

Large-Scale Empirical Studies on Effort-Aware Security Vulnerability Prediction Methods

Xiang Chen¹, Member, IEEE, Yingquan Zhao, Zhanqi Cui, Member, IEEE, Guozhu Meng¹, Member, IEEE,
Yang Liu¹, Member, IEEE, and Zan Wang¹, Member, IEEE

Abstract—Security vulnerability prediction (SVP) can identify potential vulnerable modules in advance and then help developers to allocate most of the test resources to these modules. To evaluate the performance of different SVP methods, we should take the security audit and code inspection into account and then consider effort-aware performance measures (such as ACC and P_{opt}). However, to the best of our knowledge, the effectiveness of different SVP methods has not been thoroughly investigated in terms of effort-aware performance measures. In this article, we consider 48 different SVP methods, of which 36 are supervised methods and 12 are unsupervised methods. For the supervised methods, we consider 34 software-metric-based methods and two text-mining-based methods. For the software-metric-based methods, in addition to a large number of classification methods, we also consider four state-of-the-art methods (i.e., EALR, OneWay, CBS, and MULTI) proposed in recent effort-aware just-in-time defect prediction studies. For text-mining-based methods, we consider the Bag-of-Word model and the term-frequency-inverse-document-frequency model. For the unsupervised methods, all the modules are ranked in the ascendent order based on a specific metric. Since 12 software metrics are considered when measuring extracted modules, there are 12 different unsupervised methods. To the best of our knowledge, over 40 SVP methods have not been considered in previous SVP studies. In our large-scale empirical studies, we use three real open-source web applications written in PHP as benchmark. These three web applications include 3466 modules and 223 vulnerabilities in total. We evaluate these SVP methods both in the

within-project SVP scenario and the cross-project SVP scenario. Empirical results show that two unsupervised methods [i.e., lines of code (LOC) and Halstead's volume (HV)] and four recently proposed state-of-the-art supervised methods (i.e., MULTI, OneWay, CBS, and EALR) can achieve better performance than the other methods in terms of effort-aware performance measures. Then, we analyze the reasons why these six methods can achieve better performance. For example, when using 20% of the entire efforts, we find that these six methods always require more modules to be inspected, especially for unsupervised methods LOC and HV. Finally, from the view of practical vulnerability localization, we find that all the unsupervised methods and the OneWay method have high false alarms before finding the first vulnerable module. This may have an impact on developers' confidence and tolerance, and supervised methods (especially MULTI and text-mining-based methods) are preferred.

Index Terms—Effort-aware performance measures, security vulnerability prediction (SVP), software metric, supervised method, text mining, unsupervised method.

I. INTRODUCTION

THE vast number of security vulnerabilities are reported each year, and security vulnerabilities have imposed significant damages to individuals and companies. Compared with software defects, the number of vulnerabilities in the software project is far less than the number of software defects. Meanwhile, finding security vulnerabilities requires a deep understanding of both the software and the attacker's mindset [1]. Moreover, security vulnerabilities provide opportunities for hackers, who often keep their knowledge of vulnerabilities secret and can commit the criminal activities in some cases. Due to highly negative impact of security vulnerabilities, different approaches of security vulnerability analysis and discovery have been proposed by researchers from the academic community and the industrial community [2]. Except for traditional approaches (such as static analysis, dynamic analysis, and hybrid analysis) [3], approaches using machine learning are popular in current security vulnerability analysis and discovery and have received increasing interests [4].

Motivated by the studies of software defect prediction (SDP) [5], [6], security vulnerability prediction (SVP) constructs models with machine learning to identify potential vulnerable program modules [7]. In particular, SVP extracts and labels modules by mining software historical repositories, such as version control systems and bug tracking systems. The granularity of the

Manuscript received May 27, 2018; revised October 14, 2018 and March 25, 2019; accepted June 19, 2019. This work was supported in part by the National Natural Science Foundation of China under Grant 61702041, Grant 61872263, Grant 61602267, and Grant 61202006, in part by Nantong Application Research Plan under Grant JC2018134, in part by the Science and Technology Project of the Beijing Municipal Education Commission under Grant KM201811232016, in part by Qin Xin Talents Cultivation Program for the Beijing Information Science and Technology University under Grant QXTCP C201906. Associate Editor: W. E. Wong. (*Xiang Chen and Yingquan Zhao have contributed equally to this work and are co-first authors.*) (*Corresponding author: Xiang Chen.*)

X. Chen is with the School of Information Science and Technology, Nantong University, Nantong 226019, China (e-mail: xchencs@ntu.edu.cn).

Y. Zhao is with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China (e-mail: enockchao@163.com).

Z. Cui is with the Computer School, Beijing Information Science and Technology University, Beijing 100192, China (e-mail: czq@bistu.edu.cn).

G. Meng was with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. He is now with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100864, China (e-mail: gzmeng@ntu.edu.sg).

Y. Liu is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: yangliu@ntu.edu.sg).

Z. Wang is with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China (e-mail: wangzan@tju.edu.cn).

Color versions of one or more of the figures in this article are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TR.2019.2924932

module can be set as file, class, or binary component, as needed. Then, it uses metrics to measure these extracted modules. Most of metrics are inspired by the SDP domain, such as code complexity and project development characteristics. Finally, it uses a specific classification method (such as random forest) to train a model. For a new module, we can then leverage this trained model to predict whether this module is vulnerable or nonvulnerable.

Since locating vulnerabilities should take into account cost effectiveness of SVP models, in this article, we consider effort-aware performance measures (i.e., P_{opt} and ACC). To the best of our knowledge, this is the first paper to compare the performance for a large number of SVP methods. In particular, we consider 48 different SVP methods. Among these methods, 36 methods are supervised methods and 12 methods are unsupervised methods. For the supervised methods, we consider 34 methods based on software metrics and two methods based on text mining. For the supervised methods based on software metrics, we first consider a large number of traditional classification methods, which are used in the previous empirical study to revisit their impact on the performance of traditional SDP [8]. Then, we include four state-of-the-art methods proposed in recent effort-aware just-in-time software defect prediction (JIT-SDP) studies. These four methods are EALR [9], OneWay [10], CBS [11], and MULTI [12]. For the supervised methods based on text mining, we not only consider a method based on the Bag-of-Word (BOW) model [13], but also consider a method based on the term-frequency-inverse-document-frequency (TF-IDF) model. For the unsupervised methods, all the modules are ranked in the ascendent order based a specific metric. Since 12 software metrics are considered when measuring extracted modules, there are 12 different unsupervised methods. To the best of our knowledge, for these SVP methods, over 40 methods have not been investigated in previous SVP studies.

In our large-scale empirical studies, we choose three real open-source web applications (i.e., Drupal, PHPMyAdmin, and Moodle) as our experimental subjects. These three web applications include 3466 modules and 223 vulnerabilities in total. To evaluate the performance of different SVP methods, we consider two common model evaluation scenarios: within-project SVP scenario and cross-project SVP scenario [4]. Empirical results show that two unsupervised methods (i.e., LOC and HV) and recently proposed four state-of-the-art supervised methods (i.e., MULTI, OneWay, CBS, and EALR) can achieve better performance than the other methods (including text-mining-based methods) in terms of effort-aware performance measures.

However, when analyzing proportion of modules inspected assuming only using 20% of the entire efforts (i.e., $PMI@20\%$ measure), we find that these methods always need to inspect more modules, especially for unsupervised methods. This finding illustrate why these methods can achieve better performance when considering effort-aware performance measures. Moreover, when considering the initial false alarms (nonvulnerable modules are misclassified as vulnerable modules) encountered before finding the first vulnerable module (i.e., IFA measure), we find that almost all the unsupervised methods have high false alarms. Therefore, this may have an impact on developers'

confidence and tolerance, and supervised methods are preferred, especially for MULTI and text-mining-based methods.

The main contributions of this article can be summarized as follows:

- 1) To the best of our knowledge, this is the first paper to investigate a large number of SVP methods, including 36 supervised methods and 12 unsupervised methods. More than 30 supervised methods and all the unsupervised methods have not been considered in previous SVP studies.
- 2) Large-scale empirical studies are conducted on three real open-source web applications written in PHP. Final results based on effort-aware performance measures show that two unsupervised methods (i.e., LOC and NV) and recently proposed four supervised methods (i.e., MULTI, OneWay, CBS, and EALR) have competitiveness both in the within-project SVP scenario and in the cross-project SVP scenario.
- 3) We also compare these SVP methods from proportion of modules inspected when using 20% of the entire efforts and the initial false alarms encountered before finding the first vulnerable modules. Final results can demonstrate why unsupervised methods can have better ACC and P_{opt} values. Moreover, these findings can provide guidelines to effectively use these SVP methods for vulnerability localization in practice.

The rest of this article is organized as follows. Section II introduces background of SVP and related work for our study. Section III shows methodology, including experimental subjects, all the considered supervised methods and unsupervised methods, performance measures, and the experimental setup. Section IV performs result analysis. Section V gives some discussions. Section VI analyzes threats to validity for our empirical studies. Section VII concludes this article.

II. BACKGROUND AND RELATED WORK

In this section, we first introduce the background of SVP. Then, we summarize previous studies for SVP. Since our work is motivated by the recent debates between supervised methods and unsupervised methods in the SDP domain, we review these studies in the last subsection.

A. Background of SVP

Software quality assurance teams often have limited resources for security audit and code inspection. Moreover, detecting and mitigating security vulnerabilities require manual analysis by experts who need to be trained with a security mindset [1]. Similar to SDP [5], [6], SVP resorts to machine learning and aims to identify vulnerable modules in advance. Therefore, the allocation of test resources can be optimized, and more security vulnerabilities can be detected and mitigated as soon as possible.

The brief process of SVP can be summarized as follows [13]. SVP first extracts program modules from software historical repositories, such as version control systems and bug tracking systems. The granularity of modules can be set as source code file, object-oriented class, and binary component, as needed.

Then, it uses metrics to measure these modules and labels these modules by analyzing commit message and bug reports. Most of the metrics are inspired by the SDP domain and are designed based on the analysis of code complexity, project development characteristics, or text mining. Moreover, these metrics can be easily measured in large-scale software projects. Finally, it uses a specific classification method (such as random forest) to train a model based on gathered SVP datasets. For a new program module, we can use the trained model to predict whether this module is vulnerable or nonvulnerable.

According to the above description, we can find that the SVP model construction process is similar to SDP [5], [6]. Vulnerabilities and defects are similar in that both vulnerabilities and defects can be caused by human mistakes during the development process. These mistakes may be related to the code complexity, the developer experience, or the development process. Therefore, metrics used for the SDP domain can be directly used to construct SVP models. Empirical results of previous studies (introduced in Section II-B) showed the feasibility of this solution. However, different from defects, vulnerabilities are instances of errors in the specification, development, or configuration of software such that their executions can violate implicit or explicit security policies. Therefore, the characteristic of vulnerable modules should have certain differences when compared with defective modules. This requires researchers to design specific metrics to measure vulnerable modules, and this problem has not been thoroughly studied. Moreover, the number of vulnerable modules is far less than the number of defective modules [7]. Therefore, the problem of class imbalance in SVP is more challenging than SDP.

B. Related Work of SVP

Based on the background analysis for SVP, we can find that SVP uses machine learning to identify potential vulnerable modules and does not incorporate program analysis methods [4]. Therefore, most of the SVP studies are mainly motivated by the previous studies on SDP.

Zimmermann *et al.* [7] first investigated the possibility of SVP by considering traditional metrics (such as complexity, code churn, dependence measures, and organizational structure of the company) used in SDP. Based on a commercial project (i.e., Windows Vista), they found that the traditional metrics have statistically significant correlation with the number of vulnerabilities by using Spearman's rank correlation. However, the effect of the correlation is small. They then use Logistic regression to evaluate the prediction performance. They found that the trained models have good precision values but low recall values.

Meneely and Williams [14] analyzed the relationship between developer-activity-based metrics and vulnerabilities. Based on three open-source projects (the Linux kernel, the PHP programming language, and the Wireshark network protocol analyzer), they found that the correlations exist. However, the correlations vary and are not very strong. They also evaluated the prediction performance of these metrics by using the Bayesian network. However, the performance based on precision and recall is also not satisfactory.

Shin *et al.* [15] also investigated the possibility of SVP by considering traditional metrics (complexity, code churn, and developer activity). Final results showed that the trained models can predict 70.8% vulnerabilities by inspecting only 10.9% files of Firefox web browser and predict 68.8% vulnerabilities by inspecting only 3.0% files of Red Hat Linux kernel. Later, Shin and Williams [16] investigated whether SDP models can be used to predict vulnerabilities. Their empirical results showed that SDP models can be used to predict vulnerabilities. However, false positives of these models need to be reduced while retaining high recall values. Russell *et al.* [17] utilized deep learning to learn feature directly from source code, and empirical results showed that deep feature representation learning is a promising method.

Walden *et al.* [13] compared the performance between software-metric-based SVP methods and text-mining-based SVP methods [18], [19]. To conduct empirical studies, they analyzed three large-scale open-source projects (i.e., Drupal, Moodle, and PhpMyAdmin) and shared these datasets to facilitate the follow-up studies. They found that text-mining-based methods can have better recall values. Based on their shared datasets, Zhang *et al.* [20] proposed the VULPREDICTOR method to improve the prediction performance. In particular, they first built six base classifiers based on software metrics or text mining. Then, they constructed a metaclassifier to process the outputs of six base classifiers. Empirical results showed the effectiveness of their proposed method. Tang *et al.* [21] considered code inspection cost and proposed effort-aware performance measures to evaluate SVP methods. Empirical results showed that both in effort-aware ranking-based measures and effort-aware classification-based measures, text-mining-based methods had similar performance with software-metric-based methods. Recently, Stuckman *et al.* [22] investigated the effect of dimensionality reduction methods for SVP. They considered two kinds of dimensionality reduction methods: feature selection methods and feature synthesis methods. They found that using dimensionality reduction can narrow the performance gap between text-mining-based methods and software-metric-based methods.

Based on datasets shared by Walden *et al.* [13], this is the first paper to systematically investigate the performance of a large number of SVP methods, including 36 supervised methods and 12 unsupervised methods. The performance of these methods is evaluated by effort-aware performance measures (i.e., ACC and P_{opt}), and the performance of these methods is compared in two different scenarios: the within-project SVP scenario and the cross-project SVP scenario.

C. Debates on Supervised Methods Versus Unsupervised Methods

The debates on the performance comparison between supervised methods and unsupervised methods still derive no conclusion in current SDP research. In previous SDP studies, most of the proposed methods are based on supervised methods [5]. The high performance of these methods depends on the high-quality SDP dataset. However, high-quality SDP datasets often need

high cost to extract and label program modules, and noises in SDP databases are inevitable [23], [24]. On the contrary, unsupervised methods do not need any training data and can scale to large-scale projects. These characteristics can help to apply SDP models to the industry [25], [26]. The above advantages of unsupervised methods have attracted more and more researchers' attention [27], [28]. Moreover, a recent study showed that simple unsupervised methods can be easily applied to improve the performance of regression testing [29].

For effort-aware JIT-SDP [9], [30], [31], for which the granularity of modules is set as code change, Yang *et al.* [32] found that some simple unsupervised methods can achieve better performance compared to supervised methods (including the EALR method proposed by Kamei *et al.* [9]). Their study has triggered heated discussions. Fu and Menzies [10] revisited empirical studies of Yang *et al.* [32], and they found that not all the unsupervised methods can achieve better results than supervised methods. Therefore, they proposed the OneWay method, which can select the best unsupervised method automatically based on training data. Huang *et al.* [11] later found that given the same test effort, supervised methods often need to inspect more code changes and, therefore, can achieve better performance. Liu *et al.* [33] proposed a new code-churn-based unsupervised method, which can achieve better performance than unsupervised methods proposed by Yang *et al.* [32]. Chen *et al.* [12] proposed a multiobjective-optimization-based supervised method MULTI and found that this method performed better than state-of-the-art JIT-SDP methods. Therefore, they confirmed that supervised methods are still promising in effort-aware JIT-SDP. Yan *et al.* [34] compared supervised methods and unsupervised methods in file-level software defect prediction (FL-SDP), and they found similar results found in effort-aware JIT-SDP studies. Recently, Zhou *et al.* [35] performed large-scale empirical studies by comparing two simple unsupervised methods (i.e., ManualDown and ManualUp) with previous cross-project defect prediction methods. Based on empirical studies, they recommended that future studies should consider ManualDown and ManualUp as the baseline methods.

However, for SVP, we do not find any studies investigating the performance of unsupervised methods. Moreover, this is the first paper that systematically compares unsupervised methods with many state-of-the-art supervised methods, which are widely used in effort-aware JIT-SDP [10], [11], [32] and FL-SDP [34] studies.

III. METHODOLOGY

In our large-scale empirical studies, we want to answer the following two research questions.

RQ1: How are performance comparison results among all our considered methods in the within-project SVP scenario in terms of effort-aware performance measures?

RQ2: How are performance comparison results among all our considered methods in the cross-project SVP scenario in terms of effort-aware performance measures?

For these two RQs, we consider two different SVP scenarios, and the introduction of these two scenarios can be found in Section III-E. To evaluate the performance of different SVP

methods, we only focus on effort-aware performance measures, which will be illustrated in Section III-D.

In the rest of this section, we first introduce experimental subjects used in our empirical studies. Then, we introduce all the supervised methods and unsupervised methods in detail. Later, we illustrate the motivation of using effort-aware performance measures and their corresponding meanings. Finally, we give the detail of experiment setup, including data preprocessing methods, model evaluation scenarios, and statistical analysis methods used for method comparison.

A. Experimental Subjects

In our empirical studies, we consider three real open-source web applications written in PHP programming language. The selected web applications are Drupal, Moodle, and PHPMyAdmin. Here, Drupal is a widely used content management system, Moodle is an open-source learning management system, and PHPMyAdmin is a web-based management tool for the MySQL database. These web applications are widely used in previous SVP studies [13], [20]–[22].

In these web applications, the granularity of the extracted modules is set as file. These three web applications include 3466 modules and 223 vulnerabilities in total. The detected vulnerabilities include code injection vulnerabilities, cross-site request forgery vulnerabilities, cross-site scripting vulnerabilities, and path disclosure vulnerabilities. These vulnerabilities are identified based on the vendor of common vulnerabilities and exposures or analysis on code commits and bug reports.

Extracted modules from these applications are measured in two ways. One way uses traditional software metrics inspired by traditional SDP. Another way uses text mining to extract term vectors, and the details of this way can be found in Section III-B.

For the first way, 12 metrics are considered [13]. The metric name, metric abbreviation (i.e., corresponding metric name in Weka arff file), and the introduction can be found in Table I.

The characteristics of these datasets can be found in Table II, including the dataset name, the number of extracted modules, the number (percentage) of vulnerable modules, and the number of text features (terms) after text mining.

B. Supervised Methods

In this subsection, we introduce all the 36 supervised methods we consider. Here, 34 supervised methods are software-metric-based methods, and the remaining two ones are text-mining-based methods.

1) *Software-Metric-Based Methods:* For software-metric-based methods, previous studies only considered some classification methods. For example, Walden *et al.* only considered random forest method [13], [22]. Tang *et al.* considered both random forest and Naive Bayes methods [21]. Zhang *et al.* considered random forest, Naive Bayes, and decision tree as base classifiers and then used ensemble learning to further improve the prediction performance [20]. In our empirical studies, we first consider a large number of classification methods considered in previous effort-aware JIT-SDP studies [10]–[12], [32]. Moreover, most of these methods are also used in the previous empirical study to revisit their impact on the performance of

TABLE I
TRADITIONAL METRICS USED FOR MEASURING EXTRACTED MODULES

| Name | Abbreviation | Introduction |
|-------------------------------------|--------------|--|
| Lines of code | LOC | Number of lines in a source file excluding lines without PHP tokens, such as blank lines and comments. |
| Lines of code for non-HTML | NLOC | Same as LOC, not considering content outside of PHP start/end tags. |
| Number of functions | NM | Number of function and method definitions in a source file. |
| Cyclomatic complexity | CC | The size of a control flow graph after linear chains of nodes are collapsed into one. |
| Maximum nesting complexity | NE | The maximum depth to which loops and control structures in the file are nested. |
| Halsteads volume | HV | The volume of one source file can be computed by $(N_1 + N_2)\log(n_1 + n_2)$. Here n_1 and n_2 denote the number of distinct operators and the number of distinct operands, respectively. N_1 and N_2 denote the total number of operators and the total number of operands, respectively. To measure this metric, operators are method names and PHP language operators, while operands are parameter and variable names. |
| Total external calls | NIC | The number of instances where a statement in the source file being measured invokes a function/method defined in another source file. |
| Fan-in | NICU | The number of source files (excluding the source file being measured), which contain statements that invoke a function/method defined in one source file being measured. |
| Fan-out | NOFCU | The number of source files (excluding the source file being measured), which contain functions/methods invoked by statements in one source file being measured. |
| Internal functions/methods called | NOIC | The number of functions/methods defined in the source file being measured which are called at least once by a statement in one same source file. |
| External functions/methods called | NOEFC | The number of functions/methods defined in other source files which are called at least once by a statement in one source file being measured. |
| External calls to functions/methods | NOECU | The number of source files (excluding the source file being measured) calling a particular function/method defined in the source file being measured, summed across all functions and methods in the source file being measured. |

TABLE II
CHARACTERISTICS OF DATASETS

| Project | # Module | # (%) Vulnerabilities | Text Features |
|------------|----------|-----------------------|---------------|
| Drupal | 202 | 62 (30.68%) | 3,886 |
| PHPMyAdmin | 322 | 27 (8.39%) | 5,232 |
| Moodle | 2,942 | 24 (0.82%) | 18,306 |

traditional SDP [8]. These classification methods and their abbreviations are summarized in Table III. From Table III, these methods can be briefly classified into six families. In particular, function family includes three methods: SL, RBFNet, and SMO. Lazy family includes one method: Ibk. Rule family includes two methods: Jrip and Ridor. Bayes family includes one method: NB. Tree family includes three methods: J48, LMT, and RF. Ensemble family considers four different ensemble learning methods: BG, AB, RF, and RS. In the abbreviations of the Ensemble family, BG+LMT means that this method uses LMT as the base classifier and uses Bagging as the ensemble method. It is not hard to find that these methods can cover different types of traditional supervised methods in machine learning. In our empirical studies, we also consider the same parameters setting used by Yang *et al.* [32] to perform these supervised methods.

For software-metric-based methods, we further consider four state-of-the-art methods (i.e., EALR [9], OneWay [10], CBS [11], and MULTI [12]) proposed in recent effort-aware JIT-SDP studies. For JIT-SDP, these methods are used to predict defective code changes, and for SVP, these methods should be used to predict vulnerable modules. We illustrate these methods in the context of SVP.

The EALR method is a customized method proposed by Kamei *et al.* [9]. Empirical results on JIT-SDP [9] showed that EALR can detect 35% defective code changes when using 20% of the effort. For SVP, this method uses $\text{vulnerable}(m_i)/\text{LOC}(m_i)$ as the dependent variable. Here, $\text{vulnerable}(m_i)$ is 1 if the module m_i is vulnerable; otherwise, $\text{vulnerable}(m_i)$ is 0. $\text{LOC}(m_i)$ denotes the LOC metric value of m_i . Then, it uses linear regression to construct the models.

The OneWay method [10] is a simple supervised method based on unsupervised methods proposed by Yang *et al.* [32]. This method identifies the best unsupervised method based on the analysis of the training data and then applies this best unsupervised method to the testing data. Empirical results on JIT-SDP showed that OneWay has competitive performance, and it performs better than most unsupervised methods [32]. For SVP, the OneWay method aims to identify the best unsupervised method from 12 unsupervised methods based on the training data.

The CBS method [11] is also a simple but improved supervised method. Empirical results on JIT-SDP [11] showed that CBS achieved better than EALR [9] and achieved similar results with Yang *et al.*'s unsupervised methods [32] and OneWay [10]. For SVP, this method first builds a logistic-regression-based classifier to identify vulnerable modules. Then, it sorts the identified vulnerable modules in the ascending order by their inspection cost based on the LOC metric.

The MULTI method [12] is a multiobjective-optimization-based supervised method. Empirical results on JIT-SDP showed that MULTI can perform significantly better than all of the state-of-the-art methods [10], [32]. For SVP, MULTI formalizes SVP as a multiobjective optimization problem. One objective is designed to maximize the number of identified vulnerable modules, and another object is designed to minimize the inspection costs. There exists an obvious conflict between these two

TABLE III
OVERVIEW OF THE METRIC-BASED SUPERVISED METHODS USING TRADITIONAL MACHINE LEARNING METHODS

| Family | Method | Abbreviation |
|----------|---------------------------------|--|
| Function | Simple Logistic | SL |
| | Radial Basis Functions Network | RBFNet |
| | Sequential Minimal Optimization | SMO |
| Lazy | K-nearest Neighbor | Ibk |
| | Propositional Rule | Jrip |
| Rule | Ripple Down Rules | Ridor |
| | Naive Bayes | NB |
| | C4.5 based Decision Tree | J48 |
| Bayes | Logistic Model Tree | LMT |
| | Random Forest | RF |
| | Bagging | BG+LMT, BG+NB, BG+SL, BG+SMO, and BG+J48 |
| Tree | Adaboost | AB+LMT, AB+NB, AB+SL, AB+SMO, and AB+J48 |
| | Rotation Forest | RF+LMT, RF+NB, RF+SL, RF+SMO, and RF+J48 |
| | Random Subspace | RS+LMT, RS+NB, RS+SL, RS+SMO, and RS+J48 |
| Ensemble | | |

objectives. MULTI uses logistic regression to build the models and uses NSGA-II [36] to generate a set of nondominated solutions (i.e., Pareto front), of which each solution denotes the coefficient vector of logistic regression. In this article, due to randomness variation inherent from MULTI, we perform ten independent runs to get a higher statistical confidence. Keeping in line with our previous study [12], we use MULTI-B to gather the best result of the solutions in ten Pareto fronts in the given testing data. Then, we use MULTI-M to gather the median result of the solutions in ten Pareto fronts in the testing data. Therefore, MULTI-B and MULTI-M can denote the optimal performance and the average performance of MULTI method, respectively.

2) *Text-Mining-Based Methods*: For SVP, the text-mining-based method was first proposed by Walden *et al.* [13], [18].

First, they preprocessed source files and extracted text features (denoted by terms). It included three phases: the tokenization phase, the stop words removing phase, and the word stemming phase. In particular, in the tokenization phase, they extracted identifier names and words in code comments. Then, they broke identifier names into tokens by using Camel casing convention. In the stop words removing phase, they considered a list of stop words by using Snowball.¹ Stop words are extremely common words, which appear to be of little value to differentiate one file from another. In the word stemming phase, they transformed each word to its root form. Here, they used a popular stemming algorithm proposed by Porter [37].

Then, they considered BOW model to compute the weights of terms. In this model, given a term in the source file, they only record its term frequency (TF). In this article, we further consider the TF-IDF model. In this model, the weight of a term is the product of its TF weight and its inverse document frequency weight.

After the weight of each term is computed based on either the BOW model or the TF-IDF model, random forest is used as the classifier.

For the convenience of the subsequent description, we use BOW-TM to denote the text-mining method based on the BOW model, and we use TFIDF-TM to denote the text mining method based on the TF-IDF model.

C. Unsupervised Methods

Unsupervised methods [32], [35] have attracted more interests in current SDP research. These methods do not need any training data, are very simple, and have a low model construction cost. These methods are particularly suitable for new projects without training data or with limited training data.

The idea of unsupervised methods is motivated by the findings of Koru *et al.* [38], [39] that smaller modules should be inspected first, since the relationship between the module size and the number of defects is logarithmic. The ManualUp model (i.e., smaller modules should be inspected first) proposed by Menzies *et al.* [40] further verified Koru *et al.*'s findings. Until now, many studies showed the competitiveness of these unsupervised methods based on the ManualUp model in JIT-SDP and FL-SDP in terms of effort-aware performance measures [32]–[35].

However, whether unsupervised SVP methods based on the ManualUp model have competitiveness when compared to supervised SVP methods in terms of the effort-aware performance measures has not been studied. In the context of SVP, for the j th metric, it computes the vulnerability-proneness probability for the i th module m_i as $1/v_{i,j}$. Here, $v_{i,j}$ denotes the metric value of the i th module m_i for the j th metric. This means that the smaller the metric value, the higher the vulnerability-proneness probability. Then, all the modules will be ranked in the descendant order according to the computed probability.

Based on the introduction of experimental subjects in Section III-A, these subjects considered 12 different metrics to measure all the extracted modules. Therefore, we consider 12 unsupervised methods based on settings by Yang *et al.* [32]. These methods are based on LOC, NLOC, NM, CC, NE, HV, NIC, NICU, NOIC, NOEFC, NOEFCU, and NOECU metrics, respectively.

¹[Online]. Available: <http://snowball.tartarus.org/algorithms/english/stop.txt>

D. Performance Measures

Instead of information-retrieval-based measures (such as precision, recall, and F1), we mainly consider effort-aware performance measures (i.e., ACC and P_{opt}). These measures are first proposed by Mende and Koschke [41], since Arisholm *et al.* [42] suggested that locating defects should take into account cost effectiveness of SDP models. For effort-aware JIT-SDP, Kamei *et al.* [9] used the total number of LOC modified by a code change as the effort required to inspect this change and considered ACC and P_{opt} measures. These two measures are also used in follow-up studies on effort-aware JIT-SDP [10]–[12], [32]. For effort-aware FL-SDP, Yan *et al.* [34] used the LOC of a module as the effort of code inspection and also considered these two measures.

For SVP, efforts for security audit and code inspection should be considered as well when applying SVP models in practice. In this article, we use the value of metric LOC as the proxy of the effort to inspect the module [34], [42]. To compute the value of these two measures, the modules in the test set should be sorted, and these modules should be inspected from the top to the bottom. For the supervised SVP methods, we sort the modules in the decreasing order according to the predicted vulnerable proneness of the trained model. For unsupervised SVP methods, we can directly get the sorted result by the ranking strategy based on the corresponding metric value. In particular, ACC denotes the recall of vulnerable modules when expending 20% of the entire efforts. P_{opt} is the normalized version of one effort-aware performance indicator. According to [43], P_{opt} can be formally defined as

$$P_{opt}(m) = 1 - \frac{\text{area(optimal)} - \text{area}(m)}{\text{area(optimal)} - \text{area(worst)}}. \quad (1)$$

Here, $\text{area}(m)$, area(optimal) , and area(worst) are the area under the curve corresponding to a proposed prediction method, the optimal method, and the worst method, respectively. For the optimal method, modules are sorted in the descendant order according to their actual vulnerability density, while for the worst method, modules are sorted in the ascendant order according to their actual vulnerability density. A simple example for illustrating P_{opt} can be found in Fig. 1. In this figure, the x -axis denotes the ratio of code inspection efforts and the y -axis denotes the ratio of found vulnerabilities.

It is not hard to find that the higher the value of performance measure, the better the performance of the method. Both ACC and P_{opt} measures are applicable to supervised and unsupervised SVP methods.

E. Experiment Setup

In this subsection, we first present data preprocessing operations. Then, we illustrate two model performance evaluation scenarios. Finally, we introduce the statistical analysis method used to analyze whether the performance differences between two methods are significant.

1) *Data Preprocessing*: To improve the performance of supervised SVP methods based on the software metric, we perform the following data preprocessing operations.

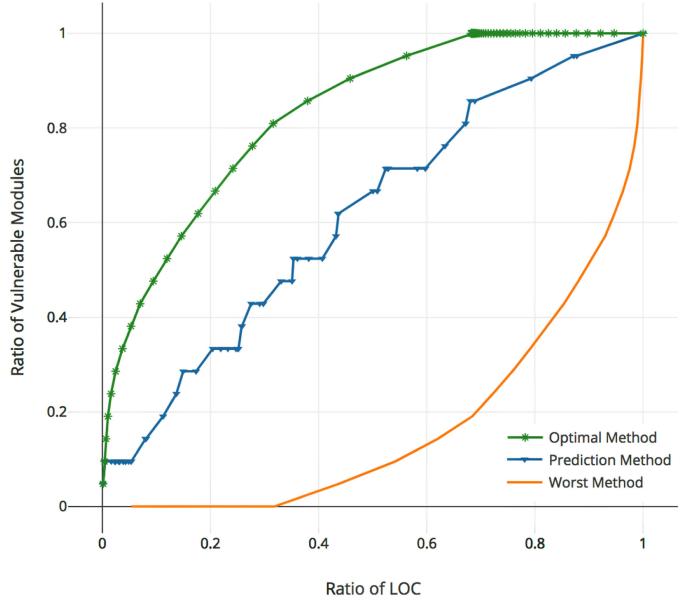


Fig. 1. Simple example for illustrating P_{opt} performance measure.

- 1) *Class imbalance learning phase*: Compared to defects, vulnerabilities are far fewer in the project, and detecting vulnerabilities is like searching for a needle in a haystack [7]. Shin and Williams [16] found that 21% of the source code files in Mozilla Firefox have faults, while only 3% of files have vulnerabilities. In our empirical subjects, the percentage of vulnerable modules for Moodle is less than 1%, as shown in Table II. Therefore, class-imbalanced methods [44] are commonly used to improve the performance of SVP. In consistent with the study, we use an unsupervised filter (i.e., SpreadSubsample) provided by Weka package and use the same setting suggested by Walden *et al.* [13].
- 2) *Redundant feature removing phase*: Redundant features may reduce the performance of trained models [45]–[50]. We remove highly redundant metrics by the method used in previous studies [9], [32].
- 3) *Feature value transformation phase*: Since most of the metrics are highly skewed [51], we perform logarithmic transformation for each numerical metric.

For supervised-based SVP methods using text mining, we perform data preprocessing only considering class imbalance learning phase suggested by Walden *et al.* [13].

Note that for supervised-based SVP methods, class imbalance phase and redundant feature removing are only performed on the training set.

2) *Model Evaluation Scenarios*: In our empirical studies, we mainly consider two common model evaluation scenarios, and the process of these two scenarios can be found in Fig. 2.

- 1) *Within-project SVP scenario*: Keeping in line with the previous study [13], we use (10×3) -fold cross validation. The process of threefold cross validation can be summarized as follows: The instances are randomly divided into three folds of equal size. Each fold has the same ratio of

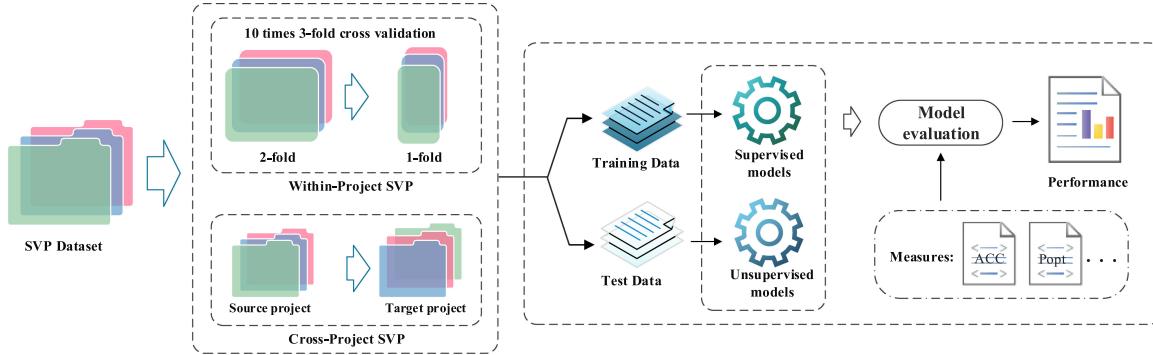


Fig. 2. Process of the within-project SVP scenario and the cross-project SVP scenario.

TABLE IV
CLIFF'S δ AND THE EFFECTIVENESS LEVEL [56]

| Cliff's δ | Effectiveness Level |
|------------------------------|---------------------|
| $ \delta < 0.147$ | Negligible |
| $0.147 \leq \delta < 0.33$ | Small |
| $0.33 \leq \delta < 0.474$ | Medium |
| $0.474 \leq \delta $ | Large |

vulnerable modules as the entire dataset. A model is trained based on the instances in two folds, and the instances in the remaining fold are used as the test data. This process is repeated three times, and each fold is used as the test set at least once. To further overcome the effect of randomness, the threefold cross validation is repeated ten times. The reason of not using tenfold cross validation is that the number of security vulnerabilities that some applications contain is too few. For example, the dataset of Moodel only contains 24 vulnerabilities. If using tenfold cross validation, each fold can only have two to three vulnerabilities.

- 2) *Cross-project SVP scenario:* In this scenario, a model is trained in one project, and then, this model is used to identify vulnerable modules in another project. This scenario is applicable for new projects without training data or projects with limited training data. In SDP, this scenario is widely investigated [52]–[55]. Since there exists randomness in the class imbalance learning phase (i.e., SpreadSubsample method), we perform each SVP method ten times independently with different random number seeds when the source project and the target project are determined.

- 3) *Statistical Analysis Method:* To examine whether there is a significant difference in the prediction performance between two methods, we first use the Benjamini–Hochberg (BH) corrected p -value [56] to examine whether a difference is statistically significant at the significance level of 0.05. If the statistical test shows a significant difference, we then use the Cliff's δ to measure the magnitude of the difference. The meaning of different Cliff's δ values and their corresponding interpretation are shown in Table IV. In summary, one method performs significantly better/worse than another method, if the BH

corrected p -value is less than 0.05, and the effectiveness level is not negligible based on Cliff's δ . On the contrary, the difference between two methods is not significant if the p -value is not less than 0.05 or the effectiveness level is negligible based on Cliff's δ .

IV. RESULT ANALYSIS

A. Result Analysis for RQ1

In this RQ, we want to compare different SVP methods in the within-project SVP scenario. Here, we use the Scott–Knott test [57] to group all SVP methods into statistically distinct ranks ($\alpha = 0.05$). In particular, it uses the hierarchical cluster analysis to partition all SVP methods into ranks. It starts by dividing the SVP methods into two ranks on the basis of mean performance metric values (i.e., ACC or P_{opt}). If the divided ranks are significantly different in statistics, it recursively executes again within each rank to further divide the ranks. It terminates when ranks can no longer be divided into statistically distinct ranks. The results of the Scott–Knott test in the within-project SVP scenario are shown in Fig. 3. The dotted lines represent groups divided by the Scott–Knott test. All methods are ordered based on their mean ranks. The distribution of P_{opt} and ACC at the within-project SVP scenario over all the three projects is shown using boxplot. The blue label denotes supervised-based methods using software metrics (i.e., Supervised_SM). The red label denotes supervised-based methods using text mining (i.e., Supervised_TM). The green label denotes unsupervised-based methods (i.e., Unsupervised). Note that since unsupervised methods LOC and NLOC have the same ranking results, we only show the result of the method LOC, and the method NLOC is omitted in the result analysis.

According to Fig. 3, we find the following: 1) based on the ACC performance measure, top two groups include six methods. In particular, LOC and HV are unsupervised methods, and OneWay, MULTI-B, CBS, and EALR are supervised methods; and (2) based on the P_{opt} performance measure, the first group includes four methods. In particular, LOC is the unsupervised method, and OneWay and MULTI-B are supervised methods.

Table V summarizes the mean ACC and P_{opt} for representative SVP methods, and for each row, the best result is bolded.

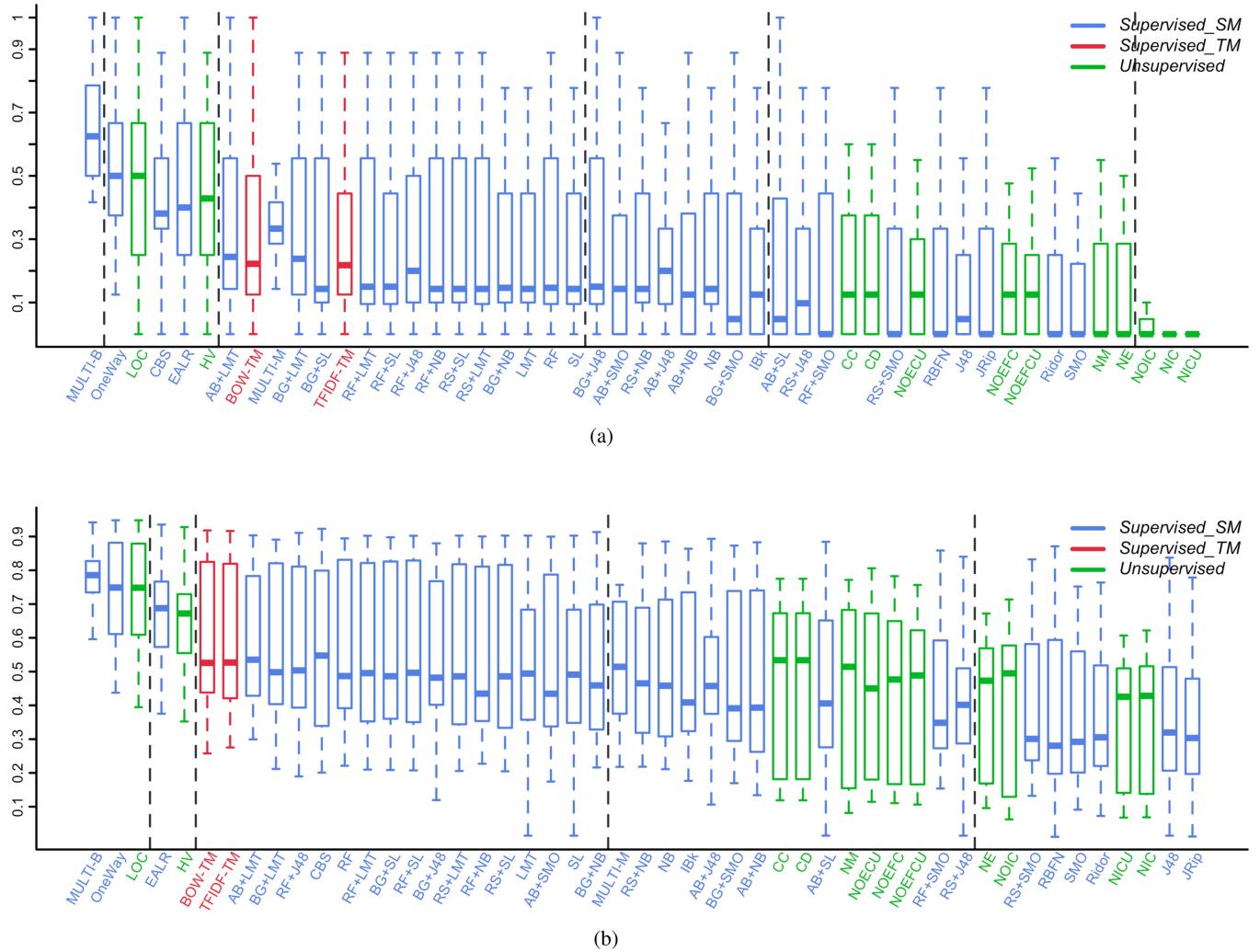
Fig. 3. Results of the Scott–Knott test in the within-project SVP scenario. (a) ACC. (b) P_{opt} .

TABLE V
RESULT FOR REPRESENTATIVE SVP METHODS IN THE WITHIN-PROJECT SVP SCENARIO

(1) ACC

| Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
|------------|---------------|--------|-------|-------|--------|---------------|----------|--------------|-------|
| | MULTI-B | OneWay | CBS | EALR | AB+LMT | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal | 0.570 | 0.490 | 0.345 | 0.355 | 0.205 | 0.162 | 0.157 | 0.479 | 0.418 |
| PHPMyAdmin | 0.839 | 0.785 | 0.722 | 0.722 | 0.648 | 0.645 | 0.610 | 0.785 | 0.722 |
| Moodle | 0.556 | 0.246 | 0.329 | 0.296 | 0.125 | 0.149 | 0.156 | 0.225 | 0.175 |
| Average | 0.655 | 0.507 | 0.466 | 0.458 | 0.326 | 0.319 | 0.308 | 0.496 | 0.438 |
| W/D/L | — | 3/0/0 | 3/0/0 | 3/0/0 | 3/0/0 | 3/0/0 | 3/0/0 | 3/0/0 | 3/0/0 |

(2) P_{opt}

| Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
|------------|---------------|--------------|-------|-------|--------|---------------|----------|--------------|-------|
| | MULTI-B | OneWay | CBS | EALR | AB+LMT | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal | 0.788 | 0.744 | 0.307 | 0.678 | 0.462 | 0.405 | 0.405 | 0.746 | 0.709 |
| PHPMyAdmin | 0.855 | 0.902 | 0.811 | 0.781 | 0.800 | 0.841 | 0.834 | 0.901 | 0.727 |
| Moodle | 0.699 | 0.569 | 0.572 | 0.524 | 0.466 | 0.529 | 0.523 | 0.566 | 0.510 |
| Average | 0.781 | 0.738 | 0.564 | 0.661 | 0.576 | 0.592 | 0.587 | 0.738 | 0.648 |
| W/D/L | - | 2/1/0 | 2/1/0 | 2/1/0 | 3/0/0 | 2/1/0 | 2/1/0 | 2/1/0 | 3/0/0 |

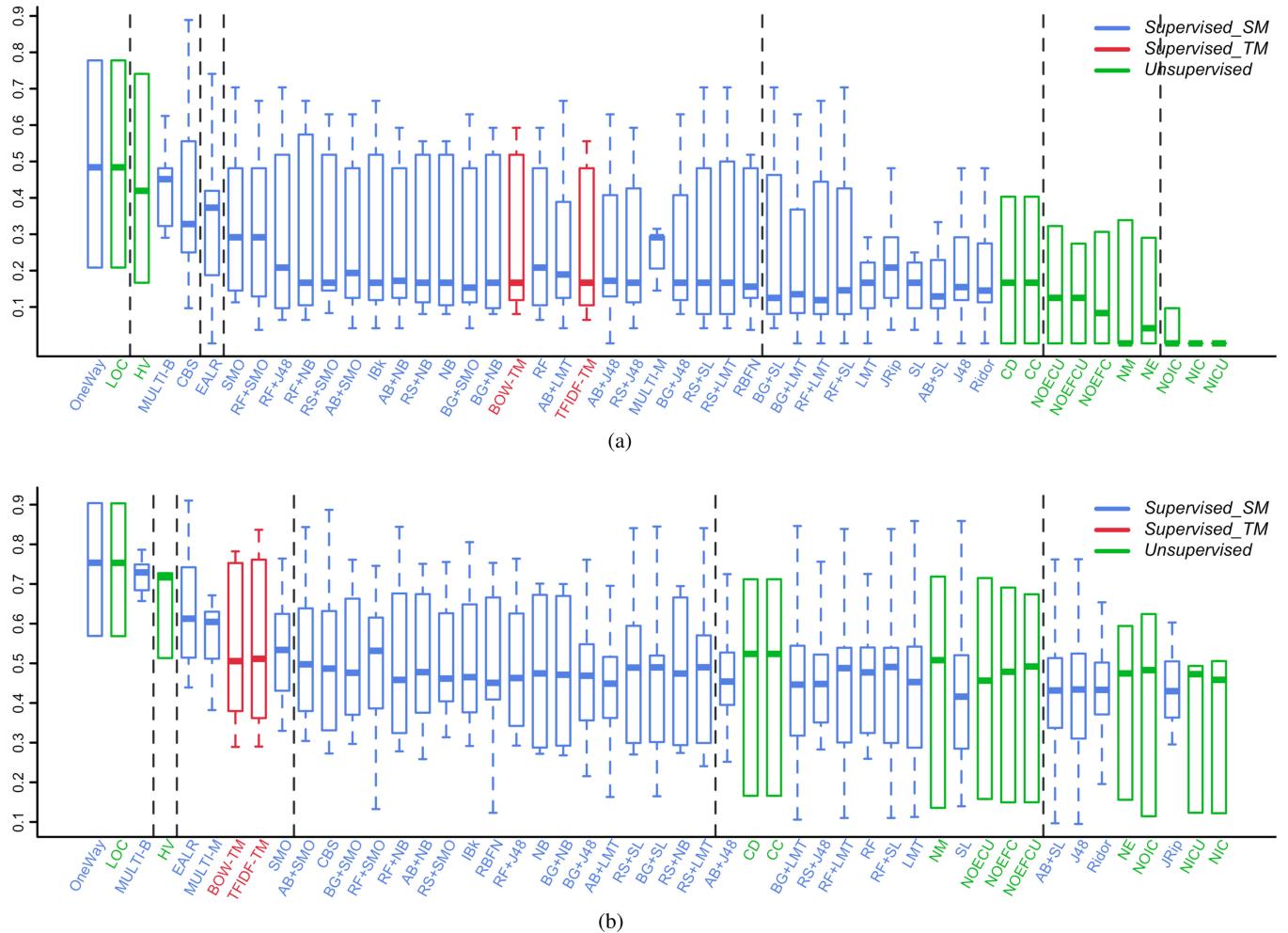


Fig. 4. Results of the Scott–Knott test in the cross-project SVP scenario. (a) ACC . (b) P_{opt} .

The selection criterion for representative SVP methods² is set as follows: For supervised methods based on the software metric, we consider four recently proposed supervised methods (i.e., MULTI-B, OneWay, CBS, and EALR) and choose a method from methods in Table III with the best performance based on a specific measure. For supervised methods based on text mining, we consider all the text mining methods (i.e., BOW-TM and TFIDF-TM). For unsupervised methods, we consider two methods (i.e., LOC and HV). In the last row of Table V, we show win/draw/loss (W/D/L) results when comparing MULTI-B with other SVP methods. W/D/L shows the number of datasets, on which MULTI-B performs significantly better, the same as, or worse than the corresponding SVP method, respectively.

According to Table V, we find the following: 1) based on the ACC performance measure, the best method is MULTI-B, with mean ACC value ranging from 0.556 to 0.839. Based on the analysis of W/D/L, MULTI-B performs significantly better than all the SVP methods on all the datasets; and (2) based on the P_{opt} performance measure, the best method is also MULTI-B,

with mean P_{opt} value ranging from 0.699 to 0.855. Based on the analysis of W/D/L, MULTI-B performs significantly better than or similar to all the SVP methods on all the datasets.

B. Result Analysis for RQ2

The result of the Scott–Knott test in the cross-project SVP scenario is shown in Fig. 4. According to Fig. 4, we find the following: 1) based on the ACC performance measure, the first group includes two methods. In particular, NLOC is one unsupervised method, and OneWay is one supervised method; and 2) based on the P_{opt} performance measure, the first group includes three methods. In particular, LOC is an unsupervised method and OneWay and MULTI-B are supervised methods.

Table VI summarizes the mean ACC and P_{opt} for representative SVP methods, and for each row, the best result is bolded. The selection criterion of representative SVP methods is consistent with the criterion used in RQ1 analysis. In the last row of Table VI, we show W/D/L results when comparing MULTI-B with other SVP methods. In this table, Drupal => PHPMyAdmin means that we use Drupal as the source project and use PHPMyAdmin as the target project.

²Notice that the selection criterion for representative SVP methods is the same in Tables V–VIII.

TABLE VI
RESULT FOR REPRESENTATIVE SVP METHODS IN THE CROSS-PROJECT SVP SCENARIO

| Source Project => Target Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
|----------------------------------|---------------|--------------|--------------|-------|-------|---------------|----------|--------------|-------|
| | MULTI-B | OneWay | CBS | EALR | SMO | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal => PHPMyAdmin | 0.421 | 0.778 | 0.585 | 0.000 | 0.493 | 0.522 | 0.511 | 0.778 | 0.741 |
| Drupal => Moodle | 0.474 | 0.208 | 0.258 | 0.379 | 0.288 | 0.150 | 0.138 | 0.208 | 0.167 |
| PHPMyAdmin => Drupal | 0.313 | 0.484 | 0.334 | 0.331 | 0.123 | 0.119 | 0.105 | 0.484 | 0.419 |
| PHPMyAdmin => Moodle | 0.533 | 0.208 | 0.279 | 0.196 | 0.292 | 0.171 | 0.171 | 0.208 | 0.167 |
| Moodle => Drupal | 0.311 | 0.484 | 0.284 | 0.415 | 0.132 | 0.103 | 0.098 | 0.484 | 0.419 |
| Moodle => PHPMyAdmin | 0.485 | 0.778 | 0.796 | 0.789 | 0.552 | 0.541 | 0.530 | 0.778 | 0.741 |
| Average | 0.423 | 0.490 | 0.423 | 0.352 | 0.313 | 0.268 | 0.259 | 0.490 | 0.442 |
| W/D/L | — | 2/0/4 | 2/2/2 | 3/1/2 | 4/0/2 | 4/0/2 | 4/0/2 | 2/0/4 | 2/0/4 |
| (2) P_{opt} | | | | | | | | | |
| Source Project => Target Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
| | MULTI-B | OneWay | CBS | EALR | SMO | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal => PHPMyAdmin | 0.722 | 0.904 | 0.428 | 0.142 | 0.730 | 0.765 | 0.770 | 0.903 | 0.727 |
| Drupal => Moodle | 0.742 | 0.569 | 0.492 | 0.614 | 0.531 | 0.494 | 0.502 | 0.569 | 0.513 |
| PHPMyAdmin => Drupal | 0.682 | 0.753 | 0.397 | 0.593 | 0.428 | 0.387 | 0.368 | 0.753 | 0.717 |
| PHPMyAdmin => Moodle | 0.760 | 0.569 | 0.506 | 0.500 | 0.540 | 0.548 | 0.532 | 0.569 | 0.513 |
| Moodle => Drupal | 0.677 | 0.753 | 0.417 | 0.728 | 0.372 | 0.303 | 0.300 | 0.753 | 0.717 |
| Moodle => PHPMyAdmin | 0.743 | 0.904 | 0.859 | 0.888 | 0.642 | 0.771 | 0.780 | 0.903 | 0.727 |
| Average | 0.721 | 0.742 | 0.517 | 0.578 | 0.541 | 0.545 | 0.542 | 0.742 | 0.652 |
| W/D/L | — | 2/0/4 | 6/0/0 | 4/0/2 | 5/0/1 | 5/0/1 | 4/0/2 | 2/0/4 | 3/0/3 |

According to Table VI, we find the following: 1) based on the ACC performance measure, the best methods are OneWay and LOC. Here, these two methods can achieve the same performance. The reason is that OneWay always identifies LOC as the best unsupervised method based on the analysis of the training data. Mean ACC values of these methods range from 0.311 to 0.533. Based on the analysis of W/D/L, the results of comparing MULTI-B with OneWay and LOC is 2/0/4 and 2/0/4, respectively. Moreover, when comparing MULTI with other methods, the number of win (W) is at least half of times in most cases; and 2) based on the P_{opt} performance measure, the best methods are also OneWay and LOC. Here, these two methods can also achieve the same performance. Mean P_{opt} values of these methods range from 0.677 to 0.760. Based on the analysis of W/D/L, the results of comparing MULTI-B with OneWay and LOC are 2/0/4 and 2/0/4, respectively. Moreover, when comparing MULTI with other methods, the number of win is at least half of times in all the cases.

C. Result Summary and Implications

Based on the result analysis on two RQs, we find that two unsupervised SVP methods (i.e., LOC and HV) and four recently proposed supervised SVP methods (i.e., MULTI, OneWay, CBS, and EALR) can achieve better performance than other SVP methods (including two text-mining-based methods) both in the within-project SVP scenario and in the cross-project SVP scenario. Therefore, we suggest that these SVP methods should be considered as baseline methods in the future effort-aware SVP studies. We especially recommend two unsupervised SVP methods, since these methods have relatively low

computation cost, are easy to implement, and have satisfactory performance.

Based on our empirical results, we surprisingly find that some simple unsupervised methods can achieve better performance. Therefore, there exists a lot of room for improvement by designing more effective SVP methods. We summarize some possible future research directions. First, there exists curse of dimensionality issue in gathered SVP datasets, especially using the text-mining-based methods. In this article, we manually remove the most highly correlated metrics to deal with the risk of multicollinearity [9]. However, more feature selection methods and dimensionality reduction methods [58] should be investigated to reduce the number of features and improve the performance of SVP models. In particular, feature selection methods can reduce the number of features in SVP datasets by selecting the most important ones, while dimensionality reduction methods can reduce the number of features by creating combined features from the original features. Second, we can focus on the class imbalanced problem. The problem of class imbalance in SVP is more challenging than SDP. In consistent with the previous study, we only use unsupervised filter (i.e., SpreadSubsample) provided by Weka package and use the same setting suggested by Walden *et al.* [13] to deal with this problem. However, more class imbalanced methods (such as oversampling methods, undersampling methods, cost-sensitive methods, ensemble methods, or hybrid methods) [59]–[61] should be investigated. Third, in this article, we only consider manually designed metrics and features by using the text-mining-based methods. In the future, we can resort to deep learning (such as long short-term memory model) and word embedding to automatically learn both semantic and syntactic features from extracted program modules [62], [63].

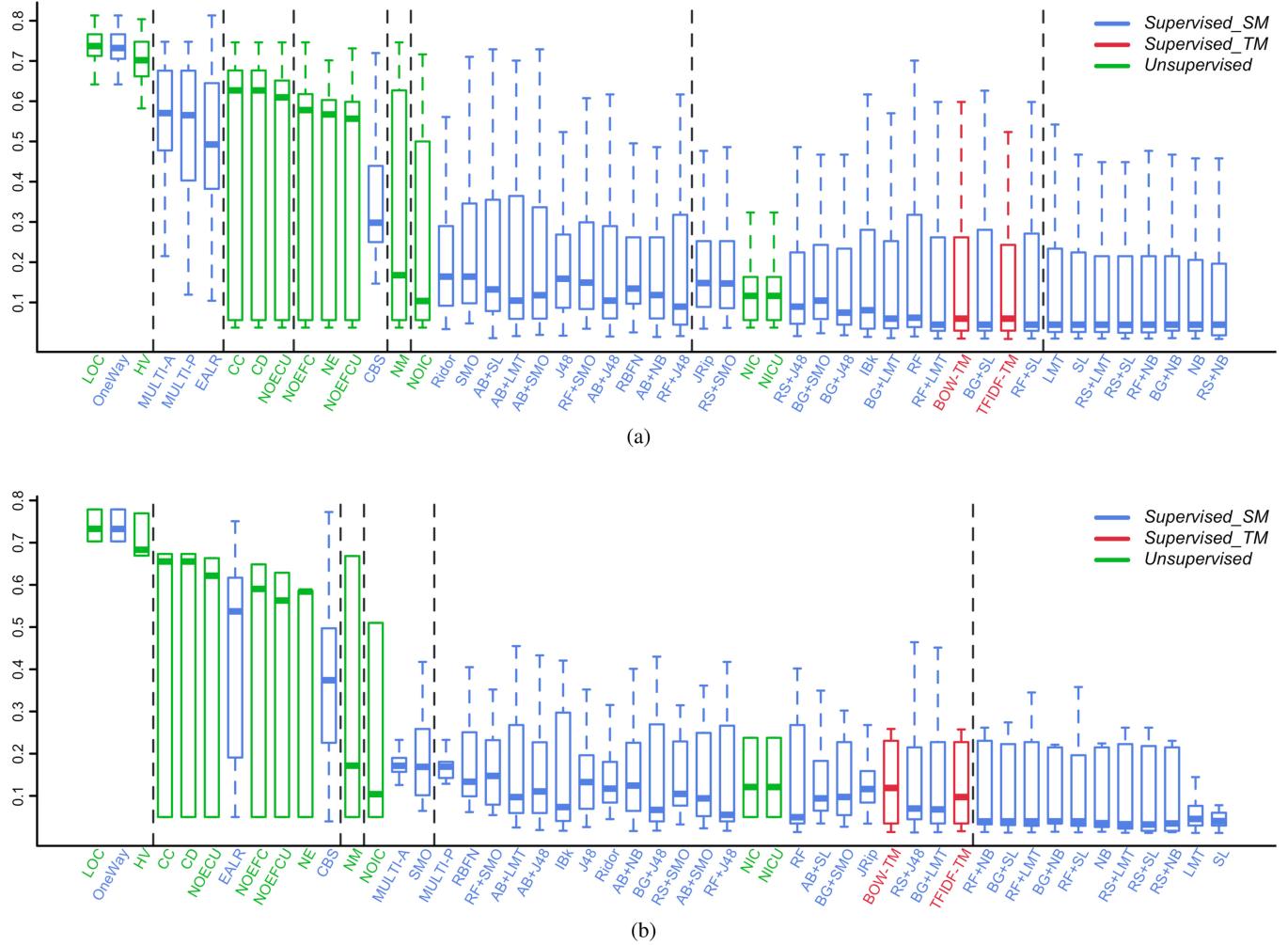


Fig. 5. Results of the Scott–Knott test in terms of $PMI@20\%$ performance measure. (a) Within-project SVP scenario. (b) Cross-project SVP scenario.

Then, we use these features to construct more powerful SVP models.

V. DISCUSSIONS

In Section IV, we mainly analyze the performance comparison results among different SVP methods both in the within-project SVP scenario and in the cross-project SVP scenario. In this section, we make further discussions. In Section V-A, we analyze why unsupervised methods can achieve better performance in terms of effort-aware performance measures and whether these methods are applicable for practical vulnerability localization. In Section V-B, we illustrate why unsupervised methods use the ManualUp model [40] in our study. Finally, in Section V-C, we analyze whether the computational cost for different supervised SVP methods is acceptable.

A. Result Analysis for $PMI@20\%$ and IFA Measures

In addition to ACC and P_{opt} , we also consider another two measures ($PMI@20\%$ and IFA) recently proposed by Huang *et al.* [11] for effort-aware JIT-SDP.

In the context of SVP, $PMI@20\%$ returns proportion of modules inspected with only 20% of the entire efforts. A higher $PMI@20\%$ value indicates that, by only using 20% of the entire efforts, developers need to inspect more modules. It means that the additional efforts required due to context switches and additional communication overhead among developers should not be ignored [64]. Therefore, when only using 20% of the entire efforts, $PMI@20\%$ and ACC evaluate SVP methods from two different perspectives. We use a simple example to illustrate the difference between ACC and $PMI@20\%$. Suppose that there are 1000 modules in the project, of which 20 modules are vulnerable. If expending 20% of the entire efforts based on the ranked list by a specific SVP method, we can only inspect 300 modules, of which five modules are vulnerable modules. Then, the ACC of this SVP method is $5/20 = 25\%$, and the $PMI@20\%$ of this method is $300/1000 = 30\%$.

The results of the Scott–Knott test in the within-project SVP scenario and the cross-project SVP scenario based on $PMI@20\%$ measure are shown in Fig. 5. From this figure, we can find that given the same inspection effort (i.e., 20% of cost), the unsupervised methods (LOC and HV methods) and the supervised method (OneWay) often need to inspect a large

TABLE VII
RESULT FOR REPRESENTATIVE SVP METHODS IN TERMS OF $PMI@20\%$ PERFORMANCE MEASURE

| Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
|------------|---------------|--------------|-------|-------|-------|---------------|----------|--------------|-------|
| | MULTI-B | OneWay | CBS | EALR | RS+NB | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal | 0.701 | 0.701 | 0.276 | 0.475 | 0.043 | 0.063 | 0.065 | 0.706 | 0.681 |
| PHPMyAdmin | 0.769 | 0.776 | 0.599 | 0.694 | 0.263 | 0.380 | 0.369 | 0.776 | 0.766 |
| Moodle | 0.690 | 0.707 | 0.258 | 0.354 | 0.017 | 0.027 | 0.029 | 0.733 | 0.668 |
| Average | 0.720 | 0.728 | 0.378 | 0.508 | 0.108 | 0.157 | 0.155 | 0.738 | 0.705 |

| (2) The Cross-Project SVP Scenario | | | | | | | | | |
|------------------------------------|---------------|--------------|-------|-------|-------|---------------|----------|--------------|-------|
| Source Project => Target Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
| | MULTI-B | OneWay | CBS | EALR | SL | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal => PHPMyAdmin | 0.308 | 0.779 | 0.375 | 0.050 | 0.062 | 0.237 | 0.237 | 0.779 | 0.769 |
| Drupal => Moodle | 0.324 | 0.732 | 0.270 | 0.193 | 0.021 | 0.017 | 0.018 | 0.733 | 0.669 |
| PHPMyAdmin => Drupal | 0.361 | 0.703 | 0.403 | 0.550 | 0.038 | 0.153 | 0.134 | 0.703 | 0.683 |
| PHPMyAdmin => Moodle | 0.353 | 0.732 | 0.442 | 0.592 | 0.030 | 0.090 | 0.073 | 0.733 | 0.669 |
| Moodle => Drupal | 0.361 | 0.703 | 0.219 | 0.568 | 0.037 | 0.037 | 0.040 | 0.703 | 0.683 |
| Moodle => PHPMyAdmin | 0.331 | 0.779 | 0.699 | 0.676 | 0.270 | 0.235 | 0.228 | 0.779 | 0.769 |
| Average | 0.340 | 0.738 | 0.401 | 0.438 | 0.076 | 0.128 | 0.122 | 0.738 | 0.707 |

number of modules. According to the description of OneWay in Section III-B, OneWay can be treated as an unsupervised method to a certain extent, since it automatically selects the best unsupervised method based on the analysis on the training set. This finding explains why unsupervised methods can achieve better prediction performance in terms of effort-aware performance measures. That is, these methods achieve higher ACC value and P_{opt} value by inspecting more modules. Table VII summarizes the mean $PMI@20\%$ for representative SVP methods in the within-project SVP scenario and the cross-project SVP scenario, respectively, and for each row, highest $PMI@20\%$ value is bolded. It is not hard to find that for two different scenarios, the unsupervised method LOC always needs to inspect the large number of modules (i.e., 73.8% modules) on average when only given 20% inspection costs.

To approximately show the code size distribution of the program modules for each project, we use the histogram to depict the frequencies of program modules in a certain range of LOC. The histogram for each project can be found in Fig. 6. From this figure, we can find that the code size distribution of modules is highly skewed, especially for the Drupal project and the Moodle project. For each of these three projects, the size of most modules is small, while the size of a few modules is large. Therefore, it is not difficult to understand that the method LOC always needs to inspect most modules.

IFA returns the number of initial false alarms encountered before we find the first vulnerable module, which is inspired by research on automatic software fault localization [65]. A higher IFA means more false positives (i.e., nonvulnerable modules are predicted as vulnerable modules) before detecting the first vulnerable module and may have an impact on developers' confidence and tolerance [66], [67].

The results of the Scott–Knott test in the within-project SVP scenario and the cross-project SVP scenario based on the IFA performance measure are shown in Fig. 7. From this figure, we can find that for almost all the unsupervised methods, many

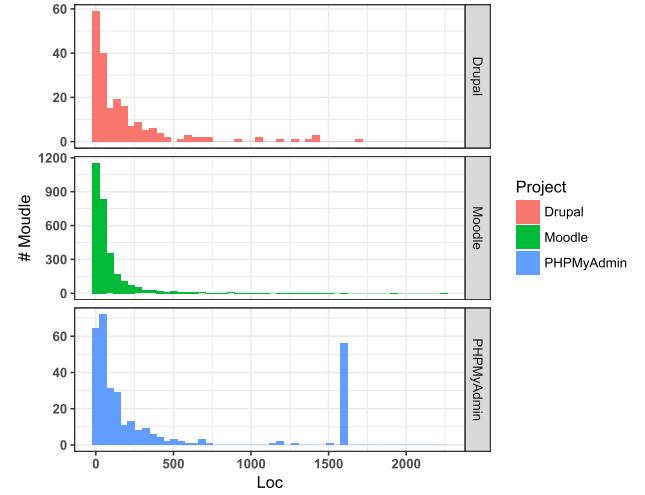


Fig. 6. Histogram for each project to show the code size distribution of the program modules.

highly ranked modules are false positives. On the contrary, MULTI-A has a smaller IFA value and thereby can improve developers' confidence. Table VIII summarizes the mean IFA for representative SVP methods in the within-project SVP scenario and the cross-project SVP scenario, respectively, and for each row, the smallest IFA value is bolded. According to Table VIII, the MULTI-B method almost can achieve the smallest IFA value. For most of the cases, the first module inspected by MULTI-B is the vulnerable module. In addition to the MULTI method, the traditional model RS+NB and two text-mining-based methods also show the competitiveness on the IFA measure. In summary, we find almost all the unsupervised methods have high false alarms, and this may have an impact on developers' confidence and tolerance. Supervised methods especially for MULTI and text-mining-based methods are preferred when considering IFA measure.

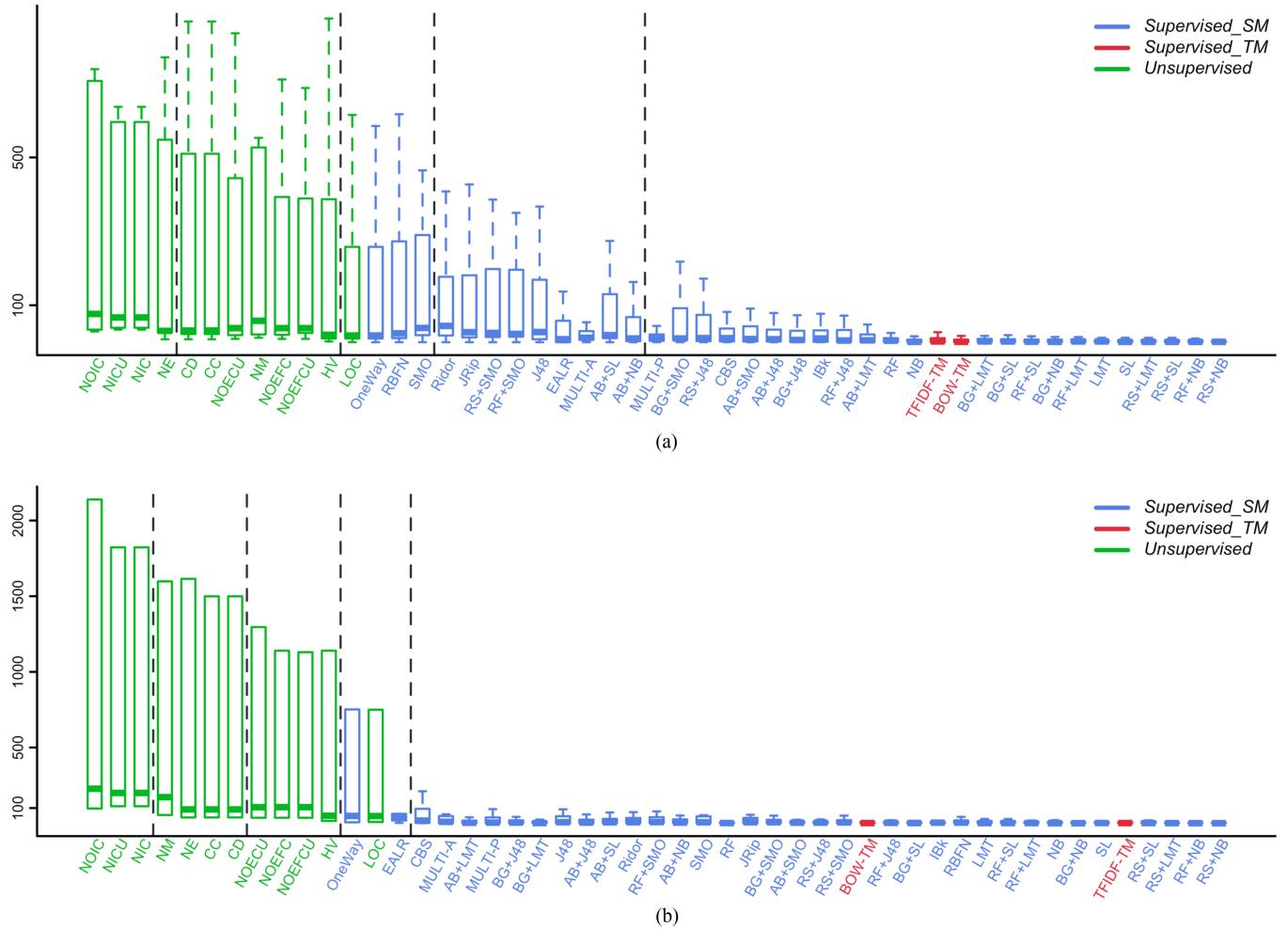


Fig. 7. Results of the Scott-Knott test in terms of IFA performance measure. (a) Within-project SVP scenario. (b) Cross-project SVP scenario.

B. Result Analysis for Comparing Unsupervised Methods With Different Ranking Strategies

In this subsection, we compare unsupervised methods with different ranking strategies. In our empirical studies, we consider unsupervised methods, which rank all the modules in the ascendent order in terms of a specific metric. Here, we further consider unsupervised methods, which rank all the modules in the descendent order.

The comparison result in the within-project SVP scenario can be found in Fig. 8, and the comparison result in the cross-project SVP scenario can be found in Fig. 9. The blue label denotes unsupervised methods based on the descendent order, and the suffix of these methods is D. The black label denotes unsupervised methods based on the ascendent order, and the suffix of these methods is A. Based on empirical results, we can find that most of the unsupervised methods based on the ascendent order outperform corresponding unsupervised methods on the descendent order. Among these unsupervised methods, the unsupervised method LOC_A can achieve the best performance except for the within-project SVP scenario using ACC. However, in this scenario, the method LOC_A is still in the first group.

C. Computational Cost Analysis

In this subsection, we analyze the model construction time for different SVP methods. Since unsupervised methods do not need any training data and are very simple (i.e., compute the vulnerability probability only by using the specific metric value), their running speed is very fast. Therefore, we only compare supervised-based SVP methods. All the methods are run on macOS High Sierra operation system (Intel i5-7360U CPU with 8 GB of memory). The average model construction time for each run in the cross-project SVP scenario is shown in Table IX. From Table IX, we can find that the model construction time of different SVP methods is acceptable. Except for MULTI, the model construction time of all the other SVP supervised methods never exceed 2 s for each run. For MULTI, since this method uses NSGA-II [36] to search for the Pareto front, the model construction time is larger than all the other methods. However, the construction time varies only from 3.362 to 6.899 s.

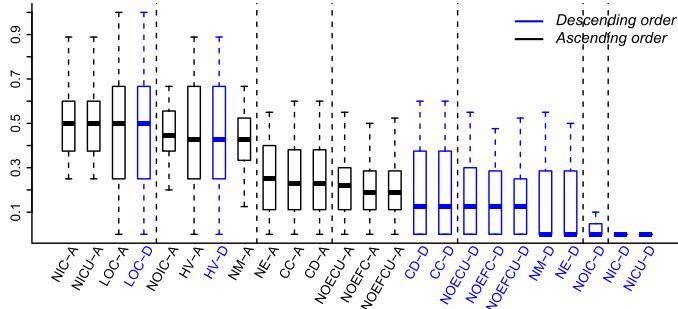
VI. THREATS TO VALIDITY

In this subsection, we mainly discuss the potential threats to validity of our empirical studies.

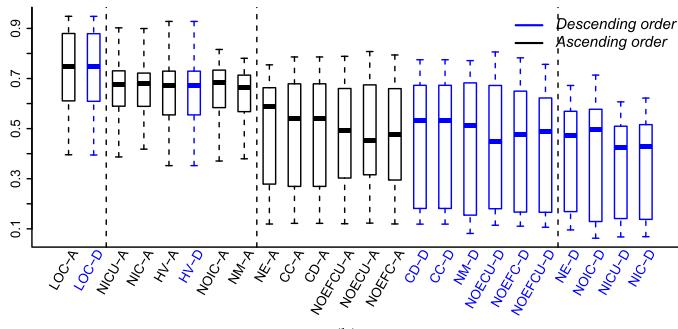
TABLE VIII
RESULT FOR REPRESENTATIVE SVP METHODS IN TERMS OF *IFA* PERFORMANCE MEASURE

| Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
|------------|---------------|---------|---------|---------|--------------|---------------|----------|--------------|---------|
| | MULTI-B | OneWay | CBS | EALR | RS+NB | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal | 0.000 | 17.267 | 1.367 | 3.333 | 0.100 | 0.100 | 0.167 | 17.267 | 19.400 |
| PHPMyAdmin | 2.233 | 8.200 | 12.933 | 8.900 | 1.700 | 2.667 | 3.967 | 8.267 | 10.033 |
| Moodle | 31.067 | 398.367 | 116.300 | 256.267 | 9.267 | 22.100 | 22.500 | 407.133 | 515.733 |

| (2) The Cross-Project SVP Scenario | | | | | | | | | |
|------------------------------------|---------------|---------|---------|----------|--------------|---------------|----------|--------------|----------|
| Source Project => Target Project | Supervised_SM | | | | | Supervised_TM | | Unsupervised | |
| | MULTI-B | OneWay | CBS | EALR | RS+NB | BOW-TM | TFIDF-TM | LOC | HV |
| Drupal => PHPMyAdmin | 11.300 | 6.000 | 7.000 | 61.600 | 0.000 | 0.400 | 0.700 | 8.000 | 15.000 |
| Drupal => Moodle | 0.300 | 752.000 | 141.800 | 36.400 | 4.500 | 7.600 | 6.200 | 750.000 | 1140.000 |
| PHPMyAdmin => Drupal | 0.000 | 48.000 | 24.300 | 24.700 | 0.000 | 5.900 | 3.800 | 47.000 | 49.000 |
| PHPMyAdmin => Moodle | 6.000 | 752.000 | 531.900 | 1224.100 | 7.500 | 55.400 | 22.700 | 750.000 | 1140.000 |
| Moodle => Drupal | 0.000 | 48.000 | 5.800 | 7.200 | 0.000 | 0.000 | 0.100 | 47.000 | 49.000 |
| Moodle => PHPMyAdmin | 0.000 | 6.000 | 8.900 | 36.100 | 0.000 | 0.100 | 0.800 | 8.000 | 15.000 |



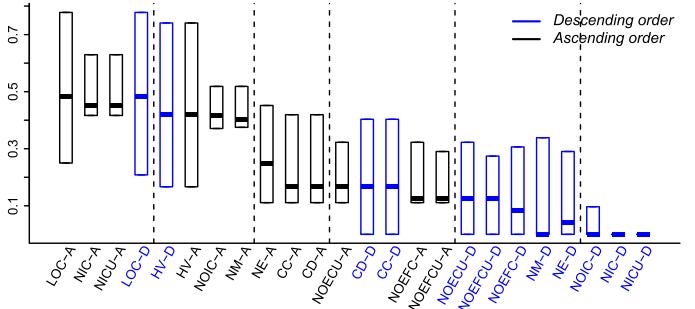
(a)



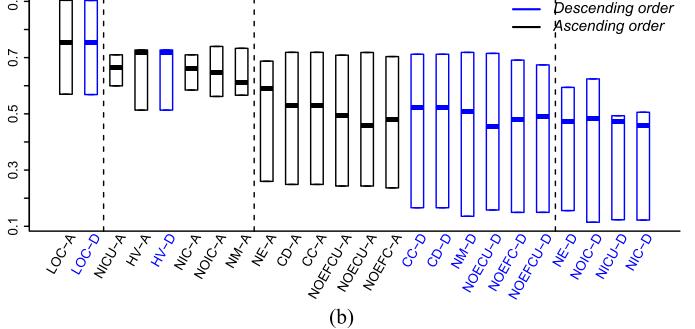
(b)

Fig. 8. Comparing Unsupervised methods with different ranking strategies in the within-project SVP scenario. (a) *ACC*. (b) *P_{opt}*.

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. First, we have double checked our experiments and implementations of different SVP methods. Still, there could be errors that we have not noticed. Second, to guarantee the correctness of classifiers and statistical analysis methods, we use mature third-party libraries, such as R and Weka. Third, we use the default value of hyperparameter for our supervised SVP methods. Hyperparameter optimization may improve the performance of SVP methods and need further investigating in our future work. Finally, we use the code shared by Walden *et al.* [13] to ensure the correctness of classical baseline methods.



(a)



(b)

Fig. 9. Comparing unsupervised methods with different ranking strategies in the cross-project SVP scenario. (a) *ACC*. (b) *P_{opt}*.

The code includes the random forest method based on software metrics and the text mining method based on the BOW model.

Threats to external validity are about whether the observed experimental results can be generalized to other subjects. We only consider web applications projects written in PHP. These datasets are high quality and widely used in previous SVP studies [13], [20]–[22]. In the future, we want to consider more commercial and open-source projects by other programming languages. Also, we want to investigate more projects in other application domains, such as mobile applications [68]. When investigating other projects, the metrics either based on traditional software metrics or based on term vectors by text mining (introduced in Section III-A) are independent of programming languages.

TABLE IX
MODEL CONSTRUCTION TIME OF DIFFERENT SUPERVISED-BASED SVP METHODS IN THE CROSS-PROJECT SVP SCENARIO (UNIT: SECOND)

| Method | Drupal | PHPMyAdmin | Moodle |
|----------|--------|------------|--------|
| BG+LMT | 0.359 | 0.114 | 0.207 |
| BG+NB | 0.177 | 0.042 | 0.040 |
| BG+SL | 0.173 | 0.085 | 0.096 |
| BG+SMO | 0.068 | 0.051 | 0.056 |
| BG+J48 | 0.041 | 0.035 | 0.027 |
| AB+LMT | 0.450 | 0.131 | 0.176 |
| AB+NB | 0.053 | 0.030 | 0.031 |
| AB+SL | 0.100 | 0.037 | 0.051 |
| AB+SMO | 0.060 | 0.035 | 0.054 |
| AB+J48 | 0.037 | 0.026 | 0.030 |
| RF+LMT | 0.369 | 0.123 | 0.183 |
| RF+NB | 0.323 | 0.056 | 0.063 |
| RF+SL | 0.265 | 0.090 | 0.117 |
| RF+SMO | 0.187 | 0.077 | 0.073 |
| RF+J48 | 0.157 | 0.047 | 0.052 |
| RS+LMT | 0.208 | 0.084 | 0.118 |
| RS+NB | 0.123 | 0.037 | 0.033 |
| RS+SL | 0.140 | 0.061 | 0.080 |
| RS+SMO | 0.058 | 0.053 | 0.051 |
| RS+J48 | 0.036 | 0.024 | 0.029 |
| EALR | 0.164 | 0.065 | 0.071 |
| OneWay | 0.070 | 0.036 | 0.028 |
| CBS | 0.134 | 0.106 | 0.111 |
| MULTI | 6.899 | 4.198 | 3.362 |
| BOW-TM | 0.369 | 0.708 | 0.251 |
| TFIDF-TM | 0.497 | 1.302 | 0.369 |

Therefore, we can directly use these metrics to measure the modules extracted from projects using other programming languages or from other application domains. However, common vulnerabilities of different applications are quite different. Therefore, designing extra specific metrics may improve the performance of SVP models when considering the characteristic of these application domains.

Threats to conclusion validity are mainly concerned with inappropriate use of statistical analysis techniques. We perform the Scott–Knott test to rank a great number of SVP methods and Scott–Knott is widely used in previous empirical studies for SDP [8], [11], [12], [32].

Threats to construct validity are about whether the performance measures used in the empirical studies reflect the real-world situation. In this article, we consider effort-aware performance measures (i.e., ACC and P_{opt}), which are more common and suitable for real test scenarios (considering the testing resources are limited). Therefore, information-retrieval-based measures (such as precision, recall, and F1) are not considered in this article.

VII. CONCLUSION

Compared to SDP, research on SVP has yet to mature. In this article, we considered 48 different supervised and unsupervised SVP methods. In our empirical studies, we used three web

applications as benchmark, including 3466 modules and 223 vulnerabilities. Based on effort-aware performance measures, we found that some unsupervised methods (i.e., LOC and HV) and recently proposed state-of-the-art supervised methods (i.e., MULTI, OneWay, CBS, and EALR) can achieve better performance both in the within-project SVP scenario and in the cross-project SVP scenario. Finally, we also thoroughly analyze the reasons why unsupervised SVP methods can achieve better performance and point out the problem when using unsupervised SVP methods in practice.

In the future, we plan to extend our research in several ways. We first want to investigate the generalization of our empirical results by considering more open-source projects or commercial projects. Second, we want to consider more novel metrics related to security vulnerabilities, since previous studies find that the performance of SVP models based on traditional software metrics is poor [7], [14], [69]. Third, in addition to cross validation, we want to investigate other model evaluation methods (such as holdout and bootstrapping [70]) for the within-project SVP scenario. Finally, we want to investigate the actual efforts required to inspect different modules.

REFERENCES

- [1] S. Heelan, “Vulnerability detection systems: Think cyborg, not robot,” *IEEE Secur. Privacy*, vol. 9, no. 3, pp. 74–77, May/Jun. 2011.
- [2] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba, “Collaborative security: A survey and taxonomy,” *ACM Comput. Surv.*, vol. 48, no. 1, 2015, Art. no. 1.
- [3] H. Shahriar and M. Zulkernine, “Mitigating program security vulnerabilities: Approaches and challenges,” *ACM Comput. Surv.*, vol. 44, no. 3, 2012, Art. no. 11.
- [4] S. M. Ghaffarian and H. R. Shahriari, “Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey,” *ACM Comput. Surv.*, vol. 50, no. 4, 2017, Art. no. 56.
- [5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.
- [6] Y. Kamei and E. Shihab, “Defect prediction: Accomplishments and future challenges,” in *Proc. Int. Conf. Softw. Anal., Evol., Reeng.*, 2016, pp. 33–45.
- [7] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista,” in *Proc. Int. Conf. Softw. Testing, Verification Validation*, 2010, pp. 421–428.
- [8] B. Ghotra, S. McIntosh, and A. E. Hassan, “Revisiting the impact of classification techniques on the performance of defect prediction models,” in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 789–800.
- [9] Y. Kamei *et al.*, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
- [10] W. Fu and T. Menzies, “Revisiting unsupervised learning for defect prediction,” in *Proc. Joint Meeting Eur. Softw. Eng. Conf./ACM SIGSOFT Symp. Found. Softw. Eng.*, 2017, pp. 72–83.
- [11] Q. Huang, X. Xia, and D. Lo, “Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction,” in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 159–170.
- [12] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, “MULTI: Multi-objective effort-aware just-in-time software defect prediction,” *Inf. Softw. Technol.*, vol. 93, pp. 1–13, 2018.
- [13] J. Walden, J. Stuckman, and R. Scandariato, “Predicting vulnerable components: Software metrics vs text mining,” in *Proc. Int. Symp. Softw. Rel. Eng.*, 2014, pp. 23–33.
- [14] A. Meneely and L. Williams, “Strengthening the empirical analysis of the relationship between Linus’ law and software security,” in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2010, Art. no. 9.
- [15] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.

- [16] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Softw. Eng.*, vol. 18, no. 1, pp. 25–59, 2013.
- [17] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *Proc. Int. Conf. Mach. Learn. Appl.*, 2018, pp. 757–762.
- [18] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proc. Int. Workshop Secur. Meas. Metrics*, 2012, pp. 7–10.
- [19] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
- [20] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction," in *Proc. Int. Conf. Eng. Complex Comput. Syst.*, 2015, pp. 40–49.
- [21] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "Predicting vulnerable components via text mining or software metrics? An effort-aware perspective," in *Proc. Int. Conf. Softw. Qual., Rel. Secur.*, 2015, pp. 27–36.
- [22] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Trans. Rel.*, vol. 66, no. 1, pp. 17–37, Mar. 2017.
- [23] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 481–490.
- [24] W. Liu, S. Liu, Q. Gu, X. Chen, and D. Chen, "FECS: A cluster based feature selection method for software fault prediction with noises," in *Proc. Annu. Comput. Softw. Appl. Conf.*, 2015, pp. 276–281.
- [25] A. Monden *et al.*, "Assessing the cost effectiveness of fault prediction in acceptance testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1345–1357, Oct. 2013.
- [26] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, Jr, "Does bug prediction support human developers? Findings from a Google case study," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 372–381.
- [27] J. Nam and S. Kim, "CLAMI: Defect prediction on unlabeled datasets," in *Proc. Int. Conf. Autom. Softw. Eng.*, 2015, pp. 452–463.
- [28] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 309–320.
- [29] S. Wang, J. Nam, and L. Tan, "QTEP: Quality-aware test case prioritization," in *Proc. Joint Meeting Eur. Softw. Eng. Conf./ACM SIGSOFT Symp. Found. Softw. Eng.*, 2017, pp. 523–534.
- [30] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018.
- [31] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.
- [32] Y. Yang *et al.*, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. Int. Symp. Found. Softw. Eng.*, 2016, pp. 157–168.
- [33] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 11–19.
- [34] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 344–353.
- [35] Y. Zhou *et al.*, "How far we have progressed in the journey? An examination of cross-project defect prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 1, 2018, Art. no. 1.
- [36] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
- [37] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.
- [38] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *IEEE Trans. Softw. Eng.*, vol. 35, no. 2, pp. 293–304, Mar./Apr. 2009.
- [39] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 473–498, 2008.
- [40] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Autom. Softw. Eng.*, vol. 17, no. 4, pp. 375–407, 2010.
- [41] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. Eur. Conf. Softw. Maintenance Reeng.*, 2010, pp. 107–116.
- [42] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.
- [43] Y. Kamei, S. Matsumoto, A. Monden, K.-I. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
- [44] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.
- [45] B. Ghotra, S. McIntosh, and A. E. Hassan, "A large-scale study of the impact of feature selection techniques on defect classification models," in *Proc. Int. Conf. Mining Softw. Repositories*, 2017, pp. 146–157.
- [46] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Trans. Softw. Eng.*, vol. 39, no. 4, pp. 552–569, Apr. 2013.
- [47] W. Liu, S. Liu, Q. Gu, J. Chen, X. Chen, and D. Chen, "Empirical studies of a two-stage data preprocessing approach for software fault prediction," *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 38–53, Mar. 2016.
- [48] S. Liu, X. Chen, W. Liu, J. Chen, Q. Gu, and D. Chen, "FECAR: A feature selection framework for software defect prediction," in *Proc. Annu. Comput. Softw. Appl. Conf.*, 2014, pp. 426–435.
- [49] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2016, pp. 309–320.
- [50] Z. Xu, J. Xuan, J. Liu, and X. Cui, "MICHAC: Defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering," in *Proc. Int. Conf. Softw. Anal., Evol., Reeng.*, 2016, pp. 370–381.
- [51] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2010, Art. no. 4.
- [52] S. Hosseini, B. Turhan, and D. Gunaratna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.
- [53] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.
- [54] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," *J. Comput. Sci. Technol.*, vol. 32, no. 6, pp. 1090–1107, 2017.
- [55] Z. Xu, J. Liu, X. Luo, and T. Zhang, "Cross-version defect prediction via hybrid active learning with kernel principal component analysis," in *Proc. Int. Conf. Softw. Anal., Evol. Reeng.*, 2018, pp. 209–220.
- [56] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Roy. Statist. Soc. Ser. B, Methodol.*, vol. 57, pp. 289–300, 1995.
- [57] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "ScottKnott: A package for performing the Scott-Knott clustering algorithm in R," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17, 2014.
- [58] C. Ni, X. Chen, F. Wu, Y. Shen, and Q. Gu, "An empirical study on Pareto based multi-objective feature selection for software defect prediction," *J. Syst. Softw.*, vol. 152, pp. 215–238, 2019.
- [59] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Trans. Softw. Eng.*, to be published.
- [60] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, to be published.
- [61] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," *Inf. Softw. Technol.*, vol. 106, pp. 161–181, 2019.
- [62] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 450–461.
- [63] C. Chen, Z. Xing, and Y. Liu, "By the community & for the community: A deep learning approach to assist collaborative editing in Q&A sites," in *Proc. 21st ACM Conf. Comput.-Supported Cooperative Work Soc. Comput.*, 2018, Art. no. 32.
- [64] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 19–29.
- [65] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

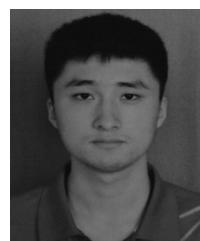
- [66] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.
- [67] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.
- [68] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of Android malware for effective malware comprehension, detection, and classification," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 306–317.
- [69] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Secur.*, 2015, Art. no. 4.
- [70] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.



Xiang Chen (M'19) received the B.Sc. degree in management from Xi'an Jiaotong University, Xi'an, China, in 2002, and the M.Sc. and Ph.D. degrees in computer software and theory from Nanjing University, Nanjing, China, in 2008 and 2011, respectively.

He is currently an Associate Professor with the School of Information Science and Technology, Nantong University, Nantong, China. He has authored more than 40 papers in referred journals or conferences, including *Information and Software Technology*, the *Journal of Systems and Software*, the IEEE TRANSACTIONS ON RELIABILITY, the *Journal of Software: Evolution and Process*, *Software Quality Journal*, the *Journal of Computer Science and Technology*, the IEEE Computer Society Signature Conference on Computers, Software and Applications, the Asia-Pacific Software Engineering Conference, and the ACM Symposium on Applied Computing. His research interests mainly include software maintenance and software testing, such as security vulnerability prediction, software defect prediction, combinatorial testing, regression testing, and fault localization.

Dr. Chen is a Senior Member of the China Computer Federation and a member of the ACM.



Yingquan Zhao received the B.S. degree in computer science from Nantong University, Nantong, China, in 2018. He is currently working toward the master's degree in software engineering with the College of Intelligence and Computing, Tianjin University, Tianjin, China.

His research interests include security vulnerability prediction, software defect prediction, and data mining.



Zhanqi Cui (M'19) received the Ph.D. degree in computer science from Nanjing University, Nanjing, China, in 2011.

He is currently an Associate Professor in Software Engineering with the Beijing Information Science and Technology University, Beijing, China. His research interests include software analysis and testing.



Guozhu Meng (M'13) received the bachelor's and master's degrees in computer science and technology from Tianjin University, Tianjin, China, in 2009 and 2012, respectively, and the Ph.D. degree in software engineering from Nanyang Technological University, Singapore, in 2017.

He is currently an Associate Professor in Software Engineering with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His research interests include mobile security, software engineering, program analysis, and vulnerability analysis and detection.



Yang Liu (M'10) received the bachelor's and Ph.D. degrees in computer science from the National University of Singapore (NUS), Singapore, in 2005 and 2010, respectively.

He performed his postdoctoral research with NUS. He is currently an Associate Professor in Software Engineering with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research interests include software engineering, cyber-physical systems, formal methods, and security.



Zan Wang (M'18) received the B.Sc. degree in applied mathematics, the master's degree in computer science, and the Ph.D. degree in information systems from Tianjin University, Tianjin, China, in 2000, 2004, and 2009, respectively.

He is currently an Associate Professor in Software Engineering with the College of Intelligence and Computing, Tianjin University. His research interests mainly include software testing and analysis, such as software bug localization, concurrency bugs detection, and automatic program repair.