2015 IEEE 14th International Conference on Machine Learning and Applications

Predicting Vulnerable Software Components through *N*-gram Analysis and Statistical Feature Selection

Yulei Pang
Department of Mathematics
Southern Connecticut State University
New Haven, Connecticut, USA
Email: pangy1@southernct.edu

Xiaozhen Xue
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
Email: xiaozhen.xue@ttu.edu

Akbar Siami Namin
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
Email: akbar.namin@ttu.edu

***Abstract*—Vulnerabilities need to be detected and removed from software. Although previous studies demonstrated the usefulness of employing prediction techniques in deciding about vulnerabilities of software components, the accuracy and improvement of effectiveness of these prediction techniques is still a grand challenging research question. This paper proposes a hybrid technique based on combining *N*-gram analysis and feature selection algorithms for predicting vulnerable software components where features are defined as continuous sequences of token in source code files, i.e., Java class file. Machine learning-based feature selection algorithms are then employed to reduce the feature and search space. We evaluated the proposed technique based on some Java Android applications, and the results demonstrated that the proposed technique could predict vulnerable classes, i.e., software components, with high precision, accuracy and recall.**

***Keywords*—Vulnerability prediction, *N*-gram, Feature selection, Wilcoxon test**

I. Introduction

Software security is becoming increasingly important for both businesses and individual users. It is a commonly held belief that detecting and identifying and fixing all possible vulnerabilities is the key to ensure security of underlying software system from possible cyberattacks. However, testing software from security standpoint, which aims at detecting and patching potential vulnerabilities, is very time consuming and expensive mainly due to the following two reasons: 1) modern software systems are large and complex, and it is quite common that a software system may contain thousands of files and interleaved components; 2) vulnerabilities are more difficult to detect compared to traditional faults, since software testers need to know attacking strategies and reason as an attacker. Therefore, any automated testing tool and technique that help detecting and fixing vulnerabilities as early as possible can improve the software quality and in particular security requirements significantly. One of such techniques used prevalently is the vulnerability prediction where components and files are prioritized letting the tester check the components with higher likelihood of being vulnerable earlier.

A common practice in prioritizing software components is based on building a classifier using popular machine learning classification techniques. Accordingly, the procedure includes collecting training data, selecting features, training a model, and testing the accuracy of the classifier with a given test data. The existing literature focus on addressing two questions: 1) which features contribute more to building a better vulnerability detection model?, and 2) which machine learning algorithms build a better model that offers more precise classification systems? It is commonly accepted that the goodness of a prediction technique heavily depends on how well features are constructed and selected. Therefore, the first question triggers a number of research questions on investigating features including adopting complex, history, metrics and other software metrics from source code analysis point of view [1], [2], [3].

Although previous studies show the usefulness of prediction models to classify vulnerable components, the improvement of the effectiveness of these techniques is still a challenging problem. In a recent study, text mining techniques were adapted to classify vulnerable software components and the results were considerably promising [4]. In utilizing text mining techniques, the tokens extracted from a given source code were viewed as features. There exists some other advanced text mining techniques, such as *N*-gram analysis, which handles not only about single tokens but also sequences of tokens. The *N*-gram analysis fits into the source code analysis scenario properly and in particular when the underlying program is written in an objected oriented language. Object oriented features such as classes, objects, methods, arguments, and variables are usually interrelated, and the consecutive appearance of some of these OO features may have some specific implications. However, the major concern when performing an *N*-gram analysis is dealing with high dimensionality, which is a major problem that affects the performance and effectiveness of training datasets and therefore classification models.

In this paper, we employ a statistical feature selection technique proposed in our previous study [4] to choose the most relevant and important attributes from a dataset, thus reducing the space of features and thus the dimensionality of the problem. We introduce a hybrid technique combining *N*-gram analysis and feature selection for the purpose of predicting vulnerable software components. The key contributions of this paper are as follows:

- Introduce a hybrid technique combining *N*-gram analysis and statistical feature selection for predicting vulnerable software components;
- Evaluate the performance of the proposed hybrid technique based on a number of Java Android programs;

978-1-5090-0287-0/15 $31.00 © 2015 IEEE
DOI 10.1109/ICMLA.2015.99

- Investigate the prevalence of vulnerable software components, and the effectiveness of proposed technique in cross-project scenarios.

The rest of this paper is organized as follows. Section II presents related work in the view of vulnerability prediction and feature selection. Section III provides a detailed explanation of the proposed technique. In Section IV, we evaluate our technique based on experimental studies conducted in this work. Section V poses and addresses two additional research questions. Section VI discusses the threats to validities. Section VII concludes the study and provides future directions.

II. Related Work

There are a good number of similar research works that address predicting *faulty* software components. Nagappan et al. [2] utilized complexity metrics on some Microsoft software systems with the goal of identifying faulty components. Nagappan and Ball [3] proposed a multiple linear regression model using code churn to predict the defect density in software systems. Ostrand et al. [5] used the historical and recorded information such as number of faults in prior releases for fault prediction.

The problem of predicting faulty components is very similar to the vulnerability prediction problem. As a result, most of the study in this area is the replication of techniques proposed for predicting faulty components. Gegick et al. [6] performed a study to investigate whether failing information can be utilized as an indicator for vulnerabilities. Shin and Williams investigated the feasibility of vulnerability prediction by reusing existing faults prediction model when complexity, code churn, and history metrics were adopted. Hovsepyan et al. [7] applied text analysis techniques to this problem and their results achieved high precision and recall. Scandariato et al. [4] defined a method for constructing features by extracting tokens from a given source code to predict vulnerable components in Android systems. Morrison et al. [8] discuss the Challenges with applying vulnerability prediction models.

Feature selections are widely used in various problem domains. One of the common uses of feature ranking techniques is their use in microarray analysis with the purpose of discovering a set of drug leads [9].The statistical feature selection algorithms have been widely used in bioinformatics, including tumor diagnosis by analyzing gene expression profiles [10], cell type specific signatures study of microRNAs on target mRNA expression [11], and investigating of asthma developed by allergic rhinitis [12]. In software engineering, Gao et al. [13] used feature selections for prediction faulty components. A more relevant example is that Xue et al. [14] used sequential analysis and statistical feature selection for localizing faulty event in GUI applications.

III. Methodology

This section explains the techniques that are adopted in this paper including constructing features using *N*-gram analysis, feature selection methods based on Wilcoxon test for reducing the space and dimensionality of features, and finally the learners we chose for building the prediction model.

A. Features Construction

Machine learning algorithms often involve constructions of vectors, where the variables are chosen systematically and data are transformed into high dimensional vectors. In this scenario, the required data mainly are extracted from a given source code of a software application. Previous studies mainly use software metrics including complexity, code churn, and fault history metrics for the purpose of prediction [1]. Some recent studies used tokens of source code as features analyzed by text mining techniques [4].

In this paper, we also build features based on mining source code. However, on the application of data mining techniques, there exist some other advanced techniques, such as *N*-gram analysis. *N*-gram is a contiguous sequence of n items in a given sequence of text. This technique is widely used in bioinformatics science. We also used it in software testing domain where faulty features were identified in GUI interfaces using N-gram analysis [14].

In object oriented languages, e.g., Java, source-code level tokens consist of identifiers, keywords, separators, operators, literals and comments. We excluded separators and operators to ease the computation. The tokens to use in computation involve classes, objects, methods, arguments, and variables. There exist some associations among these types of tokens that highlight the appropriateness of considering consecutive sequences of tokens than a single token in computation and analysis. For example, when we declare a class as public, a 2-gram sequence "public class" can be viewed as a feature. Similarly, if we want to call a print function, a 3-gram sequence "System out print" or "System out print" can be regarded as features.

Program 1 A simple Java Program.

```
1 public class UseArgument {
2  public static void main(String args[]){
3    System.out.print("Hi, ");
4    System.out.print(args[0]);
5    System.out.println(". How are you");
6  }
7 }
```

For instance, consider a simple program used in a Java programming course in Princeton University III-A[1]. The program is tokenized and transformed into a list of tokens: {*public, class, UseArgument, static, void, main, String, args, System, out, print, Hi, 0, println, How, are, you*}. For each class file we build a vector whose elements represent how many times each token appears in the list, i.e., the frequency. The vector of this file can be represented as $\{2, 1, 1, 1, 1, 1, 1, 2, 3, 3, 2, 1, 1, 1, 1, 1, 1\}$. The list basically holds *1*-gram features. Similarly, the *2*-gram features in the programs are {*(public, class), (class, UseArgument), (UseArgument, public), (public, static), (static, void), (void, main), (main, String), (String, args), (args, System), (System, out), (out, print), (print, Hi), (Hi, System), (print, args), (args, 0), (0, System), (out, println), (println, How), (How, are), (are, you)*}, and consequently the responding vector for this class file is $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$. It is important to note that the dimension of the *2*-gram features can be much higher than *1*-gram features. Similarly, the *n*-gram ($N > 2$) features can be highly dimensional depending on the size of the programs.

[1] http://introcs.cs.princeton.edu/java/11hello/UseArgument.java.html

B. Statistical Feature Selection

When features are generated based on *N*-gram analysis, one major challenge we often face is the feature space exploration or as it is known the curse of dimensionality. In such circumstances, the dimensionality is very high, affecting the quality of training datasets and therefore classification models. In order to deal with this issue efficiently, feature selection technique techniques are used in this paper. Feature selection algorithms aim at identifying a subset of features sharing a certain property. In practice, many machine learning algorithms rank features according to their importance or relevance, e.g. feature ranking methods. Feature selection is especially important when there are many features and comparatively few samples, i.e., data points, which is exactly the case in the target problem of this paper, i.e., the prediction problem of identifying vulnerable software components. In this paper, we use feature ranking approach for excluding a large number of irrelevant or less important features where a small number of features are kept and the rest is discarded. A common practice is that each feature is scored by a feature quality measure. A test measurement based on hypotheses testing to rank the features can be useful. Our experience showed that the ranking method performed very well in software testing domain and when applied to the fault localization problem [14].

Hypotheses test is an essential method in statistical inference. The question of interest is simplified into two competing hypotheses: the null hypothesis and the alternative hypothesis. More specifically, the null hypothesis H_0 assumes that there is no differences between two groups; whereas, the alternative hypothesis H_1 implies that there is a statistically significant differences between two sample groups. The probability value (p-value) of a statistical hypothesis test is the probability of falsely rejecting a null hypothesis if it is in fact true (Type I error). Accordingly, small p-values suggest that the null hypothesis is unlikely to be true. The magnitude of p-values may be an indicator for measuring the strength of rejecting the underlying null hypothesis. We formulate the null and alternative hypotheses as followings, with the purpose of ranking features by computed p-values.

- $H_0^{f_i} : \mu_{non}^{f_i} = \mu_{vul}^{f_i}$ for the *i*-th feature f_i.
- $H_1^{f_i} : \mu_{non}^{f_i} \neq \mu_{vul}^{f_i}$ for the *i*-th feature f_i.

The Wilcoxon rank-sum test, also called Mann–Whitney–Wilcoxon, is a non-parametric alternative to two-sample *t*-test, which does not rely on the assumption of data complying with any distribution [15]. Figure 1 visualizes the ranking algorithm proposed. The data are organized in a form of matrix. Each column represents a feature, and each row stands for one source file. The non-vulnerable files appear in the top part from row 1 to I, and the vulnerable files followed in the bottom part from row $I+1$ to $I+J$. All the numbers in this matrix are the values of corresponding feature for the underling files. For the 1st feature, the non-vulnerable C_n and the vulnerable C_v groups are built, and the p-value of the Wilcoxon test for the each feature is computed based on the C_n and C_v as shown in the figure.

C. The Learner

There are many studies addressing the use of different machine learning algorithms for vulnerability prediction such as

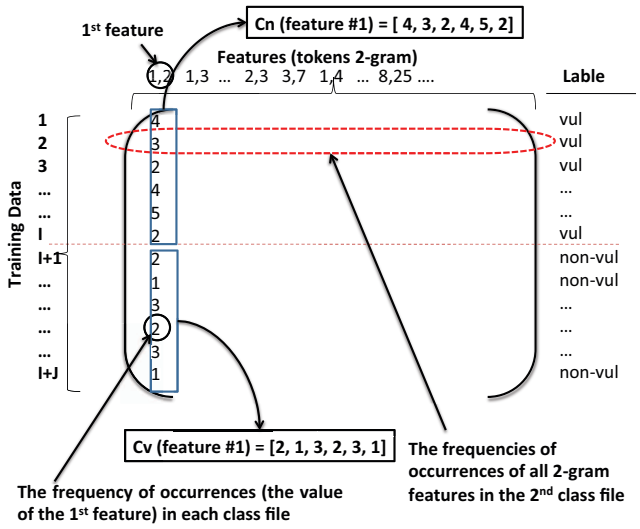

Fig. 1. feature ranking.

support vector machines, logistic regressions, neural network, decision Trees, random forest, and *k*-Nearest Neighbor [16], [17], [18]. It is a common belief that the performance of these learning techniques depends greatly on the characteristics of the data to be classified. In this study, in order to assess the performance of these techniques in addressing the vulnerability prediction problem, we have conducted a preliminary experimental study comparing these techniques. More specifically, we compared the performance of six algorithms based on a small set of subject programs. We observed that the best results were obtained when support vector machines were used. Therefore, we focused our efforts only on support vector machines. Support vector machine (SVM) is a machine learning algorithm widely used for classification and prediction problems. SVM utilizes a kernel function to perform both linear and non-linear classifications. For a simple binary high-dimensional linear classification problem, the SVM algorithm builds a hyper-plane with the intention of maximizing the distance between the hyperplane and its nearest data points on each side of the hyperplane.

IV. Experiments

In order to show the effectiveness of the proposed technique, we conducted a number of experiments. In this section, first we describe the subjects programs and the experimentation setup, and then introduce the evaluation metrics under the context of binary classification. We finally report the results of the analysis.

A. Subject Programs

TABLE I. Subject programs.

Program	Category	Class	Downloads
BoardGameGeek	game	70	50K
Connectbot	communication	51	5M
CoolReader	book reader	51	5M
AnkiDroid	education	28	500K

Table I lists the subject programs that we have used and

can be downloaded from F-Droid [2]. The table also reports the type of the applications, the number of classes, and the number of downloads that reflects the popularity of these applications: `BoardGameGeek` is a board game; `Connectbot` is a Secure Shell (SSH) client; `CoolReader` is an XML/CSS based eBook reader; and `AnkiDroid` is a flashcard management and editing tool. The reason we use these application is that 1) they are some medium-sized daily used applications; 2) they are used by other researchers [4], 3) the benchmark is available online[3], and some raw data can be used directly.

B. Data Collection

We need two sets of information for each class: the label and the feature. We developed some Java utility programs to extract and organize the feature information. We downloaded the label information from the benchmark online. We extracted all the *N*-features ($1 \leq n \leq 5$). We developed some R scripts to perform the Wilcoxon test based feature selection. Only one fifth of features were used and the rest were excluded from the experiments. For each application, we replicated the experiments 10 times using 5-fold cross-validation by randomly choosing one fifth of data as training data, and predict the rest four fifths as testing data. We developed some R scripts to perform the training and testing data splits. By calling "svm()" function in "LIBSVM" library, we obtained the predicted potentially vulnerable classes. We carried out the grouping process on a Dell laptop with Windows XP environment with an Intel processor (1.66HZ, 2 cores) with 1 GB physics memory. The labels of all classes for a program were obtained within 20 minutes.

C. Evaluation Metrics

In the field of statistics and in particularly for the prediction purposes, four key terms are usually computed for assessing the performance of a classifier: true positives(tp), true negatives(tn), false positives(fp), and false negatives(fn). The terms positive and negative refer to the classifier's prediction, also known as the expectation, and the terms true and false refer to whether that prediction corresponds to the external judgment, also known as the observation [19]. Three major measurement metrics, i.e. precision and recall are usually used to assess how well a binary classification is performed [19]. The accuracy is the degree of closeness of measurements of a quantity to the quantity's actual (true) value. The precision, also called as reproducibility or repeatability, is the degree to which repeated measurements under unchanged conditions show similar results. The recall measurement in this context, is also referred to as the true positive rate or sensitivity, is the ratio of true positives over the sum of true positives and false negatives or the percentage of flows in an application class that are correctly identified.

$$Accuracy = \frac{tp + tn}{tp + fp + fn + tn} \tag{1}$$

$$Precision = \frac{tp}{tp + fp} \tag{2}$$

$$Recall = \frac{tp}{tp + fn} \tag{3}$$

[2]https://f-droid.org/

[3]https://sites.google.com/site/textminingandroid/

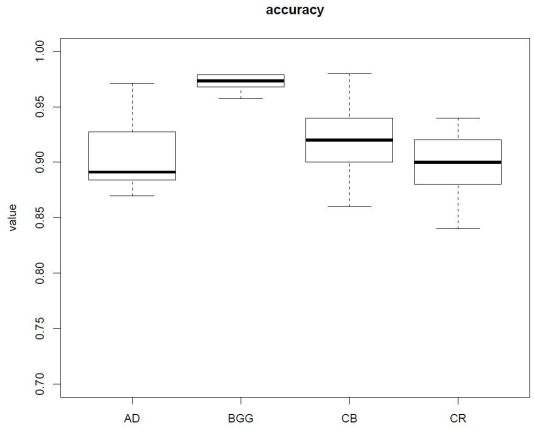

Fig. 2. The Accuracy Value of the 4 Applications; AD: AnkiDriod; BGG: BoardGameGeek; CB: ConnectBot; CR: CoolReader

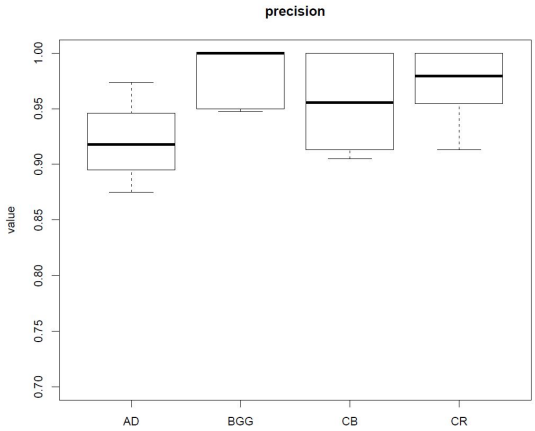

Fig. 3. The Precision Value of the 4 Applications; AD: AnkiDriod; BGG: BoardGameGeek; CB: ConnectBot; CR: CoolReader

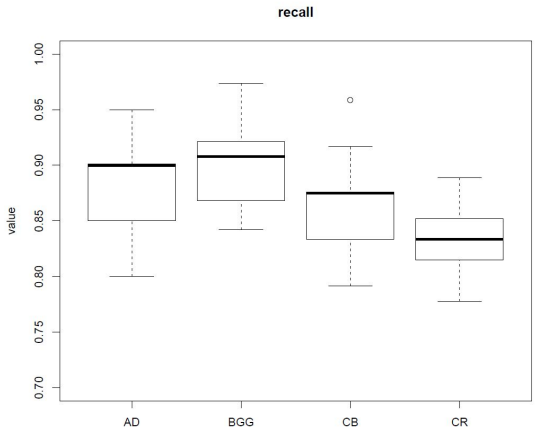

Fig. 4. The Recall Value of the 4 Applications; AD: AnkiDriod; BGG: BoardGameGeek; CB: ConnectBot; CR: CoolReader

TABLE II.

Program	Accuracy	Precision	Recall
AD	90.58%	92.22%	89.00%
BGG	97.23%	97.95%	90.26%
CB	91.60%	95.93%	86.25%
CR	89.60%	97.02%	83.33%
AVE	92.25%	95.78%	87.21%

D. Evaluation

Table II summarizes and reports the results in terms of means for accuracy, precision and recall, respectively, when the proposed technique is applied to the four applications. Overall the average accuracy is 92.25%; the precision is 95.78%; and the recall is 87.21%. Figures 2, 3 and 4 depict the data for each program. As demonstrated in the boxplots, the average accuracy are 90.58%, 97.23%, 91.60% and 89.60%; the average precision are 92.22%, 97.95%, 95.93% and 97.02%, the average recall are 89.00%, 90.26%, 86.25% and 83.33%, respectively. Shin et al. [1] have suggested that precision and recall values around 0.7 are reasonable for prediction models. With respect to the Shin et al. judgement, the performance of the proposed technique for these Java Android applications is fascinating. High precision demonstrated that once a class is labeled as vulnerable, it is likely to be an actual vulnerable class. High accuracy indicates that for most cases, both the actual vulnerable and actual non-vulnerable classes are classified properly. High recall indicates that we can identify all of the actual vulnerable class.

V. Discussion

In addition to measuring the effectiveness of the proposed technique, we further pose two research questions and address them through additional experiments:

RQ1: How prevalence are the vulnerable software components in typical daily-used android based applications? An answer to this question could identify the importance of technique.

RQ2: Is the propose technique effective in the cross-project identification? The answer to this question could help generalizing the technique.

A. The Prevalence of Vulnerabilities

The impact of vulnerabilities can be very high. On the other hand, the Android operating system continues to gain market share among smart phone users across the world. It was reported that Android had reached over 50% market share in the United States and Great Britain and over 70% in Germany and China [20]. Given the fact the Android subject programs studied in this paper are pretty popular it is of interests to study how provenance is the vulnerable components in these daily used applications.

Figure 5 shows that the four common Java Android programs approximately 40% of their source code files contain some vulnerabilities, ranging from 21% in $BoardGameGeek$ to 55% in $AnkiDroid$. Regarding $RQ1$, we may conclude that vulnerable components account for a considerable large percentage in these daily used Java Android applications. We may conclude that the identification of vulnerable components is still a major concern for the software security research community.

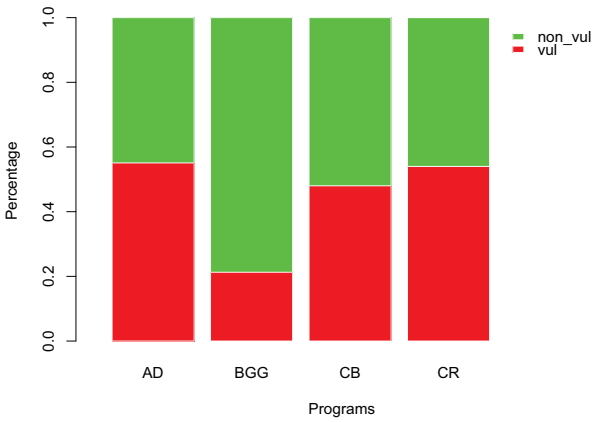

Fig. 5. The percentage of vulnerable software components for each program; red color area means vulnerable components

B. The effectiveness of cross-project prediction

For the purpose of generalizing the vulnerable components prediction techniques, other researchers have investigated whether a predictor trained from the data of another software project could be applied, i.e., cross-project predictions. Cross-project prediction is notoriously difficult for many mining-based software engineering approaches, since software projects are often very different from each other in their usages, structures, programming rules, etc. What makes the situation even worse is that it is difficult to measure and handle such differences [21]. Therefore, it is also interesting to study whether the proposed approach works in cross-project scenario.

TABLE III. Experiments in cross-project scenario; Program A: the program where vulnerable components are from; Program B: the target program

Prog A	Prog B	Accuracy	Precision	Recall
AD	BGG	62.11%	65.38%	62.96%
AD	CR	80.00%	84.00%	77.78%
BGG	CD	52.39%	55.56%	55.56%
BGG	CR	66.66%	69.23%	66.67%
CD	CR	58.77%	60.71%	62.96%
CD	AD	48.22%	52.17%	44.44%
CR	BGG	64.18%	69.57%	59.26%
CR	AD	74.64%	76.92%	74.07%
AVE		63.37%	66.69%	62.96%

In this experiment, we combine a pair of applications together to investigate if the observed venerable components could help identifying vulnerabilities in another application. For example, we employ vulnerable and non-vulnerable classes in application A as training data, to classify software components in Application B. We repeated the experiments several times in each scenario, as shown in Table III. The results clearly indicate that the proposed technique deteriorated significantly in the cross-project scenario. Regarding $RQ2$, this small-scale experiment illustrates that the performance of the proposed technique deteriorated significantly in the cross-project scenario. The results are consistent with other prediction technique.

VI. Threats to Validity

Some raw data used in this experimental study are downloaded online, for example, the results were obtained using fortify tools by other researchers. Another issue related to the external is that different languages and different platforms may have different features and thus the performance of the proposed technique for different cases may vary. All the data are extracted from Java source code, and the XML files which also contains important information are excluded [4]. Instead of using any external tools, we developed some Java and R scripts to extract the features and values, to conduct the statistical feature selection, and to perform the binary classification. In order to remove the internal threats, the intermediate data was tested by a graduate student major in software engineering. The basic prediction model used in this paper is support vector machine. In practice, other models are also used like neural network, logistic regression and Naive Bayes. The results obtained from different models may vary. For the evaluation of prediction, we used accuracy, precision, and recall to measure the performance. However, in practice there may be other metrics and representation demonstrating how well a classifier performs. It is interesting to analyze the result based on other metrics.

VII. Conclusion

In this paper, we proposed a technique which combines *N*-grams analysis and statistical feature selection for prediction vulnerable software components. The sequences of tokens in the source code are regarded as features, and the Wilcoxon test was employed to reduce the feature space. Based on our experimental study we concluded that the proposed technique could classify vulnerable classes with high precision, accuracy and recall. Further experiments are needed to validate the results we obtained in this paper. In particular, experiments on either other language based programs are of interest. Also, the idea of the proposed technique is to use machine learning algorithm, which are grounded on features study. In practice there are other ways to construct features, we would explore further in this direction. There exists some other classification techniques, like the ensemble methods we used in previous study [22] [23]. We would explore the possibility of applying them into vulnerable software component classification problem.

Acknowledgement

This work is supported by the Connecticut State University American Association of University Professors (CSU-AAUP) Research Grants in Southern Connecticut State University.

References

[1] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[2] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[4] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *Software Engineering, IEEE Transactions on*, 2014.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340–355, 2005.

[6] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 31–38.

[7] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*. ACM, 2012, pp. 7–10.

[8] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. ACM–Association for Computing Machinery, 2015.

[9] T. Golub, D. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. Mesirov, H. Coller, M. Loh, J. Dowing, M. Caliquri, C. Bloomfield, and E. Lander, "Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring," *Science*, vol. 286, pp. 531 – 537, 1999.

[10] L. Deng, J. Ma, and J. Pei, "Rank sum method for related gene selection and its application to tumor diagnosis," *Chinese Science Bulletin*, vol. 49, no. 15, pp. 1652–1657, 2004.

[11] P. Sood, A. Krek, M. Zavolan, G. Macino, and N. Rajewsky, "Cell-type-specific signatures of micrornas on target mrna expression," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103, no. 8, pp. 2746–2751, 2006.

[12] C. Möller, S. Dreborg, H. A. Ferdousi, S. Halken, A. Høst, L. Jacobsen, A. Koivikko, D. Y. Koller, B. Niggemann, L. A. Norberg *et al.*, "Pollen immunotherapy reduces the development of asthma in children with seasonal rhinoconjunctivitis (the pat-study)," *Journal of allergy and clinical immunology*, vol. 109, no. 2, pp. 251–256, 2002.

[13] K. Gao, T. M. Khoshgoftaar, and R. Wald, "Combining feature selection and ensemble learning for software quality estimation," in *The Twenty-Seventh International Flairs Conference*, 2014.

[14] X. Xue, Y. Pang, and A. S. Namin, "Feature selections for effectively localizing faulty events in gui applications," in *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*. IEEE, 2014, pp. 306–311.

[15] S. Siegel, "Nonparametric statistics for the behavioral sciences." 1956.

[16] 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.

[17] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[18] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 215–224.

[19] D. L. Olson and D. Delen, *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[20] B. Wolfe, K. Elish, and D. Yao, "High precision screening for android malware with dimensionality reduction," in *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*. IEEE, 2014, pp. 21–28.

[21] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can i clone this piece of code here?" in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 170–179.

[22] X. Xue, Y. Pang, and A. S. Namin, "Trimming test suites with coincidentally correct test cases for enhancing fault localizations," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE, 2014, pp. 239–244.

[23] Y. Pang, X. Xue, and A. S. Namin, "Identifying effective test cases through k-means clustering for enhancing regression testing," in *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, vol. 2. IEEE, 2013, pp. 78–83.