Received: 2 February 2020 | Revised: 22 June 2020 | Accepted: 1 July 2020

DOI: 10.1002/smr.2303

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach

Kazi Zakia Sultana | Vaibhav Anu | Tai-Yin Chong

Department of Computer Science, Montclair State University, Montclair, New Jersey, USA

Correspondence
Vaibhav Anu, Department of Computer Science, Montclair State University, Montclair, NJ, USA.
Email: anuv@montclair.edu

Abstract

[*Context*]A software vulnerability becomes harmful for software when an attacker successfully exploits the insecure code and reveals the vulnerability. A single vulnerability in code can put the entire software at risk. Therefore, maintaining software security throughout the software life cycle is an important and at the same time challenging task for development teams. This can also leave the door open for vulnerable code being evolved during successive releases. In recent years, researchers have used software metrics-based vulnerability prediction approaches to detect vulnerable code early and ensure secure code releases. Software metrics have been employed to predict vulnerability specifically in C/C++ and Java-based systems. However, the prediction performance of metrics at different granularity levels (class level or method level) has not been analyzed. In this paper, we focused on metrics that are specific to lower granularity levels (Java classes and methods). Based on statistical analysis, we first identified a set of class-level metrics and a set of method-level metrics and then employed them as features in machine learning techniques to predict vulnerable classes and methods, respectively. This paper describes a comparative study on how our selected metrics perform at different granularity levels. Such a comparative study can help the developers in choosing the appropriate metrics (at the desired level of granularity). [*Objective*] The goal of this research is to propose a set of metrics at two lower granularity levels and provide evidence for their usefulness during vulnerability prediction (which will help in maintaining secure code and ensure secure software evolution). [*Method*] For four Java-based open source systems (including two releases of Apache Tomcat), we designed and conducted experiments based on statistical tests to propose a set of software metrics that can be used for predicting vulnerable code components (i.e., vulnerable classes and methods). Next, we used our identified metrics as features to train supervised machine learning algorithms to classify Java code as vulnerable or non-vulnerable. [*Result*] Our study has successfully identified a set of class-level metrics and a second set of method-level metrics that can be useful from a vulnerability prediction standpoint. We achieved recall higher than 70% and precision higher than 75% in vulnerability prediction using our identified class-level metrics as features of machine learning. Furthermore, method-level metrics showed recall higher than 65% and precision higher than 80%.

KEYWORDS

software evolution, software maintenance, software metrics, software security, vulnerability prediction

1 | INTRODUCTION

Precautionary measures can be taken for mitigating risks and reducing the probability of insecure software release in different phases of the software development process. Secure coding during the development phase is not only useful for avoiding the introduction of vulnerabilities, but also useful for reducing security testing efforts. There are security patterns and guidelines for developers to follow[1-4] that can reduce the probability of writing vulnerable code. Although these rules help the developers in writing secure code, they cannot help in identifying the probable source of vulnerability when testers are trying to locate vulnerabilities (or in situations where testers/developers are looking to correct the code when one or more vulnerabilities have been exploited by the attackers). There are many static analysis tools (e.g., Findbugs[5]) that can locate specific vulnerabilities in the source code. These tools often suffer from high false positive (FP) rates and fail to detect all the vulnerable files.[6-10] Zitser et al.[9] analyzed five modern static analysis tools (ARCHER, BOON, PolySpace C Verifier, Splint, and UNO) and found that in PolySpace and Splint, the average false alarm rates were roughly 50%. According to Ayewah et al.,[7] static analysis tools sometimes report true but trivial issues (trivial issues are ones that might happen but would have minimal adverse impact if they did). For example, reporting a bug that would only impact logging output, or assertions, can be classified as a trivial issue. Although accurate logging messages are important, bugs in logging code might be considered as of lower importance.[7] These tools, generally, do not consider what the code is supposed to do. Instead, they usually look for unusual code, such as an awkward or dubious computation, or a statement that results (if executed) in an exception but might be dead code that can never be executed. Bad coding, which ideally should be cleaned up, might not impair the actual functionality of the software.[7] In contrast, oftentimes, bad coding may lead to serious error/s if resolved. Therefore, there is a real need for techniques/methods that can help determine which file has high probability of being vulnerable. Such techniques can ensure that developers do not need to depend on the results of static analyzers. Instead, they can focus on investigating selected files (that are likely to contain vulnerabilities) and ensure secure code. In addition, the developer can apply static analyzer tools on the selected files later to locate bad coding leading to vulnerability instead of using the tool for the entire code base.

One way to minimize the probability of code being vulnerable is to examine a project's source code repository and its associated vulnerability history to mine some code constructs (e.g., code patterns, software metrics) that can be used to build a vulnerability prediction model. In earlier studies, various software metrics and traceable patterns (Java class-level and method-level patterns that can be automatically traced or recognized) were extracted from code and used as features in a supervised machine learning (ML) prediction model.[11-18] Class-level traceable patterns are called *micro patterns*, whereas method-level traceable patterns are called *nano-patterns*. Gil and Maman[19] defined 27 micro patterns organized into eight categories (categories were created with respect to the formal conditions on the structure of Java classes). Nano-patterns are essentially method-level patterns that capture properties of methods within a class.[20] Although the above-mentioned pattern-based models were shown to have higher recall rates when predicting vulnerable classes/methods, there are some limitations in using them as features for vulnerability prediction models as they have lower precision than software metrics in vulnerability prediction.[18,21,22]

Software metrics (e.g., cyclomatic complexity, source line of code) were evolved to quantify the characteristics of a software. The numerical values of these metrics are generally used for software quality assurance, cost estimation, and resource allocation. Although researchers frequently use metrics-based prediction models, many FPs are introduced as the metrics are not directly related to coding constructs. In a recent survey,[23] a panel of seven software metrics experts answered questions regarding the current standing of software metrics. They agreed that it is not possible to pick one universal metric. The usefulness of a software metric depends on the context of the software. Therefore, in our study, we have evaluated the performance of the software metrics in the context of a particular granularity (class vs. function) and proposed useful metrics at two granularity levels (class level vs. method level in Java). Analyzing the performance at different granularity levels (class vs. function or method) is important because it will help the developers to decide which set of metrics they will use to predict vulnerability in their context. If the developers get appropriate set of metrics for vulnerability prediction, they will be able to locate vulnerable component accurately if any vulnerability is exposed after the release.

Multiple existing research studies have evaluated the effectiveness of the software metrics for vulnerability prediction,[11,12,15] but they do not compare the performance of metrics at different granularity levels, which can help the developers to realize applicability in their (developer's) specific context. To that end, *the high-level research goal* of this study is *to analyze the usefulness of software metrics for vulnerability prediction at different granularity levels in Java projects*. Using vulnerability datasets from several Java projects, we have performed statistical significance tests (e.g., Mann–Whitney *U* test) to propose two separate sets of metrics, one at Java class level and another at method level, and then evaluated the performance of the ML classifiers using these metrics as features.

The major contributions of this paper are as follows.

- We have analyzed the performance of metrics at varying granularity levels (class level vs. method level) for Java-based projects. This will help the developers to use the metrics at their chosen granularity level (i.e., class or method levels).
- We have proposed sets of metrics at class- and method-level granularity that can help the developers to start with a concrete set of metrics when building the vulnerability prediction models for their own projects. They do not need to identify their system-specific metrics.

- In this study, our research provides insights into how well the metrics (that are significant for a specific system) perform in vulnerability prediction in the later releases of that system. Similarly, a software developer can identify the significant metrics for the system and use them to build the vulnerability prediction model following our approach. This will guide the developers of a system to ensure secure software evolution in future releases.

The rest of this paper is organized as follows: Section 2 describes terms that are frequently used in this paper, followed by Section 3 that describes the related work. Section 4 presents the methodology of our experiments. Section 5 presents the findings of our experiments. A discussion of results is provided in Section 6. Section 7 discusses the validity threats for this research, and Section 8 discusses conclusion and future work.

2 | DEFINITIONS

In this section, we provide brief descriptions of various concepts that are frequently referred in this paper.

- *Class-level software metrics.* Class-level software metrics identify the characteristics of a class (the structural complexity of a class, dependency on other classes, number of methods per class). For example, McCabe's cyclomatic complexity counts the number of independent paths through a program unit (i.e., number of decision statements plus one). AvgCyclomatic[24] is a class-level software metric that takes the average of McCabe's cyclomatic complexity metrics for all nested functions or methods in a class.
- *Method-level software metrics.* Method-level software metrics identify the characteristics of a Java method (the structural complexity of a method, dependency on other methods, properties of parameters, return conditions, etc.). For example, MaxNesting[25] is a method-level metric that indicates the maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.
- *Supervised ML.* In this paper, we have collected the feature values from vulnerable and non-vulnerable classes or methods and labeled data (marked as vulnerable or non-vulnerable) and then trained the machine so that it can classify any class or method as vulnerable based on the specified learning algorithm. By doing so, supervised ML can help predict vulnerable component (in our case, class or method) based on the values of those features in that component. We selected two supervised learning techniques (support vector machine [SVM] and logistic regression [LR] as used in Chowdhury and Zulkernine[15]).
- *Precision.* Precision can be defined as the ratio of the number of predicted vulnerable classes (or methods) that are actually vulnerable to the total number of classes (or methods) predicted as vulnerable.[26]
- *Recall.* Recall in context of vulnerable class prediction is defined as the ratio of the number of predicted vulnerable classes that are actually vulnerable to the total number of vulnerable classes in the system. Similar definition of recall applies for vulnerable method prediction. This measure is significant in the case of vulnerability prediction because the higher the recall is for vulnerability prediction, fewer vulnerabilities will remain undetected. However, there is a trade-off between recall and precision. For example, if all the components predicted as vulnerable by the model are actually vulnerable, the model's precision will be 100%. On the other hand, there is still a possibility that many vulnerable components remain as undetected or wrongly predicted as non-vulnerable. Similarly, if a model can successfully detect all the vulnerable components, its recall will be 100%, but precision will degrade if it predicts many non-vulnerable components as vulnerable.
- *FP rate.* The FP rate indicates the percentage of non-vulnerable classes or methods that are wrongly predicted as vulnerable. A high FP rate renders a predictor less useful.
- *F-measure.* F-measure is a weighted average of precision and recall.[26] It gives equal importance to both precision and recall by considering their harmonic mean.[16,26] The formula for F-measure is defined as follows[27]:

$$F_\beta = \frac{(\beta^2 + 1)Precision \times Recall}{\beta^2 \times Precision + Recall}. \quad (1)$$

In the above equation, β controls a balance between precision and recall. β=1 makes F_1-measure equivalent to the harmonic mean of precision and recall. β>1 gives more weight to recall and β<1 makes it more precision oriented.

- *Receiver operating characteristic (ROC) curve.* The ROC curve shows the trade-off between the true positive (TP) rate and the 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.[28] An increase in recall (TP rate) also results in an increase in FP rate.[28] Therefore, the ROC measure helps to track the optimum point where the metric performs well with respect to the TP and FP rates.

3 | RELATED WORK

A software metric can characterize or measure the degree to which a property is contained by a software.[29] That property may be related to the structure of the software (e.g., cyclomatic complexity), or the management of the software (e.g., defect density). In our study, we will focus on the software metrics that measure different structural properties of the software. In previous studies, researchers either used some traditional software metrics (complexity metrics, code churn, etc.) or they explored some new metrics (e.g., text features, import statement) for vulnerability prediction. Therefore, we have divided this section into two subsections: one highlighting the studies that used traditional structural software metrics and another focusing on other studies that used some newly explored metrics for vulnerability prediction.

3.1 | Traditional software metrics-based vulnerability prediction models

Most of the research focused on creating vulnerability prediction models using complexity metrics.[11,12,15,16] Shin and Williams[13] showed that complexity metrics-based fault prediction models can be useful for vulnerability prediction to a certain extent. According to their findings, traditional metrics including code churn, complexity, and fault history exhibit similar performance in vulnerability prediction as they exhibit in fault prediction models.[13] They found that certain metrics including nesting complexity metrics can help in locating vulnerable code.[30] Shin et al.[12] studied the performance of metrics such as developer activity metrics alongside the various complexity metrics for predicting vulnerabilities. Another study identified development history metrics as stronger indicators of vulnerabilities than code complexity metrics as their research indicated that complexity metrics alone do not have statistically significant power to precisely and accurately predict vulnerabilities.[31] Chowdhury and Zulkernine[15,16] designed an empirical study using four alternative data mining and statistical techniques to explore the relation of complexity, coupling, and cohesion with vulnerability. They found that these metrics can be useful when employed in the early stages of development. Alves et al.[32] concluded that software metrics can help in distinguishing between vulnerable and non-vulnerable code, but they have no strong correlation with the number of vulnerabilities in the analyzed code. Zimmermann et al.[33] conducted a study to find the vulnerability prediction performance of conventional metrics including complexity, churn, coverage, dependency measures, and organizational structure of the company. These metrics showed higher precision and lower recall in vulnerability prediction. They also found that vulnerability prediction is more complex than fault prediction as program domain and usage of the program components have impacts on the accuracy of vulnerability prediction. Younis et al.[34] trained their model using a set of vulnerabilities previously exploited by attackers. During their study, they characterized the vulnerable functions with no exploit and the ones with an exploit using a small sample of metrics (they used eight software metrics). Walden et al.[35] employed various code- and design-based software metrics as vulnerability predictors and used text mining techniques to build vulnerability prediction models for web applications written in PHP (Hypertext Preprocessor).

3.2 | Other metrics-based vulnerability prediction models

Neuhaus et al.[36] introduced a new metric based on import statements and function calls to construct a vulnerability prediction model. The imports in programming languages (e.g., '#include' in C++, 'import' in Java) allow developers to reuse services provided by other libraries. They leveraged ML techniques to predict vulnerabilities with an average precision of 70% and recall of 45%. Zhang et al.[37] proposed a composite algorithm VULPREDICTOR to predict vulnerable files by analyzing software metrics and text features together, which was built using an ensemble of many classifiers. They claimed that VULPREDICTOR performs better than Walden et al.'s approaches for a wide range of files.[37] Scandariato et al.[38] used bag-of-words representation considering a Java source file as a series of terms with associated frequencies. However, the reason why the 'bag-of' or term frequency approaches are relevant to vulnerabilities was not clearly defined. Zhou and Sharma[39] designed a K-fold stacking classifier and experimented commit messages and bug reports in open source projects to identify undisclosed vulnerabilities. They achieved improved precision by 54.55% while maintaining the same recall rate compared with the work in vulnerability identification using commit messages.[39] They chose commit message as the only feature from commit texts, and therefore, in case of short-length commit messages, they may miss security-related information to identify vulnerability. Li et al.[40] presented the first deep learning-based vulnerability detection system (VulDeePecker) to automate feature extraction process and reduce the false negatives as incurred by other vulnerability detection systems. As deep learning method is yet to be explored in vulnerability prediction, the applicability of the approach is limited to certain types of vulnerability.[40] Abunadi and Alenezi[41] presented an empirical study on cross-project prediction techniques in vulnerability prediction. In this study, they randomly selected a set of software metrics as features and then employed them in cross-project validation using five different ML techniques. Feature selection may have an impact on the overall result of their study.[41]

Even though the above-mentioned research studies have successfully shown the usefulness of software metrics during vulnerability prediction, they suffer from some major drawbacks: (1) all the above-mentioned research focused only on a few selected metrics and failed to study the usefulness of many remaining metrics, and (2) the studies did not categorize the metrics at file or class or function levels. A file can be too large or too small based on the number of classes and methods it contains. According to Giger et al.,[42] file-level bug prediction requires more time to examine and locate the vulnerable area. Hence, the previous studies did not present a concrete set of metrics (either at class or function or file levels), which could be more useful for the developers in secure software development.

Therefore, if we analyze the metrics based on their granularity and determine their possible impact on vulnerability, it will be more useful for the developers and they can decide which metrics to use to locate the potential source of vulnerability in their code. Additionally, it would be more useful for the developers if they get a concrete set of metrics at different granularity levels (file or class or function) and use them as features to train their model. To that end, in our research, we have investigated all software metrics and determined which metrics can be useful when predicting vulnerable code components (in this paper, we refer to class and method as code components). Thus, we identified two sets of metrics at class and method levels, and then, we evaluated their performance in vulnerability prediction.

4 | METHODOLOGY

In this section, we will describe the research questions (RQs), system selection, the methodologies followed for vulnerability collection, metrics extraction, and model development for vulnerability prediction.

4.1 | Research questions

The goal of this research is to analyze the performance of software metrics in vulnerability prediction at different granularity levels (class level vs. method level) in Java projects. Based on our research goal, we formulated four RQs.

4.1.1 | Research question 1

What software metrics are best suited for vulnerability prediction at class level of Java-based software systems?

The focus of this RQ is on *identifying a set of software metrics that can be a useful feature set for training supervised ML algorithms to predict vulnerable Java classes*. Based on the normality distribution of data, we have performed Welch's test (for metrics with normally distributed data) and Mann–Whitney *U* test (for the rest of the metrics) to identify the above-mentioned set of metrics for each system under study. For different systems, we found different sets of significant metrics after the statistical tests, but we considered the common set of metrics (metrics that were found significant for all systems under study).

4.1.2 | Research question 2

What software metrics are best suited for vulnerability prediction at method level of Java-based software systems?

The focus of this RQ is on *identifying a set of software metrics that can be a useful feature set for training supervised ML algorithms to predict vulnerable Java methods*. We again performed the Mann–Whitney *U*test and Welch's test to identify the most useful metrics for each system under study. It should be noted that, for different systems and their respective code bases, the set of most useful metrics might be different. We considered the common set of metrics (similar to the process described for Java class-level metric selection in RQ1).

4.1.3 | Research question 3

Can the software metrics identified in RQ1 be employed by developers to predict vulnerability-prone classes in Java-based software systems?

In a sense, RQ3 can be restated as follows: *how do the class-level software metrics (identified in RQ1) perform as features for predicting vulnerable classes in Java-based software systems?* The results of the related experiments evaluate class-level software metrics as predictors for security vulnerabilities in Java classes. We present various performance measures including FP rate, recall, precision, ROC, and F-measure while using metrics as features in ML for classifying Java classes as vulnerable or non-vulnerable.

4.1.4 | Research question 4

Can the software metrics identified in RQ2 be employed by developers to predict vulnerability-prone methods in Java-based software systems?

In a sense, RQ4 can be restated as follows: *how do the method-level software metrics (identified in RQ2) perform as features for predicting vulnerable methods in Java-based software systems?* The results of the related experiments evaluate method-level software metrics as predictors for security vulnerabilities in Java methods. We present various performance measures including FP rate, recall, precision, ROC, and F-measure while using metrics as features in ML for classifying Java methods as vulnerable or non-vulnerable.

4.2 | System selection

We selected three different Java projects: Apache Tomcat (releases 6 and 7)[*], Apache CXF,[†] and the Stanford SecuriBench dataset.[43]

We selected Apache projects as they release their vulnerability reports as security advisories, including the name of the classes and methods affected by a vulnerability on the Apache website. Apache Tomcat vulnerability reports are available at the Tomcat security page[‡] and Apache CXF vulnerability reports are available at CXF security page[§]. Stanford SecuriBench is a set of open source programs used as a test bed for static and dynamic security tools.[44,45] We conducted the experiment for J2EE web applications Personalblog 1.2.6 and Roller 0.9.9 in the Stanford dataset. We used a static analyzer tool called early security vulnerability detector (ESVD)[46] to identify vulnerable code for the Stanford SecuriBench projects as it was shown to have fewer FPs in its results with higher precision and recall for the Stanford projects.[47] ESVD showed 97% precision and 78% recall rate in Personalblog, and in Roller, it had 99% precision and 89% recall rate.[47] For other projects, the recall rates are less than 65%. Therefore, we selected Personalblog and Roller for our study.

Next, for all Apache projects, the most current version at the time of our study was deemed to be the non-vulnerable version. This is because vulnerabilities reported in each project had already been fixed and did 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 study. The last version may contain other vulnerabilities that may be reported in subsequent releases, but we checked to ensure that the vulnerabilities reported in earlier versions (that we considered in this study) have been fixed by developers and did not exist in that release's current version. In the case of Stanford, we considered the classes and methods as non-vulnerable where no vulnerabilities were detected by ESVD tool (again, an interrater reliability of 95% was achieved between the two authors who verified ESVD's vulnerability reports).

4.3 | Vulnerability collection

In this section, we will describe how we collected vulnerable and non-vulnerable data from the systems under study. Apache projects publish the name of the vulnerability (e.g., denial of service, information disclosure) along with the common vulnerabilities and exposures (CVE)-id. The vulnerability types found in Stanford dataset are also reported by the ESVD tool. We collected those vulnerability names for all the vulnerable classes of the systems considered in this study and listed them in Table 1.

4.3.1 | Apache Tomcat

Apache Tomcat security reports provide information about the vulnerability type, its CVE-id, affected versions, revision number, fixed version, and the affected classes. Figure 1 shows that the denial of service (CVE-2014-0230) vulnerability was fixed in revision 1659537 of version 6.0.44. If we follow the link to its revision number,[¶] we obtain the list of classes modified to fix the vulnerability as shown in Figure 2. If a vulnerability affects the versions 7.1, 7.2, and 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 as shown in Table 2. We found 14 versions affected with vulnerabilities in Tomcat-6 and 18 versions in Tomcat-7. In these versions, the total number of affected methods is 124 for Tomcat-6 and 106 for Tomcat-7. The total number of affected classes is 104 for Tomcat-6 and 63 for Tomcat-7. The source code for Apache Tomcat is available in Apache Archives of Tomcat.[48]

[*] https://tomcat.apache.org/
[†] https://cxf.apache.org/
[‡] https://tomcat.apache.org/security.html
[§] https://cxf.apache.org/security-advisories.html
[¶] https://svn.apache.org/viewvc?view=revision&revision=1659537

TABLE 1 Vulnerabilities

Apache projects	Stanford SecuriBench
SQL injections (Structured Query Language)	Cross-site scripting
Security manager bypass	SQL injections
Request smuggling	
Information disclosure	
Frame injection in documentation Javadoc	
Session fixation	
DIGEST authentication weakness	
Denial of service	
Bypass of security constraints	
Bypass of CSRF (Cross Site Request Forgery) prevention filter	
Authentication bypass and information disclosure	
Multiple weaknesses in HTTP DIGEST authentication	
Cross-site scripting	
Security manager file permission bypass	
Remote denial of service and information disclosure	
Arbitrary file deletion	

FIGURE 1 Apache Tomcat security page

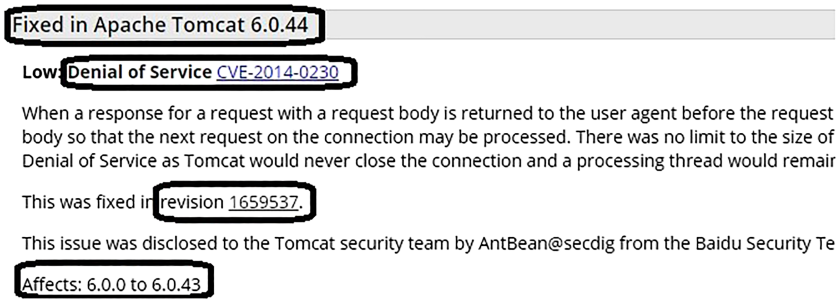
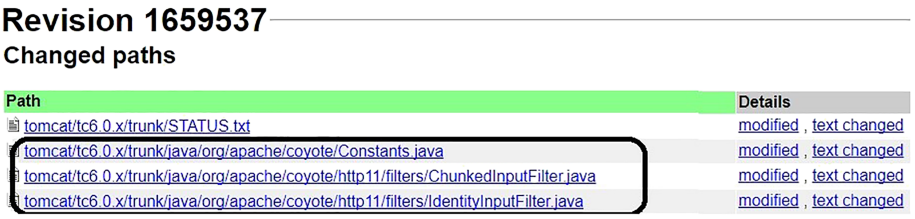


FIGURE 2 Affected class names for a vulnerability



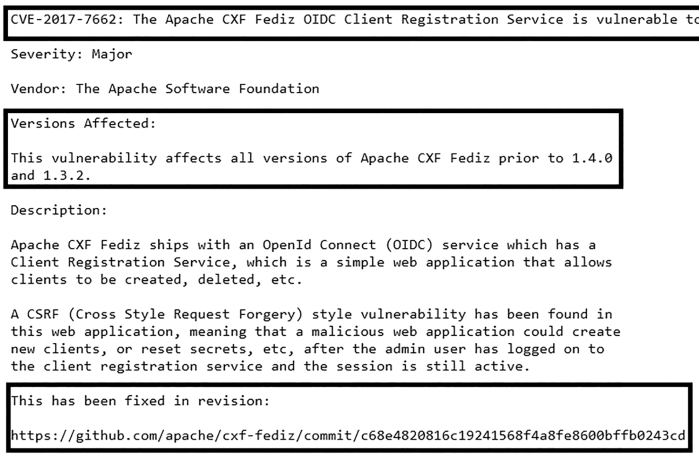
4.3.2 | Apache CXF

The vulnerability reports of Apache CXF provide information about the vulnerability type, its CVE-id, affected versions, revision number, and fixed version. We collected the last affected code versions for the listed vulnerabilities as shown in Table 2. We considered 12 versions for Apache CXF. The reports also provide the list of classes that were modified to fix the vulnerability. According to Figure 3, CSRF attacks (CVE-2017-7662) affected the versions of Apache CXF named Fediz prior to 1.4.0 and 1.3.2. Next, by following the link,[#] we obtain the list of classes involved in fixing the vulnerability. We found 45 vulnerable methods and 33 vulnerable classes in Apache CXF as shown in Table 2. The source code for Apache CXF is located in Apache Archives of CXF.[48]

[#]https://github.com/apache/cxf-fediz/commit/c68e4820816c19241568f4a8fe8600bffb0243cd

TABLE 2 Java-based systems considered during this study

Systems	Affected versions	Vulnerable methods	Vulnerable classes	Non-vulnerable methods	Non-vulnerable classes
Tomcat-6	6.0.16, 6.0.18, 6.0.26, 6.0.27, 6.0.29, 6.0.30, 6.0.32, 6.0.33, 6.0.35, 6.0.36, 6.0.37, 6.0.39, 6.0.41, 6.0.43	124	104	8645	2211
Tomcat-7	7.0.6, 7.0.10, 7.0.11, 7.0.16, 7.0.20, 7.0.21, 7.0.22, 7.0.27, 7.0.29, 7.0.32, 7.0.39, 7.0.42, 7.0.47, 7.0.50, 7.0.52, 7.0.53, 7.0.54, 7.0.57	106	63	10 296	2789
Apache CXF	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.2, 2.7.7, 2.7.9, 2.7.10, fediz-core-1.2.0, 3.0.2, 3.0.6, 3.1.8	45	33	26 366	737
Stanford SecuriBench	Personalblog 1.2.6 and Roller 0.9.9	156	91	2734	365


CVE-2017-7662: The Apache CXF Fediz OIDC Client Registration Service is vulnerable to CSRF attacks

Severity: Major

Vendor: The Apache Software Foundation

Versions Affected:

This vulnerability affects all versions of Apache CXF Fediz prior to 1.4.0 and 1.3.2.

Description:

Apache CXF Fediz ships with an OpenId Connect (OIDC) service which has a Client Registration Service, which is a simple web application that allows clients to be created, deleted, etc.

A CSRF (Cross Style Request Forgery) style vulnerability has been found in this web application, meaning that a malicious web application could create new clients, or reset secrets, etc, after the admin user has logged on to the client registration service and the session is still active.

This has been fixed in revision:

https://github.com/apache/cxf-fediz/commit/c68e4820816c19241568f4a8fe8600bffb0243cd

FIGURE 3 Apache CXF security page

4.3.3 | Stanford SecuriBench

We used ESVD tool to identify vulnerable classes and methods for the following projects: *Personalblog 1.2.6* and *Roller 0.9.9*. ESVD usually reports 11 types of vulnerabilities including command injection, cookie poisoning, cross-site scripting, log forging, and SQL injection. Of them, we considered only cross-site scripting and SQL injection types as others are not found in Apache projects. In addition, two authors manually checked the vulnerabilities reported by ESVD to determine if FPs existed in the dataset. The two authors achieved a 95% agreement on the vulnerabilities detected by ESVD and only then considered them in this study. As ESVD has plugin for Eclipse, we could manually verify the vulnerabilities as detected by ESVD (by examining the code in Eclipse). For example, ESVD lists the identified vulnerabilities along with their location in the source code as in Figure 4. We can validate why the vulnerability occurred and then decide if we would consider them in our experiment.

To do the manual validation of the identified vulnerable components by ESVD, the authors randomly picked 25 vulnerable methods (out of 156) and two authors independently studied them and rated them as either vulnerable or non-vulnerable. Next, the authors picked 25 non-

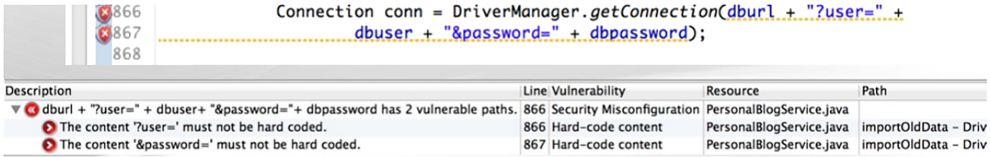


FIGURE 4 Early security vulnerability detector plugin for Eclipse

vulnerable methods (out of 2734) and two authors independently rated them as vulnerable or non-vulnerable. They repeated the same thing with classes. This way, two authors independently rated 100 code components (including classes and methods) as vulnerable and non-vulnerable. Then, they compared the ratings and found that both authors agreed with each other's ratings in most cases. For the cases that the authors did not agree upon, they discussed and were able to reach agreement on 95% of the cases (at which point they stopped further evaluations).

We explored the vulnerable code and found 12 vulnerable methods in Personalblog 1.2.6 and 144 vulnerable methods in Roller 0.9.9. We also detected a total of 91 vulnerable classes in both projects combined.

4.4 | Software metrics extraction

We used a commercial tool called Understand 4.0[49] (created by SciTools) to compute the class-level and method-level software metrics. Along with extracting the metrics,[50] the tool also extracts the value of the metric. The metrics extracted by Understand tool are listed in SciTools.[50] We first created a project in Understand 4.0 for every version considered in Table 2 and ran a scheduler to extract the specified metrics (both class level and method level). This process took almost 30 s for each project version. For each system version under study, we generated a separate CSV (comma-separated values) file containing the method-level metrics for every method in that version. After that, we separated the vulnerable methods and their metrics from this CSV file and placed them in another CSV file. We also extracted the metrics from the non-vulnerable versions where the vulnerabilities have been fixed. We followed the same procedure to collect class-level metrics for the vulnerable and non-vulnerable classes in each system under study. An experimental package consisting of the dataset prepared by the authors for this research has been made available for use by the interested researchers.[51]

4.5 | Determining significant metrics using statistical test

As our collected software metrics have numerical values (which are generated by Understand 4.0 tool), for each metric, we can determine if the metric's values in vulnerable and non-vulnerable classes have equal means or not. If the mean of the values for a metric in vulnerable classes is significantly different from the mean of values of that metric in non-vulnerable classes, it signifies that the metric may be a strong candidate as one of the features for the vulnerability prediction model using supervised ML algorithms.

First, we conducted the Kolmogorov–Smirnov test (normality test)[52] using Statistical Package for the Social Sciences (SPSS)[53] and divided the metrics into two categories: metrics that follow a normal distribution and metrics that do not. After that, we performed the Mann–Whitney *U* test (for metrics not following normal distribution) and Welch's test (for metrics following normal distribution) to identify the significant set of metrics for each system under study. The Mann–Whitney *U* test is used to compare differences between two independent groups when the dependent variable is either ordinal or continuous, but not normally distributed.[54] Welch's test[55] for unequal variances, also called Welch's *t* test, is a modification of a Student's *t* test and is used to verify if two sample means are significantly different. Then, we considered the metrics for which means were significantly different (p value <0.05) in two groups (vulnerable vs. non-vulnerable). It should be noted that, for different systems, the set of significant metrics might be different. Hence, we first identified the set of significant metrics for each system under study separately. Next, we considered the common set of metrics for all systems under study.

4.6 | Computing effect size

We also computed effect size as p value is not enough to interpret how substantial the effect of vulnerability on the software metrics is. Effect size is a quantitative measure of the magnitude of the experimental effect. The larger the effect size, the stronger the relationship between two variables. Cohen's d is an appropriate effect size for the comparison between two means if the data follow the normal distribution.[56] It can be measured using the following formula:

effect size =

$$\frac{\left(Mean_{Non-vulnerable} - Mean_{vulnerable}\right)}{\sqrt{\frac{\left(N_{Non-vulnerable}-1\right)\times SD^2_{Non-vulnerable} + \left(N_{vulnerable}-1\right)\times SD^2_{vulnerable}}{N_{Non-vulnerable}+N_{vulnerable}-2}}}. \quad (2)$$

In the above equation, SD refers to the standard deviation and N is the sample size. For the metrics not following normal distribution, we applied Cliff's delta to measure effect sizes. Cliff's delta measures how often one value in one distribution is higher than the values in the second distribution and it does not require any assumptions about the shape or spread of the two distributions.[57] We followed the approach described in https://www.real-statistics.com/non-parametric-tests/mann-whitney-test/cliffs-delta/ for calculating Cliff's Delta using Microsoft Excel.

4.7 | Vulnerability prediction

In our study, we identified a set of significant class-level and method-level metrics as features from the program components (i.e., class or method) of the four Java-based software systems. There are two separate groups of components for each project, *vulnerable* and *non-vulnerable*. We created labeled data (marked as vulnerable or non-vulnerable) and then trained the machine so that it can classify any component as vulnerable based on its learning algorithm. Supervised ML can help predict vulnerable components based on the values of the features. We applied two supervised learning techniques (SVM and LR) as they have been successfully used in the previous studies.[15] In addition, SVMs can efficiently perform a non-linear classification by implicitly mapping the inputs into high-dimensional feature spaces.[58] SVM is a robust and powerful supervised learning method that can best segregate two classes with a right hyperplane. LR is a class of regression where the independent variable is used to predict the dependent variable.[59] We chose LR as we have a binary or dichotomous dependent variable (vulnerable or non-vulnerable), and it does not need to assume linear relationship between the dependent and independent variable.[59] As in this study, we do not aim to find the best ML algorithm for vulnerability prediction, we chose these two algorithms as they seemed to be appropriate for our problem. Considering two algorithms can help us to see if there is any major discrepancy between the results found in two algorithms.

4.7.1 | Tool selection

Waikato Environment for Knowledge Analysis (WEKA) is a popular, open source toolkit for ML and data mining.[60] We used WEKA 3.8 for our study. The parameters for each of the techniques were initialized using the default settings for WEKA.

4.7.2 | Data balancing

As our dataset was not balanced, we needed to create a balanced dataset having the same number of vulnerable and non-vulnerable classes or methods. In this study, we applied the *ClassBalancer* filter in WEKA 3.8.[61] This filter reweighs the instances in the data so that each method has the same total weight. This filter changes only the weight of the first batch of the data, which is used for training the model (90% in 10-fold cross validation); *the testing dataset (10% in 10-fold cross validation) is not re-weighted by this balancer.*

4.7.3 | Data analysis

We conducted experiments using two ML algorithms (SVM and LR) for all the projects under study. In each case, we applied 10-fold cross validation to ensure that the trained model will work accurately for an unknown dataset.[26] For every system considered in this study, we created one .arff file (WEKA file format) for WEKA. The file contains the metric values of all the classes/methods collected from the history of the system. If any class is vulnerable, the last column value will be 1; otherwise, it will be 0. In this way, we created four .arff files for four systems under study. Then, we used WEKA tool for training the machine using 10-fold cross validation for each system. WEKA generates the values of performance measures (recall, precision, FP rate, F-measure, and ROC).

5 | RESULTS

This section describes the results obtained from analyzing the data collected during this study. This section is organized around the four RQs described in Section 4.

5.1 | RQ1: What software metrics are best suited for vulnerability prediction at class level of Java-based software systems?

In order to answer this RQ, we first used the Understand 4.0 tool to extract the class-level metrics for each of the four systems separately (Table 2). As an example, when supplied with the known vulnerable and non-vulnerable classes for Apache CXF system, Understand 4.0 tool identified 39 relevant software metrics along with their values in CXF's classes (third column in Table 3). Separate values for each metric were generated for each vulnerable class versus non-vulnerable class. Thereafter, for each identified class-level metric in CXF system, we ran Welch's or

Mann–Whitney test (based on normality test) to compare the mean of the metric's values in vulnerable versions versus the mean of the metric's values in the non-vulnerable version.

Table 3 shows the results of the test. In Table 3, each metric in gray background signifies that the result of the *t* test for the metric was at $p>0.05$. This means that the mean of all values of the particular metric in vulnerable classes was *not* significantly different than the mean of all values of that particular metric in non-vulnerable classes. That is, a metric with gray background is *not* a potentially important vulnerability predictor for the system under study. Consequently, this means that the rest of the metrics listed in Table 3 were found to be potentially important/useful vulnerability predictor for the system under study. As an example, for Apache CXF system, out of the 39 initially identified metrics, we found that 30 metrics were useful/significant from a vulnerability prediction standpoint (examples of such metrics include AvgCyclomatic, AvgCyclomaticModified, etc., as can be seen in the third column of Table 3). As *p*value is not sufficient to interpret how substantial the effect of the metric on vulnerability prediction is, we also identified the effect sizes for each metric and shown them in parentheses (Table 3).

We performed the significance test (Mann–Whitney or Welch's *t* test) for all four systems and identified significant vulnerability predictor metrics for each system. As our goal is to propose a set of class-level metrics that are generalizable for all (or most) software systems, we have selected a set of common metrics that proved to be significant/useful as vulnerability predictors for all four systems under study (Tomcat-6, Tomcat-7, CXF, and Stanford). In the next paragraph, we list the set of 22 class-level metrics that have been found to be common statistically significant vulnerability predictors across all four systems.

We found that *AvgCyclomatic*, *AvgCyclomaticModified*, *AvgCyclomaticStrict*, *CountClassCoupled*, *CountDeclMethodPrivat*, *CountLine*, *CountLineBlank*, *CountLineCode*, *CountLineCodeDecl*, *CountLineCodeExe*, *CountLineComment*, *CountSemicolon*, *CountStmt*, *CountStmtDecl*, *CountStmtEx*, *MaxCyclomatic*, *MaxCyclomaticModified*, *MaxCyclomaticStrict*, *MaxNesting*, *SumCyclomatic*, *SumCyclomaticModified*, and *SumCyclomaticStrict* are the common class-level metrics (that showed statistically significant difference in vulnerable vs. non-vulnerable classes in all four systems).

5.2 | RQ2: What software metrics are best suited for vulnerability prediction at method level of Java-based software systems?

We replicated the same procedure (that was described in RQ1 for identifying the set of useful class-level metrics) to identify the set of common useful method-level metrics. For each of the four systems, we first procured the known vulnerable methods (reported in a particular version). Next, for each system separately, we supplied the methods to the Understand tool. The tool provided us with the values of the method-level metrics for both vulnerable and non-vulnerable methods. Next, we ran Mann–Whitney or Welch's test (based on normality distribution) to identify those metrics for which the values were significantly different in the vulnerable methods from the values in the non-vulnerable methods for every system. Finally, we selected a set of 18 significant/useful metrics that have been found to be common across all four systems.

We found that *CountInput*, *CountLine*, *CountLineBlank*, *CountLineCode*, *CountLineCodeDecl*, *CountLineCodeExe*, *CountLineComment*, *CountOutput*, *CountPathLog*, *CountSemicolon*, *CountStmt*, *CountStmtDecl*, *CountStmtExe*, *Cyclomatic*, *CyclomaticModified*, *CyclomaticStrict*, *Essential*, and *MaxNesting* are the common significant method-level metrics (that showed statistically significant difference in vulnerable and non-vulnerable methods of four systems under study).

5.3 | RQ3: Can the software metrics identified in RQ1 be employed by developers to predict vulnerability-prone classes in Java-based software systems?

One of the primary goals of the current research is to provide software development teams with an initial set of class-level software metrics that can be used as a starter kit to build a feature set for training ML algorithms in vulnerability prediction. In RQ1, we built a set consisting of 22 class-level metrics that can be used for predicting vulnerable classes in all (or most) of the systems developed using Java. To that end, we trained two supervised ML algorithms (SVM and LR) using our proposed set of 22 class-level metrics as feature set. The performance measures achieved with the SVM and LR are shown in Tables 4 and 5, respectively.

The insights from comparing Tables 4 and 5 are as follows.

- Our proposed set of 22 common class-level metrics, when used as a feature set to train ML algorithms, showed moderate-to-high recall and precision (between 70% to 90%) in the case of all four systems (and for both SVM and LR algorithms).
- With respect to vulnerability prediction at class level, the SVM algorithm performed better than the LR algorithm in terms of recall (which is a very important measure from vulnerability prediction standpoint as high recall rate means that the vulnerability prediction model will help in ensuring that fewer vulnerabilities will remain undetected).

TABLE 3 Results of Mann–Whitney and Welch's test at Java class level

Tomcat-6	Tomcat-7	Apache CXF	Stanford SecuriBench
AvgCyclomatic (0.72)	AvgCyclomatic (0.74)	AvgCyclomatic (0.89)	AvgCyclomatic (0.40)
AvgCyclomaticModified (0.73)	AvgCyclomaticModified (0.73)	AvgCyclomaticModified (0.89)	AvgCyclomaticModified (0.40)
AvgCyclomaticStrict (0.72)	AvgCyclomaticStrict (0.73)	AvgCyclomaticStrict (0.88)	AvgCyclomaticStrict (0.38)
AvgEssential (0.79)	AvgEssential (0.74)	AvgEssential (0.76)	*AvgEssential (0.1)
CountClassBase (0.20)	*CountClassBase (0.3)	*CountClassBase (0.1)	*CountClassBase (0.3)
CountClassCoupled (0.75)	CountClassCoupled (0.78)	CountClassCoupled (0.68)	CountClassCoupled (0.56)
*CountClassDerived (0.1)	*CountClassDerived (0.1)	*CountClassDerived (0.02)	*CountClassDerived (0.003)
*CountDeclClassMethod (0.1)	CountDeclClassVariable (0.22)	*CountDeclClassMethod (0.84)	CountDeclClassVariable (0.50)
CountDeclClassVariable (0.61)	CountDeclClassMethod (0.59)	*CountDeclClassVariable (0.2)	*CountDeclClassMethod (0.1)
CountDeclInstanceMethod (0.64)	CountDeclInstanceMethod (0.71)	*CountDeclInstanceMethod (0.3)	*CountDeclInstanceMethod (0.2)
CountDeclInstanceVariable (0.65)	CountDeclInstanceVariable (0.7)	*CountDeclInstanceVariable (0.1)	*CountDeclInstanceVariable (0.008)
CountDeclMethod (0.69)	CountDeclMethod (0.74)	*CountDeclMethodDefault (0.2)	*CountDeclMethod (0.2)
CountDeclMethodAll (0.49)	CountDeclMethodAll (0.55)	CountDeclMethod (0.30)	*CountDeclMethodAll (0.13)
*CountDeclMethodDefault (0.13)	*CountDeclMethodDefault (0.2)	CountDeclMethodAll (0.43)	*CountDeclMethodDefault (0.3)
CountDeclMethodPrivate (0.49)	CountDeclMethodPrivate (0.42)	CountDeclMethodPrivate (0.46)	*CountDeclMethodPrivate (0.53)
CountDeclMethodProtected (0.59)	CountDeclMethodProtected (0.62)	*CountDeclMethodProtected (0.5)	*CountDeclMethodProtected (0.3)
CountDeclMethodPublic (0.51)	CountDeclMethodPublic (0.57)	*CountDeclMethodPublic (0.2)	*CountDeclMethodPublic (0.03)
CountLine (0.81)	CountLine (0.84)	CountLine (0.61)	CountLine (0.4)
CountLineBlank (0.82)	CountLineBlank (0.86)	CountLineBlank (0.56)	CountLineBlank (0.4)
CountLineCode (0.80)	CountLineCode (0.83)	CountLineCode (0.66)	CountLineCode (0.43)
CountLineCodeDecl (0.78)	CountLineCodeDecl (0.79)	CountLineCodeDecl (0.55)	CountLineCodeDecl (0.4)
CountLineCodeExe (0.81)	CountLineCodeExe (0.84)	CountLineCodeExe (0.68)	CountLineCodeExe (0.48)
CountLineComment (0.77)	CountLineComment (0.8)	CountLineComment (0.24)	CountLineComment (0.3)
CountSemicolon (0.80)	CountSemicolon (0.83)	CountSemicolon (0.62)	CountSemicolon (0.43)
CountStmt (0.80)	CountStmt (0.83)	CountStmt (0.66)	CountStmt (0.40)
CountStmtDecl (0.77)	CountStmtDecl (0.79)	CountStmtDecl (0.52)	CountStmtDecl (0.4)
CountStmtExe (0.80)	CountStmtExe (0.84)	CountStmtExe (0.71)	CountStmtExe (0.42)
MaxCyclomatic (0.79)	MaxCyclomatic (0.8)	MaxCyclomatic (0.85)	MaxCyclomatic (0.45)
MaxCyclomaticModified (0.79)	MaxCyclomaticModified (0.81)	MaxCyclomaticModified (0.85)	MaxCyclomaticModified (0.45)
MaxCyclomaticStrict (0.78)	MaxCyclomaticStrict (0.8)	MaxCyclomaticStrict (0.85)	MaxCyclomaticStrict (0.40)
MaxEssential (0.75)	MaxEssential (0.73)	MaxEssential (0.73)	*MaxEssential (0.3)
*MaxInheritanceTree (0.1)	*MaxInheritanceTree (0.4)	*MaxInheritanceTree (0.3)	*MaxInheritanceTree (0.5)
MaxNesting (0.72)	MaxNesting (0.74)	MaxNesting (0.79)	MaxNesting (0.58)
PercentLackOfCohesion (0.63)	PercentLackOfCohesion (0.57)	PercentLackOfCohesion (0.25)	*PercentLackOfCohesion (0.2)
RatioCommentToCode (0.24)	RatioCommentToCode (0.4)	RatioCommentToCode (-0.35)	*RatioCommentToCode (0.3)
SumCyclomatic (0.80)	SumCyclomatic (0.84)	SumCyclomatic (0.74)	SumCyclomatic (0.3)
SumCyclomaticModified (0.80)	SumCyclomaticModified (0.84)	SumCyclomaticModified (0.74)	SumCyclomaticModified (0.3)
SumCyclomaticStrict (0.80)	SumCyclomaticStrict (0.83)	SumCyclomaticStrict (0.75)	SumCyclomaticStrict (0.3)
SumEssential (0.80)	SumEssential (0.8)	SumEssential (0.62)	*SumEssential(0.2)

Note: The metrics with asterisk (*) follow a normal distribution, and for them, we conducted Welch's test. For other metrics, we conducted Mann–Whitney *U* test. For the gray-highlighted metrics, we found *p* value greater than 0.05 (i.e., these metrics were found *not useful* from a vulnerability prediction perspective). For remaining metrics, $p<.05$ (i.e., useful metrics for vulnerability prediction). The value mentioned in parenthesis is the effect size for each metric.

TABLE 4 Performance measures of the machine learning classifier (support vector machine) using the common class-level metrics (as in research question 1) as features

System	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	14.6	83.9	75.9	79.7	80.7
Tomcat-7	12.2	87.3	83.9	85.6	85.9
Apache CXF	10.7	87.9	77.4	82.3	83.4
Stanford	24.3	75.5	74.7	75.1	75.2

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

TABLE 5 Performance measures of the machine learning classifier (logistic regression) using the common class-level metrics (as in research question 1) as features

System	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	12.6	85.1	72.2	78.1	87
Tomcat-7	10	88.7	78.6	83.4	88.1
Apache CXF	8.9	89.3	74.2	81	92.6
Stanford	21.9	76.1	69.6	72.7	80.2

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

Tables 4 and 5 also present the other performance measures including FP rate, precision, F-measure, and ROC while using the proposed common class-level metrics as features for every system under study. In SVM, the lowest recall is 74.7%, and in LR, the lowest recall is 69.6%. The FP rate is less than 24.3% in SVM and less than 21.9% in LR. In SVM, the lowest precision is 75.5%, and in LR, it is 76.1%. The lowest F-measure in LR is 72.7%, whereas in SVM, it is 75.1%.

5.4 | RQ4: Can the software metrics identified in RQ2 be employed by developers to predict vulnerability-prone methods in Java-based software systems?

In order to evaluate the performance of the set of 18 method-level metrics (proposed in RQ2) as features, we trained the SVM and LR algorithms using the proposed set of method-level metrics as feature set. The performance measures achieved by SVM and LR algorithm are shown in Tables 6 and 7, respectively.

The following are insights from Tables 6 and 7.

- Our proposed set of method-level metrics, when used as a feature set to train ML algorithms, achieved moderate-to-high recall and precision (between 65% to 90%). A significant result was that the recall rate of SVM (Table 6) is as high as 90.8% (for Stanford system). This result of our proposed set of metrics motivates further investigation.
- Similar to the results obtained for class-level metrics, the SVM algorithm again outperformed the LR algorithm (especially for recall). Except for Tomcat-7 system (where SVM's recall rate was lower than LR's), for the rest of the three systems, SVM achieved higher recall rates.

Tables 6 and 7 also present the other performance measures including FP rate, precision, F-measure, and ROC while using method-level metrics as features for every system under study. With respect to SVM (Table 6), the lowest recall rate is 69.3%, and in LR (Table 7), the lowest recall

TABLE 6 Performance measures of the machine learning classifier (support vector machine) using the common method-level metrics (as in research question 2) as features

System (%)	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	17.4	80.4	71	75.4	76.8
Tomcat-7	15.7	81.5	69.3	74.9	76.8
Apache CXF	12.6	85.3	73.3	78.9	80.4
Stanford	21.4	80.9	90.8	85.6	84.7

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

TABLE 7 Performance measures of the machine learning classifier (logistic regression) using the common method-level metrics (as in research question 2) as features

System	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	14.1	82.6	66.9	74	85.2
Tomcat-7	14.6	83	71.3	76.7	84.8
Apache CXF	10.2	86.7	66.7	75.4	87.3
Stanford	15.6	84.6	85.5	85	89.2

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

is 66.7%. The FP rate is less than 21.4% in SVM and less than 15.6% in LR in the case of all four systems. With SVM, the lowest precision is 80.4%, and with LR, it is 82.6%. The lowest F-measure with LR is 74%, whereas with SVM, it is 74.9%.

A discussion on the results is provided in the next section.

6 | DISCUSSION

This section provides a discussion on the implications of some of the major results.

6.1 | Algorithm selection: SVM versus LR

At *class level*, a comparison between the performance of the two ML algorithms suggested that for all four systems, the SVM algorithm consistently yielded better recall rate than LR (Table 4 vs. Table 5). In the case of our research, we focus more on getting higher recall rate as a high recall rate suggests that a high percentage of 'actually vulnerable classes' has been identified as vulnerable by the algorithm. The LR algorithm performed better with respect to other performance measures like precision and F-measure. F-measure (a weighted average of precision and recall) can measure how precise and robust the classifier is. As can be seen in Tables 4 and 5, the F-measure values for LR are consistently better than SVM (for all four systems). At *method level*, the SVM algorithm achieved better recall than the LR algorithm for Tomcat-6, Apache CXF, and Stanford systems (Table 6 vs. Table 7).

Overall, these results indicate that SVM algorithm is a more appropriate choice for vulnerability prediction when using class-level or method-level software metrics as features and when recall rate is important. In the case of precision, LR performs better than SVM. It can also be noted that the difference in the results of SVM and LR is not significant, which indicates that the result is almost identical in two ML algorithms, and therefore, developers can use one of them to train their machine.

6.2 | Usefulness of the proposed set of software metrics

According to Sections 5.3 and 5.4, when ML algorithms are trained with our proposed set of class-level or method-level metrics, the algorithms can classify a class/method as vulnerable or non-vulnerable with high recall and precision.

Furthermore, we performed an additional analysis where we obtained the performance of SVM algorithm when it was trained by using all the significant metrics for an individual system instead of using our proposed set of 22 common class-level metrics. As an example, for Apache CXF system, we identified 30 useful metrics (third column of Table 3). So, we used these 30 metrics as feature set. We found that the recall in SVM was 74.2% as shown in Table 8. This is lower than the recall value 77.4%, which we obtained when we used our proposed set of 22 class-level metrics (Table 4). For two systems (Tomcat-7 and Apache CXF), the recall values were higher when using our proposed set of 22 metrics as features. For the other two systems (Tomcat-6 and Stanford), the recall values were higher when the feature set consisted of all significant class-level metrics specific to the system. However, in the case of all four systems, the difference in the recall rate was not significant when using our proposed set of metrics versus when using all system-specific metrics. *This finding is significant as it indicates that our proposed set of 22 class-level metrics can be successfully employed as feature set (in SVM) to predict vulnerable classes in a Java-based system.* Similar results were also obtained for LR algorithm. When comparing the results in Table 5 versus Table 9, it can be inferred that there was no significant difference between the recall and precision rates when using our proposed set of metrics versus when using system-specific metrics.

According to Table 3, there are some class-level metrics that do not contribute to training the ML algorithm as their distributions do not have statistically significant difference in the vulnerable and non-vulnerable classes. For example, *CountDeclInstanceMethod* is the number of instance methods (methods defined in a class that are only accessible through an object of that class) and *CountDeclInstanceVariable* is the number of

TABLE 8 Performance measures of the system-specific significant class-level metrics using support vector machine

System	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	13.2	86	81.5	83.7	84.1
Tomcat-7	11.3	87.7	80.4	83.9	84.5
Apache CXF	7.7	90.6	74.2	81.6	93.7
Stanford	24.9	76.2	79.7	78	77.4

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

TABLE 9 Performance measures of the system-specific significant class-level metrics using logistic regression

System	FP rate (%)	Precision (%)	Recall (%)	F-measure (%)	ROC (%)
Tomcat-6	10.5	87.1	70.4	77.8	82.8
Tomcat-7	9.5	88.5	73.2	80.1	85.6
Apache CXF	7.7	90.6	74.2	81.6	93.7
Stanford	19.5	78.7	72.2	75.3	79.4

Abbreviations: FP, false positive; ROC, receiver operating characteristic.

variables that are only accessible through an object of a class. These metrics are not counted in our set of metrics because they do not show statistically significant difference in vulnerable and non-vulnerable data. As instance methods and instance variables are only accessible through an object of that class, they are less likely to cause vulnerable code. On the other hand, *MaxNesting* has statistically significant difference between vulnerable and non-vulnerable data both at class level and method level. *MaxNesting* is the maximum nesting level of control constructs (if, while, for, switch, etc.) in the function. This metric is highly likely to make code vulnerable because the higher the number of nesting levels, the higher the chance of making code complex and vulnerable.

Overall, these results indicate that our proposed set of metrics can be used by development teams as starting point for building their own feature set for training ML algorithms to predict vulnerable classes and methods. The developers can first compute the values of our proposed set of metrics for their vulnerable and non-vulnerable components found in their earlier versions and then train their machine with the extracted values (Figure 5). A new class or method can be predicted as either vulnerable or non-vulnerable based on the values of these metrics computed for that class or method (Figure 5). As our proposed generic set of metrics can be applied for any system, the developers will only need to extract these values from their vulnerability history and then train the machine with the extracted values and finally classify the test code as vulnerable or non-vulnerable as shown in Figure 5.

6.3 | Analysis of the performance measures

According to Tables 4-6, and 7, recall rates are higher in SVM compared with the recall rates reported in LR. These measures in the tables are the average values at all points of the ROC curve as reported by WEKA. They can actually vary based on the threshold point on the ROC curve. Therefore, in the next subsection, we have analyzed the results in terms of the ROC curve. The FP rates of the Stanford dataset in Tables 4 and 5 are higher than other systems. However, precision, recall, F-measure, and ROC are lower in the Stanford dataset. Table 3 presents that as compared with other systems, the Stanford dataset has more metrics that do not have statistically significant difference between vulnerable classes and the non-vulnerable classes. The other metrics showing statistically significant difference between vulnerable classes and the non-vulnerable classes do not have large effect sizes. Therefore, we can infer that the class-level metrics did not work well as features of ML techniques in the

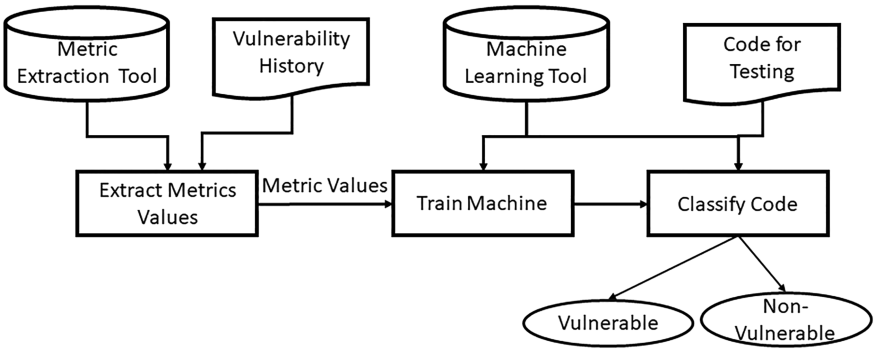


FIGURE 5 Flow chart for machine learning-based vulnerability prediction

case of vulnerability prediction of the Stanford dataset. Similarly, according to Tables 6and 7, FP rates of method-level metrics are also high in the Stanford dataset. On the other hand, class-level metrics showed high precision (above 80%) and high recall (above 70%) in Tomcat-6, Tomcat-7, and Apache CXF (Tables 4 and 5). In Table 3, we can see that most of the class-level metrics that show statistically significant difference between vulnerable classes and the non-vulnerable classes in Tomcat-6, Tomcat-7, and Apache CXF have large effect sizes (above 0.5). The class-level metrics that we proposed in our study have effect sizes more than 0.7 (Table 3).

6.4 | Trade-off between recall and FP rate

Generally, the expectation is to achieve a higher recall with a lower FP rate when predicting vulnerable code. But the FP rate usually increases with the increase of recall.[28] We plot a graph of TP rate versus FP rate in order to explore the trade-off between recall and the FP rate. Such plots are known as ROC curves, which are used to visualize the performance of a predictor in detecting the true group (in our case, vulnerable classes or methods). Figure 6 presents the ROC curves using class-level metrics in LR for four systems under study. For three out of four systems (Figure 6), class-level metrics can correctly identify about 60% of the vulnerable classes while keeping the FP rate below 10% and about 80% of the vulnerable classes when the FP rate is kept below 20%. In the case of method-level metrics, 60% of the vulnerable methods are detected keeping FP rate at 10% and more than 75% vulnerable methods are detected when keeping FP rate at 20% as shown in Figure 7.

Overall, we recommend that development teams should try to achieve high recall rates because this way, a lower number of vulnerabilities will go undetected. In doing so, they may have to sacrifice the FP rate, which will mean that more non-vulnerable class/methods will be predicted as vulnerable. But if they analyze the ROC for the metrics, they may decide the point at which the TP rate and FP rate are at their optimum level.

6.5 | Impact of effect size

We mentioned the effect size for every metric in Table 3 (in parentheses). In our proposed set of metrics, we found that only one metric (*CountLineComment*) has effect size less than 0.3 in Apache CXF. All other metrics considered in the study (Table 3) have effect size greater than 0.5

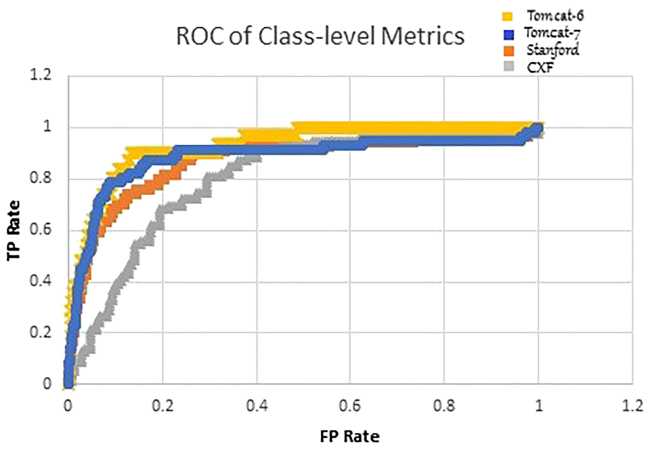


FIGURE 6 ROC for class-level metrics in logistic regression. FP, false positive; ROC, receiver operating characteristic; TP, true positive

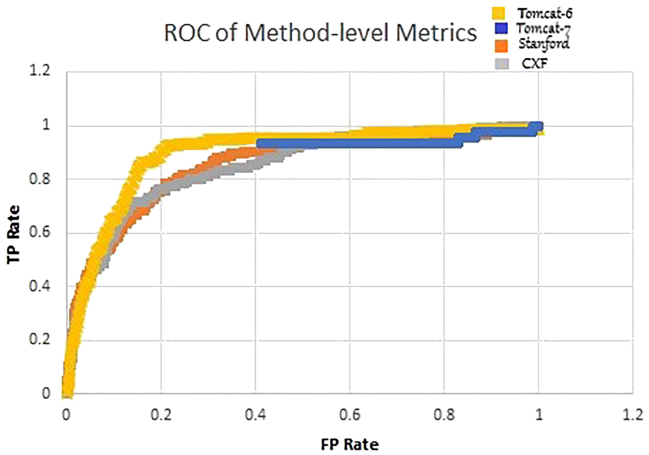


FIGURE 7 ROC for method-level metrics in logistic regression. FP, false positive; ROC, receiver operating characteristic; TP, true positive

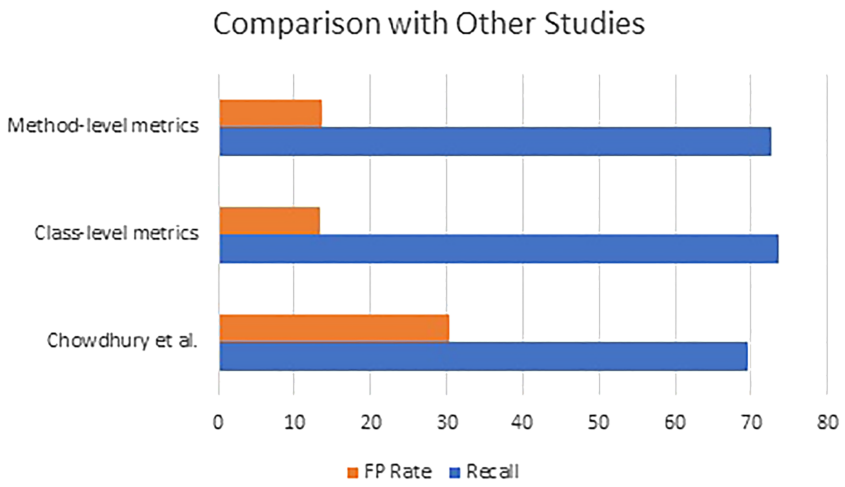


FIGURE 8 Comparison with other studies. FP, false positive

(with most of them having effect size greater than 0.7) in Tomcat-6, Tomcat-7, and Apache CXF. With a Cohen's *d* of 0.7, 75.8% data of one group will be above the mean of another group, 72.6% of the two groups will overlap, and there is a 69% chance that a person picked at random from one group will have a higher score than a person picked at random from another group (probability of superiority).[‖] The effect sizes of the metrics in the Stanford dataset were between 0.3 to 0.6.

6.6 | Comparison with similar research work

Chowdhury and Zulkernine[16] showed that complexity, coupling, and cohesion can be effective metrics for vulnerability prediction in early stages of software development. They did not distinguish between class- and method-level software metrics in their study. The study was performed on the source files (files with .c, .cpp, .java, and .h extensions) of different releases of Mozilla Firefox. In our case, we did the study for Java-based systems and extracted class- and method-level metrics. We compared the average FP rate and recall of their study with those of our study using LR in Figure 8. Predictions for class-level metrics resulted in a lower FP rate (13.35% vs. 30.3%) and higher recall rate (73.7% vs. 69.5%) compared with the study by Chowdhury and Zulkernine.[16] Predictions for method-level metrics resulted in a lower FP rate (13.6% vs. 30.3%) and higher recall rate (72.6% vs. 69.5%) compared with the study by Chowdhury and Zulkernine.[16]

We compared our work with Chowdhury and Zulkernine's work because they covered most of the object-oriented software metrics.[16] Although their experiments were based on C++ project, object-oriented metrics can also be applied for Java systems, and therefore, in our study, we used all of those metrics. As our study is based on Java projects, we did not use their systems. By comparing recall rate found in our experiment with the recall rate found in their work, we can show that considering metrics at lower granularity level (class or method) than at file level could give better performance in vulnerability prediction. Moreover, separating the class-level metrics from the method-level metrics can improve the prediction performance. Although the dataset and feature set used by Chowdhury and Zulkernine[16] are different than the dataset and features used in this paper, we presented the comparison to promote a future research on using metrics for vulnerability prediction at lower granularity levels (class or method) as well as separating the metrics based on their granularity and then assessing the overall performance of the ML classifier.

7 | VALIDITY THREATS

7.1 | Internal validity

This threat refers to the possibility of having confounded/unanticipated relationships among variables. In this study, we are not claiming causation. We do not claim that these metrics are the direct cause of the vulnerabilities found in the code components (i.e., methods, classes). Instead, we recommend rigorous testing of those methods/classes that are predicted vulnerable by using our approach. It can be assumed that such methods/classes contain properties that can be related to vulnerable code.

[‖] https://rpsychologist.com/d3/cohend/

7.2 | Construct validity

We analyzed the performance of class- and method-level metrics as vulnerability predictors. Although our selected metrics represent the characteristics of a Java method or a class, they may not capture all properties. In addition, we collected the vulnerable classes/methods from the Apache vulnerability reports (reported by development teams) and also by using a static analysis tool. One threat comes from the fact that these reports and tools may miss vulnerabilities (e.g., undiscovered vulnerabilities) that may still exist in the non-vulnerable version.

7.3 | External validity

The experiment was conducted on two Apache projects and two J2EE web applications with known vulnerabilities. Although our results were pretty consistent across these four systems, there is a need to verify the results with other systems. As three projects out of four are selected from Apache, they may share some common characteristics that can have an impact on the identification of vulnerable classes and methods. We intend to address this threat in our future studies by analyzing prediction performance on more systems (Java vs. non-Java and open source vs. industrial data) selected from different domains and organizations. Furthermore, for replication proposes, we have provided the complete dataset that was prepared for this research here.[51]

7.4 | Conclusion validity

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. We proposed sets of class-level and method-level software metrics that are significantly related to vulnerable classes and methods, respectively, based on our statistical inferences. Although we conducted normality test and also mentioned effect sizes to achieve better results, we will need to do further investigation on the effect sizes as well as the relation between individual metric and vulnerable component (class or method).

8 | CONCLUSION

In this study, we proposed a set of class- and method-level software metrics that perform well in terms of precision and recall for all (or most) Java-based systems (when used as features for predicting vulnerable methods/classes). We analyzed the performance of these metrics so that developers can choose the appropriate metrics for vulnerability prediction based on the granularity level (class/method) adequate for their project and need. We found that our proposed set of class and method-level metrics showed similar (and in some cases, better) performance compared with the performance of system-specific significant metrics (when used for vulnerability prediction). This signifies that development teams can use our proposed set of class- or method-level metrics for vulnerability prediction for any Java project. Once a method or class is predicted as vulnerable by our proposed metrics-based prediction model, developers can be more cautious when using that method or class (during later releases). Furthermore, after a class/method is predicted as vulnerable, more rigorous/focused testing can be performed. This will ensure secure software evolution by giving teams the ability to perform targeted testing on potentially vulnerable classes/methods (thus supporting risk prevention and risk reduction).

In the future, we will extend our work by redefining the metrics so that they are more security specific (we expect that this will improve vulnerability prediction accuracy). We will also investigate the impact of these metrics on different types of systems (open source vs. industrial or Java vs. non-Java). Furthermore, we intend to extend the current research by performing analyses to determine whether some metrics are associated to particular type of vulnerabilities (as an example, determining the type of metrics that are associated with SQL injection vulnerability type).

ORCID

Vaibhav Anu https://orcid.org/0000-0001-8104-4942

REFERENCES

1. Graff MG, Van Wyk KR. *Secure Coding: Principles and Practices*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.; 2003.
2. Wheeler DA. Secure Programming for Linux and Unix Howto; 2003.
3. Seacord R. *Secure Coding in C and C++*. 1st ed. Upper Saddle river, NJ, USA: Addison-Wesley Professional; 2005.
4. Howard M, Leblanc DE. *Writing Secure Code*. 2nd ed. Redmond, WA, USA: Microsoft Press; 2002.
5. FindBugs IBM. Why and how to use findbugs. https://www.ibm.com/developerworks/java/library/j-findbug1/
6. Nadeem M, Williams BJ, Allen EB. High false positive detection of security vulnerabilities: a case study. In: Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). ACM; 2012; New York, NY, USA:359-360.

7. Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y. Evaluating static analysis defect warnings on production software. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07). ACM; 2007; New York, NY, USA:1-8.
8. Tripp O, Guarnieri S, Pistoia M, Aravkin A. Aletheia: improving the usability of static security analysis. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14). ACM; 2014; New York, NY, USA:762-774.
9. Zitser M, Lippmann R, Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12); 2004; Newport Beach, CA, USA:97-106.
10. Reynolds ZP, Jayanth AB, Koc U, Porter AA, Raje RR, Hill JH. Identifying and documenting false positive patterns generated by static code analysis tools. In: 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP); 2017; Buenos Aires, Argentina:55-61.
11. Shin Y, Williams L. 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 (ESEM '08). ACM; 2008; NY, USA:315-317.
12. Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans Softw Eng*. 2011;37(6):772-787.
13. Shin Y, Williams L. Can traditional fault prediction models be used for vulnerability prediction? *Empir Softw Eng*. 2013;18(1):25-59.
14. Chowdhury I, Chan B, Chowdhury MZ. Security metrics for source code structures. In: Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems (SESS '08). ACM; 2008; NY, USA:57-64.
15. Chowdhury I, Zulkernine M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10). ACM; 2010; NY, USA:1963-1969.
16. Chowdhury I, Zulkernine M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J Syst Archit*. 2011;57(3):294-313.
17. Sultana KZ, Deo A, Williams BJ. A preliminary study examining relationships between nano-patterns and software security vulnerabilities. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Vol. 1; 2016; Atlanta, GA, USA:257-262.
18. Sultana KZ, Deo A, Williams BJ. Correlation analysis among java nano-patterns and software vulnerabilities. In: 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE); 2017; Singapore:69-76.
19. Gil JY, Maman I. Micro patterns in java code. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05). ACM; 2005; San Diego, CA, USA:97-116.
20. Singer J, Brown G, Luján M, Pocock A, Yiapanis P. Fundamental nano-patterns to characterize and classify java methods. *Electron Notes Theoret Comput Sci*. 2010;253(7):191-204. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).
21. Sultana KZ, Williams BJ. Evaluating micro patterns and software metrics in vulnerability prediction. In: 2017 6th International Workshop on Software Mining (softwaremining)(SOFTWAREMINING); 2017; Urbana-Champaign, IL, USA:40-47.
22. Sultana KZ, Williams BJ, Bosu A. A comparison of nano-patterns vs. software metrics in vulnerability prediction. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC); 2018; Nara, Japan:355-364.
23. Voas J, Kuhn R. What happened to software metrics? *Computer*. 2017;50(5):88-98.
24. McCabe TJ. A complexity measure. *IEEE Trans Softw Eng*. 1976;2(4):308-320.
25. Harrison WA, Magel KI. A complexity measure based on nesting level. *SIGPLAN Not*. 1981;16(3):63-74.
26. Witten IH, Frank E. *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco, USA: Morgan Kaufmann; 2005.
27. Chinchor N. Muc-4 evaluation metrics. In: Proceedings of the 4th Conference on Message Understanding (MUC4 '92). Association for Computational Linguistics; 1992; Stroudsburg, PA, USA:22-29.
28. Moshtari S, Sami A. 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 (SAC '16); 2016; Pisa, Italy:1415-1421.
29. Fenton NE, Pfleeger SL. *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. Boston, MA, USA: PWS Publishing Co.; 1998.
30. Shin Y, Williams L. Is complexity really the enemy of software security? In: Proceedings of the 4th ACM Workshop on Quality of Protection (QoP '08). ACM; 2008; NY, USA:47-50.
31. Ghaffarian SM, Shahriari HR. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput Surv*. 2017;50(4):56:1-56:36.
32. Alves H, Fonseca B, Antunes N. Software metrics and security vulnerabilities: dataset and exploratory study. In: 2016 12th European Dependable Computing Conference (EDCC); 2016; Gothenburg, Sweden:37-44.
33. Zimmermann T, Nagappan N, Williams L. Searching for a needle in a haystack: predicting security vulnerabilities for windows vista. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10). IEEE Computer Society; 2010; DC, USA:421-428.
34. Younis A, Malaiya Y, Anderson C, Ray I. To fear or not to fear that is the question: code characteristics of a vulnerable function with an existing exploit. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16). ACM; 2016; NY, USA:97-104.
35. Walden J, Stuckman J, Scandariato R. Predicting vulnerable components: software metrics vs text mining. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering; 2014; Naples, Italy:23-33.
36. Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07). ACM; 2007; NY, USA:529-540.
37. Zhang Y, Lo D, Xia X, Xu B, Sun J, Li S. Combining software metrics and text features for vulnerable file prediction. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS); 2015; Gold Coast, QLD, Australia:40-49.
38. Scandariato R, Walden J, Hovsepyan A, Joosen W. Predicting vulnerable software components via text mining. *IEEE Trans Softw Eng*. 2014;40(10):993-1006.
39. Zhou Y, Sharma A. Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM; 2017; New York, NY, USA:914-919.
40. Li Z, Zou D, Xu S, et al. Vuldeepecker: A deep learning-based system for vulnerability detection. CoRR abs/1801.01681 (2018). arXiv:1801.01681; 2018.
41. Abunadi I, Alenezi M. Towards cross project vulnerability prediction in open source web applications. In: Proceedings of the the International Conference on Engineering & MIS 2015 (ICEMIS '15). ACM; 2015; New York, NY, USA:42:1-42:5.

42. Giger E, D'Ambros M, Pinzger M, Gall HC. Method-level bug prediction. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12). ACM; 2012; NY, USA:171-180.
43. SecuriBench S. SecuriBench program statistics. https://suif.stanford.edu/~livshits/securibench/stats.html
44. Livshits VB. Finding security errors in Java applications using lightweight static analysis. Work-in-Progress Report, Annual Computer Security Applications Conference; 2004; Tucson, AZ, USA.
45. Livshits VB, Lam MS. Finding security errors in Java programs with static analysis. In: Proceedings of the 14th Usenix Security Symposium; 2005; Seattle, WA, USA:271-286.
46. Sampaio L. Early security vulnerability detector—ESVD. https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/. Accessed: 2017-08-04; 2017.
47. Alonso C. Early vulnerability detection for supporting secure programming. http://docplayer.net/1619013-Early-vulnerability-detection-for-supporting-secure-programming.html. Accessed: 2017-08-04; 2015.
48. (ASF) ASF. ASF distribution directory. http://archive.apache.org/
49. SciTools. Understand. https://scitools.com/features/
50. SciTools. Understand. https://scitools.com/support/metrics_list/?metricGroup=complex
51. Sultana KZ, Anu V. Zakia et al vulnerability dataset. https://drive.google.com/drive/folders/1zWILeVMBXKSbiswUlwOOk2KRWrMShbeD
52. Tutorials SPSS. SPSS Kolmogorov–Smirnov test for normality. https://www.spss-tutorials.com/spss-kolmogorov-smirnov-test-for-normality/#kolmogorov-smirnov-normality-test-what-is-it
53. *IBM SPSS Statistics for Windows (Version: 25)*. Armonk, NY: IBM Corp.; 2017.
54. LAERD. Mann–Whitney U test using SPSS statistics.
55. StatisticsHowTo. Welch's test for unequal variances. http://www.statisticshowto.com/welchs-test-for-unequal-variances/
56. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*. New York, NY: Lawrence Erlbaum Associates; 1988.
57. Cliff N. Dominance statistics: ordinal analyses to answer ordinal questions; 1993.
58. Cortes C, Vapnik V. Support-vector networks. *Machine Learn*. 1995;20(3):273-297.
59. Hosmer DW, Lemeshow S. *Applied Logistic Regression*, Vol.398. New York, NY: John Wiley & Sons; 2000.
60. WEKA. Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka
61. Frank E. Classbalancer. http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html. Accessed: 2017-08-04; 2016.

How to cite this article: Sultana KZ, Anu V, Chong T-Y. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. *J Softw Evol Proc*. 2021;33:e2303. https://doi.org/10.1002/smr.2303