2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)

Using Software Metrics for Predicting Vulnerable Code-Components: A Study on Java and Python Open Source Projects

Tai-Yin Chong, Vaibhav Anu, Kazi Zakia Sultana
Department of Computer Science, Montclair State University
{chongt2, anuv, sultanak}@montclair.edu

***Abstract*—Software vulnerabilities often remain hidden until an attacker exploits the weak/insecure code. Therefore, testing the software from a vulnerability discovery perspective becomes challenging for developers if they do not inspect their code thoroughly (which is time-consuming). We propose that vulnerability prediction using certain software metrics can support the testing process by identifying vulnerable code-components (e.g., functions, classes, etc.). Once a code-component is predicted as vulnerable, the developers can focus their testing efforts on it, thereby avoiding the time/effort required for testing the entire application. The current paper presents a study that compares how software metrics perform as vulnerability predictors for software projects developed in two different languages (Java vs Python). The goal of this research is to analyze the vulnerability prediction performance of software metrics for different programming languages. We designed and conducted experiments on security vulnerabilities reported for three Java projects (Apache Tomcat 6, Tomcat 7, Apache CXF) and two Python projects (Django and Keystone). In this paper, we focus on a specific type of code component: Functions. We apply Machine Learning models for predicting vulnerable functions. Overall results show that software metrics-based vulnerability prediction is more useful for Java projects than Python projects (i.e., software metrics when used as features were able to predict Java vulnerable functions with a higher recall and precision compared to Python vulnerable functions prediction).**

***Index Terms*—software security, software metrics, vulnerability prediction, software reliability, machine learning**

I. Introduction

Vulnerability detection during software development is essential so as to reduce the testing effort and also to reduce the cost incurred due to software security problems. Security patterns and secure-programming guidelines given to developers [1]–[3] can reduce the probability of writing vulnerable code. However, separate set of rules needs to be developed for each programming language. Furthermore, the existing static analysis tools [4], [5] only cover those issues that make code susceptible to generic bugs and the tools often do not consider security bugs. To alleviate such issues faced by developers, researchers have examined the source code repositories and vulnerability history to mine some code constructs (e.g., software patterns) that may be associated to vulnerable code. In past studies, software nano-patterns and metrics were extracted from vulnerable code and used as features in machine learning predictors [6]–[11]. Nano-patterns based models did not prove effective for vulnerability prediction as they yielded high false-positive rates (i.e., code-components that were not vulnerable were predicted as vulnerable). The metrics-based models had lower recall rates. Owing to this inefficacy of nano-patterns and metrics for vulnerability prediction, we focus on analyzing the performance of software metrics as features in vulnerability prediction of systems in different languages. Originally, software metrics were evolved to quantify the characteristics of a software. The values of software metrics are generally used for various purposes including cost estimation, software quality assurance, and resource allocation. In this study, we considered the function-level software metrics. Note that for Java projects, function-level metrics are more commonly called method-level metrics. Examples of function-level metrics are provided in Table I.

The primary focus of our research is to compare the performance of software metrics as vulnerability predictors for different programming languages. So, for our preliminary investigation, we started our experiments with projects written in two different languages (Java and Python). We tried to determine if there is any difference in the predictive performance of software metrics for two different languages (Java vs Python) at function-level.

The major contributions of this paper are as follows:

- We have analyzed the performance of metrics at function-level for both programming language contexts (Java vs. Python). As the code component that we considered (i.e., function) is a lower granularity level than file, the developers will be able to trace the origin of vulnerability more easily. This is because, if our prediction model predicts a function as vulnerable, then the developers just need to focus their testing efforts on that particular function (instead of reviewing the code in the entire file).
- This study also provides a comparison of the performance of software metrics in vulnerability prediction for two different programming language contexts (Java vs. Python).

In Section II, we have described the research questions, system selection, the methodologies followed for vulnerability collection, metrics extraction process, and algorithms used for vulnerability prediction. Section III presents the results of our experiment along with their implications. We discuss the limitations of our work in Section IV and finally conclude the paper in Section V.

978-1-7281-1664-8/19/$31.00 ©2019 IEEE
DOI 10.1109/CSE/EUC.2019.00028

TABLE I
FUNCTION-LEVEL SOFTWARE METRICS

Metrics	Description
Count Input [12]	Number of inputs a function uses plus the number of unique sub-programs calling the function. Inputs include parameters and global variables that are used in the function.
Count Output [12]	The number of outputs that are SET. This can be parameters or global variables. So Functions calls + Parameters set/modify + Global Variables set/modify.
LOC [13]	The number of lines that contain source code. A line can contain source and a comment and thus count towards multiple metrics.
Cyclomatic complexity	McCabes cyclomatic complexity counts the number of independent paths through a program unit (i.e., number of decision statements plus one).
Modified cyclomatic [14]	The Cyclomatic Complexity except that each decision in a multi-decision structure (switch in C/Java) statement is not counted and instead the entire multi-way decision structure counts as 1.
Max nesting [15]	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.

II. METHODOLOGY

The following subsections present the research questions, system selection, the methodologies of vulnerability collection, metrics extraction, and vulnerability prediction.

A. Research Questions

Based on our research goal, we formulated two research questions to evaluate our results.

1) Research Question 1 (RQ1): **How do software metrics perform when used as features for predicting vulnerable functions/methods in Java projects?**

The experiments related to this question evaluated usefulness of function-level software metrics as predictors for potential security vulnerabilities in Java functions (also known as methods). We obtained various performance measures including false positive rate, recall, precision, and ROC while using metrics as features for classifying Java functions as vulnerable vs non-vulnerable.

2) Research Question 2 (RQ2): **How do software metrics perform when used as features for predicting vulnerable functions in Python projects?**

The experiments related to this question evaluated usefulness of function-level software metrics as predictors for potential security vulnerabilities in Python functions. We obtained various performance measures including false positive rate, recall, precision, and ROC while using metrics as features for classifying Python functions as vulnerable vs non-vulnerable.

B. System Selection

We selected three Java projects: Apache Tomcat 6[1], Tomcat 7, and Apache CXF[2]. Apache projects release their vulnerability reports under what they call Security Advisories. Each Security Advisory provides the name of the classes affected by a vulnerability along with the type of vulnerability (Denial of Service, Information Disclosure etc.). Vulnerability reports for Apache Tomcat and CXF are available at their security pages[3,4]respectively. For the current paper, we were only interested in collecting vulnerable functions/methods from the affected classes.

TABLE II
SYSTEMS

Systems	Language	Number of Vulnerable Functions Identified
Tomcat-6	Java	124
Tomcat-7	Java	106
Apache-CXF	Java	45
Django	Python	78
Keystone	Python	120

For all Apache projects, we considered the last reported version at the time of data collection as the non-vulnerable version, as vulnerabilities reported in each project had already been fixed and do not exist (in the same form) in the last version of that release. For example, 6.0.48 was the last version in Tomcat-6, 7.0.75 was the last version in Tomcat-7 and 3.1.10 was the last version in CXF at the time of data collection. Two authors reviewed the downloaded code to ensure that the vulnerability has been removed from the non-vulnerable version. Authors also note the following validity threat: although the last version may contain other vulnerabilities that may be reported in subsequent releases, the vulnerabilities reported in earlier versions and considered in this study have been fixed and do not exist in that release's final version. It was challenging to find Python projects that were both open source and had enough vulnerability data. We chose Django and Keystone as they are open source projects and they have extensive version histories.

C. Vulnerability Collection

1) Apache Tomcat: Apache Tomcat security reports provide information about the vulnerability type, its CVE-id (Common Vulnerabilities and Exposures), affected versions, revision number, fixed version, and the affected classes. Figure 1 shows the Remote Code Execution (CVE-2017-12617) vulnerability was fixed in four revisions including 1809978, 1809992, 1810014 and 1810026 of version 7.0.82. If we follow the link to first revision number[5], we obtain the list of code-components modified to fix the vulnerability as shown in Figure 2. After that, if we click "text changed" link, it shows us the lines of code that were added or removed for fixing the vulnerability. We manually extracted the name of the function that contained the modified lines of code for fixing the issue. In this way, we collected the vulnerable functions from the affected versions. If a vulnerability affects versions 7.1, 7.2, 7.3 and is fixed in 7.5, we considered 7.3 as the last affected version. In this way, we collected the last affected code versions for the listed vulnerabilities in all Apache systems under study. We found 14 versions affected with vulnerabilities in Tomcat-6 and 18 versions in Tomcat-7. In these versions, the total number of affected functions was 124 for Tomcat-

[1]https://tomcat.apache.org/
[2]http://cxf.apache.org/
[3]https://tomcat.apache.org/security.html
[4]http://cxf.apache.org/security-advisories.html
[5]https://svn.apache.org/viewvc?view=revision&revision=1809978

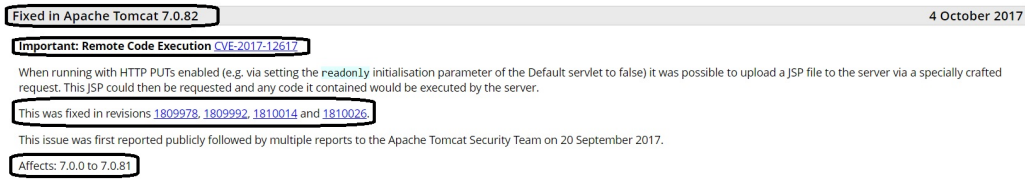

Fig. 1. Apache Tomcat Security Page

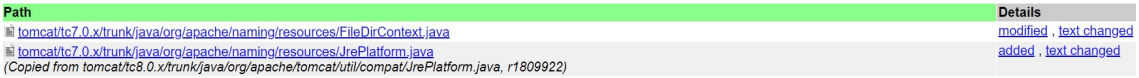

Fig. 2. Affected code-component names for a vulnerability

6 and 106 for Tomcat-7 (cf. Table II). The source code for Apache Tomcat is available in Archives of Tomcat[6].

2) Apache CXF: The vulnerability reports of Apache CXF provide the vulnerability type, its CVE-id, affected versions, revision number and fixed version. We collected 12 affected code versions for Apache CXF. We found 45 vulnerable functions in Apache CXF (cf. Table II). The source code for Apache CXF is located in Apache Archives of CXF[7].

3) Django: CVEDetails [16] lists the vulnerabilities reported for the Django project. This archive provides the CVE-id linked to a more detailed report on the particular vulnerability[8]. The report provides the type of threat, CVSS (Common Vulnerability Scoring System) scores, and affected versions. The report contains references to the security releases of the project [17] which has information about the changed files for fixing that specific vulnerability. In the Github patch [18], we could see the additions and deletions of code and determine the vulnerable functions for each file. We found 78 vulnerable functions in Django project.

4) Keystone: We used CVEDetails archive [19] for collecting vulnerability reports of the Keystone project. Through the CVE id, we could find more detailed reports[9] including the type of threat, CVSS scores, and affected versions. The detailed report page would at times link to another page[10] where all the patch file links (additions and deletions to the code for fixing vulnerability) are available. The patch files allowed us to determine the vulnerable functions. We found 120 vulnerable functions in Keystone project.

D. Software Metrics Extraction

We used a commercial tool called Understand 4.0 [20] to compute the source code metrics. We first created a project in Understand for every version of every system and ran a scheduler to extract the specified metrics we needed. This process took almost 30 seconds for each project version. For each system version under study, we generated a separate comma-separated values (CSV) file containing the function-level metrics as listed in Table III for every function in the system. We extracted the vulnerable functions and their metrics from this CSV file and placed them in a separate CSV file. Then we followed the same procedure to collect function-level metrics for the non-vulnerable functions in a particular project. As can be seen in Table III, Understand 4.0 identified and generated values for 18 function-level metrics for each of the three Java projects. For each of the two Python projects, values were generated for 17 function-level metrics by the tool.

[6]http://archive.apache.org/dist/tomcat/
[7]http://archive.apache.org/dist/cxf/
[8]https://www.cvedetails.com/cve/CVE-2019-6975/
[9]https://www.cvedetails.com/cve/CVE-2018-14432/
[10]https://www.openwall.com/lists/oss-security/2018/07/25/2

E. Vulnerability Prediction

In this paper, we identified (using the Understand 4.0 tool) a set of function-level metrics as features (Table III provides a list of the identified metrics). We collected the feature values from vulnerable and non-vulnerable functions. There are two groups for each project: vulnerable and non-vulnerable. We collected labeled data (marked as vulnerable or non-vulnerable) and then trained the machine so that it can classify a supplied function as vulnerable or non-vulnerable based on its (machine's) learning algorithm. Supervised machine learning can help predict vulnerable components based on the values of the features. We applied two supervised learning techniques (Support Vector Machine, and Logistic Regression) as they have been successfully used in earlier research studies that focused on vulnerability prediction [8]. SVM is a supervised learning model where the training points are separated by the widest possible gap. The new point is classified based on its side of the gap. SVM can perform both the linear and non-linear classification by mapping the input into higher dimensional space [21]. Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function [22].

1) Tool Used: We developed vulnerability predictors by using two machine learning techniques (SVM and LR). Waikato Environment for Knowledge Analysis (WEKA) is a popular open source toolkit for machine learning and data mining [23]. We used WEKA 3.8 for our study. The parameters of the techniques were initialized using the default settings in WEKA.

TABLE III
FUNCTION-LEVEL METRICS USED IN THIS STUDY AS FEATURE-SETS TO TRAIN THE ML ALGORITHMS

List of Function-level Metrics used for Java in this Study (as feature-set)	List of Function-level Metrics used for Python in this Study (as feature-set)
CountInput, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountOutput,CountPathLog, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, Cyclomatic, CyclomaticModified, CyclomaticStrict, Essential, MaxNesting	CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountPath, CountPathLog, CountStmt, CountStmtDecl, CountStmtExe, Cyclomatic, CyclomaticModified, CyclomaticStrict, Essential, MaxNesting, RatioCommentToCode

2) Data Balance: As our dataset was not balanced, we needed to create a balanced dataset consisting of the same number of vulnerable and non-vulnerable functions. Earlier studies either considered under-sampling of the majority class or over-sampling of the minority class. The problem with under-sampling is that information may be lost. In the case of over-sampling, duplicating the instances of the minority category (vulnerable functions) can make the system biased to the minority category when the number of vulnerable functions is too small. In this study, we applied the *ClassBalancer* filter[11]. This filter re-weights the instances so that each category has the same total weight.

3) Data Analysis: We executed two machine-learning algorithms for the vulnerable and non-vulnerable functions of all the projects under study. For each algorithm, we applied 10-fold cross-validation to ensure that the trained model will work accurately for an unknown dataset in practice [24]. In 10-fold cross-validation, 90% data is used as training and remaining 10% is used for testing and the process runs for 10 times considering different 10% data as test-data in every run. The result of this experiment shows how accurately a model trained with historical data can predict vulnerable functions.

4) Performance Measures:

- **Precision:** Precision for vulnerable function prediction can be defined as the ratio of the number of predicted vulnerable functions that are actually vulnerable to the total number of functions retrieved as vulnerable [24].
- **Recall:** Recall of vulnerable function prediction is defined as the ratio of the number of predicted vulnerable-functions that are actually vulnerable to the total number of vulnerable functions in the system. This measure is significant in the case of vulnerability prediction because the higher the recall is for vulnerable component prediction, fewer vulnerability will remain undetected.
- **False Positive Rate:** The FP rate indicates the percentage of non-vulnerable functions that are wrongly predicted as vulnerable. A high FP rate makes a predictor ineffective (i.e., renders the predictor unfit for use).
- **ROC:** ROC (Receiver Operating Characteristic) curve shows the trade-off between the True Positive (TP) rate and the False Positive (FP) rate. The area under the ROC curve is a measure that evaluates the performance of the binary classifier in terms of TP and FP rate. An area close to 1 indicates a high-performance model and an area about 0.5 indicates low-performance model [25].

[11] http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

III. RESULTS AND DISCUSSION

This section discusses the results of data analysis performed for the two research questions (Section II-A1 and II-A2).

A. Research Question 1 - How do software metrics perform when used as features for predicting vulnerable functions/methods in Java projects?

In order to answer this question, we used our identified set of function-level metrics as feature-set to train two Machine learning (ML) algorithms: Support Vector Machine (SVM) and Logistic Regression (LR).

For each Java system, the feature-set consisted of 18 function-level metrics that we identified using the Understand tool (Table III). Table IV and Table V present the performance measures (e.g. FP rate, precision, and recall) obtained when using the function-level metrics as features for the three Java systems under study. Table IV presents the performance measures for SVM whereas Table V presents the measures for LR.

Major insights from Table IV and Table V:

- One major insight was that the performance measures obtained by both algorithms (SVM and LR) for all three Java systems under study were pretty consistent. As can be seen in Table IV, SVM algorithm achieved a recall of 71%, 69.3%, and 73.3% for Tomcat-6, Tomcat-7, and Apache-CXF, respectively. Furthermore, the LR

TABLE IV
PERFORMANCE MEASURES OF FUNCTION-LEVEL METRICS AS FEATURES FOR JAVA VULNERABILITY PREDICTION (USING **SVM**)

System	FP Rate	Precision	Recall	ROC
Tomcat-6	17.4%	80.4%	71%	76.8%
Tomcat-7	15.7%	81.5%	69.3%	76.8%
Apache CXF	12.6%	85.3%	73.3%	80.4%

TABLE V
PERFORMANCE MEASURES OF FUNCTION-LEVEL METRICS AS FEATURES FOR JAVA VULNERABILITY PREDICTION (USING **LR**)

System	FP Rate	Precision	Recall	ROC
Tomcat-6	14.1%	82.6%	66.9%	85.2%
Tomcat-7	14.6%	83.0%	71.3%	84.8%
Apache CXF	10.2%	86.7%	66.7%	87.3%

algorithm achieved a recall of 66.9%, 71.3%, and 66.7% for the three systems. Overall, it can be concluded that the recall remained consistently around 70% for all three systems and for both ML algorithms. Similarly, the precision remained consistently around 81-82% for all three systems for both algorithms. The fact that the recall and precision were consistently moderate-to-high (in the 70 and 80 percents) motivates further investigation of using software metrics as feature sets for training ML classifiers to predict vulnerable Java code-components (in our case, functions are the code-components that we have focused upon).

- Overall, when compared to the LR algorithm, the SVM algorithm was able to achieve high recall in predicting vulnerable functions. For two systems, Tomcat-6 and Apache CXF, the SVM algorithm achieved a recall of 71% and 73.3% respectively. Only in the case of Tomcat-7, LR algorithm achieved slightly higher recall than SVM (71.3% vs. 69.3%). In the context of our research, achieving a higher recall rate is of utmost importance as a higher recall would ensure that most of the vulnerable functions are predicted as vulnerable by the model and therefore, high recall ensures that vulnerable functions do not go undetected. So, a major insight obtained from Tables IV and V is that SVM algorithm is more suitable for predicting Java vulnerable functions than LR algorithm (when using software metrics as feature-sets).

B. Research Question 2 - How do software metrics perform when used as features for predicting vulnerable functions in Python projects?

Similar to the analysis performed for RQ1, we used our identified set of Python function-level metrics as feature-set to train two Machine learning (ML) algorithms: SVM and LR. For each Python system, the feature-set consisted of 17 function-level metrics that we identified using the Understand 4.0 tool (cf. Table III).

Table VI and Table VII present the performance measures obtained when using the function-level metrics as features for the two Python systems under study. Table VI presents the performance measures for SVM whereas Table VII presents the measures for LR.

TABLE VI
PERFORMANCE MEASURES OF FUNCTION-LEVEL METRICS AS FEATURES FOR PYTHON VULNERABILITY PREDICTION (USING **SVM**)

System	FP Rate	Precision	Recall	ROC
Django	62.2%	44.6%	50.0%	43.9%
Keystone	68.6%	37.9%	41.9%	36.6%

TABLE VII
PERFORMANCE MEASURES OF FUNCTION-LEVEL METRICS AS FEATURES FOR PYTHON VULNERABILITY PREDICTION (USING **LR**)

System	FP Rate	Precision	Recall	ROC
Django	66.7%	40.6%	45.6%	33.5%
Keystone	56.9%	41.0%	39.5%	29.4%

Major insights from Table VI and Table VII:

- One major result found was that for both the Python systems under study (Django and Keystone), the performance measures achieved by both algorithms (SVM and LR) were quite low. The highest recall achieved for Python systems was achieved by SVM algorithm for Django system and the recall was only 50% (cf. Table VI). The recall remained consistently in the range of 40-50% for both the systems and for both algorithms. The other performance measures such as precision also remained consistently low (precision remained around 40% for both the systems and both algorithms).
- Similar to Java systems, in the case of Python systems also, SVM algorithm was able to achieve high recall than LR. However, there was only a slight difference in the recall achieved by SVM vs LR (for Django system, SVM algorithm achieved 50% recall compared to 45.6% recall achieved by LR algorithm. For Keystone system, SVM algorithm achieved 41.9% recall compared to 39.5% recall achieved by LR algorithm).

Based on the consistently low performance measures (recall and precision) achieved by machine learning classifiers (SVM and LR) for both the Python systems under study (Django and Keystone), it can be inferred that using software metrics as feature-sets for predicting vulnerable Python code-components does not yield beneficial/meaningful results. These results are quite contrary to the results that were found for Java-based systems, where the performance measures yielded by the two algorithms were consistently high (ranging from 70-86% as shown in Tables IV and V). In the next subsection (Section III-C), we compare the usefulness of our software metrics based vulnerability prediction approach in the context of Java systems vs Python systems.

C. Comparing the usefulness of software metrics based vulnerability prediction approach for Java-based systems vs Python-based systems

The results from RQ1 and RQ2 have shown that our metrics-based vulnerability prediction approach is more suited for Java systems than Python systems. Figure 3 compares the performance measures that were obtained for SVM algorithm for all three Java systems vs the two Python systems. As can be seen in Figure 3, for both performance measures (precision and recall), SVM yielded better performance for Java systems than Python systems. Similar results were also found for LR algorithm (we have only shown the comparison for SVM algorithm due to space limitations). One possible reason behind better performance of the metrics in Java could be that Java vulnerabilities are more related to the constructs of code (for example, code complexity, number of lines in code, or branching levels inside the decision structure) and therefore, they could be better classified by the metrics. On the other hand, Python vulnerabilities may not be directly related to

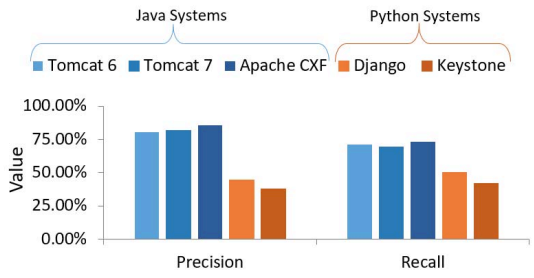

Fig. 3. Usefulness of Software Metrics for Vulnerability Prediction in Java vs Python (using SVM)

code-constructs that are characterized by the software metrics.

IV. Threats to Validity

In this section, we briefly enumerate the threats to the validity of the results obtained in this study. One major threat is that we collected the vulnerable functions from the vulnerability reports (reported by development teams). Authors note that this dependency on reported vulnerabilities could have resulted in some undiscovered vulnerabilities that may still exist in the non-vulnerable version. Furthermore, we conducted our experiments on three Java projects and two Python applications with known vulnerabilities. Although, our results were pretty consistent separately across the three Java and two Python systems, there is still a need to verify the results with other systems. We intend to address this threat in our future studies by analyzing more systems. Another validity threat is that, although we selected metrics that represent the characteristics of a Java or a Python function, they (the metrics) may not capture all properties. To mitigate this, we tried to capture as many metrics as possible (18 metrics for Java systems and 17 for Python systems) when building the feature-sets to train the classifiers.

V. Conclusion

In this study, we analyzed the performance of function-level software metrics in the vulnerability prediction of Java and Python systems. Our analysis showed that software metrics can be successfully utilized for predicting vulnerable code-components (such as functions/methods) in the case of Java-based systems. The moderate-to-high recall and precision obtained for Java systems (cf. Tables IV and V) motivates further research effort to improve the performance of our metrics-based vulnerability prediction approach for Java systems. Although there is existing research that analyzes the vulnerability prediction ability of software metrics, none of them have addressed the issue in the context of programming languages. Our comparative analysis for Java vs. Python will assist the developers to use the metrics in appropriate context. We expect that our experimental analysis will open up a new direction of research (i.e., using software metrics for vulnerability prediction and then analyzing their performance for different contexts such as different programming languages, code-components, etc.). In future, we will include more software projects and advanced machine learning techniques to make the results generalizable and extend the work to include other programming languages.

References

[1] M. G. Graff and K. R. V. Wyk, *Secure Coding: Principles and Practices*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2003.

[2] R. Seacord, *Secure Coding in C and C++*, 1st ed. Addison-Wesley Professional, 2005.

[3] M. Howard and D. E. Leblanc, *Writing Secure Code*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2002.

[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[5] "Pmd source code analyzer project," https://pmd.github.io/pmd/index.html, Last accessed on 2019-04-29.

[6] 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*, ser. ESEM '08, 2008, pp. 315–317.

[7] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, nov 2011.

[8] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. NY, USA: ACM, 2010, pp. 1963–1969.

[9] ——, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, mar 2011.

[10] K. Z. Sultana, A. Deo, and B. J. Williams, "A preliminary study examining relationships between nano-patterns and software security vulnerabilities," in *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[11] ——, "Correlation analysis among java nano-patterns and software vulnerabilities," in *IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, 2017.

[12] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*

[13] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. PWS Publishing Co., 1998.

[14] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[15] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," *SIGPLAN Not.*, vol. 16, no. 3, pp. 63–74, Mar. 1981.

[16] "Cve details," https://www.cvedetails.com/vulnerability-list/vendor_id-10199/product_id-18211/Djangoproject-Django.html/, Last accessed on 2019-04-29.

[17] "Django security releases issued: 2.1.6, 2.0.11 and 1.11.19," https://www.djangoproject.com/weblog/2019/feb/11/security-releases/, Last accessed on 2019-04-29.

[18] "Django," https://github.com/django/django/commit/402c0caa851e265410fbcaa55318f22d2bf22ee2/, Last accessed on 2019-04-29.

[19] "Cve details," https://www.cvedetails.com/vulnerability-list/vendor_id-11727/product_id-22720/Openstack-Keystone.html, Last accessed on 2019-04-29.

[20] "scitools: Visualize your code," https://scitools.com/, Last accessed on 2019-04-29.

[21] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*, 2000.

[22] T. Mitchell, *Generative and discriminative classifiers: naive bayes and logistic regression*. NY, USA: McGraw-Hill, Inc.

[23] E. Frank. (2016-12-19) Classbalancer. http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html. Accessed: 2017-08-04.

[24] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco, USA: Morgan Kaufmann, 2005.

[25] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16, 2016, pp. 1415–1421.