Combining Software Metrics and Text Features for Vulnerable File Prediction

Yun Zhang*, David Lo†, Xin Xia*‡, Bowen Xu*, Jianling Sun*, Shanping Li*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
yunzhang28@zju.edu.cn, davidlo@smu.edu.sg, {xxkidd, max_xbw, sunjl, shan}@zju.edu.cn

***Abstract*—In recent years, to help developers reduce time and effort required to build highly secure software, a number of prediction models which are built on different kinds of features have been proposed to identify vulnerable source code files. In this paper, we propose a novel approach VULPREDICTOR to predict vulnerable files; it analyzes software metrics and text mining *together* to build a *composite* prediction model. VULPREDICTOR first builds 6 underlying classifiers on a training set of vulnerable and non-vulnerable files represented by their software metrics and text features, and then constructs a meta classifier to process the outputs of the 6 underlying classifiers. We evaluate our solution on datasets from three web applications including Drupal, PHPMyAdmin and Moodle which contain a total of 3,466 files and 223 vulnerabilities. The experiment results show that VULPREDICTOR can achieve F1 and EffectivenessRatio@20% scores of up to 0.683 and 75%, respectively. On average across the 3 projects, VULPREDICTOR improves the F1 and EffectivenessRatio@20% scores of the best performing state-of-the-art approaches proposed by Walden et al. by 46.53% and 14.93%, respectively.**

***Keywords*—Vulnerable File, Machine Learning, Text Mining**

I. INTRODUCTION

Many software systems encounter security problems during their lifetime and ensuring software security is a critical part of a software development process. Developers often need to invest much testing and debugging effort to build high quality software that are free from security issues. However, due to limited resources, it is often not possible to thoroughly check every file in a software system. Developers often need to focus their checks on more vulnerable files. Unfortunately, the identification of these vulnerable files is not trivial since there are often many files in a software system, and only few of them are vulnerable.

To deal with this problem, Walden et al. have proposed prediction models that can identify more vulnerable files to optimize the allocation of testing and debugging resources – more resources should be allocated to files that are more likely to be vulnerable [1]. They compare two prediction models built on software metrics and text features respectively, using the random forest classification algorithm. An evaluation of these models on a hand-curated dataset containing 223 vulnerabilities from 3 web applications shows that the model built on text features perform better. However, the effectiveness of the prediction models is still relatively low and needs to be improved.

‡Corresponding author.

In this paper, we propose a two-tier composite approach called VULPREDICTOR to predict vulnerable files. To improve prediction accuracy, it considers software metrics and text features *together*, and *combine multiple* prediction models together. In the first tier, VULPREDICTOR builds 6 underlying prediction models from a training set of files, represented by their software metrics and text features, and labeled as vulnerable or not. Given a new file to predict (as vulnerable or not), each of the 6 underlying prediction models predicts the probability of the new file to be vulnerable. In the second tier, VULPREDICTOR builds another prediction model (referred to as the meta classifier) based on the prediction results of the 6 underlying classifiers.

We use two well-known metrics to evaluate the effectiveness of a prediction algorithm: F1 [2] and cost effectiveness [3], [4], [5], [6], [7] scores. We use F1-score [2], which is a summary measure that combines precision and recall, as one of the metrics. F1-score evaluates if an increase in precision (recall) outweighs a reduction in recall (precision). Cost effectiveness evaluates an algorithm's effectiveness for a given cost threshold, such as a certain percentage of files in a project to inspect. When a team has limited time and resources to perform inspection, it is crucial that the top percentage of files predicted to be vulnerable are correctly predicted. In this paper, we use EffectivenessRatio@20% (ER@20%), c.f., [7], as the default cost effectiveness metric. The ER@K% score of an algorithm is the ratio of the number of vulnerable files detected by the algorithm to the number detected by a *perfect* algorithm that ranks all vulnerable files first followed by non-vulnerable ones, considering the top K% of the source code files.

We evaluate the algorithms on datasets from 3 open-source web applications written in PHP, i.e., Drupal, PHPMyAdmin and Moodle, which are provided by Walden et al. [1]. The datasets contain a total of 3,466 files along with their labels (i.e., vulnerable or not) and 223 of them are vulnerable ones. The experiment results show that our VULPREDICTOR can achieve F1-scores of 0.683, 0.34, and 0.071, and ER@20% scores of 75%, 51.9%, and 29.2%, for Drupal, PHPMyAdmin, and Moodle datasets, respectively. We compare VULPREDICTOR with Walden et al.'s approaches that build models from text features ($Walden^{Textmining}$) and software metrics ($Walden^{Metrics}$) [1]. On average across the 3 projects, VULPREDICTOR improves the F1 and EffectivenessRatio@20% scores of the best performing state-of-the-art approach proposed by Walden et al. (i.e., $Walden^{Textmining}$) by 46.53% and 14.93%, respectively. Moreover, the experiments show that

978-1-4673-8581-7/15 $31.00 © 2015 IEEE
DOI 10.1109/ICECCS.2015.15

VULPREDICTOR achieves better F1 and ER@20% scores than its 6 underlying classifiers.

The main contributions of this paper are as follows:

1) We propose a composite algorithm VULPREDICTOR to predict vulnerable files in software applications. It analyzes software metrics and text features *together* and is built on an *ensemble* of many classifiers.
2) We evaluate VULPREDICTOR on datasets from 3 web applications. The experiment results show that VULPREDICTOR performs better than Walden et al.'s approaches and VULPREDICTOR's 6 underlying classifiers.

The remainder of this paper is organized as follows. We elaborate the motivation of our work in Section II. We describe the overall framework and details of VULPREDICTOR in Section III. We present our experiments and their results in Section IV. We review related work in Section V. We conclude and mention future work in Section VI.

II. MOTIVATION

In this section, we elaborate the intuition why VULPREDICTOR employs an ensemble of classifiers. Many classification algorithms proposed in machine learning field can be used to build a vulnerability prediction model, however it is not clear if any one of those can always perform best in all cases. If there exists an algorithm that always performs better than other algorithms, then there is no necessity to build an ensemble of classifiers. To investigate this, we use different classification algorithms to build models from software metrics and text features extracted from a set of training source code files, and investigate if any one of the models outperforms the rest.

We apply different classifiers on two datasets containing vulnerable and non-vulnerable files of Drupal and PHPMyAdmin. For each file, we extract their software metrics and text features. From software metrics extracted from a set of labeled (i.e., vulnerable or non-vulnerable) files, we use Random Forest, Naive Bayes, and Decision Tree to build three prediction models (aka. classifiers) – one using each algorithm. From text features data, we use Random Forest, Naive Bayes, and Complement Naive Bayes to build another three models. Thus, in total we have 6 models/classifiers. We use 10-fold cross validation setting to evaluate the classifiers and use precision, recall, and F1-score to evaluate the effectiveness of these single classifiers. More specifically, we divide all the files in a dataset into 10 equal-sized folds, and we choose 9 folds of the data for training, and evaluate the effectiveness of an approach in the remaining test fold; the above process iterates 10 times (using different test fold) and the aggregate precision, recall, and F1-score across the 10 iterations is reported.

Tables I and II present the precision, recall, and F1 scores of the 6 classifiers for Drupal and PHPMyAdmin datasets. We notice that some classifiers can achieve high precision, e.g., Random Forest applied on text features for Drupal, Random Forest applied on software metrics for PHPMyAdmin, and Naive Bayes applied on software metrics for Drupal; some others have high recall, e.g., Decision Tree applied on software metrics for Drupal, Naive Bayes applied on text features for PHPMyAdmin, and Complement Naive Bayes applied on text features for PHPMyAdmin. For Drupal, Decision Tree applied on software metrics achieves the best F1-score, but for PHPMyAdmin, Naive Bayes applied on software metrics achieves the best F1-score. This means different classifiers are able to identify different vulnerable files, and for different applications and vulnerabilities, the effectiveness of these classifiers are different. This finding motivates us to combine different and complementary classifiers to predict vulnerable files more effectively.

TABLE I. PRECISION, RECALL AND F1 SCORES OF 6 SINGLE CLASSIFIERS BUILT FROM DRUPAL DATASET.

Algorithm	Precision	Recall	F1-score
$RandomForest^{metrics}$	0.633	0.500	0.559
$NaiveBayes^{metrics}$	0.658	0.403	0.500
$DecisionTree^{metrics}$	0.582	0.629	0.605
$RandomForest^{text}$	0.725	0.468	0.569
$NaiveBayes^{text}$	0.400	0.581	0.474
$ComplementNaiveBayes^{text}$	0.387	0.581	0.465

TABLE II. PRECISION, RECALL AND F1 SCORES OF 6 SINGLE CLASSIFIERS BUILT FROM PHPMYADMIN DATASET.

Algorithm	Precision	Recall	F1-score
$RandomForest^{metrics}$	0.667	0.148	0.242
$NaiveBayes^{metrics}$	0.237	0.333	0.277
$DecisionTree^{metrics}$	0.417	0.185	0.256
$RandomForest^{text}$	0.500	0.037	0.069
$NaiveBayes^{text}$	0.111	0.926	0.198
$ComplementNaiveBayes^{text}$	0.112	1	0.201

III. OUR PROPOSED APPROACH

In this section, we first present the overall framework of our VULPREDICTOR approach. Then we elaborate the basic classifiers used by VULPREDICTOR, and how these classifiers are combined in VULPREDICTOR.

A. Overall Framework

Figure 1 presents the overall framework of VULPREDICTOR. The framework contains two phases: model building phase and prediction phase. In the model building phase, VULPREDICTOR builds a composite model from training source code files that have known labels (vulnerable or not). In the prediction phase, this model is used to predict whether a new source code file is vulnerable or not.

Given a training set of files, we first preprocess the source code files by tokenizing them, removing stop words, and stemming the tokens (Step 1). In the tokenization process, we extract identifier names and words in comments. We break identifer names into tokens following the Camel casing convention. In the stop word removal process, we remove frequently appearing words that provide little help to differentiate one file from another. We use a list of stop words that is available from Snowball [1]. In the stemming process, we reduce each word to its root form, for example, words "reads" and "reading" are both reduced to "read". We use a popular stemming algorithm proposed by Porter [8]. Next, we extract features from the files (Step 2); we extract two kinds of features:

[1] http://snowball.tartarus.org/algorithms/english/stop.txt

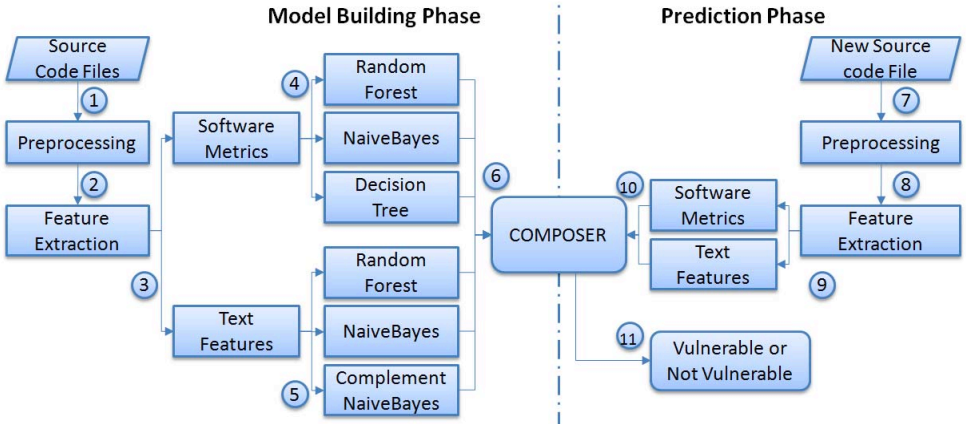


Fig. 1. Overall Framework of VULPREDICTOR

software metrics and text features (Step 3). We extract the software metrics proposed by Walden et al. [1], and they are briefly described in Table III. The text features are the tokens extracted in Step 1 and their associated frequencies. We build six underlying classifiers from the extracted features; from the software metrics, we build 3 classifiers using Random Forest, Naive Bayes, and Decision Tree classification algorithms (Step 4); and from the text features, we also build 3 classifiers using Random Forest, Naive Bayes, and Complement Naive Bayes classification algorithms (Step 5). Next, we construct a meta classifier COMPOSER that combines the six underlying classifiers using the Random Forest classification algorithm (Step 6). COMPOSER analyzes the confidence scores that are output by the six classifiers and produces a final confidence score.

After COMPOSER has been built, in the prediction phase, it is used to predict whether a new source code file is vulnerable or not. For each new file, our framework first preprocesses and extracts features from it (Step 7, 8), and represents it as software metrics and text features similar to the training phase (Step 9). Next, these features are input into the classifier application step (Step 10). The features are first input to the respective underlying classifiers and then the confidence scores produced by these classifiers are input into COMPOSER. Finally, COMPOSER outputs the prediction result: vulnerable or not vulnerable (Step 11).

B. Basic Classifiers

In this paper, we use 4 state-of-the-art classification algorithms, namely Random Forest, Naive Bayes, Decision Tree, and Complement Naive Bayes, to construct the 6 underlying classifiers of *VulPredictor*. We use these 4 since they were successfully used in many previous works [9], [10]. We describe these 4 classification algorithms in a nutshell in the following paragraphs.

1) ***Decision Tree****:* Decision Tree is a classification algorithm that constructs a model (aka. a classifier) in the form of a tree whose leaf nodes are class labels and internal nodes are tests on feature values [11]. Decision tree algorithm builds the classifier in two phases: Tree Building and Tree Pruning. In the tree building phase, a decision tree model is built by recursively splitting a training data set based on a locally optimal criterion (to create an internal node) until all or most of the data instances (in our case: files) belonging to each of the partitions have the same class label. To improve the generalization power of a decision tree model, tree pruning is used to prune away leaves and branches supported by only a single or very few data instances. In our experiment, we use C4.5 decision tree algorithm [12].

2) ***Random Forest****:* Random Forest algorithm builds not only one decision tree but many of them [13]. It takes advantage of both bagging and random feature selection for tree building; each of the decision trees is built using a bootstrap sample of the training data, and Random Forest selects a subset of features randomly to split at each node when growing a tree instead of using all the features. Multiple decision trees are learned and the output of the decision trees are combined to a single prediction using majority voting.

3) ***Naive Bayes****:* Naive Bayes classification algorithm is based on Bayes theorem with independence assumptions among features [14]. Given a data instance (in our case: a file) and its feature values, it first computes conditional probabilities of various feature values to be observed given a class label (in our case: vulnerable or not-vulnerable). Next, it uses these conditional probabilities to assign to the data instance the class label that maximizes the product of the conditional probabilities and the probability of the class label. These probabilities are estimated using a set of labeled training data (in our case: a set of files and their vulnerability labels). A Naive Bayes model is easy to build, with no complicated iterative parameter estimation which makes it particularly suited when the dimensionality of the inputs (i.e., number of features) is high. Naive Bayes is highly scalable, requiring a number of parameters to be estimated linear in the number of features. It is particularly well-suited for textual data where the number of features (i.e., number of unique pre-processed words) are high.

4) ***Complement Naive Bayes****:* Complement Naive Bayes [15] is a Naive Bayes variant that tends to work better when the classes in the training set are imbalanced (i.e., data instances of one class is much more than those of the other classes). It estimates feature probabilities for each class x based on the complement of x, i.e. on all other classes'

TABLE III. SOFTWARE METRICS

Metrics	Descriptions
Lines of code	Number of lines in a PHP source file, excluding lines without PHP tokens, such as comments and blank lines. When tokens span multiple lines, only one line is counted.
Lines of code (non-HTML)	Same as lines of code, except HTML contents embedded in PHP files are not considered.
Number of functions	Number of function definitions in a PHP file.
Cyclomatic complexity	Size of a control flow graph computed by adding one to the number of loops and decision statements in a PHP file.
Maximum nesting complexity	Maximum depth of nested loops and control structures in a PHP file.
Halstead's volume	$(N_1 + N_2) \times \log(n_1 + n_2)$ where N_1 and N_2 are the total number of operators (i.e., method names and PHP language operators) and operands (i.e., method parameters and variable names) in a PHP file respectively, and n_1 and n_2 are the number of unique operators and operands respectively in a file.
Total external calls	Number of invocations of functions, which are defined in a different file, in a target file.
Fan-in	Number of files which contain statements that invoke a function defined in the file being measured.
Fan-out	Number of files containing functions invoked by statements in the file being measured.
Internal functions called	Number of functions defined in the measured file which are called at least once by a statement in the same file.
External functions called	Number of functions defined in other files which are called at least once by a statement in the file being measured.
External calls to functions	The number of files calling a particular function defined in the file being measured, summed across all functions in the file being measured.

samples, instead of on the training samples of class x itself.

C. COMPOSER

VULPREDICTOR is a two tier composite algorithm which predicts the label of an instance (i.e., predicts if a file is vulnerable or not). In the first tier, VULPREDICTOR builds 6 underlying classifiers on a training set of labeled files represented by their software metrics and text features by running the 4 classical classification algorithms described in Section III-B. In the second tier, VULPREDICTOR builds a meta classifier COMPOSER which combines the 6 underlying classifiers.

To construct COMPOSER, the confidence scores output by the 6 underlying classifiers for each instance in the training set are collected to create a new dataset, and this dataset is used to train COMPOSER by running the Random Forest classification algorithm. To predict the label of a new file (i.e., vulnerable or not), VULPREDICTOR first uses the 6 underlying classifiers to compute six confidence scores. Next, these confidence scores are used as input to the meta classifier COMPOSER to output the final confidence score of the new file to be vulnerable.

Due to class imbalance phenomenon, if we use the default threshold (i.e., 0.5) to decide whether a new file is vulnerable or not based on its final confidence score (i.e., vulnerable if the final confidence score is at least 0.5 and not vulnerable otherwise), the prediction result can be poor. To improve prediction effectiveness, we propose to predict the label of file fv, denoted as $Label(fv)$, by learning a suitable vulnerable threshold *threshold* for each project. After a suitable threshold is learned, $Label(fv)$ is decided by the following equation:

$$Label(fv) = \begin{cases} Vulnerable, & if\, VulP(fv) \geqslant threshold \\ Not\ Vulnerable, & Otherwise \end{cases} \quad (1)$$

In the above equation, $VulP(fv)$ is the final confidence score of file fv, which is produced by VULPREDICTOR. File fv is classified as Vulnerable if $VulP(fv)$ is larger than or equal to *threshold*; otherwise it is classified as Not Vulnerable.

Algorithm 1 $EstimateThreshold$: Estimation of Threshold.

```
1: EstimateThreshold(FV, Sample)
2: Input:
3: FV: Training Files and Their Labels
4: sp: Dividing Coefficient (default value: 80%)
5: Output: threshold
6: Method:
7: Divide FV into two subsets FV_1 and FV_2 according to sp;
8: Build a Random Forest model from FV_1;
9: for all Files fv ∈ FV_2 do
10:    Compute confidence score VulP(fv);
11: end for
12: for all threshold from 0 to 1, every time increase threshold
    by 0.01 do
13:    Predict the labels of files in FV_2 according to Equation (1);
14:    Compute the F1-score on FV_2;
15: end for
16: Return threshold which maximizes the F1-score for files in FV_2
```

The value of the threshold varies between 0 and 1. To learn a good threshold value, we propose a greedy algorithm shown in Algorithm 1. We input a training set FV, and a dividing coefficient sp. We first divide a collection of training files FV into two subsets FV_1 and FV_2 based on sp (Line 7). By default, we set the dividing coefficient as 80%, i.e., 80% of the training files are in FV_1. Then, we build a model from the the

subset FV_1 (Line 8). Next, for each file in FV_2, we compute its confidence score $VulP(fv)$ (Line 9-11). Finally, to tune the best threshold value, we gradually increase *threshold* from 0 to 1 (every time we increase *threshold* by 0.01), and for each file fv in FV_2, we predict its label according to Equation (1). We output the threshold that maximizes the F1-score for files in FV_2 (Lines 12-16).

IV. Experiments and Results

In this section, we evaluate the effectiveness of VulPredictor and compare it with other approaches. The experimental environment is an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop running Windows 7 (32-bit).

A. Experiment Setup

We evaluate VulPredictor on datasets containing a total of 223 vulnerabilities from 3 open-source web applications written in PHP, i.e., Drupal, PHPMyAdmin and Moodle, which are provided by Walden et al. [1]. Each of the datasets contains a set of classes labeled as vulnerable or not and their corresponding software metrics and text features. Among the 233 vulnerabilities, 19 of them are code injection vulnerabilities which allow attackers to modify arbitrary server-side variables and HTTP headers, 12 of them are cross-site request forgery vulnerabilities which allow for external malicious HTML to induce the user to perform unwanted actions, 86 of them are cross-site scripting vulnerabilities which allow for malicious Javascript to be executed in a user's browser, 14 of them are path disclosure vulnerabilities which allow for the installation path of the application to be maliciously obtained, 73 of them are authorization issues including: privilege bypass vulnerabilities, information disclosure vulnerabilities and vulnerabilities related to missing or inadequately implemented encryption, and the remaining 19 are miscellaneous vulnerabilities related to phishing, man-in-the-middle attacks, and other unspecified attack vectors. Table IV provides summary information about the three applications, including number of files, number of vulnerable files, percentage of vulnerable files (P-rate), and the total number of text features. Moodle is the largest application and has the lowest positive rate, which makes it difficult to predict the rare vulnerable files.

To validate VulPredictor and to reduce training set selection bias, we perform 10-fold cross-validation. Cross validation is a standard evaluation setting, which is widely used to evaluate past software engineering studies [16], [17], [18]. The files of an application are randomly divided into 10 folds of equal size. Each fold has the same percentage of vulnerable files as the entire version (stratification). Of these 10 folds, 9 folds are use to train a classifier, while the remaining one fold is used to test the effectiveness of the classifier. The process is repeated ten times with different folds used for testing. The average effectiveness scores across the ten iterations are reported.

B. Evaluation Metrics

We use two metrics for our evaluation: F1-score and cost effectiveness. These two metrics measure the effectiveness of a vulnerable file prediction approach in two separate situations. F1-score measures effectiveness assuming a developer can check all files predicted as vulnerable, while cost-effectiveness measures effectiveness given a cost budget (i.e., number of files to check).

1) F1-score: F1-score, which is the harmonic mean of precision and recall, is a standard and widely used measure to evaluate classification algorithms [2]. There are four possible outcomes for a file in a target application: A file can be classified as buggy when it truly is buggy (true positive, TP); it can be classified as buggy when it is actually clean (false positive, FP); it can be classified as clean when it is actually buggy (false negative, FN); or it can be classified as clean and it truly is clean (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as:

Precision: the proportion of files that are correctly labeled as buggy among those labeled as buggy, i.e., $Precision = \frac{TP}{TP+FP}$.

Recall: the proportion of buggy files that are correctly labeled, i.e., $Recall = \frac{TP}{TP+FN}$.

F1-score: a summary measure that combines both precision and recall - it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision).

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (2)$$

There is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). The trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall alone [2]. For this reason, we compare the prediction results using F1-score, which is a harmonic mean of precision and recall.

2) Cost Effectiveness: Cost effectiveness, which evaluates prediction accuracy given a cost limit, is another widely used evaluation metric to evaluate past software engineering studies [3], [4], [5], [6]. In our setting, the cost is the number of files to check, and the benefit is the number of vulnerable files found. By default, we set the number of files to check as 20% of the total number of files. In this paper, we use EffectivenessRatio@20% (ER@20%) as the default cost effectiveness metric. The ER@20% score of a technique is the ratio of the number of vulnerable files detected by the technique to the number detected by a *perfect* technique that ranks all vulnerable files first, considering the top 20% vulnerable files.

To compute ER@20%, we sort files in the evaluation data that are predicted as vulnerable based on the confidence level that a prediction technique outputs for each of them. Aside from outputting labels (in our case: vulnerable or not), VulPredictor, and many other classifiers can also output confidence levels. A file with a higher confidence level is deemed to be more likely to be a vulnerable file by a prediction technique. We then count the number of vulnerable files that appear in the top 20% of the sorted files. We also count the number of vulnerable files that can be identified by a hypothetical perfect technique that ranks all vulnerable files first, when only the top 20% of the bug reports are checked. Based on these two numbers, ER@20% of VulPredictor is computed as:

TABLE IV. DESCRIPTIVE STATISTICS FOR THE APPLICATIONS.

Application	Vulnerable files	Total files	P-rate(%)	Text features
Drupal	62	202	30.68	3886
PHPMyAdmin	27	322	8.39	5232
Moddle	24	2942	0.82	18306

$$ER@20\% = \frac{VulPredictor(20\%)}{Perfect(20\%)} \quad (3)$$

In the above equation, $VulPredictor(20\%)$ refers to the number of vulnerable files in the first 20% of the ranked list produced by VULPREDICTOR, and $Perfect(20\%)$ refers to refers to the number of vulnerable files in the first 20% of the ranked list produced by the perfect technique.

C. Research Questions and Findings

We are interested to answer the following research questions:

RQ1 How effective is VULPREDICTOR? How much improvement can it achieve over the state-of-the-art approaches by Walden et al.?

Motivation. We need to investigate the effectiveness of VULPREDICTOR,and compare it with Walden et al.'s approaches [1]. Walden et al. build prediction models from only either software metrics or text features (but not both), using random forest algorithm. Their result shows that the prediction models built from text features perform better than those built from code metrics. Answer to this research question would shed light to whether and to what extent VULPREDICTOR improves over state-of-the-art approaches proposed by Walden et al.

Approach. To answer this research question, we perform 10-fold cross-validation 100 times and report the average performance of VULPREDICTOR and Walden et al.'s approaches. We compute precision, recall, F1-score, and ER@20% of VULPREDICTOR and Walden et al.'s approaches when they are applied to the 3 datasets listed in Section IV-A. We then compare the results achieved by VULPREDICTOR and Walden et al.'s approaches.

Results. Tables V presents precision, recall, F1, and ER@20% scores of VULPREDICTOR and Walden et al.'s approaches that build model solely either from text features ($Walden^{Textmining}$) or software metrics ($Walden^{Metrics}$) respectively. From the table, we can see that for the 3 datasets, the F1-scores of VULPREDICTOR are 0.683, 0.34, and 0.071, which outperform the F1-scores of $Walden^{Textmining}$ by 8.24%, 70%, and 61.36%, and $Walden^{Metrics}$ by 21.53%, 49.78%, and 102.86%, respectively. The ER@20% scores of VULPREDICTOR for the 3 datasets are 75%, 51.9%, and 29.2%, which outperform $Walden^{Textmining}$'s scores by 11.11%, 16.89%, and 16.8%, and $Walden^{Metrics}$'s scores by 23.33%, 28.71%, and 40.18%, respectively. As the Moodle dataset is highly imbalanced, the F1 and ER@20% scores for this dataset are relatively low, but VULPREDICTOR still achieves better results that Walden et al.'s approaches. The results show that the improvements that VULPREDICTOR achieves over Walden et al.'s approaches are substantial (8.24% - 102.86%).

Notice that the results of Walden et al.'s approaches in our paper are not the same as the results in their paper [1]. That's because we use 100 times 10-fold cross-validation; cross-validation randomizes the dataset to reduce training set selection bias, and the results for different cross-validation runs are different with different seeds.

> VULPREDICTOR *can achieve F1 and ER@20% scores of up to 68.3% and 75% respectively, and it outperforms the state-of-the-art approaches by Walden et al. by 8.24% to 102.86%.*

RQ2 Does VULPREDICTOR perform better than its parts?

Motivation. VULPREDICTOR builds a composite model on top of 6 underlying classifiers. To demonstrate the benefit of the compositional approach of VULPREDICTOR, we need to compare its effectiveness against that of each of the underlying classifiers.

Approach. To answer this research question, we compute precision, recall, F1 and ER@20% scores of VULPREDICTOR and its 6 underlying classifiers when they are applied to the 3 datasets listed in Section IV-A. The 6 underlying classifiers are built using 4 classification algorithms, namely Random Forest (RF), Naive Bayes (NB), Decision Tree (DT), and Complement Naive Bayes (CNB). We then compare the results achieved by VULPREDICTOR and the 6 stand-alone classifiers.

Results. Table VI presents F1 and ER@20% scores of VULPREDICTOR and the 6 underlying classifiers when applied to the 3 datasets. From the table, we can see that VULPREDICTOR achieves the best F1 and ER@20% scores among all the classifiers, for all 3 datasets. For example, for Drupal, the F1-score of VULPREDICTOR is 0.683, which outperforms the F1-score of the 6 underlying classifiers by 22.18%, 36.6%, 12.89%, 20.04%, 44.09%, and 46.88%, respectively; the ER@20% of VULPREDICTOR is 75%, which outperforms the ER@20% of the 6 underlying classifiers by 15.38%, 11.11%, 36.36%, 3.45%, 42.86%, and 76.47%, respectively.

For the 6 single classifiers, we notice that the effectiveness of the 6 classifiers are varying when applied to different datasets. This means that different classifiers are able to identify different vulnerable files better. VULPREDICTOR combine these complementary classifiers to improve prediction effectievness.

> *The composite model generated by* VULPREDICTOR *outperforms the six classifiers that it builds upon.*

RQ3 In terms of cost effectiveness, how effective are VULPREDICTOR and Walden et al.'s approaches when we inspect different percentages of files?

TABLE V. PRECISION, RECALL, F1-SCORE, AND ER@20% SCORES OF VULPREDICTOR COMPARED WITH WALDEN ET AL.'S APPROACHES.

Dataset	Algorithm	Precision	Recall	F1-score	ER@20%
Drupal	VULPREDICTOR	**0.672**	**0.694**	**0.683**	**75%**
	$Walden^{Textmining}$	0.603	0.661	0.631	67.50%
	$Walden^{Metrics}$	0.473	0.694	0.562	57.50%
PHP	VULPREDICTOR	**0.346**	**0.330**	**0.340**	**51.90%**
	$Walden^{Textmining}$	0.308	0.148	0.200	44.40%
	$Walden^{Metrics}$	0.164	0.370	0.227	37.00%
Moodle	VULPREDICTOR	**0.250**	**0.042**	**0.071**	**29.20%**
	$Walden^{Textmining}$	0.023	0.800	0.044	25%
	$Walden^{Metrics}$	0.018	0.704	0.035	20.83%

TABLE VI. F1 AND ER@20% SCORES OF VULPREDICTOR COMPARED WITH THOSE OF ITS 6 UNDERLYING CLASSIFIERS

	Drupal		PHPMyAdmin		Moodle	
Algorithm	F1-score	ER@20%	F1-score	ER@20%	F1-score	ER@20%
VULPREDICTOR	**0.683**	**75%**	**0.340**	**51.90%**	**0.071**	**29.20%**
$RF^{metrics}$	0.559	65%	0.242	44.40%	0	20.83%
NB^{metric}	0.500	67.50%	0.277	44.40%	0.037	25%
DT^{metric}	0.605	55%	0.256	14.80%	0	16.67%
RF^{text}	0.569	72.50%	0.069	51.90%	0	12.50%
NB^{text}	0.474	52.50%	0.198	29.60%	0.016	8.33%
CNB^{text}	0.465	42.50%	0.201	18.50%	0.014	16.67%

Motivation. By default, we set the percentage of files to inspect as 20%, which is the setting used for RQ1. In this research question, we investigate the effectiveness of VULPREDICTOR and Walden et al.'s approaches when different percentages of files are inspected. Answer to this research question can shed light as to whether VULPREDICTOR outperforms Walden et al.'s approaches considering other cost settings.

Approach. To answer this research question, we calculate the cost effectiveness scores of VULPREDICTOR and Walden et al.'s approaches ($Walden^{Textmining}$ and $Walden^{Metrics}$) when they are applied to the 3 datasets listed in Section IV-A. We investigate different percentages of files to inspect, from 5% to 60%, at 5% interval. Next, we plot the cost effectiveness graphs that show the percentages of vulnerable files that can be detected by inspecting different percentages of files.

Results. Figure 2 presents the cost effectiveness graphs of VULPREDICTOR, $Walden^{Textmining}$, and $Walden^{Metrics}$ for Drupal, PHPMyAdmin and Moodle datasets. From the graphs, we notice that for percentage range from 5% to 50%, VULPREDICTOR performs the best. When the percentage range is between 50% and 60%, VULPREDICTOR and $Walden^{Textmining}$ achieve similar cost-effectiveness scores.

VULPREDICTOR *performs better than Walden et al.'s approaches for a wide range of percentages of files to be inspected.*

RQ4 How much time does it take for the VULPREDICTOR to run to completion?

Motivation. The efficiency of a vulnerability prediction algorithm will affect its practical usage. Thus, in this research question, we investigate the time efficiency of VULPREDICTOR. We also compare the time efficiency of VULPREDICTOR with those of Walden et al.'s approaches. Answer to this research question can shed light as to whether the time efficiency of VULPREDICTOR is reasonable.

Approach. In order to answer this question, we report the model training and testing time of VULPREDICTOR and Walden et al.'s approaches. Model training time refers to the time it takes to convert a training data into a prediction model, which includes the time needed to estimate the vulnerable threshold.Testing time refers to the time it takes for a trained model to predict the labels of files in a test set. We compare the model training time and testing time of VULPREDICTOR with those of Walden et al.'s approaches.

Results. Table VII presents the average model training and testing time for VULPREDICTOR and Walden et al.'s approaches for the 3 datasets. The longest average model training time of VULPREDICTOR is only less than 1.5 minutes (for Moodle). The average model training time of VULPREDICTOR is longer than those of Walden et al.'s approaches; however, this is reasonable since VULPREDICTOR needs to learn 6 underlying classifiers and 1 meta classifier. Notice that a trained model does not need to be updated all the time and it can be used to label many files. The average testing time of VULPREDICTOR for the 3 datasets are a fraction of a second and are close to those of Walden et al.'s approaches. We also find that the average training time of $Walden^{Textmining}$ is longer than that of $Walden^{Metrics}$, as the term vectors analyzed by $Walden^{Textmining}$ are large in size.

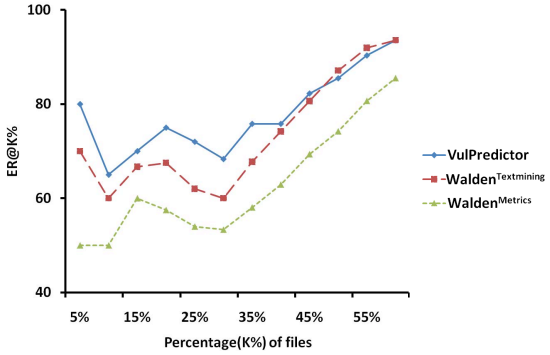
The average training time and testing time of VULPREDICTOR *are at most 73.225 seconds and 0.064 seconds, which are reasonable.*

D. Threats to Validity

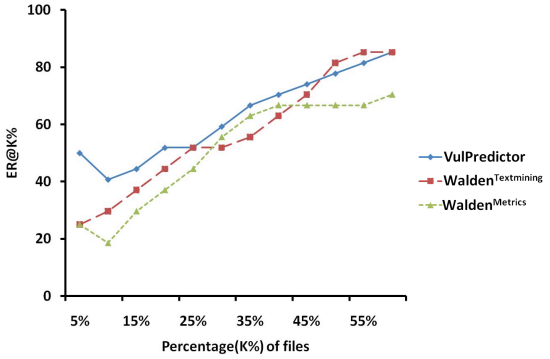
Threats to internal validity relate to errors in our experiments and implementation. We have double checked our experiments and implementation. Still, there could be errors that we did not notice. To reduce training set selection bias, we run

TABLE VII. AVERAGE MODEL TRAINING AND TESTING TIME (IN SECONDS) FOR VULPREDICTOR COMPARED WITH WALDEN ET AL.'S APPROACHES

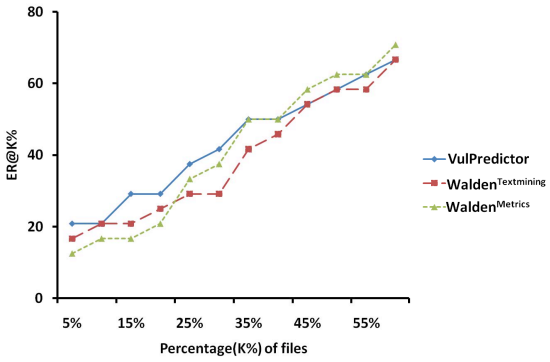
	VULPREDICTOR		$Walden^{Textmining}$		$Walden^{Metrics}$	
Datasets	Train_time	Test_time	Train_time	Test_time	Train_time	Test_time
Drupal	1.246	0.003	0.240	0.006	0.060	0.003
PHPMyAdmin	1.184	0.006	0.267	0.009	0.072	0.003
Moodle	73.225	0.064	3.130	0.032	0.166	0.025



(a) Drupal



(b) PHPMyAdmin



(c) Moodle

Fig. 2. Cost effectiveness graphs for the 3 datasets.

10-fold cross-validation, and record the average performance. Threats to external validity relate to the generalizability of our results. We have evaluated our approach using 3,466 files from 3 web applications. In the future, we plan to reduce this threat further by using even more files from more applications. Threats to construct validity refer to the suitability of our evaluation metrics. We use F1 and cost effectiveness scores which are also used by many past software engineering studies to evaluate the effectiveness of various prediction techniques [19], [4], [20], [5], [6]. Thus, we believe there is little threat to construct validity.

V. RELATED WORK

In this section, we briefly review some previous studies on vulnerability prediction and analysis in Section V-A and classification in software engineering in Section V-B.

A. Vulnerability Prediction and Analysis

There have been a number of studies on vulnerability prediction. Models and tools have been proposed to predict vulnerable files in software projects. Neuhaus et al. propose a tool Vulture that predicts vulnerable files by looking at their features [21]. It is based on the observation that files that have similar imports or function calls have similar vulnerability labels. Nguyen et al. introduce a way to predict vulnerable files by analyzing code metrics extracted using dependency graphs [22]. They have evaluated their prediction model using the JavaScript Engine of Firefox. Chowdhury et al. present a framework to predict vulnerabilities automatically based on complexity, coupling, and cohesion metrics [23]. They consider four alternative data mining techniques: C4.5 Decision Tree, Random Forest, Logistic Regression, and Naive Bayes, and compare their prediction effectiveness. Shar et al. propose a set of static code attributes to predict the existence of two common web application vulnerabilities [24]. These attributes represent the characteristics of routines that web applications generally use for input validation and sanitization to prevent web security breach.

Meneely et al. manually collect 124 Vulnerability-Contributing Commits (VCCs), and analyze these VCCs quantitatively and qualitatively [25]. Zimmermann et al. present an empirical study on vulnerabilities in Windows Vista [26]. In that study, they empirically evaluate the relationships between complexity, churn, coverage, organizational structure, and several other measures with vulnerabilities. Shin et al. investigate the suitability of complexity, code churn, and developer activity metrics as indicators of software vulnerabilities [27]. They perform two empirical studies on the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel which are both large, widely used open-source projects. Shin et al. perform statistical analysis on nine complexity metrics, and the results show that complexity metrics are able to predict vulnerabilities [28]. Alhazmi et al. find that the values of vulnerability densities fall within a range of values and may be influenced by a few factors, such as sharing of codes between successive versions of a software system [29].

The latest work in this area is done by Walden et al.. Walden et al. propose two approaches that collects either software metrics or text features and uses a Random Forest classification algorithm to create a discriminative model (classifier) from either software metrics or text features. They evaluate the effectiveness of their approaches on vulnerability datasets from three open source web applications written in PHP: Drupal, Moodle, and PHPMyAdmin. The experiment results show that the classifier built from text features has a higher recall than the classifier built from software metrics. In this paper, we extend Walden et al.'s work by proposing a novel approach VULPREDICTOR which analyzes software metrics and text mining *together* to build a *composite* prediction model. The experiment results show that VULPREDICTOR outperforms Walden et al.'s approaches in terms of both F1 and EffectivenessRatio@20% scores.

B. Classification In Software Engineering

There have been a number of software engineering studies that employ classification algorithms. Antoniol et al. propose a classification approach that can predict if an issue report is a bug report or a non-bug report [30]. Their approach is extended by Zhou et al. [31]; different from Antoniol et al.'s work that only use text features from issue reports, Zhou et al. use both text features and non-text features (i.e., values of reporter, assignee, priority, severity, and component fields) extracted from bug reports to classify issue reports into bug reports and non-bug reports more accurately. Kochhar et al. also extend Antoniol et al.'s approach by proposing an automated technique that reclassifies an issue report into fine-grained categories (e.g., bug, request for improvement (RFE), documentation, refactoring, etc.) [32]. The technique extracts various feature values from a bug report and predicts if the bug report needs to be reclassified and its reclassified category.

Xia et al. propose an automatic, high accuracy predictor to predict reopened bugs, which uses 3 classifiers for different kinds of features [33]. Tian et al. propose an automated approach based on machine learning that recommends a priority level to a bug report based on multiple influencing factors [34]. Zhang et al. investigate 7 composite algorithms, which integrate multiple machine learning classifiers, to improve cross-project defect prediction [35]. Xia et al. propose a novel cross-project build co-change prediction approach, which constructs an ensemble of classifiers by iteratively building classifiers and assigning them weights according to their prediction error rates [36]. Xia et al. propose *ELBlocker* to identify blocking bugs [7]. *ELBlocker* first randomly divides training data into multiple disjoint sets, and for each disjoint set, it builds a classifier; next, it combines these multiple classifiers, and determines an appropriate imbalance decision boundary to differentiate blocking bugs from non-blocking bugs.

Menzies et al. presents an automated method named SEVERIS (SEVERity ISsue assessment), which assists test engineers in assigning severity levels to defect reports [17]. SEVERIS is based on standard text mining and machine learning techniques applied to defect reports, they evaluate their proposed approach on bug reports from NASA and predict fine-grained bug severity levels. Lamkanfi et al. extend Menzies et al.'s work by proposing another text mining approach which analyzes textual description of bug reports that can predict coarse-grained bug severity levels with more accuracy [37]. Xia et al. propose a novel fuzzy set based feature selection algorithm which selects features that have high ability to categorize bugs based on their fault triggering conditions [33]. Thung et al. propose a text mining solution which can categorize defects into IBM's Orthogonal Defect Classification (ODC) defect types by learning a discriminative model based on texts from bug reports and code features extracted from bug fixing changes [38]. Kochhar et al. propose an automated text classification technique that extracts various feature values from a bug report and predicts if the bug report needs to be reclassified and its reclassified category [32].

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose VULPREDICTOR to predict vulnerable files in software. VULPREDICTOR analyzes software metrics and text features together and is built on an ensemble of several classifiers. VULPREDICTOR is a two tier composite algorithm; in the first tier it builds 6 underlying classifiers on a training set of vulnerable and non-vulnerable files represented by their software metrics and text features, and in the second tier a meta classifier is built to process the outputs of the first-tier classifiers. We evaluate VULPREDICTOR on datasets of 3 web applications and measure its performance in terms of F1-score and EffectivenessRatio@20% (ER@20%). The experiment results show that VULPREDICTOR can achieve F1 and ER@20% scores of up to 0.683 and 75%, respectively. On average across the 3 projects, VULPREDICTOR improves the F1 and ER@20% scores of the best performing state-of-the-art approach proposed by Walden et al. by 46.53% and 14.93%, respectively.

In the future, we plan to experiment with more vulnerabilities from additional projects in addition to the three projects considered in this work. We also plan to improve the effectiveness of VULPREDICTOR further. One direction is to investigate additional features that we can use in addition to the software metrics and text features considered in this work. We plan to investigate the effectiveness of code smells [39] to be used as features, and leverage the recent trend in using deep learning [40] to construct additional composite features that can improve the effectiveness of a classification algorithm. We also want to investigate the viability of a history-aware vulnerability prediction algorithm based on a hypothesis that a file that is vulnerable before is likely to be vulnerable again in the future. In this work, we consider an intra-project setting, where training and test data come from the same project. In the future, we plan to consider a cross-project setting, where a model built on training datasets from other projects can be used to predict vulnerable files of a given project. Furthermore, in this work, we focus on file level granularity (i.e., predicting which files contain vulnerabilities), in the future, we plan to look at other granularities, either more coarse-grained (e.g., packages), or more fine-grained (e.g., methods, or lines). In this work, we do not differentiate vulnerabilities into different types, in the future, we plan to develop a vulnerability prediction that can also infer vulnerability types.

Acknowledgment. This research was partially supported by China Knowledge Centre for Engineering Sciences and Technology (No. CKCEST-2014-1-5), National Key Technology R&D Program of the Ministry of Science and Technology of

China (No. 2015BAH17F01), and the Fundamental Research Funds for the Central Universities. We would also like to thank Walden et al. for sharing their datasets [1].

REFERENCES

[1] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 23–33.

[2] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

[3] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 215–224.

[4] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 61.

[5] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 147–157.

[6] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 279–289.

[7] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[8] M. Porter, "An algorithm for suffix stripping," *Program*, 1980.

[9] Y. C. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 852–864, 2010.

[10] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[11] E. Ceylan, F. O. Kutlubay, and A. B. Bener, "Software defect identification using machine learning techniques," in *Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on*. IEEE, 2006, pp. 240–247.

[12] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[13] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[14] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Citeseer, 1998, pp. 41–48.

[15] J. D. Rennie, L. Shih, J. Teevan, D. R. Karger *et al.*, "Tackling the poor assumptions of naive bayes text classifiers," in *ICML*, vol. 3. Washington DC), 2003, pp. 616–623.

[16] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 134–143.

[17] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[18] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 386–396.

[19] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 432–441.

[20] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, 2008.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 529–540.

[22] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*. ACM, 2010, p. 3.

[23] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[24] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 310–313.

[25] A. Meneely, H. Srinivasan, A. Musa, A. Rodriguez Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 65–74.

[26] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 421–428.

[27] Y. Shin, A. Meneely, L. Williams, J. Osborne *et al.*, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 772–787, 2011.

[28] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 315–317.

[29] O. Alhazmi, Y. Malaiya, and I. Ray, "Security vulnerabilities in software systems: A quantitative perspective," in *Data and Applications Security XIX*. Springer, 2005, pp. 281–294.

[30] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, 2008, p. 23.

[31] Y. Zhou, Y. Tong, R. Gu, and H. C. Gall, "Combining text mining and data mining for bug report classification," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 311–320.

[32] P. S. Kochhar, F. Thung, and D. Lo, "Automatic fine-grained issue report reclassification," in *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*. IEEE, 2014, pp. 126–135.

[33] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2014.

[34] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, pp. 1–30, 2014.

[35] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. IEEE, 2015.

[36] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 311–320.

[37] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[38] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[39] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 75–84.

[40] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.