therefore, extracting features from a concrete specific path consisting related defect information would be helpful to refine the process of feature extraction. Firstly, we use a target-oriented path generation algorithm to produce an intraprocedural path from the control flow graph (CFG) of the function that contains potential defects. Secondly, the irrelevant path nodes are filtered out by path slicing, instead of computing a backward slice in the program directly (Zhang et al., 2020). Finally, our approach put forward a set of extended VC features, named path-variable characteristic (PVC), extracted from the path nodes.

Additionally, there is a major challenge in raising the accuracy of identifying defects from new software projects since it is difficult to build a machine learning classifier without labeled training instances. To address this limitation, researchers have applied transfer learning-based approaches (Nam et al., 2013; 2018) in classifying unlabeled defects from new projects. The key idea of these approaches is aiming to extract common knowledge from one source project and transfer it to another target project, and a classification model is trained with labeled defects in the source project to predict defects in the target project.

In this paper, we raise a two-stage transfer learning approach, called feature ranking-matching based transfer learning (FRM-TL), to identify defects across projects based on the proposed PVC features. Feature ranking aims to find a minimized feature space for the dataset of the evaluated projects while preserving the original data properties, and then the selected features with similar distributions are matched between source and target projects.

Specifically, we investigate the following research questions:

- RQ1: Do PVC features outperform traditional features for within-project defect identification (WPDI)?
- RQ2: How do PVC features perform in cross-project defect identification (CPDI)?

To answer the two research questions, we conduct an empirical study on the open-source projects and implement a tool to extract the values of the PVC features for each project to prepare the experimental dataset. The main findings of our work include:

- Our experimental results on RQ1 (with details described in Section 5.1) demonstrate that the performance of WPDI when using PVC features is promising. By comparing the traditional features, the PVC features could achieve better WPDI results with statistical significance.
- Our experimental results on RQ2 (with details described in Section 5.2) demonstrate that our proposed transfer learning model FRM-TL is feasible and its CPDI performance is better or comparable to other baseline cross-project models with statistical significance, that is, the PVC features can achieve reasonable performance in CPDI tasks.

The main contributions of this paper are presented as follows:

- A set of novel fine-grained features, named path-variable characteristic, are proposed based on path analysis for both WPDI and CPDI.
- An effective transfer learning model, called feature ranking-matching based transfer learning, is introduced for CPDI.
- Experiments are conducted on eight open-source projects to evaluate the performance of our approach in both WPDI and CPDI.

The remainder of this paper is organized as follows. We survey related work in Section 2. Section 3 introduces in detail the proposed approach. We provide the experimental setup in Section 4. Section 5 shows the analyzing results of our research. We disclose the threats to the validity of our research in Section 6. Section 7 concludes this paper and presents the future work.

2. Related work

Two lines of research are most related to the work described in this paper. First, we discuss the related work of using machine learning techniques in identifying software defects. Second, we briefly introduce the related work of applying transfer learning methods in cross-project defect prediction (CPDP) scenario since no study has proposed such methods for CPDI scenario.

2.1. Defect identification using machine learning

To date, most approaches have proposed hand-engineered features (Wang et al., 2018) to learn distinguishable patterns between true defects and false positives. The principle of defect identification using machine learning is to train a model based on the proposed features and apply the model to identify new alarm as either actionable or unactionable. The unactionable alarms are not reported to the developers for these alarms are more likely to be false positives. Ruthruff et al. (2008) proposed a logistic regression model based on 33 features extracted from the alarms themselves to predict actionable alarms found by FindBugs, and a screening methodology was used to quickly discard features with low predictive power in order to build cost-effectively predictive models. Heckman and Williams (2009) evaluated 15 machine learning algorithms based on distinct sets of alarms characteristics out of 51 candidate features, which is one of the most comprehensive studies in identifying actionable alarms and achieves high performance. Yoon et al. (2014) used the abstract syntax trees to represent structural characteristics and classified new alarms using the probability computed by a support vector machine classifier.

Flynn et al. (2018) developed and tested four classification models for SA alarms mapped to CERT rules, using a novel combination of multiple SA tools and 28 features extracted from the alarms. Koc et al. (2019) applied neural networks to automatically learn vector representation from the SA report texts for identifying false positive SA alarms.

To the best of our knowledge, there are no prior studies of designing a set of hand-engineered features at variable-level based on path analysis techniques. In this paper, we utilize CFG to generate paths for extracting syntactic information from the source code files and leverage the extracted PVC features to build machine learning models for automatically identifying software defects. The detailed feature extraction process will be described in Section 3.2.

2.2. Cross-project defect prediction using transfer learning

Due to the lack of labeled training instances, it is usually hard to build accurate models for CPDP. Thus, researchers have proposed various approaches based on transfer learning to improve the poor performance of CPDP (Li et al., 2018; Herbold et al., 2018; Hosseini et al., 2019). The underlying idea of CPDP using transfer learning is to neutralize the negative effect of different data distributions between source and target projects by extracting knowledge from one source project and applying the learned knowledge to the target one during the training and prediction processes. Watanabe et al. (2008) conducted CPDP experiments on two software projects with the same feature set by using the average feature values to compute the similarity between the source and target projects. Turhan et al. (2009) proposed the nearest-neighbour filter to collect instances in the source project that are the nearest-neighbours of instances in the target project for building prediction models. Ma et al. (2012) used the Transfer Naive Bayes algorithm to build a defect prediction model by weighting instances between the source and target projects. Herbold (2013) computed the Euclidean distance of the data distributions between source and target projects, in order to select better source projects for improving the CPDP performance. On the similar lines, Liu et al. (2019) utilized two built supervised regression models to measure the similarity of data distributions between source and target projects. Ryu et al. (2017) proposed a transfer cost-sensitive boosting method for CPDP, which is a combination of transfer learning and class imbalance learning. The proposed approach performs boosting that assigns the similarity weight between the source and target projects based on the distributional characteristics and the class imbalance. However, the boosting method requires a small number of labeled target instances, which hinders its usage in the CPDP scenario that the target project has no historical labeled instances. To improve the performance of CPDP, Xu et al. (2019) raised a novel balanced distribution adaptation based transfer learning method, simultaneously considering the marginal and conditional distribution differences and adaptively assigning different weights to them. Nam et al. (2013) proposed a state-of-the-art transfer learning model called TCA+ extended from Transfer Component Analysis (TCA) technique by optimizing the data normalization process. This approach is to find a shared latent space for both source and target projects, where their data distributions are similar. Experimental results showed TCA+ was a promising method to improve CPDP. Additionally, Qiu et al. (2019) adopted TCA to learn transferable joint features from both deep learning-generated and hand-engineered features across projects, which enhances the transferability of hybrid features in CPDP scenarios.

This research mainly draws the idea from (Nam et al., 2013; 2018) and focuses on the application of CPDI scenario. We first propose a top ranking-based approach to select a PVC subset in the source project. And then the selected PVC features from source project are matched to the target project based on the similarity of feature distribution for building CPDI models. In Section 3.4, we will explain our proposed transfer learning method in detail.

3. Methodology

Fig. 1 overviews the procedure of automatically identifying defects proposed in this paper. Our approach consists of four major steps: 1) collecting SA alarm instances and labeling them; 2) applying path analysis techniques to extract fine-grained features from source code; 3) mapping features into integer vectors for building both WPDI and CPDI models; 4) presenting a two-stage transfer learning model based on feature ranking and matching to raise the accuracy of CPDI. In the following subsections, we describe the proposed approaches in detail.

3.1. Data preparation

The evaluated projects are originally analyzed statically by the SA tool, and then, the reported alarms are inspected manually by the developers in order to label each alarm as either **TRUE** (actionable) or **FALSE** (unactionable). To label a reported SA alarm, developers need to review the related code according to defect information. If a data-flow from an untrusted source without any sanity check can be detected by the developers, the reported alarm is a true defect. Otherwise, the reported alarm is a false positive. After that, we use the CFG, a general model to represent the execution flow between statements in a given function, to produce paths for extracting syntactic information from source code files, which is fully explained in the next subsection.

3.2. Path analysis-based feature extraction

For the sake of easy explanation, we introduce a C language function `str_add_str` from `spell` shown in Fig. 2 as a motivating example. Zhang et al. (2020) used line 117 as the seed statement to compute a backward slice for variable `str1` in the function directly, and then a set of statements at lines {114,115,117} were produced for feature extraction. However, we can easily find in the code that variable `str1` at line 117 is contained in a **for** loop structure, that is to say, variable `str1` is influenced by the predicate within the **for** statement at line 116. Thus, variables `pos` and `str2` should also be considered as the causes of resulting in the potential null pointer dereference (NPD) defect at line 117. But our previous approach (Zhang et al., 2020) is incompetent to extract the syntactic information of variables `pos` and `str2` due to the limitation of path-insensitiveness. To refine the process of feature extraction in this paper, we extract features from the generated specific paths containing related defect information to achieve path sensitive analysis.

3.2.1. Target-oriented path generation algorithm

As is shown in Fig. 3, the proposed approach is performed by procedure *generatingOneIntraPath*, which is a target-oriented path generation algorithm (Zhang et al., 2017). According to the given coverage criteria, the algorithm firstly generates an initial path `keyPath` covering the input target from the CFG of the given function. The initial path consists of a sequence of nodes from the entry node of the CFG to the target node. This step is repeated until a feasible `keyPath` is selected. The feasibility of `keyPath` is determined by the interval arithmetic technique implemented in function `isIntraPathFeasible`, which is a data flow analysis-based approach that computes the value interval of each variable in the path nodes to check its feasibility. Secondly, since the nodes of `keyPath` may be discontinuous on the CFG, the backtracking search strategy is employed for filling the path segment `keyPath`

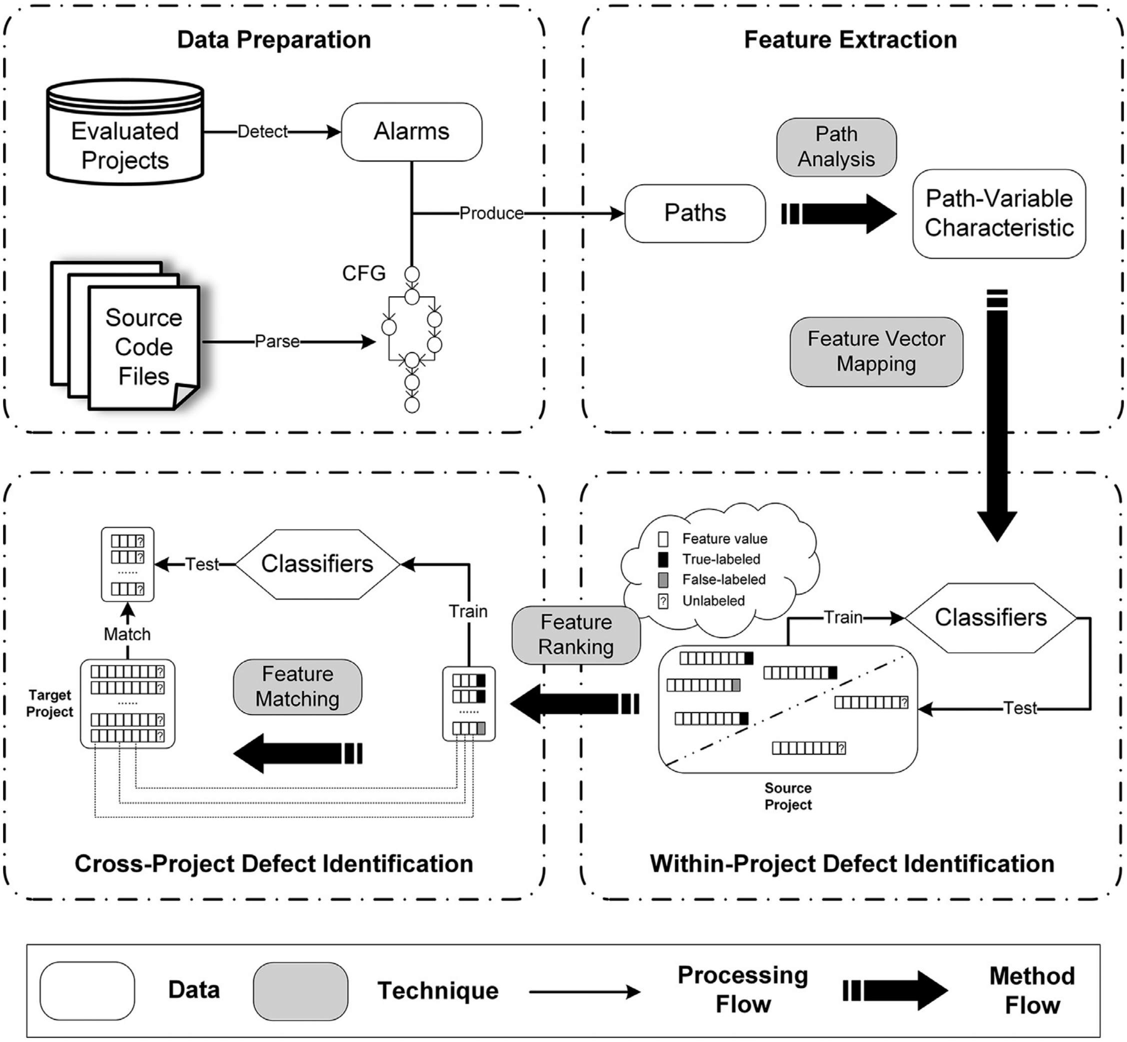
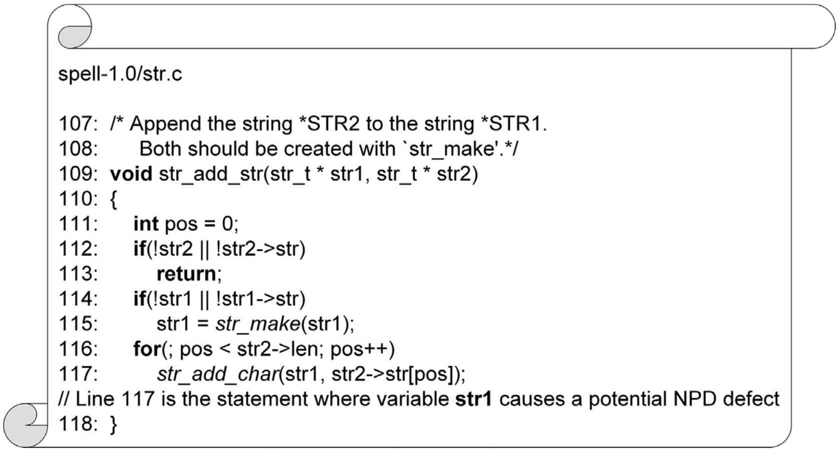


Fig. 1. Automated defect identification process.



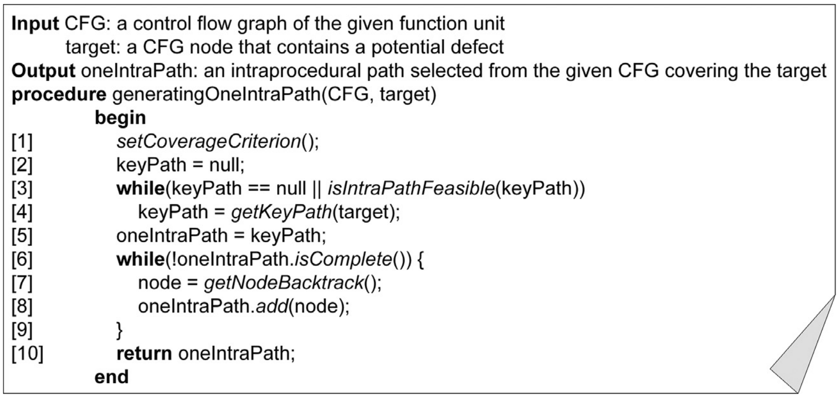
```
spell-1.0/str.c

107:  /* Append the string *STR2 to the string *STR1.
108:       Both should be created with `str_make'.*/
109:  void str_add_str(str_t * str1, str_t * str2)
110:  {
111:      int pos = 0;
112:      if(!str2 || !str2->str)
113:          return;
114:      if(!str1 || !str1->str)
115:          str1 = str_make(str1);
116:      for(; pos < str2->len; pos++)
117:          str_add_char(str1, str2->str[pos]);
// Line 117 is the statement where variable str1 causes a potential NPD defect
118:  }
```

Fig. 2. A motivating example.

as a complete intraprocedural path oneIntraPath. Finally, the algorithm returns a complete intraprocedural path covering the target node that contains the potential defect.

3.2.2. Path-variable characteristics

For our paper, we first utilize a target-oriented path generation algorithm to produce one intraprocedural path for each reported alarm to extract syntactic information from source code files. When analyzing the generated paths, each path node can be matched to one type of abstract syntax tree (AST) nodes. Specifically, four types of AST nodes are selected as tokens: 1) declaration nodes such as type declarations, method declarations, etc; 2) method invocation nodes; 3) assignment nodes; 4) control-flow nodes, i.e., **if** statements, **for** statements, and **while** statements.



```
Input CFG: a control flow graph of the given function unit
      target: a CFG node that contains a potential defect
Output oneIntraPath: an intraprocedural path selected from the given CFG covering the target
procedure generatingOneIntraPath(CFG, target)
         begin
            setCoverageCriterion();
            keyPath = null;
            while(keyPath == null || isIntraPathFeasible(keyPath))
               keyPath = getKeyPath(target);
            oneIntraPath = keyPath;
            while(!oneIntraPath.isComplete()) {
               node = getNodeBacktrack();
               oneIntraPath.add(node);
            }
            return oneIntraPath;
         end
```

Fig. 3. Algorithm for generating one intraprocedural path covering the defect target.

Table 1
Overview of the extracted PVC features.

ID	Name	Description
PVC$_1$	IP_LOC	number of source code statements counted from the definition statement of the IP variable to the statement containing a potential defect
PVC$_2$, **PVC**$_3$	{Method, File}_LOC	number of source code statements in *method, file*
PVC$_4$	Var_Type	type of the related variable(e.g., integer variable)
PVC$_5$	Var_Property	scope of the related variable(e.g., global variable)
PVC$_6$	Var_Init	initial status of the related variable after defined
PVC$_7$ − **PVC**$_{10}$	Def_IP_{C, V, Lib, User}	IP variable assigned by *a **constant**value, a **variable**value, a **library function**return value, an **user-defined function**return value*
PVC$_{11}$ − **PVC**$_{14}$	Def_Other_{C, V, Lib, User}	other variables assigned by *a **constant**value, a **variable**value, a **library function**return value, an **user-defined function**return value*
PVC$_{15}$ − **PVC**$_{18}$	Def_If_{C, V, Lib, User}	related variables assigned by *a **constant**value, a **variable**value, a **library function**return value, an **user-defined function**return value* within an **if** structure
PVC$_{19}$ − **PVC**$_{22}$	Def_For_{C, V, Lib, User}	related variables assigned by *a **constant**value, a **variable**value, a **library function**return value, an **user-defined function**return value* within a **for** structure
PVC$_{23}$ − **PVC**$_{26}$	Def_While_{C, V, Lib, User}	related variables assigned by *a **constant**value, a **variable**value, a **library function**return value, an **user-defined function**return value* within a **while** structure
PVC$_{27}$ − **PVC**$_{29}$	Use_IP_{V, Lib, User}	IP variable referenced by *a **variable**value, a **library function**, an **user-defined function***
PVC$_{30}$ − **PVC**$_{32}$	Use_IP_{If, For, While}	IP variable referenced by the predicate in *an **if**statement, a **for**statement, a **while**statement*
PVC$_{33}$ − **PVC**$_{35}$	Use_Other_{V, Lib, User}	other variables referenced by *a **variable**value, a **library function**, an **user-defined function***
PVC$_{36}$ − **PVC**$_{38}$	Use_Other_{If, For, While}	other variables referenced by the predicate in *an **if**statement, a **for**statement, a **while**statement*
PVC$_{39}$ − **PVC**$_{41}$	Use_If_{V, Lib, User}	related variables referenced by *a **variable**value, a **library function**, an **user-defined function*** within an **if** structure
PVC$_{42}$ − **PVC**$_{44}$	Use_For_{V, Lib, User}	related variables referenced by *a **variable**value, a **library function**, an **user-defined function*** within a **for** structure
PVC$_{45}$ − **PVC**$_{47}$	Use_While_{V, Lib, User}	related variables referenced by *a **variable**value, a **library function**, an **user-defined function*** within a **while** structure

Then, we exclude the irrelevant path nodes by computing a backward path slicing, and two types of path nodes remain for feature extraction: 1) the nodes containing the inspecting point (IP) variable that results in the reported alarm; 2) the nodes containing other variables that influence the IP variable. Finally, a set of fine-grained features at variable-level are extracted based on the selected tokens and the def-use relationship of the related variables (both IP variable and other variables) in the remaining path nodes. Additionally, we adopt the LOC-based metrics used in (Zhang et al., 2020) to improving the generalization performance of our proposed model. Table 1 lists the extracted PVC features with detailed description.

Fig. 4 demonstrates an example of extracting PVC features from the motivation code in Fig. 2. The path node n_7 is used as the defect target node for the path generation algorithm, which will produce an intraprocedural path $p = entry \to n_1 \to n_2 \to n_4 \to n_6 \to n_7$. In this case, `str1` is considered as the IP variable while `pos` and `str2` are considered as other variables, and then seven PVC features based on the path analysis will be extracted from the motivation code.

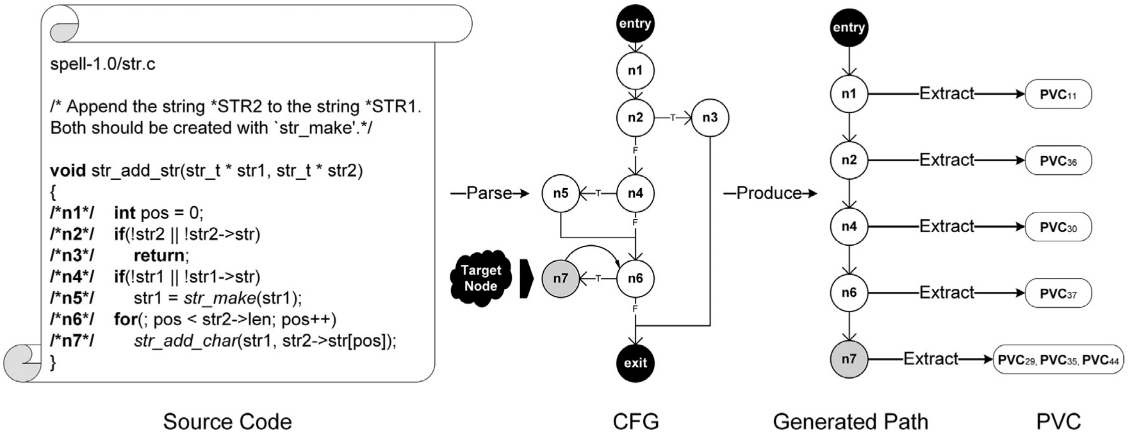
3.3. Feature vector mapping and postprocessing

To train machine learning classifiers for defect identification, we first build a mapping between integer values and PVC features, and encode reported alarms to integer feature vectors. Since the extracted PVC features are distinct for each reported alarm, the integer feature vector of each reported alarm may have different lengths. Thus, we append 0 to PVC features that are not extracted for each reported alarm and make the lengths of all integer feature vectors consistent. The mapping process can be represented as the following mapping function:

$$\mathbb{A} \longmapsto \mathbb{FV} = (\mathbb{PVC}, R(\mathbb{A})) \tag{1}$$

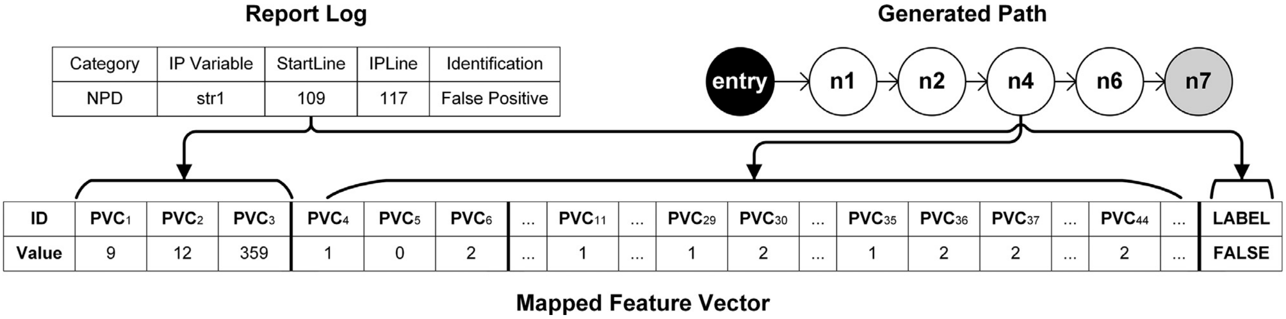
where $\mathbb{A}$ is the alarm reported by the SA tool, $\mathbb{FV}$ is the mapped integer feature vector, $\mathbb{PVC}$ is a list of integer numbers denoting the value of the 47 PVC features extracted from $\mathbb{A}$, and $R(\mathbb{A})$ is the manual inspection result of $\mathbb{A}$, which is labeled either **TRUE** (actionable) or **FALSE** (unactionable).

Fig. 5 demonstrates an example of constructing a feature vector from the code in Fig. 2. The upper left corner of the figure is the alarm's report log obtained from the SA tool, and the upper right corner of the figure is an intraprocedural path generated from the



```
spell-1.0/str.c

/* Append the string *STR2 to the string *STR1.
Both should be created with `str_make'.*/

void str_add_str(str_t * str1, str_t * str2)
{
/*n1*/    int pos = 0;
/*n2*/    if(!str2 || !str2->str)
/*n3*/        return;
/*n4*/    if(!str1 || !str1->str)
/*n5*/        str1 = str_make(str1);
/*n6*/    for(; pos < str2->len; pos++)
/*n7*/        str_add_char(str1, str2->str[pos]);
}
```

Fig. 4. An example of feature extraction.



Report Log

Category	IP Variable	StartLine	IPLine	Identification
NPD	str1	109	117	False Positive

ID	PVC1	PVC2	PVC3	PVC4	PVC5	PVC6	...	PVC11	...	PVC29	PVC30	...	PVC35	PVC36	PVC37	...	PVC44	...	LABEL
Value	9	12	359	1	0	2	...	1	...	1	2	...	1	2	2	...	2	...	FALSE

Mapped Feature Vector

Fig. 5. An example of feature vector construction.

CFG of the code in Fig. 2. The ellipses in the sample feature vector represent the PVC features that value 0.

Feature vectors usually suffer from mislabeling issue (Tantithamthavorn et al., 2015). Therefore, defect identification models trained on noisy data may be unreliable. To prune mislabeling data, we use the *RemoveMisclassified* filter implemented in Weka (Witten et al., 2016), an open-source software developed for data mining tasks, to eliminate the instances that are incorrectly classified. The *majority voting* classifier implemented in (Zhang et al., 2020) is utilized to base the misclassifications. Since our objective is not to find the best training set, we use the default parameters in Weka and save the effort of well tuning the parameters.

3.4. Feature ranking-matching based transfer learning

The lower part of Fig. 1 demonstrates the overview of CPDI based on the two-stage transfer learning approach. In the figure, we have two datasets from the source project and the target project with homogeneous feature sets. Each row and column of a dataset indicates a alarm and a PVC feature respectively, and the last column represents the label of each alarm.

We first apply the feature selection technique to the source project during the feature ranking process. Feature selection, selecting a subset of features using feature evaluator with one search method, is of great importance to avoid reducing classifier performance because of redundant and irrelevant features. We apply widely used feature selection techniques to rank the PVC features in order to find a minimized feature space that preserves the original data properties of the source project. After that, selected PVC features from the source project are matched up to the PVC features with similar distribution from the target project.

In the following subsections, we explain the feature ranking and feature matching in detail.

3.4.1. Feature ranking in the source project

According to several benchmark studies of feature selection techniques (Catal and Diri, 2009; Gao et al., 2011; Xu et al., 2016; 2017; Ghotra et al., 2017), the impact of feature selection techniques varies across the trained models, and the overlap of features selected by different feature selection techniques is low in most cases. Additionally, empirical results (Xu et al., 2016) indicate that ranking-based feature selection methods can achieve acceptable results with fewer features and less time compared to subset-based feature selection methods. Thus, we propose a top ranking-based approach to select a PVC subset in the source project. This approach adopts the idea of majority voting that combing different filter-based feature ranking methods and selects top 15% of the PVC features as suggested by Gao et al. (2011). We utilize six single attribute evaluators implemented in Weka, that is *CorrelationAttributeEval, GainRatioAttributeEval, InfoGainAttributeEval, OneRAttributeEval, ReliefFAttributeEval* and *SymmetricalUncertAttributeEval* respectively, with *Ranker* search method to evaluate the PVC features of the source project.

Fig. 6 presents the flowchart of the proposed approach. We first conduct experiments under six attribute evaluators $\{AE_1, AE_2, \ldots, AE_6\}$ for the PVC features in the source project. Each attribute evaluator AE_i evaluates the worth of the given PVC f and produces a label for f from the set of class label $\{r_1,\ r_2\}$, where r_1 denotes **Selected** and r_2 denotes **Unselected**. Label **Selected** means that f ranks into top 15% of the PVC features in the source project. If f is ranked into top 15% of the PVC features by most attribute evaluators, f is labeled to this class. Eq. (2) shows the rule of class identification for top ranking-based feature selection approach, where $AE_i^j(f)$ denotes the output of attribute evaluator AE_i

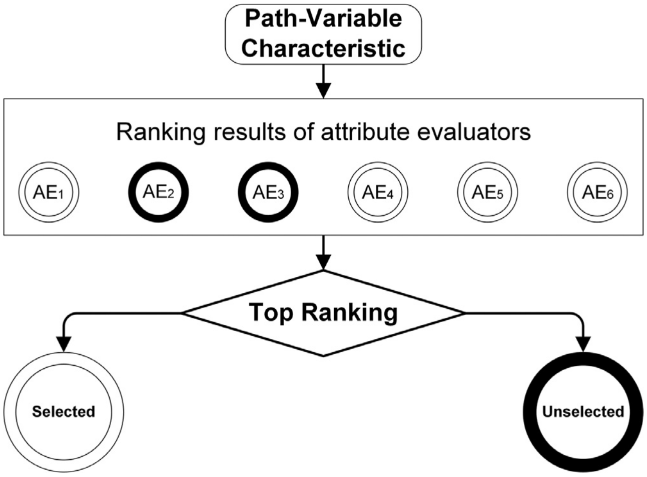

Path-Variable Characteristic
Ranking results of attribute evaluators
AE1
AE2
AE3
AE4
AE5
AE6
Top Ranking
Selected
Unselected

Fig. 6. Flowchart of top ranking-based feature selection approach.

on class label r_j.

$$AE(f) = \begin{cases} r_1, & \text{if } \sum_{i=1}^{6} AE_i^1(f) \geq 3; \\ r_2, & \text{otherwise.} \end{cases} \quad (2)$$

3.4.2. Feature matching between the source and target projects

To match PVC features between the source and target project, each source and target PVC pair is evaluated using the Kolmogorov-Smirnov test (KS test) coefficient to measure the distribution similarity. The KS test is one of the most useful and general non-parametric two-sample methods that quantifies a distance between the empirical distribution functions of two samples. Since the PVC features extracted from the evaluated projects have different distributions and variances, the KS test is a suitable statistical test to compute a p-value as a matching score for each source and target PVC pair, showing the probability of whether two samples are significantly different or not.

The key idea of feature matching is computing matching scores for all source and target PVC pairs. We used the *kolmogorovSmirnovTest* implemented in the Apache Commons Math library[1], a statistics package providing frameworks and implementations for basic statistical tests, to compute the matching score. The matching score is denoted by $p(S_i, T_j)$, which is a p-value computed by the KS test of $\mathbf{PVC}_i$ from the source project and $\mathbf{PVC}_j$ from the target project. The value of $p(S_i, T_j)$ tends to be zero when two PVC features are significantly different. Then, we remove poorly matched PVC pairs whose matching score is less than a given cutoff threshold. From the remaining source and target PVC pairs, we select a group of matched PVC pairs that has the maximum weights, that is, whose sum of matching scores is the largest, without duplicated PVC features.

To exemplify, Fig. 7 illustrates a sample matching. There are two source PVC features ($\mathbf{PVC}_1$ and $\mathbf{PVC}_{18}$) and three target PVC features ($\mathbf{PVC}_4$, $\mathbf{PVC}_{15}$ and $\mathbf{PVC}_{33}$). Therefore, there are six possible matching PVC pairs, (S_1, T_4), (S_1, T_{15}), (S_1, T_{33}), (S_{18}, T_4), (S_{18}, T_{15}), and (S_{18}, T_{33}). The matching scores of all PVC pairs are computed by the KS test and shown in the rectangles. For example, the matching score between $\mathbf{PVC}_1$ and $\mathbf{PVC}_4$ is $p(S_1, T_4) = 1.0$. If the cutoff threshold is 0.5, the matching PVC pairs (S_1, T_{15}) and (S_{18}, T_4) will be excluded. Thus, we can only consider the candidate matching PVC pairs (S_1, T_4), (S_1, T_{33}), (S_{18}, T_{15}), and (S_{18}, T_{33}) in this example. After applying the cutoff threshold, there are three groups of matched PVC pairs without duplicated PVC features, that is, {(S_1, T_4), (S_{18}, T_{15})}, {(S_1, T_4), (S_{18}, T_{33})} and {(S_1, T_{33}), (S_{18}, T_{15})}, respectively. The sum of matching scores of these three groups is respectively 1.7 (=1.0+0.7), 1.8 (=1.0+0.8) and 1.6 (=0.9+0.7). Thus, we select the second group {(S_1, T_4), (S_{18}, T_{33})} as the set of matched PVC pairs for the given source and target projects with the cutoff threshold of 0.5 in this example.

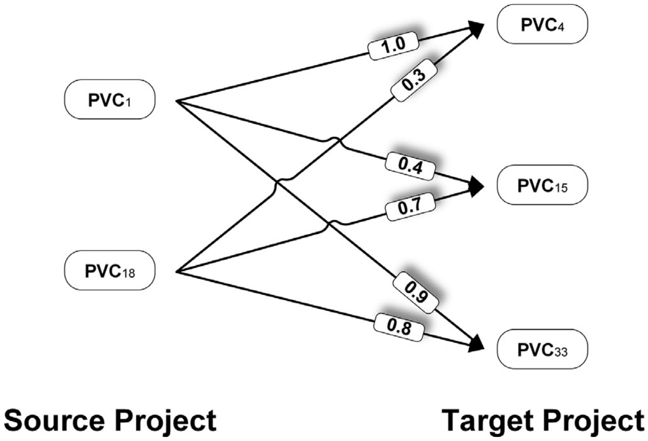


Fig. 7. An example of feature matching between the source and target projects.

4. Experimental setup

We conduct several experiments to evaluate the performance of the proposed PVC features in both WPDI and CPDI. Our experiments are all run on a 3.7 GHz Intel Core i3-6100 machine with 4GB RAM.

4.1. Static analysis tool

DTS (Yang et al., 2008) is a defect pattern-driven tool. Each defect pattern is defined using a defect pattern state machine (DPSM). A DPSM can be represented as a triple (*S*, *T*, *C*), in which:

- S is a state set and $S = \{S_{start}, S_{error}, S_{end}, S_{other}\}$, where S_{start} denotes the initial state, S_{error} denotes the error state, S_{end} denotes the end state and S_{other} denotes other intermediate states,
- T is a state transition set, which is defined as T: $S \times C \rightarrow S$,
- C is a transition condition set, which denotes the state transition conditions.

DTS proposes the interval computation technique in control-flow and data-flow analysis, of which the purpose is to compute the state of DPSM. If a DPSM is transited to an error state then a defect is reported by DTS.

4.2. Datasets and ground truth building

Our experiments are conducted on eight open-source projects (listed in Table 2 with full description). The selection of these projects is based on their sizes (they are large enough to have many SA alarms), diversity (they cover a wide range of functionality) and the fact that they have source code repositories which makes the feature extraction process easier. The sixth column indicates the number of alarms of the evaluated projects, which range from 45 to 695. The defective rates of the evaluated projects have a minimum value of 25.5% and a maximum value of 71.1%, which are listed in the seventh column.

[1] http://commons.apache.org/proper/commons-math/.

Table 2
Basic facts about the evaluated projects.

Project	Release	Description	Language	Size (KLoC)	# of Alarms	Defective Rate (%)
Juliet[a]	1.3	A collection of test cases from SARD suites	C	11	695	59.4
antiword[b]	0.37	A free MS Word reader for Linux	C	20	45	71.1
spell[c]	1.0	A spell checking program	C	2	62	66.1
sphinxbase[d]	0.3	A speech Recognition toolkit	C	28	141	33.3
uucp[e]	1.07	A complete UUCP package	C	53	677	70.8
lucene-solr[f]	4.6.1	A text search engine	Java	283	103	27.2
phoenix[g]	4.4	A client-embedded JDBC driver	Java	99	51	25.5
poi[h]	3.10	A library to access OOXML format files	Java	410	92	57.6

[a] https://samate.nist.gov/SRD/testsuite.php
[b] https://launchpad.net/ubuntu/+source/antiword/0.37-11
[c] https://ftp.gnu.org/gnu/spell/
[d] https://sourceforge.net/projects/cmusphinx/files/sphinxbase/
[e] http://www.airs.com/ian/uucp.html
[f] https://github.com/apache/lucene-solr
[g] http://phoenix.apache.org/source.html
[h] https://github.com/apache/poi/

To evaluate the proposed approach, we need to prepare the experimental dataset[2] for each evaluated project. In detail, we need to extract the values of all PVC features listed in Table 1. We implement our own tool embedded in DTS to extract the values for each PVC feature using the following steps:

- Obtain the source code files of a specific release of the evaluated project by using *git checkout* from its Git repository or downloading from the website.
- Compile the source code files using DTS.
- Generate intraprocedural paths via CFG for the reported SA alarms (with details described in Section 3.2.1).
- Extract the value of PVC features based on the generated paths (with details described in Section 3.2.2).

Since the SA tools would produce false positives, we should firstly classify the reported alarms as actionable and unactionable accurately to build the ground truth. In this paper, we use the manual inspection results from previous studies or downloaded websites as labels for the datasets. Specifically, as for the Juliet dataset, we use the manual inspection results commented on the downloaded source code files as ground truth, and the manual inspection results from our previous work (Zhang et al., 2020) are used as ground truth for the other four C projects. As for the three Java projects, we use the labeled warning information from (Wang et al., 2018) as ground truth.

4.3. Machine learning classifiers

Weka contains a series of machine learning algorithms for classification tasks with understandable output results to developers. We select 12 classification algorithms from six categories implemented in Weka for model building: decision tree (J48 and LMT), Bayes classifier (NaiveBayes (NB) and BayesNet (BN)), instance-based algorithm (IBk and KStar), rule-based algorithm (PART and JRip), function-based model (SimpleLogistic (SL) and SMO) and ensemble learning method (Vote and RandomForest (RF)). The selection of these classifiers is based on their popularity and diversity (Herbold et al., 2018; Tantithamthavorn et al., 2019; Hosseini et al., 2019). Additionally, parameter tuning of machine learning classifiers usually requires a lot of expertise and effort. Thus, we use the default parameters for all classifiers in Weka. According to the empirical results presented in Fig. 4a(a) and b(i) from (Tantithamthavorn et al., 2019), the selected classifiers tend to robust to parameter settings when considering the AUC and MCC metrics. That is to say, they may have negligible to small impact on the performance improvement of defect identification models when parameter optimization is applied.

4.4. Experimental design

For WPDI, datasets from the same project are split into the training set and the test set. When building models, we carry out tenfold cross-validation to evaluate the effectiveness of classification. In cross-validation, datasets are randomly split into ten approximately equal subsets, and nine of the subsets are used to train a model and the last subset to test. The process is repeated ten times in order that each of the ten subsets would be tested once. Since randomness would occur inevitably in splitting datasets and affect the prediction performance (Arcuri and Briand, 2011), we repeat the tenfold cross-validation 100 times (i.e., 1000 tests) for each model and report the average prediction results. To evaluate the performance of PVC features in WPDI, we compare our proposed PVC features with traditional features. The first baseline of traditional features consists of 19 widely used common metric (CM) features, including LOC-based metrics, McCabe metrics and Halstead metrics, etc. Table 3 shows an overview of the CM features used in this paper, and the work from (Menzies et al., 2007) contains the full definition of these features. In order to have homogeneous feature sets between different programming languages, the object-oriented based CK metrics are excluded in this paper. We use Prest (Kocaguneli et al., 2009), an intelligent open-source tool for software metrics extraction, to collect CM features for experimental analysis. The second baseline of traditional features is the VC features used in (Zhang et al., 2020).

For CPDI, we build a prediction model using all instances from the source project and predict instances from the target project by using the built model. Unlike WPDI, the evaluation of CPDI does not involve any randomness, because all instances from the source project constitute the training set and all instances from the target project constitute the test set. We perform the evaluation of each project pair once. Additionally, we choose a cutoff of 0.05 for the KS test, which is a commonly accepted significance level in the statistical test. To evaluate the performance of our proposed transfer learning approach FRM-TL, we compare FRM-TL to three baseline approaches. We conduct CPDI without transfer learning as the first baseline (CPDI-ALL). That is to say, all the PVC features are used to build the defect identification models between the source and target datasets. As the second baseline, we include the CPDP approach proposed by He et al. (2014), which uses 16 distribution characteristics of values of each instance with all features. The 16 distribution characteristics are *mode, median, mean, harmonic*

[2] https://github.com/WayYuZhang/SoftwareDefectIdentification/tree/master/Dataset.

Table 3
Overview of the selected CM features.

Category	Feature
Size	loc_total
McCabe	cyclomatic_complexity, cyclomatic_density
Halstead	num_operators, num_operands, num_unique_operators, num_unique_operands, halstead_length, halstead_volume, halstead_level, halstead_difficulty, halstead_effort, halstead_time
Miscellaneous	branch_count, call_pairs, condition_count, decision_count, decision_density, parameter_count

mean, minimum, maximum, range, variation ratio, first quartile, third quartile, interquartile range, variance, standard deviation, coefficient of variance, skewness, and *kurtosis* (He et al., 2014). In this paper, we extract the 16 distribution characteristics from the PVC feature set as features (CPDI-IFS) to build the defect identification models. The third baseline is the classic transferable feature learning approach TCA (Pan et al., 2011), which maps the source and target datasets onto the same subspace while minimizing data difference and maximizing data variance. In this paper, we implement our own version of TCA (CPDI-TCA) based on the supplied source code provided by its author (Pan et al., 2011).

To compare our approach to baselines, we adopt the Scott-Knott Effect Size Difference (ESD) test in this paper, which is a mean comparison approach that leverages a hierarchical clustering to partition the set of treatment means into statistically distinct groups with non-negligible difference (Tantithamthavorn et al., 2019). We used the `sk_esd` function implemented in the *ScottKnottESD*[3] R package to compare the identification performance of the approaches we examined. Additionally, to measure the effect size of results among baselines and our approach, we compute the Cohen effect size estimate d by using the `checkDifference` function implemented in the *ScottKnottESD* R package. The magnitude is accessed using the thresholds as follows: negligible ($d \leq 0.2$), small ($0.2 < d \leq 0.5$), medium ($0.5 < d \leq 0.8$), and large ($d > 0.8$).

4.5. Evaluation metrics

Since the software defect datasets usually suffer from the class imbalance problem, performance metrics such as precision and recall, which are highly affected by defective ratios, may be not appropriate for defect identification tasks. To measure the identification performance in this paper, the following three comprehensive metrics are adopted to evaluate model prediction results, which are commonly used in the previous studies (Nam et al., 2018; Li et al., 2018; Herbold et al., 2018; Tantithamthavorn et al., 2019).

4.5.1. Indicators derived from confusion matrix

The measurement of model prediction performance is usually based on the analysis of a confusion matrix. This matrix consists of four numbers: 1) true positive (TP): the number of predicted actionable alarms that are truly actionable; 2) false negative (FN): the number of predicted unactionable alarms that are actually actionable; 3) false positive (FP): the number of predicted actionable alarms that are actually unactionable; 4) true negative (TN): the number of predicted unactionable alarms that are truly unactionable. The two metrics, F2-measure (F2) and Matthews correlation coefficient (MCC), are derived from the confusion matrix. Here is a brief introduction:

$$Precision(P) = \frac{TP}{TP + FP} \tag{3}$$

$$Recall(R) = \frac{TP}{TP + FN} \tag{4}$$

$$F_\beta = \frac{(1+\beta^2) \times P \times R}{\beta^2 \times P + R} \tag{5}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP) \times (TP+FN) \times (TN+FP) \times (TN+FN)}} \tag{6}$$

Eq. (5) shows the general formula of F-measure, which is a weighted harmonic mean value of precision (P) and recall (R). In this paper, we use the F2-measure ($\beta = 2$) for evaluating the performance of defect identification models, which has a greater impact on recall. The higher is the F2-measure, the better is the performance.

MCC measures the correlation between the observed and predicted classifications by taking into account all components of the confusion matrix, which can be calculated as shown in Eq. (6). Its return value is on a scale $[-1, 1]$, where values close to 1 indicate a perfect prediction, values close to -1 represent total disagreement between observation and prediction, and values close to 0 indicate no better than random prediction.

4.5.2. Area under ROC curve

The third metric is the area under ROC curve (AUC), which is defined using the receiver operating characteristic (ROC). The AUC is known as a useful metric for comparing different models and widely used because AUC is unaffected by the class imbalance problem as well as being independent from the prediction threshold (Tantithamthavorn et al., 2019). The higher AUC indicates better prediction performance of the defect identification model, and the AUC of 0.5 means the performance of a random predictor.

5. Results and analysis

This section presents the experimental results. We focus on the performance of our proposed PVC features and answer the research questions raised in Section 1.

5.1. Answering RQ1

To answer this question, we use different features to build WPDI models to compare the impact of three sets of features: CM features, VC features and PVC features. The first two are used as baselines. In total, we conduct 96 sets of WPDI comparison experiments (8 datasets × 12 machine learning classifiers), each of which uses instances of the same project.

5.1.1. Results for RQ1

Table 4 presents the mean F2, MCC and AUC values for each evaluated project when using the three sets of features across the 12 machine learning classifiers. The results of F2, MCC and AUC for each evaluated project are shown by row. If there are better results between CM features and PVC features, the higher values of each comparison experiment are in bold as shown in Table 4. Between VC features and PVC features, the higher values are underlined in the table. To exemplify, by using the 12 machine learning classifiers for `antiword`, the mean AUC of using PVC features is 0.935, while the mean AUC is only 0.702 with the first baseline of using

[3] https://github.com/klainfo/ScottKnottESD.

Table 4
Comparison results of within-project defect identification on the evaluated projects under 3 different evaluation metrics.

Project	F2			MCC			AUC		
	CM	VC	PVC	CM	VC	PVC	CM	VC	PVC
Juliet	0.764(4.749,L)	0.960(6.163,L)	**0.993**	0.091(19.677,L)	0.913(2.485,L)	**0.976**	0.553(17.706,L)	0.982(1.123,L)	**0.995**
antiword	0.776(2.181,L)	0.908(1.217,L)	**0.950**	0.344(4.340,L)	0.658(1.369,L)	**0.800**	0.702(4.427,L)	0.846(1.707,L)	**0.935**
spell	0.848(1.995,L)	0.917(-0.443,S)	**0.907**	0.603(1.168,L)	0.627(0.553,M)	**0.679**	0.853(0.454,S)	0.843(0.545,M)	**0.881**
sphinxbase	0.606(1.203,L)	0.705(0.850,L)	**0.825**	0.516(1.163,L)	0.583(1.083,L)	**0.759**	0.784(1.241,L)	0.809(1.274,L)	**0.918**
uucp	0.916(0.202,S)	0.920(0.143,N)	**0.929**	0.594(1.462,L)	0.624(1.377,L)	**0.805**	0.846(1.687,L)	0.842(1.335,L)	**0.943**
lucene-solr	0.464(3.315,L)	0.607(4.097,L)	**0.878**	0.313(6.945,L)	0.478(4.547,L)	**0.857**	0.721(3.937,L)	0.792(3.443,L)	**0.953**
phoenix	0.299(0.766,M)	0.217(1.295,L)	**0.399**	0.143(1.584,L)	0.047(2.115,L)	**0.406**	0.569(1.558,L)	0.547(1.656,L)	**0.747**
poi	0.771(1.550,L)	0.819(1.945,L)	**0.888**	0.566(2.060,L)	0.553(2.378,L)	**0.740**	0.802(1.801,L)	0.796(2.177,L)	**0.907**

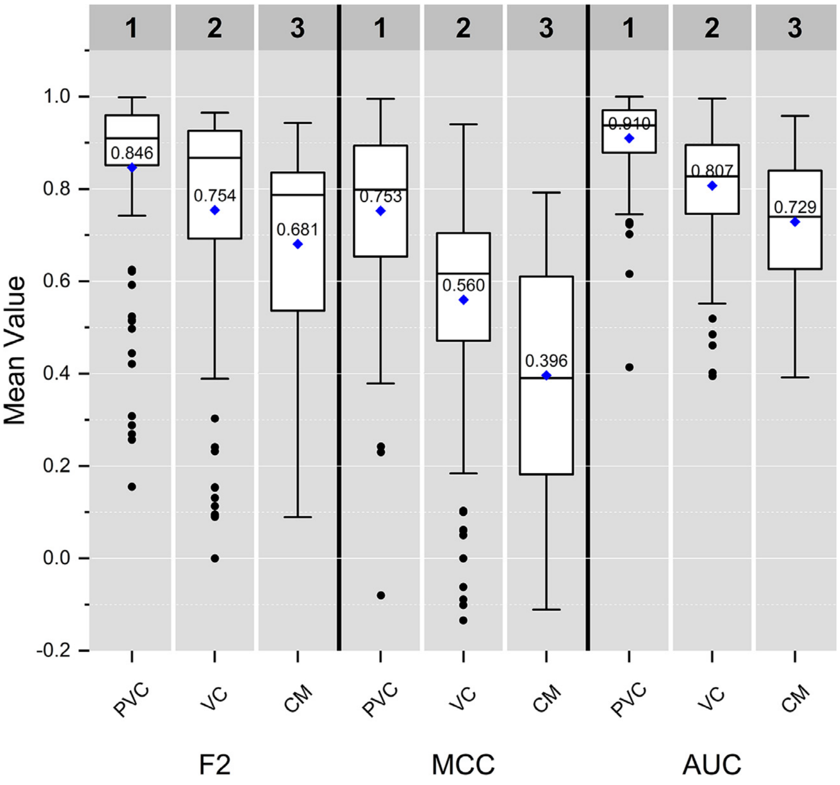


Fig. 8. The Scott-Knott ESD ranking of baselines and pvc features under 3 different evaluation metrics.

CM features, and the mean AUC is 0.846 with the second baseline of using VC features. For this comparison, the only difference lies in the feature sets used for model building, meaning that the same classification algorithms, the same parameters and the same evaluated project are used.

The values in parentheses in Table 4 show Cohen's *d* and its magnitude for the effect size among baselines and PVC features, where N denotes negligible, S denotes small, M denotes medium, and L denotes large. The positive values of Cohen's *d* indicate that the PVC features improve the baseline in terms of the effect size. For example, the Cohen's *d* of mean F2 values between CM features and PVC features for Juliet is 4.749 and its magnitude is L, namely, the PVC features outperform the CM features in Juliet with the large magnitude of the effect size when considering the mean F2.

Fig. 8 presents the Scott-Knott ESD test of the mean F2, MCC and AUC results (across all projects and classifiers) using box plots. The ranking results of different evaluation metrics are partitioned by the solid vertical lines in black, while the ranking results of different feature sets are grouped by the solid vertical lines in white. The bottom and top of the boxes indicates the first and third quartiles respectively, while the solid horizontal line in a box indicates the median value in each performance distribution, and the blue diamond in a box indicates the mean value in each performance distribution. Black circles in the figure are outliers.

The following results can be observed from Table 4 and Fig. 8:

- The PVC features rank into the first group in the Scott-Knott ESD test of all the three evaluation metrics.
- The PVC features achieve a mean F2 of 0.846, while the CM features achieve a mean F2 of 0.681, and the VC features achieve a mean F2 of 0.754. By comparing the two baselines, the average improvement in the mean F2 is 24.2% and 12.2% respectively.
- When considering the mean F2 results, the magnitude of Cohen's *d* values between CM features and PVC features are medium (M) or large (L) in 7 out of 8 projects (except uucp), while the PVC features show significantly better results to VC features in terms of effect size in 75% projects.
- The PVC features achieve a mean MCC of 0.753, while the CM features achieve a mean MCC of 0.396, and the VC features achieve a mean MCC of 0.560. By comparing the two baselines, the average improvement in the mean MCC is 90.2% and 34.5% respectively.

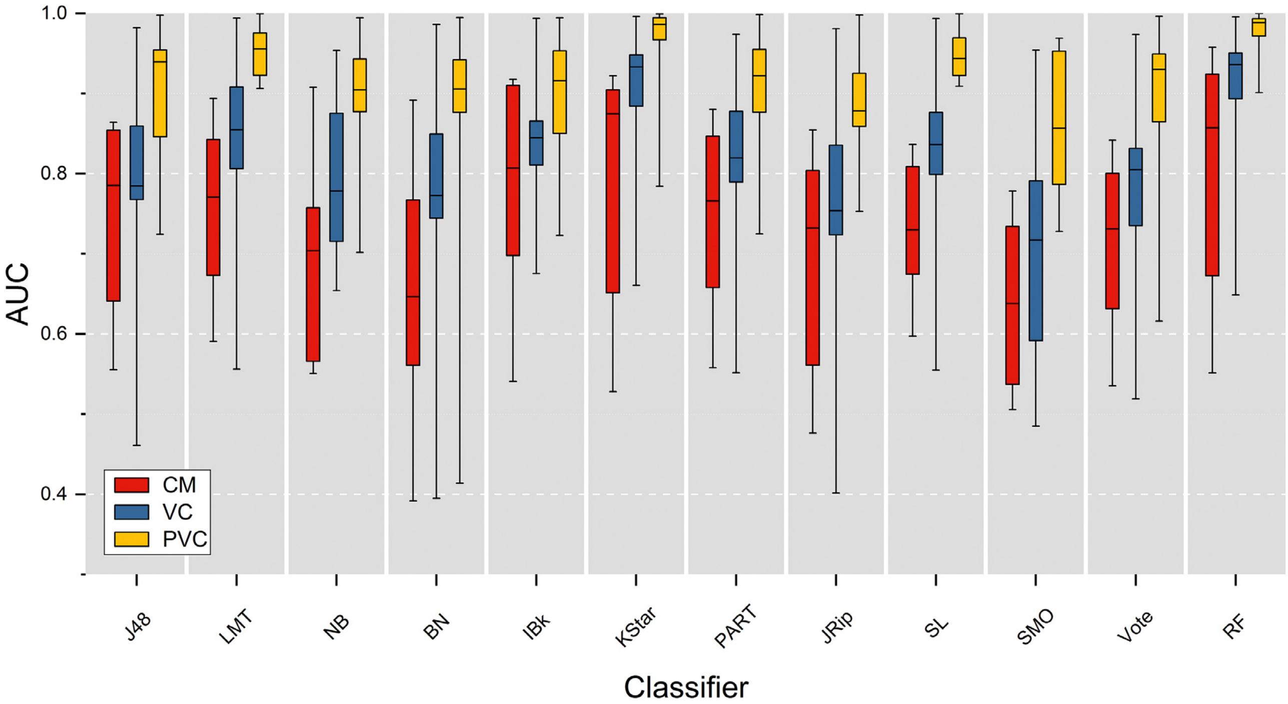


Fig. 9. Performance comparison of different classifiers.

- The PVC features outperform the two baselines in all targets with the medium (M) or large (L) magnitude of the effect size in terms of the mean MCC results.
- The PVC features achieve a mean AUC of 0.910, while the CM features achieve a mean AUC of 0.729, and the VC features achieve a mean AUC of 0.807. By comparing the two baselines, the average improvement in the mean AUC is 24.8% and 12.8% respectively.
- When considering the mean AUC results, the magnitude of Cohen's *d* values between the two baselines and PVC features are large (L) in 7 out of 8 projects (except spell). In other words, the PVC features improve the two baselines in most targets in terms of effect size.

5.1.2. Performance in different machine learning classifiers

In Fig. 9, we present the identification performance (in terms of the mean AUC) of different machine learning classifiers using box plots. The red, blue and yellow box plots represents the performance distributions of selected classifiers when using the CM, VC and PVC features respectively.

We can observe that the *RF* classifier measures up the best performance across all evaluated projects. This is reasonable because the *RF* classifier is an ensemble learning method which combines the prediction result of numerous single classifier. However, the performance of the *BN* classifier heavily depends on the evaluated projects, which varies significantly over different projects. On average, the selected 12 machine learning classifiers have fairly consistent performance when using PVC features, and the improvement compared to the two baselines is not tied to a particular machine learning classifier.

5.1.3. Answer to RQ1

Overall, our proposed PVC features are effective in automatically identifying SA alarms, which improve the performance of within-project defect identification. The comparison results indicate that the PVC features outperform the two baselines with statistical significance, namely, we can enhance the WPDI performance by using the PVC features extracted from the generated paths instead of the traditional features.

Table 5
Feature ranking results for the evaluated projects.

Project	Selected features
Juliet	$\mathbf{PVC}_1$, $\mathbf{PVC}_2$, $\mathbf{PVC}_7$, $\mathbf{PVC}_{29}$, $\mathbf{PVC}_{30}$, $\mathbf{PVC}_{41}$
antiword	$\mathbf{PVC}_1$, $\mathbf{PVC}_2$, $\mathbf{PVC}_3$, $\mathbf{PVC}_{12}$, $\mathbf{PVC}_{33}$, $\mathbf{PVC}_{36}$, $\mathbf{PVC}_{41}$
spell	$\mathbf{PVC}_1$, $\mathbf{PVC}_2$, $\mathbf{PVC}_7$, $\mathbf{PVC}_8$, $\mathbf{PVC}_{10}$, $\mathbf{PVC}_{15}$, $\mathbf{PVC}_{26}$, $\mathbf{PVC}_{31}$, $\mathbf{PVC}_{44}$
sphinxbase	$\mathbf{PVC}_1$, $\mathbf{PVC}_2$, $\mathbf{PVC}_3$, $\mathbf{PVC}_5$, $\mathbf{PVC}_6$, $\mathbf{PVC}_9$, $\mathbf{PVC}_{10}$
uucp	$\mathbf{PVC}_2$, $\mathbf{PVC}_3$, $\mathbf{PVC}_4$, $\mathbf{PVC}_6$, $\mathbf{PVC}_{11}$, $\mathbf{PVC}_{31}$, $\mathbf{PVC}_{34}$
lucene-solr	$\mathbf{PVC}_2$, $\mathbf{PVC}_3$, $\mathbf{PVC}_{10}$, $\mathbf{PVC}_{12}$, $\mathbf{PVC}_{26}$, $\mathbf{PVC}_{33}$, $\mathbf{PVC}_{35}$
phoenix	$\mathbf{PVC}_2$, $\mathbf{PVC}_{13}$, $\mathbf{PVC}_{14}$, $\mathbf{PVC}_{17}$, $\mathbf{PVC}_{27}$, $\mathbf{PVC}_{28}$, $\mathbf{PVC}_{34}$, $\mathbf{PVC}_{35}$
poi	$\mathbf{PVC}_3$, $\mathbf{PVC}_4$, $\mathbf{PVC}_5$, $\mathbf{PVC}_{10}$, $\mathbf{PVC}_{11}$, $\mathbf{PVC}_{14}$, $\mathbf{PVC}_{28}$, $\mathbf{PVC}_{33}$, $\mathbf{PVC}_{34}$

5.2. Answering RQ2

To answer this question, we first conduct the feature ranking process for each evaluated project as described in Section 3.4.1. Table 5 presents a brief overview of the selected PVC features for each project. The number of selected PVC features varies from 6 (Juliet) to 9 (spell and poi). In total, 28 out of 47 PVC features are selected for the evaluated projects. Based on the selected PVC features in Table 5, we can easily find out that the features vary significantly over different projects. Furthermore, the LOC-based features ($\mathbf{PVC}_1$, $\mathbf{PVC}_2$ and $\mathbf{PVC}_3$) are contained in all evaluated projects, which implies that the number of source code statements at different levels are predictive of the actionability of the alarm.

Then, the KS test is utilized to select a group of matched PVC pairs between the source and target project, which has the maximum matching scores. Since we choose a cutoff of 0.05 for the KS test, we remove matched PVC pairs with the matching score ≤ 0.05 and build an identification model using matched PVC pairs with the score > 0.05. The number of matched PVC pairs varies by each identification combination. For example, the number of matched PVC pairs is one in sphinxbase ⇒ lucene-solr (the

Table 6
Comparison results of cross-project defect identification on the evaluated projects under 3 different evaluation metrics.

F2	CPDI-ALL	CPDI-IFS	CPDI-TCA	FRM-TL
Juliet	**0.594**(-0.775,M)	0.596(-0.635,M)	0.571*(-0.773,M)	0.469
antiword	**0.644**(-0.107,N)	0.569(0.300,S)	0.554(0.356,S)	0.625*
spell	0.453(1.814,L)	0.713(0.619,M)	0.517(1.413,L)	**0.778***
sphinxbase	**0.540**(-0.383,S)	0.241(1.990,L)	0.360(0.656,M)	0.467*
uucp	0.612(0.497,S)	0.706(-0.033,N)	0.591(0.591,M)	**0.700***
lucene-solr	**0.462**(-1.772,L)	0.177(-0.126,N)	0.394*(-1.891,L)	0.147
phoenix	**0.331**(-1.108,L)	0.206(-1.442,L)	0.228*(-1.681,L)	0.095
poi	0.310(1.548,L)	0.427(0.802,L)	0.508(0.418,S)	**0.577***
MCC	CPDI-ALL	CPDI-IFS	CPDI-TCA	FRM-TL
Juliet	**0.071**(-0.465,S)	-0.012(0.064,N)	-0.074(0.429,S)	-0.003*
antiword	0.091(0.039,N)	0.029(0.353,S)	-0.118(1.135,L)	**0.104***
spell	0.092(0.170,N)	0.111(0.084,N)	0.083(0.201,S)	**0.127***
sphinxbase	0.060(0.087,N)	0.001(0.327,S)	0.017(0.254,S)	**0.081***
uucp	0.014(0.969,L)	0.030(1.366,L)	-0.060(1.161,L)	**0.191***
lucene-solr	**-0.027**(-0.012,N)	-0.010(-0.073,N)	0.006*(-0.189,N)	-0.028
phoenix	**0.064**(-0.459,S)	0.047(-0.176,N)	-0.107(0.705,M)	0.013*
poi	**0.009**(-0.139,N)	0.035(-0.286,S)	-0.044(0.102,N)	-0.017*
AUC	CPDI-ALL	CPDI-IFS	CPDI-TCA	FRM-TL
Juliet	**0.504**(-0.291,S)	0.492(-0.175,N)	0.459(0.100,N)	0.469*
antiword	0.542(0.191,N)	0.540(0.177,N)	0.416(1.043,L)	**0.564***
spell	**0.542**(-0.057,N)	0.563(-0.217,S)	0.528(0.033,N)	0.533*
sphinxbase	0.514(0.247,S)	0.516 (0.255,S)	0.465(0.556,M)	**0.552***
uucp	0.498(0.696,M)	0.514(0.467,S)	0.456(0.760,M)	**0.575***
lucene-solr	0.496(0.168,N)	0.483(0.297,S)	0.503(0.109,N)	**0.518***
phoenix	0.543(0.188,N)	0.498(0.198,N)	0.395(0.990,L)	**0.519***
poi	0.508(0.201,S)	0.508(0.211,S)	0.484(0.349,S)	**0.531***

symbol ⇒ is used to denote a identification combination) while that is seven in `poi` ⇒ `phoenix`. We put all the matched PVC pairs of each identification combination in the online appendix.[4]

5.2.1. Results for RQ2

We conduct 480 sets of CDPI experiments (40 identification combinations × 12 machine learning classifiers). Each experiment takes two different evaluated projects as one identification combination, while the source project is as the training set and the target project is as the test set. We compare our proposed CPDI approach FRM-TL to three baselines: CPDI-ALL, CPDI-IFS and CPDI-TCA. Table 6 shows the mean F2, MCC and AUC values of the four CPDI approaches by each source project across all target projects and classifiers. The results of F2, MCC and AUC for each source project are shown by row. If there are better results between CPDI-ALL and our approach, the higher values of each comparison experiment are in bold as shown in Table 6. Between CPDI-IFS and our approach, the higher values are underlined in the table. Between CPDI-TCA and our approach, the higher values are shown with an asterisk (*).

The values in parentheses in Table 6 show Cohen's *d* and its magnitude for the effect size among baselines and FRM-TL. If a Cohen's *d* is positive, FRM-TL improves the baseline in terms of the effect size. To exemplify, the Cohen's *d* of mean F2 values between CPDI-ALL and FRM-TL for `spell` is 1.814 and its magnitude is L, that is, FRM-TL outperforms CPDI-ALL in `spell` with the large magnitude of the effect size when considering the mean F2. Furthermore, we present the Scott-Knott ESD test of the mean F2, MCC and AUC results (including all CPDI experiments) using box plots in Fig. 10.

The followings are the observations of the results from Table 6 and Fig. 10:

- Our proposed approach FRM-TL ranks into the first group in the Scott-Knott ESD test of all the three evaluation metrics.
- FRM-TL achieves a mean F2 of 0.482, which respectively outperforms CPDI-IFS and CPDI-TCA by 6.2% and 3.7%. However, FRM-TL does not lead to better with statistical significance but comparable against CPDI-ALL. There are 5 out of 8 source projects show worse results against CPDI-ALL, which is mainly concentrated in the identification combinations from the source projects `lucene-solr` and `phoenix`. In the next subsection, we discuss and analyze why these results could happen.
- When considering the mean F2 results, the magnitude of Cohen's *d* values between CPDI-ALL and FRM-TL are positive in 3 out of 8 projects, while FRM-TL shows better or comparable results to CPDI-IFS in terms of effect size in 75% projects (except `Juliet` and `phoenix`), and the 5 out of 8 source projects (`antiword`, `spell`, `sphinxbase`, `uucp` and `poi`) lead to better or comparable results against CPDI-TCA.
- FRM-TL achieves a mean MCC of 0.059, while CPDI-ALL achieves a mean MCC of 0.047, CPDI-IFS achieves a mean MCC of 0.029, and CPDI-TCA achieves a mean MCC of -0.037. Overall, FRM-TL outperforms the baselines with statistical significance when considering results from all CPDI experiments.
- When considering the mean MCC results, FRM-TL shows better with statistical significance or comparable results against the baselines in all source projects in terms of effect size.
- FRM-TL achieves a mean AUC of 0.533, while CPDI-ALL achieves a mean AUC of 0.518, CPDI-IFS achieves a mean AUC of 0.514, and CPDI-TCA achieves a mean AUC of 0.463. By comparing the three baselines, the average improvement in the mean AUC is 2.9%, 3.7% and 15.1% respectively.
- When considering the mean AUC results, the magnitude of Cohen's *d* values between the three baselines and FRM-TL are positive in 20 out of 24 targets, namely, FRM-TL improves the three baselines in most targets in terms of effect size.

5.2.2. Evaluation of the matched PVC features

As shown in Table 6, identification combinations in some source projects have very poor CPDI performance. In combination `poi` ⇒ `phoenix`, all the evaluation metric results are lower than the

[4] https://github.com/WayYuZhang/SoftwareDefectIdentification/tree/master/ExperimentalResult.

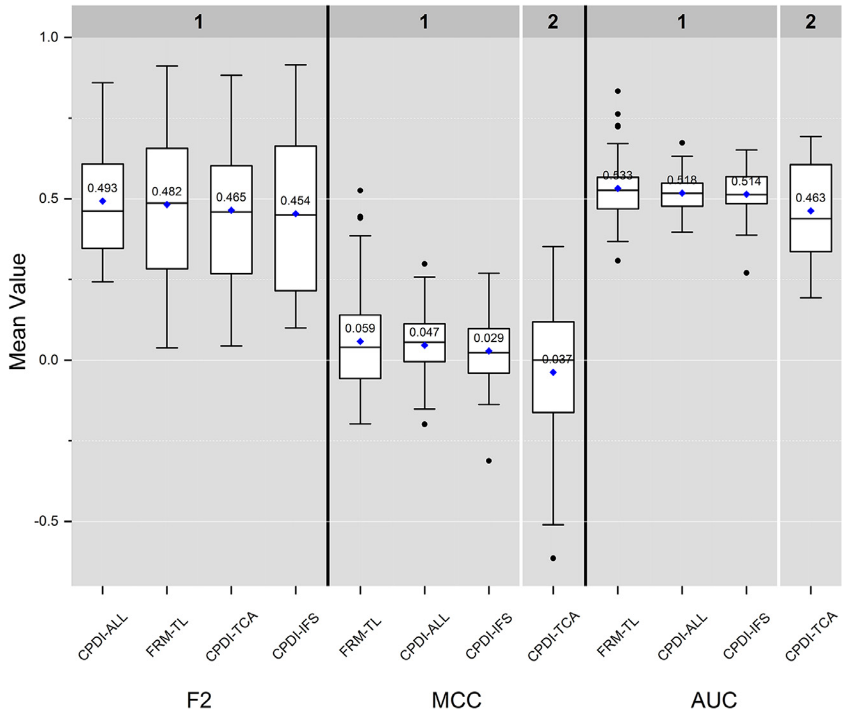


Fig. 10. The Scott-Knott ESD ranking of baselines and FRM-TL under 3 different evaluation metrics.

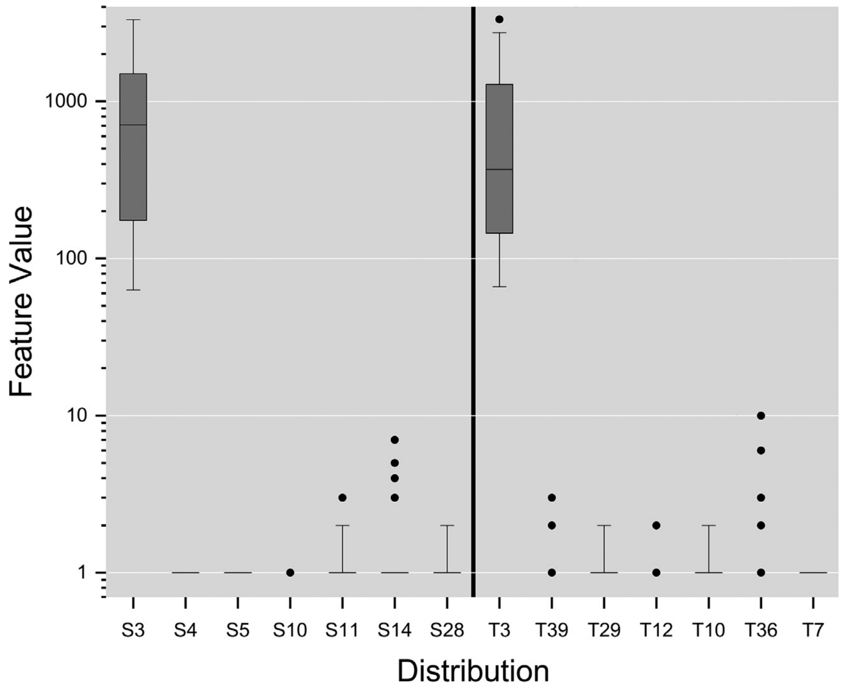


Fig. 11. Distribution of the matched PVC Pairs in poi ⇒ phoenix.

mean values of poi. Thus, we discuss this identification combination as a representative example.

In Fig. 11, we use box plots to show the distributions of all matched PVC pairs in combination poi ⇒ phoenix. The seven box plots on the left part represent distributions of the matched PVC features from the source project poi while the seven box plots on the right part represent those from the target project phoenix. Each scale of the X-axis represents a PVC feature, and there is a correspondence between their positions according to the matched PVC pairs of combination poi ⇒ phoenix, that is (S_3, T_3), (S_4, T_{39}), (S_5, T_{29}), (S_{10}, T_{12}), (S_{11}, T_{10}), (S_{14}, T_{36}) and (S_{28}, T_7) respectively. As shown in Fig. 11, all the five PVC pairs have similar feature distributions.

Since the PVC pairs are matched based on the similarity of source and target feature distributions, our proposed FRM-TL model is supposed to achieve better CPDI performance compared to the baseline approach. However, when the different tendencies of defect-proneness occur between the source and target PVC fea-

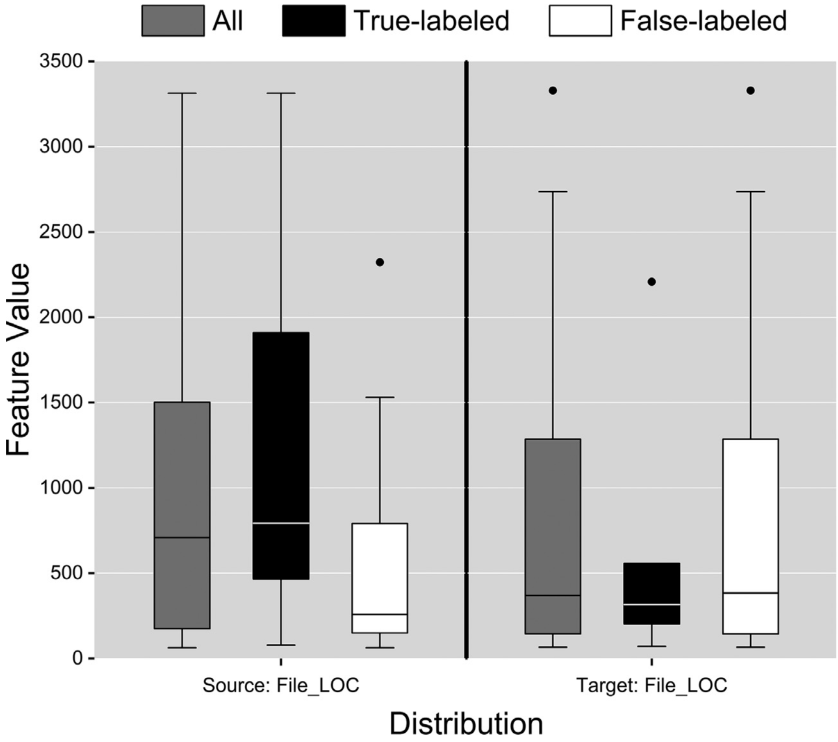


Fig. 12. Distribution of the features in PVC Pair (S_3, T_3).

tures, our proposed approach may not produce reasonable identification performance. Fig. 12 explains how the defect-prone tendencies differ between the source and target features of PVC pair (S_3, T_3) and why the identification combination poi ⇒ phoenix has a poor result. The gray, black and white box plots represents the distributions of matched PVC features in all, true-labeled and false-labeled instances respectively. The three box plots on the left part represent the distributions of the source PVC feature while those on the right part represent the distributions of the target PVC feature.

Interestingly, the two matched PVC features are both **File_LOC**, which might be considered as a matching of two similar features. As shown in Fig. 12, the median value of true-labeled instances in the source PVC feature is higher than that of false-labeled instances, indicating that a defect identification model would label an instance as actionable when the feature value of the instance is more than 800 in the case of project poi. However, the median value of true-labeled instances in the target PVC feature is lower than that of false-labeled instances so that a lower feature value would lead to a higher defect-proneness result in project phoenix. This inconsistent tendency of defect-proneness in the matched PVC pairs would harm the identification performance even if the two matched features within the PVC pair are similar. We regard this kind of matching as a noisy output of the feature-matching process, which could also be discovered in other identification combinations that cannot achieve reasonable CPDI performance.

As observed, a defect identification model using the matched PVC pairs that have a consistent defect-prone tendency could achieve a high F-measure result. However, filtering out the noisy feature-matching pairs is a challenging task as the instances in the test set are not labeled in advance. Thus, designing a filter for removing the noisy matching will remain as future work, which is helpful for enhancing the performance of defect identification models.

5.2.3. Performance in different matching score cutoff thresholds

Additionally, we apply different cutoff thresholds for matching scores (0.05 and $0.10, 0.20, \ldots, 0.90$) to observe the differences in CPDI performance. Fig. 13 presents the mean F2, MCC and AUC results (across all identification combinations and classifiers) in terms of different cutoff thresholds. The horizontal axis represents the cutoff thresholds used in this paper.

According to the statistics from Fig. 13, the FRM-TL model with a cutoff of 0.90 could achieve better CPDI performance compared to those with other cutoff thresholds. By comparing the result achieved by the cutoff 0.05, the average improvement in the mean F2, MCC and AUC is respectively 1.7%, 30.5% and 1.0%. However, there are fluctuations when the cutoff threshold increased. These results were somewhat expected to have in mind that, as described in Section 5.2.2, some noisy outputs of the feature matching process might not be filtered out if their matching scores are higher than the cutoff threshold, which is not helpful for improving the CPDI performance. Overall, we can easily observe that the mean evaluation metric values are gradually improved as the cutoff threshold increased. This means some negative PVC pairs can be filtered out when a cutoff threshold increased, thus improving the CPDI performance of our proposed approach.

5.2.4. Answer to RQ2

To summary, the two-stage transfer learning approach FRM-TL is superior in identifying SA alarms from new projects than the baseline approaches in most cases, that is, our proposed PVC features can achieve reasonable performance in cross-project defect identification.

6. Threats to validity

Three main threats to validity in this paper, that is, external validity, internal validity and construct validity, respectively, are illustrated as follows.

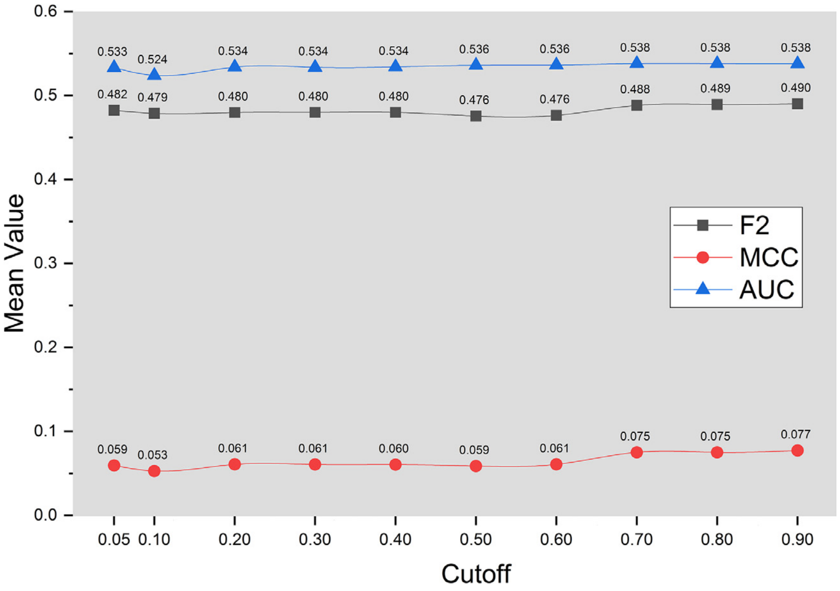


Fig. 13. Performance comparison in different cutoff thresholds.

6.1. External validity

The principal threat to external validity is that the evaluated projects in this paper may not be of enough generalization for all software projects. As a result, projects exclude in the eight projects might yield better or worse performance based on our approach. But additional running of our proposed model on other projects will minimize this threat to validity. Since our model is only evaluated on open-source projects, its performance on closed-source projects is unknown.

6.2. Internal validity

For our paper, dataset preparation is the main concern of internal validity. Oversights of manual inspection could invalidate a few of the model results. Multiple examination by different developers will minimize this threat to internal validity. Furthermore, our evaluated benchmark consists of 1866 reported alarms which may not be large enough to train defect identification models with high confidence. We repeated the experiments using different random seeds to reduce the randomness that might be caused by having limited data.

6.3. Construct validity

Firstly, since we use the default parameters for machine learning classifiers in Weka, the experimental results could be improved when parameter optimization is applied. Thus, our results may be affected by the other parameter tuning machine learning classifiers. We remain to conduct experiments with parameter optimization as future work. Secondly, we adopt F2-measure, MCC and AUC for model evaluation in this paper. However, validating prediction models in terms of other evaluation measures is also required in practice. We will conduct experiments using more evaluation measures in our future work.

7. Conclusion and future work

In order to mitigate the workload of manual inspection caused by SA tools, this paper presents a machine learning-based approach for automatically identifying software defects reported by SA tools. Firstly, our approach proposes to leverage path analysis techniques to produce a set of fine-grained features, called PVC, from the source code files of the evaluated projects for automated defect identification. Specifically, we utilize CFG to generate paths for extracting syntactic information from the source code files and leverage the extracted PVC features to build machine learning models for identifying SA alarms automatically. Then, we raise a two-stage transfer learning approach, called FRM-TL, to enhance the performance of cross-project defect identification by applying the feature ranking and feature matching techniques.

Our empirical results on eight open-source projects show that the proposed PVC features at variable-level are promising and can yield significant improvement on both within-project and cross-project defect identification. By comparing the two traditional features (CM features and VC features), our proposed PVC features respectively improve the within-project defect identification on average by 24.2% and 12.2% in F2, 90.2% and 34.5% in MCC, and 24.8% and 12.8% in AUC. For cross-project defect identification, our proposed transfer learning approach based on the matched PVC feature pairs outperforms the baseline approaches in most cases.

In the future, as is mentioned in Section 5.2.2, there is a major challenge in filtering out the noisy feature matching. Thus, designing a filter to remove the matched feature pairs with inconsistent defect-prone tendencies is an interesting problem to address. We also plan to leverage the representation leaning techniques to learn deep semantic structure automatically from the source code files for model building, which would be promising to increase the accuracy of defect identification.

Declaration Completing 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. The authors declare that they have no conflict of interest.

CRediT authorship contribution statement

Yuwei Zhang: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Dahai Jin:** Conceptualization, Validation. **Ying Xing:** Writing - review & editing. **Yunzhan Gong:** Supervision, Funding acquisition.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Nos. U1736110 and 61702044).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2020.110585.

References

Arcuri, A., Briand, L.C., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1–10.

Catal, C., Diri, B., 2009. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Inf. Sci. 179 (8), 1040–1058.

Fan, G., Wu, R.X., Shi, Q.K., Xiao, X., Zhou, J.G., Zhang, C., 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In: Proceedings of the 41st International Conference on Software Engineering, pp. 72–82.

Flynn, L., Snavely, W., Svoboda, D., VanHoudnos, N.M., Qin, R., Burns, J., Zubrow, D., Stoddard, R., Marce-Santurio, G., 2018. Prioritizing alerts from multiple static analysis tools, using classification models. In: Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies, pp. 13–20.

Gao, K.H., Khoshgoftaar, T.M., Wang, H.J., Seliya, N., 2011. Choosing software metrics for defect prediction: an investigation on feature selection techniques. Softw., Pract. Exper. 41 (5), 579–606.

Ghotra, B., McIntosh, S., Hassan, A.E., 2017. A large-scale study of the impact of feature selection techniques on defect classification models. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 146–157.

He, P., Li, B., Ma, Y.T., 2014. Towards Cross-Project Defect Prediction with Imbalanced Feature Sets. CoRR.

Heckman, S., Williams, L., 2009. A model building process for identifying actionable static analysis alerts. In: Proceedings of the 2rd International Conference on Software Testing Verification and Validation, pp. 161–170.

Heckman, S., Williams, L., 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf. Softw. Technol. 53 (4), 363–387.

Herbold, S., 2013. Training data selection for cross-project defect prediction. In: Proceedings of the 9th International Conference on Predictive Models in Software Engineering, pp. 6:1–6:10.

Herbold, S., Trautsch, A., Grabowski, J., 2018. A comparative study to benchmark cross-project defect prediction approaches. IEEE Trans. Softw. Eng. 44 (9), 811–833.

Hosseini, S., Turhan, B., Gunarathna, D., 2019. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans. Softw. Eng. 45 (2), 111–147.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th International Conference on Program Comprehension, pp. 200–210.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018. Summarizing source code with transferred API knowledge. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 2269–2275.

Koc, U., Saadatpanah, P., Foster, J.S., Porter, A.A., 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, pp. 35–42.

Koc, U., Wei, S.Y., Foster, J.S., Carpuat, M., Porter, A.A., 2019. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In: Proceedings of the 12th International Conference on Software Testing, Validation and Verification, pp. 288–299.

Kocaguneli, E., Tosun, A., Bener, A.B., Turhan, B., Caglayan, B., 2009. Prest: an intelligent software metrics extraction, analysis and defect prediction tool. In: Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering, pp. 637–642.

Le, W., Soffa, M.L., 2007. Refining buffer overflow detection via demand-driven path-sensitive analysis. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 63–68.

Li, Z.Q., Jing, X.Y., Zhu, X.K., 2018. Progress on approaches to software defect prediction. IET Softw. 12 (3), 161–175.

Liu, C., Yang, D., Xia, X., Yan, M., Zhang, X.H., 2019. A two-phase transfer learning model for cross-project defect prediction. Inf Softw. Technol. 107, 125–136.

Ma, Y., Luo, G.C., Zeng, X., Chen, A.G., 2012. Transfer learning for cross-company software defect prediction. Inf. Softw. Technol. 54 (3), 248–256.

Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.

Muske, T., Serebrenik, A., 2016. Survey of approaches for handling static analysis alarms. In: Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation, pp. 157–166.

Nam, J., Fu, W., Kim, S., Menzies, T., Tan, L., 2018. Heterogeneous defect prediction. IEEE Trans. Softw. Eng. 44 (9), 874–896.

Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: Proceedings of the 35th International Conference on Software Engineering, pp. 382–391.

Pan, S.J., Tsang, I.W., Kwok, J.T., Yang, Q., 2011. Domain adaptation via transfer component analysis. IEEE Trans. Neural Netw. 22 (2), 199–210.

Qiu, S.J., Lu, L., Cai, Z.Y., Jiang, S.Y., 2019. Cross-project defect prediction via transferable deep learning-generated and handcrafted features. In: Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering, pp. 431–552.

Raghothaman, M., Kulkarni, S., Heo, K., Naik, M., 2018. User-guided program reasoning using bayesian inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 722–735.

Ruthruff, J.R., Penix, J., Morgenthaler, J.D., Elbaum, S., Rothermel, G., 2008. Predicting accurate and actionable static analysis warnings: an experimental approach. In: Proceedings of the 30th International Conference on Software Engineering, pp. 341–350.

Ryu, D., Jang, J.I., Baik, J., 2017. A transfer cost-sensitive boosting approach for cross-project defect prediction. Softw. Qual. J. 25 (1), 235–272.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Ihara, A., Matsumoto, K., 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In: Proceedings of the 37th International Conference on Software Engineering, pp. 812–823.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2019. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.

Turhan, B., Menzies, T., Bener, A.B., Stefano, J.D., 2009. On the relative value of cross–company and within-company data for defect prediction. Empir. Softw. Eng. 14 (5), 540–578.

Wang, J.J., Wang, S., Wang, Q., 2018. Is there a “golden” feature set for static warning identification?: an experimental evaluation. In: Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement, pp. 17:1–17:10.

Wang, S., Liu, T.Y., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering, pp. 297–308.

Watanabe, S., Kaiya, H., Kaijiri, K., 2008. Adapting a fault prediction model to allow inter language reuse. In: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, pp. 19–24.

Witten, I.H., Frank, E., Hall, M.A., Pal, C.J., 2016. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann.

Xu, Z., Liu, J., Xia, Z., Yuan, P.P., 2017. An empirical study on the equivalence and stability of feature selection for noisy software defect data. In: Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering, pp. 191–196.

Xu, Z., Liu, J., Yang, Z.J., An, G.G., Jia, X.Y., 2016. The impact of feature selection on defect prediction performance: an empirical comparison. In: Proceedings of the 27th International Symposium on Software Reliability Engineering, pp. 309–320.

Xu, Z., Pang, S., Zhang, T., Luo, X.P., Liu, J., Tang, Y.T., Yu, X., Xue, L., 2019. Cross project defect prediction via balanced distribution adaptation based transfer learning. J. Comput. Sci. Technol. 34 (5), 1039–1062.

Yang, Z.H., Gong, Y.Z., Xiao, Q., Wang, Y.W., 2008. DTS-A software defects testing system. In: Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation, pp. 269–270.

Yoon, J., Jin, M., Jung, Y., 2014. Reducing false alarms from an industrial-strength static analyzer by SVM. In: Proceedings of the 21st Asia-Pacific Software Engineering Conference, vol. 2, pp. 3–6.

Zhang, X.Z., Gong, Y.Z., Wang, Y.W., 2017. Heuristic guided selective path exploration for loop structure in coverage testing. IJOSSP 8 (2), 59–75.

Zhang, Y.W., Xing, Y., Gong, Y.Z., Jin, D.H., Li, H.H., Liu, F., 2020. A variable-level automated defect identification model based on machine learning. Soft Comput. 24 (2), 1045–1061.

Zhao, Y.S., Wang, Y.W., Gong, Y.Z., Chen, H.H., Xiao, Q., Yang, Z.H., 2011. STVL: improve the precision of static defect detection with symbolic three-valued logic. In: Proceedings of the 18th Asia Pacific Software Engineering Conference, pp. 179–186.

Yuwei Zhang is currently a Ph.D. candidate in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. He received his BS degree in network engineering from Beijing University of Posts and Telecommunications in 2016. His research interests include software testing and machine learning.

Dahai Jin received his Ph.D. degree in information security from Armored Engineering Institute of the PLA, Beijing, in 2006. He is currently an associate professor in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. His research interests include software testing and static analysis.

Ying Xing received her Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, in 2014, where she was a Postdoctoral Fellow, in 2014, and is currently an Assistant Professor with the Automation School. Her research interests include artificial intelligence and automatic test data generation.

Yunzhan Gong received his Ph.D. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 1991. He is currently a professor and supervisor of doctoral students in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing. His research interests include fault tolerant computing and software testing.