Predicting Security Vulnerabilities using Source Code Metrics

Sundarakrishnan Ganesh
Department of Computer Science and Media Technology
Linnaeus University
Vaxjo, Sweden
sg222wn@student.lnu.se

Tobias Ohlsson and Francis Palma
Department of Computer Science and Media Technology
Linnaeus University
Kalmar, Sweden
{tobias.olsson,francis.palma}@lnu.se

2021 Swedish Workshop on Data Science (SweDS) | 978-1-6654-1830-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/SweDS53855.2021.9638301

***Abstract*—Large open-source systems generate and operate on a plethora of sensitive enterprise data. Thus, security threats or vulnerabilities must not be present in open-source systems and must be resolved as early as possible in the development phases to avoid catastrophic consequences. One way to recognize security vulnerabilities is to predict them while developers write code to minimize costs and resources. This study examines the effectiveness of machine learning algorithms to predict potential security vulnerabilities by analyzing the source code of a system. We obtained the security vulnerabilities dataset from Apache Tomcat security reports for version 4.x to 10.x. We also collected the source code of Apache Tomcat 4.x to 10.x to compute 43 object-oriented metrics. We assessed four traditional supervised learning algorithms, i.e., Naive Bayes (NB), Decision Tree (DT), K-Nearest Neighbors (KNN), and Logistic Regression (LR), to understand their efficacy in predicting security vulnerabilities. We obtained the highest accuracy of 80.6% using the KNN. Thus, the KNN classifier was demonstrated to be the most effective of all the models we built. The DT classifier also performed well but under-performed when it came to multi-class classification.**

***Index Terms*—Prediction, Security Vulnerabilities, Machine Learning, Source Code, Software Metrics.**

I. Introduction

The system's security is profoundly conditioned on the quality of the underlying source code. Poor code quality frequently makes the code susceptive to third-party access, thus, drawing the system exposed to security threats. Security vulnerabilities identification is essential to implement and operate secured systems. A study revealed that a cyberattack in Equifax led to 143 million accounts being exposed, for which Equifax accused Apache Struts' open-source nature [1].

System security has been a subject of interest concerning vulnerability identification and avoidance. Previously, the revealing of vulnerabilities or bugs was conducted through manual peer reviews, which became tiresome as developers began producing a thousand lines of code. Thus, the code inspection tools, e.g., JSLint [2] and SonarQube [3], were introduced, helping to recognize vulnerabilities based on familiar code patterns. Nevertheless, the problem of revealing vulnerabilities as early as possible persists. Software testing methods, e.g., *fuzz testing* are demonstrated useful to identify vulnerabilities; however, at later steps of development cycle [4].

As outlined by NordVPN, the number of common vulnerabilities and exposures increased by 12,174 in 2020 with laborious patching efforts. Without patches installed, attackers may exploit the security vulnerabilities [5]. This study guides by recognizing security vulnerabilities in open-source systems like the Jakarta servlet engine: Apache Tomcat.

Large software projects generate a plethora of source code that makes it challenging for the developers to determine the source of a bug or security vulnerability. Moreover, security vulnerabilities are usually not revealed until the very end of development. Security vulnerabilities might boost the cost of quality assurance significantly. Therefore, this study strives to construct machine learning models that will suggest developers on the likelihood of vulnerability occurrences in the source code using static metrics. Thus, we answer two research questions – RQ1: *What is the best feature set for predicting security vulnerabilities?* and RQ2: *Can we employ machine learning to predict security vulnerabilities with high accuracy?*

The key contributions of this paper are: (1) a dataset consisting of security vulnerabilities and source code metrics for Apache Tomcat version 4.x until 10.x; (2) identification of the features set that contribute most to forecasting security vulnerabilities; (3) a best-performing machine learning model to predict the type and severity of the security vulnerabilities. We achieved the highest accuracy of 80.6% using the KNN classifier. The other three classifiers: Decision Tree (DT), Logistic Regression (LR), and Naive Bayes (NB), had an accuracy of 77.2%, 61.8%, and 71.7%, respectively. The DT classifier under-performed for multi-class classification.

In the rest of the paper: Section II provides some background information on security vulnerabilities and the metrics we use. Section III highlights the related work while Section IV describes our method. Section V presents experiments and results with a discussion in Section VI. Finally, Section VII concludes the paper and outlines future plans.

II. Background

Security vulnerabilities are identified via common vulnerabilities and exposures (CVEs) categories.

**This study was conducted as part of the bachelor degree project report by Sundarakrishnan Ganesh under the supervision of Francis Palma. The accepted report is available online on* http://lnu.diva-portal.org

A. Types of Security Vulnerabilities

Apache Tomcat has 22 unique security vulnerabilities as listed on the Apache website: *AJP Request Injection and potential Remote Code Execution*, *Arbitrary File deletion*, *Authentication weakness*, *Cross-site scripting*, *Denial of Service*, *DIGEST Authentication weakness*, *Frame injection in documentation Javadoc*, *HTTP/2 DoS*, *HTTP/2 request mix-up*, *Information disclosure*, *Limited directory traversal*, *Local Privilege Escalation*, *Multiple weaknesses in HTTP DIGEST authentication*, *Remote Code Execution*, *Remote Code Execution on Windows*, *Remote Code Execution via session persistence*, *Request Smuggling*, *Security Bypass*, *Session fixation*, *Session hi-jacking*, *Unrestricted Access to Global Resources*, and *WebSocket DoS* [6].

For example, *Denial of Service* is an attack in which a resource is rendered unavailable for its expected use. If a service is swamped with requests, regular users may not be able to send requests to it [7]. *Remote Code Execution* attack is written into an automated script that yields remote access to a potentially compromised system with administrative rights [8]. *Information Disclosure* occurs when a website inadvertently discloses sensitive information such as passwords or financial details to its users [9]. *AJP Request Injection and Potential Remote Code Execution* vulnerability transpires when the AJP (Apache Jserv Protocol) connections are entrusted without validation [6]. The *Arbitrary File Deletion* vulnerability transpires as a result of a *Directory Traversal* attack where, utilizing malformed input, an attacker deletes files available via the web application.The *Request Smuggling* vulnerability transpires when incoming HTTP requests are not validated, and thus, an attacker sends malformed HTTP requests to the application resulting in *information disclosure*, *security bypass*, or *directory traversal* attack [10].

B. Code Quality and Vulnerability Proneness

Chowdhury and Zulkernine [11] showed that code quality metrics such as coupling, cohesion, and complexity metrics are critical in the occurrence of a vulnerability in a system. Poor quality code can significantly increase the probability of an external attacker exploiting certain parts of the system. This relationship between code quality and vulnerabilities was recognized in 2001 when the *Code Red* exploited a buffer overflow attack on Microsoft's Internet Information Services.

C. Source Code Metrics

In this study, we consider 30 source code metrics from [12] that we use in predicting security vulnerabilities (Table I).

Table I: List of 30 Source Code Metrics from [12].

- **CBO** (Coupling Between Objects) is the number of dependencies in a class.
- **DIT** (Depth of Inheritance Tree) represents the number of fathers in a class.
- **totalFieldsQty**, **protectedFieldsQty**, **defaultFieldsQty**, **finalFieldsQty** metrics are the number of fields of various types.
- **totalMethodsQty** (Number of Methods) is the number of methods of various return types and scopes.
- **visibleMethodsQty** (Number of Visible Methods) is the number of non-private methods.
- **NoSI** (Number of Static Invocations) represents the number of invocations to static methods, but is limited to JDT resolved methods.
- **RFC** (Response for a Class) is the number of unique method invocations in a class, but fails with overloaded methods.
- **LOC** (Lines of Code) is the total lines of code in a class.
- **LCOM** (Lack of Cohesion of Methods) is the count of the number of method pairs whose similarity is zero.
- **TCC** (Tight Class Cohesion) is the cohesion of a class and results are delivered between 0 and 1.
- **LCC** (Loose Class Cohesion), same as TCC, but also includes the number of indirect connections between visible classes for the cohesion calculation.
- **returnQty**, **loopQty**, **comparisonsQty**, **tryCatchQty**, **paranthesizedExpsQty**, **stringLiteralsQty**, **numbersQty**, **mathOperationsQty**, and **variablesQty** represent the number of return statements, number of loops, comparison operators, try and catch statements used, parenthesized expressions, string literals, numeric literals, arithmetic symbols, and variables respectively.
- **maxNestedBlocksQty** (Max Nested Blocks) is the highest number of blocks nested within each other.
- **anonymousClassesQty**, **innerClassesQty**, and **lambdsQty** are the number of anonymous classes, inner classes, and lambda expressions, respectively.
- **uniqueWordsQty** and **logStatementsQty** are the count of unique words and log statements in a class, respectively.
- *Has Javadoc* metric checks whether the source code has javadoc or not.
- **privateMethodsQty**, **protectedMethodsQty**, **synchronizedMethodsQty** based on modifiers specify whether a class is public, abstract, private, protected, or native.
- **assignmentsQty** and **fieldQty** represent how often each variable and local field was used inside each class, respectively.
- *Method Invocations* is the number of directly invoked methods, variations are local invocations and indirect local invocations.
- **WMC** (Weight Method Class), also known as McCabe's complexity, counts the number of branch instructions in a class.

III. Related Work

Several studies performed the detection of security vulnerabilities or defects early in the life-cycle of the software systems [13]–[20]. For example, Harer et al. [15] discussed the use of traditional and deep machine learning models to detect vulnerabilities in C/C++ programs using control-flow graphs to predict vulnerabilities. Their features are extracted mainly based on the build and source of the code. Pang et al. [16] used a deep learning algorithm paired with a statistical feature selection algorithm to predict vulnerable classes in Java-based Android applications. Livshits et al. [17] proposed a solution to reduce vulnerabilities using static analysis of the Java code by providing warnings about 29 vulnerabilities based on the Tainted-Object-Propagation analysis. Awni et al. [18] employed several datasets to test the performance of three machine learning models in their bug prediction study. Their supervised machine learning algorithms predict future defects based on historical information and it was showed that the model based on Decision Tree is the best performing than Linear Regression. Gupta and Saxena [20] proposed that object-oriented metrics like Coupling Between Objects (CBO) and Lines of Code (LOC) are related to the occurrence of a vulnerability in a class more than Depth of Inheritance Tree (DIT) and Number of Children (NoC), which tend to be less sensitive during prediction. The authors used static code metrics to train their models and showed that the Linear Regression classifier provides good accuracy. They trained and tested on the individual and combined datasets and obtained maximum accuracy of 76.27%. Rinkaj et al. [19] studied the correlation between object-oriented metrics and defects, and

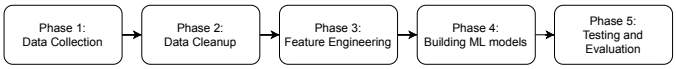


Figure 1: Research Method.

developed a KNN (K-Nearest Neighbor) model and compared its effectiveness in defect prediction against the parametric regression techniques.

IV. Method

Figure 1 shows our five-phase research method. The following sections detail each of the phases.

Phase 1: Data Collection: The data collection phase begins with collecting the security vulnerability dataset from the Apache Tomcat website via a web scraper built using node.js. For each security vulnerability, we collect the type (22 vulnerability types) and severity (4 severity levels) of the vulnerability and the Java classes in Apache Tomcat affected by the vulnerability. Finally, all the data were consolidated into a single CSV file. We exclude the duplicate vulnerability entries. Afterwards, we downloaded the source code of each Apache Tomcat version. We also collected class-level code metrics for each version of source code using a tool called CK [21]. This process resulted in 43 code metrics for each class in each version and stored in another CSV. Finally, the former and the latter CSVs are merged using right join, which resulted in 12,214 rows in the raw dataset. In total, 183 unique classes are involved in 22 security vulnerabilities in all versions of Tomcat, the distribution of which is shown in Figure 2. Moreover, 19 classes had *high* vulnerability, 65 classes with *important* vulnerability, 83 classes with *low* vulnerability, and 16 with *moderate*.

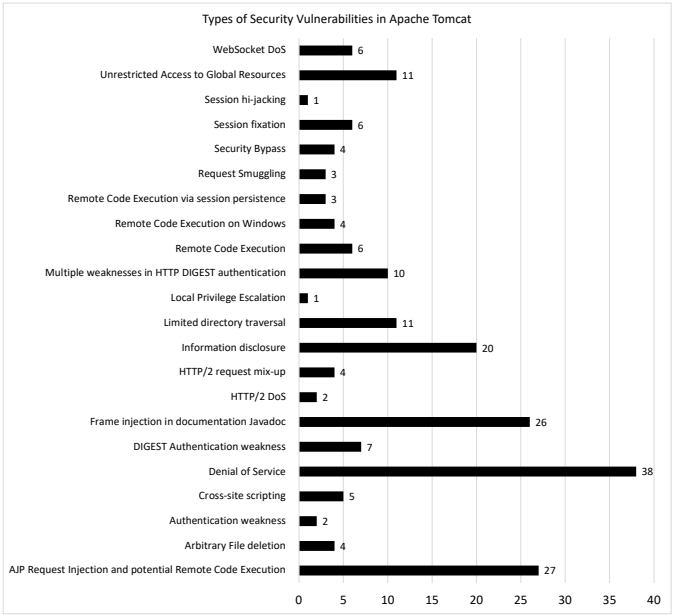


Figure 2: Types of Security Vulnerabilities in Apache Tomcat.

Phase 2: Data Cleanup: Upon forming the dataset, the second phase involves the dataset clean up, i.e., managing the irrelevant information for prediction. At first, we dropped all the null (i.e., NA) rows. Then, using Python's *pandas* framework, the duplicate rows were removed, leaving unique instances. Our raw vulnerability dataset consisted of a high negative to positive class ratio, i.e., the dataset is imbalanced. Hence, we used oversampling methods to handle the imbalance in the dataset. A total of 10,039 rows remained after data cleanup.

Phase 3: Feature Selection: Aniche-Ck's tool [12] for metrics computation produces 43 metrics, i.e., features, for each class from which features with higher impact on the predictor variable will be selected using feature selection techniques. In this study, we experimented with two feature selection techniques: (1) *Sequential Forward Selection* (SFS) starts with an empty model and then start fitting the model with individual feature one at a time and select the feature with the minimum p-value [22]; and (2) *Recursive Forward Elimination* (RFE), which is based on greedy optimization [23].

Phase 4: Building ML Models: In this study, the target variable is the *security vulnerability*, thus we constructed and employed *classification* algorithms. Initially, we experimented with three machine learning algorithms: Naïve Bayes classifier (NB), Logistic Regression (LR), and Decision Tree (DT) classifiers. NB's decision rule relies on Bayes' theorem [24] while the LR works for discrete or categorical outcomes, making it an extension of linear regression. Then, predicted values are mapped to probabilities. In DT, observations about an item are followed by conclusions about its target feature using a choice tree where the leaves serve as category labels and branches produce category labels according to the alternatives [25].

The ML models must be optimized independently. Beginning with the DT model, it was pruned since it was overfitted. Hence, the pruned DT model provided more accurate results instead of a hyper-accurate outcome. This helped predicting the vulnerable state of a class, i.e., whether or not a class possessed a security vulnerability. However, the DT model proved insufficient to predict the severity and the types of the vulnerabilities since DT works well as a binary classifier, thus, yielded poor accuracy results on predicting severity types. Therefore, a new model, the KNN classifier, was considered to predict the vulnerability severity for the concerned classes.

Cross Validation Techniques: In this study, we experimented with two cross-validation techniques: *hold-out* method and *k*-fold cross-validation. The hold-out, a.k.a., one-fold validation, is one of the most commonly used approaches to validate a machine learning model, randomly splitting a given dataset into a training and testing set into a 70%-30% ratio. The *k*-fold cross-validation is a re-sampling procedure used to evaluate machine learning models on a limited data sample. This technique requires an input parameter k, which determines the number of folds required by the dataset. Thus, $k = 10$, for instance, leads to 10 fold validation. For *k*-fold cross-validation, we split the full dataset into k groups, and for each group, consider it as hold-out and the remaining groups as the

training set, and repeat this process *k* times.

Phase 5: Testing and Evaluation: We measure the accuracy based on Equation 1, where TP, TN, FP, and FN refers to true positive, true negative, false positive, and false negative, respectively.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

We measure *precision* as the ratio between true positives and all positives, and *recall* is the ratio of true positives it correctly identifies among all the positives, as shown in Equation 2.

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN} \quad (2)$$

V. Experiments and Results

Classes are affected by the security vulnerabilities exist in the source code of each Apache Tomcat version. They are treated as positive instances, and the unaffected classes are considered negative ones. Consolidating the vulnerability information and the source code of all Tomcat versions, our final dataset had 12,214 rows. The data cleaning phase, e.g., null rows, duplicate rows, corrupt data, yielded 10,084 rows. Raw dataset and all the models built in this study are made available online [26].

We conduct experiments in two steps. First, we predicted whether the classes were associated with security vulnerabilities (yes or no). In a further experiment, we attempt to predict the types and the severity of the vulnerabilities.

In the first experiment, the training data was fit into the Decision Tree (DT), Logistic Regression (LR), and Naive Bayes (NB) models and the target feature (vulnerability) was predicted. After the prediction, the accuracy, precision, and recall values were recorded. Furthermore, an AUC-ROC (Area under the ROC Curve) curve graph was plotted with obtained results. AUC-ROC shows the discriminative power of the model between classes. The ROC curve is plotted with TPR (true positive) against the FPR (false positive rate).

In another experiment, with the K-Nearest Neighbors (KNN) algorithm, we fit the training data into the KNN model and predict the vulnerability. In this case, the value of the n_nearest_neighbors was set to 109, being the square root of the total number of rows in the final dataset. After the prediction, we reported the accuracy, precision, and recall.

A. Optimizing ML Models

This section explains the performance optimization of our ML models by tailoring various hyper-parameters for each model. From the basic implementation of the ML models, the DT model is witnessed to be overfitted. The model produced hyper-accurate results, which would result in poor performance with test data. To settle this issue, we used a post-pruning method (i.e., cost complexity-based pruning). Using this method, the ccp_alphas values are plotted in a chart. Furthermore, a chart that illustrates the relationship between the number of nodes and depth is outlined. Finally, we visualized the correlation between the accuracy and the ccp_alpha values. The optimal ccp_alpha value is inferred from the plots and is initialized with the DT model.

For the LR model, the 'sag' solver was used due to the larger dataset size. Moreover, the new "scikit.learn" [27] version for logistic regression has n_jobs that enables the user to control the CPU usage for the algorithm, which was also tuned for optimal performance.

Furthermore, although the algorithms mentioned above would perform well for a binary classification such as the "vulnerability status", they would perform poorly with multi-class classifications, i.e., the severity and the vulnerability types. Thus, the KNN was considered for multi-class classification.

In the following, we show the results and answer our two research questions.

B. *Best Feature Set for Predicting Vulnerabilities (RQ1)*

RQ1 aims to report the best feature set for prediction because the best feature set will play a significant role in the accuracy of the machine learning models.

1) Prediction using the First Feature Set: This experiment focuses on prediction using the feature set acquired using the Sequential Forward Selector (SFS) algorithm. Feature selection based on SFS resulted in 31 features as the final features when selecting the best number of features. The algorithm chooses features as a whole subset. Hence, the individual importance of each feature is not recorded in this method. The results of this experiment are the performance indices of the ML models when the first feature set is fitted. The SFS method chooses the following 31 features (metrics) as best contributing:

- CBO, DIT, RFC, LCOM, TCC, LCC, NoSI, LOC, totalMethodsQty, privateMethodsQty, protectedMethodsQty, visibleMethodsQty, synchronizedMethodsQty, totalFieldsQty, protectedFieldsQty, defaultFieldsQty, finalFieldsQty, returnQty, loopQty, tryCatchQty, paranthesizedExpsQty, stringLiteralsQty, numbersQty, assignmentsQty, mathOperationsQty, variablesQty, maxNestedBlocksQty, anonymousClassesQty, innerClassesQty, uniqueWordsQty, logStatementsQty

2) Prediction using the Second Feature Set: This experiment focuses on prediction using the feature set acquired using the Recursive Forward Elimination (RFE) algorithm. A set of 22 features were fit into the training and test data, and the three models predicted the target feature vulnerability. The results of this experiment are the performance indices of the ML models when the second feature set is fitted. RFE algorithm chose 22 features in total. The features along with their rankings by the algorithm are as shown in Figure 3 with the features marked with 1 being the top-ranked ones.

C. *Machine Learning to Predict Vulnerabilities (RQ2)*

RQ2 aims to report the best performing model in terms of prediction accuracy.

1) Choosing the Best Performing Model: This experiment focuses on comparing the ML models to select the best classifier. It will be conducted using the finalized feature set for optimal performance. Table II shows the performance of the DT classifier when no pruning is performed, i.e., the model

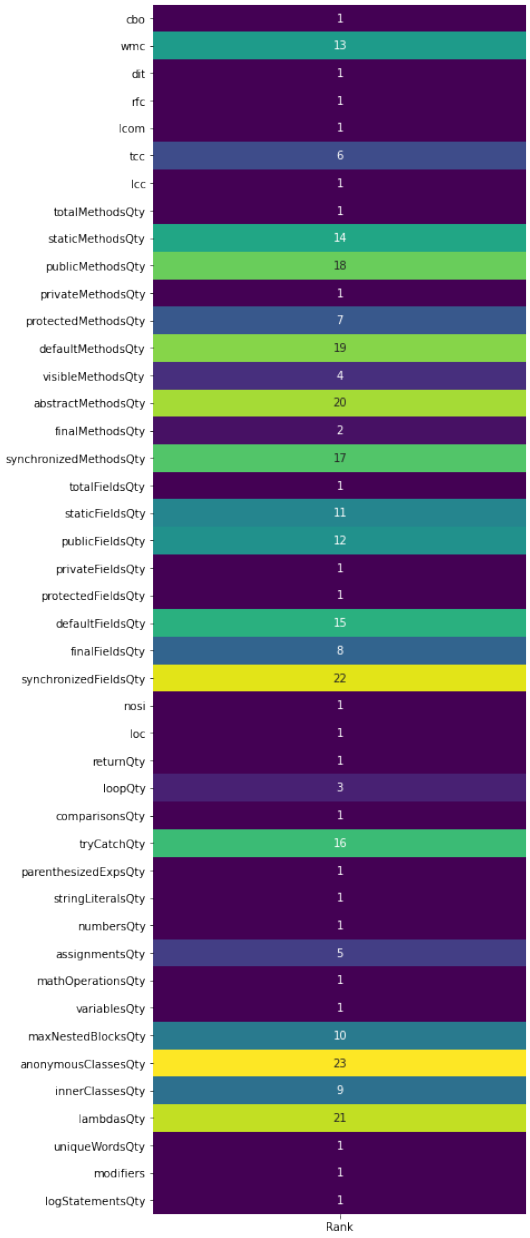


Figure 3: RFE Ranked Features.

Table II: Performance of the DT Model Without and With Pruning.

Measures	Without Pruning (in %)	With Pruning (in %)
Accuracy	97.7	76.6
Train Score	99.8	77.1
Test Score	97.4	80.1

is overfitted along with the performance metrics of the DT classifier post pruning.

Tables III and IV present prediction performance utilizing the SFS and RFE feature selection methods, respectively. Notably, precision and recall can be recorded only for binary

Table III: Prediction Results using the SFS Algorithm.

Measures	DT	LR	NB
Accuracy (in %)	76.8	61.9	71.6
Precision (in %)	81.8	57.8	87.3
Recall (in %)	68.9	88.5	50.5

Table IV: Predicting Performance using RFE algorithm.

Measures	KNN	DT	LR	NB
Accuracy (in %)	80.6	77.2	61.8	71.7
Precision (in %)	78.4	81.6	57.8	87.4
Recall (in %)	84.4	70.2	87.3	50.6

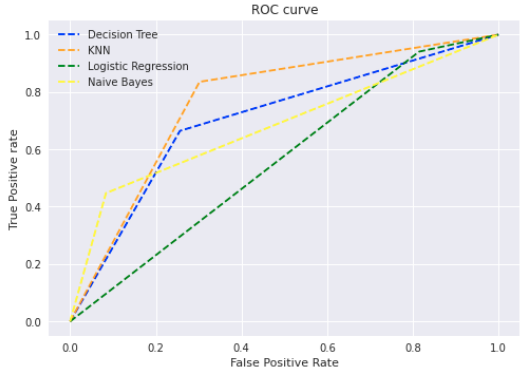


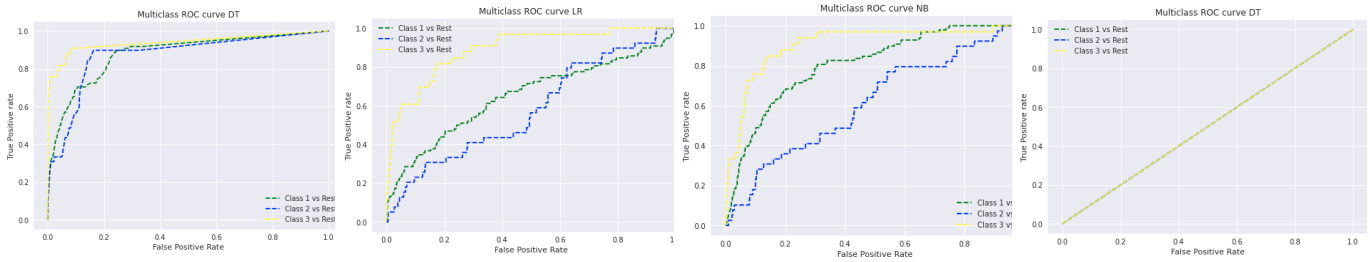
Figure 4: AUC-ROC Curve for Predicting Vulnerability Types.

classification, where the result is usually 1 (vulnerable) or 0 (not vulnerable). Based on the feature set selected by the SFS method, DT performed best in terms of accuracy (76.8%). In contrast, the NB model outperformed others in precision (87.3%), and LR performed best in recall (88.5%). The scenario is different when using the RFE selected features and an additional model, KNN. More concretely, KNN outperformed three other models in terms of accuracy (80.6%).

Figure 4 shows the AUC-ROC curve for the performance by four models in predicting the types of security vulnerability. As Figure 4 confirms, KNN outperforms three other ML modes.

Figure 5 depicts the AUC-ROC curves for the performance by four models in predicting the severity of the vulnerability. As Figure 5d shows, DT model under-performed in predicting the severity of the vulnerability being this a multi-class classification problem (e.g., *important* vs. *moderate* vs. *low*). Encore, KNN outperformed its contenders in terms of the AUC-ROC curve in predicting the severity of the vulnerability.

We also made a comparison with other static code analysis benchmark tools, e.g., SonarQube [3]. Table V shows the comparison between the best performance (with the best performing models) and SonarQube w.r.t. various measures. SonarQube had a higher accuracy but very low precision and recall. Overall, SonarQube could better classify the negative classes than the positive ones. This could be because it uses predefined rules to detect vulnerabilities in the source code



(a) KNN (b) LR (c) NB (d) DT

Figure 5: AUC-ROC Curve for Predicting Severity of Vulnerability by Different Classifiers.

Table V: Comparison with SonarQube as Benchmark Tool.

Measures	**Best Performance (Model)**	**SonarQube** [3]
Accuracy (in %)	80.6 (KNN)	98.2
Precision (in %)	87.4 (NB)	12.5
Recall (in %)	88.5 (LR)	4.1

and does not rely on metrics. Also, the vulnerabilities in Apache Tomcat are more project-specific. In contrast, the vulnerabilities detected by SonarQube are more general to the entirety of Java-based applications. It is difficult to detect new types of vulnerabilities unless the corresponding rules are updated in SonarQube [3].

VI. Analysis and Discussion

The feature sets derived with SFS and RFE include 31 and 22 features, respectively. With these differing features, both the feature sets produced almost similar performance when fitted into ML models. However, the feature set by RFE performed similarly to the one by SFS with a significantly smaller number of features, and the set by RFE is the better feature set.

In the beginning, SFS features were fitted into the DT model, and the models' performance was recorded. We observed that the DT model is overfitted, and hence, we pruned the DT model. The DT is a binary classifier model and one of the fastest among the four. Later, the RFE feature set was fitted into the DT model, which yielded the same prediction accuracy of 77.5%. Therefore, we obtained the purpose of feature selection and obtained the best results with the smallest feature set.

We witnessed that the DT model functioned well as a binary classifier compared to NB and LR models. However, the DT model delivered poor in predicting vulnerability severity. Thus, we introduced a multi-class classifier, namely KNN. Our results imply that the KNN classifier performed significantly better than other models (DT, NB, LR) when predicting the severity of a vulnerability. The KNN classifier reached an accuracy score of 74% while predicting the severity of the vulnerabilities. At the same time, the DT, NB, and LR models provided accuracy scores of 40%, 38%, and 43%, respectively.

Threats to Validity: This study focused on the prediction of security vulnerabilities in the Apache Tomcat open-source project. To minimize the threats to *external validity*, we considered seven different versions of Apache Tomcat. However, other systems need to be analyzed to further generalize our findings. To minimize the threats to *internal validity*, we experimented with several ML models and two feature selection methods. However, the results reported in this study are further subject to improvement using more sophisticated ML models and feature selection techniques. Also, vulnerability predictions are made at the class level. However, a finer – method or line-level – prediction would be useful for the developers. To minimize the threats to the *reliability* and *repeatability validity*, the dataset and model implementations are made available online [26].

VII. Conclusion and Future Work

This study responded to two research questions on the best feature set and the best performing model in predicting security vulnerabilities. The best feature set was determined by employing two feature selection techniques and examining the performance results. To achieve the best performing model, the experiments were carried out using four supervised machine learning models. Among the Decision Tree (DT), Naive Bayes (NB), and Logistic Regression (LR), our results suggested that the pruned DT delivered better than the other two models in predicting security vulnerability. However, as a binary classifier, DT could not handle severity and the types with multiple classes. The presence of vulnerability is merely a binary value (yes/no), but the severity has four classes (High, Important, Moderate, Low). Thus, the KNN model was considered, and it performed better than the DT model while predicting all the target classes. Consequently, the KNN classifier was selected as the better performing model among the four. This answers RQ2 on choosing the best performing model.

The models and feature sets used in this experiment are subject to enrichment for better performance. More sophisticated feature selection methods can be applied. Also, ensemble methods, simple multi-layer perceptron, and deep neural networks can be applied to make more accurate predictions. Moreover, the results obtained using this experiment are distinct only to Apache Tomcat, and more experiments are required with different systems.

REFERENCES

[1] C. Osborne. (2018) Open-source Vulnerabilities Plague Enterprise Codebase Systems. [Online]. Available: https://www.zdnet.com/article/enterprise-codebases-plagued-by-open-source-vulnerabilities/

[2] "Jslint," https://www.jslint.com, accessed: 2021-10-15.

[3] "Sonarqube," https://www.sonarqube.org/, accessed: 2021-10-15.

[4] I. Andrianto, M. I. Liem, and Y. D. W. Asnar, "Web application fuzz testing," in *2017 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2017, pp. 1–6.

[5] NordVPN, "Five vulnerabilities attackers leveraged most in 2020," Dec 2020. [Online]. Available: https://www.qualitydigest.com/inside/management-article/five-vulnerabilities-attackers-leveraged-most-2020-011321.html

[6] "Apache tomcat," https://tomcat.apache.org, accessed: 2021-10-15.

[7] Nsrav, 2021. [Online]. Available: https://owasp.org/www-community/attacks/Denial_of_Service

[8] A. W. Bayles, E. Brindley, J. C. Foster, C. Hurley, and J. Long, "Chapter 8 - classes of attack," in *Infosec Career Hacking*, A. W. Bayles, E. Brindley, J. C. Foster, C. Hurley, and J. Long, Eds. Burlington: Syngress, 2005, pp. 241–289. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597490115500143

[9] Harley, "Webapps 101: Information disclosure vulnerabilities and portswigger lab examples," Jan 2021. [Online]. Available: https://infinitelogins.com/2021/01/02/information-disclosure-vulnerabilities-portswigger-lab-examples/

[10] "What is HTTP Request Smuggling? by PortSwigger Ltd." https://portswigger.net/web-security/request-smuggling, accessed: 2021-11-16.

[11] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011, special Issue on Security and Dependability Assurance of Software Architectures. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762110000615

[12] M. Aniche, *Java code metrics calculator (CK)*, 2015, available in https://github.com/mauricioaniche/ck/.

[13] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *2010 Third international conference on software testing, verification and validation*. IEEE, 2010, pp. 421–428.

[14] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[15] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, and et al., "Automated software vulnerability detection with machine learning," Aug 2018. [Online]. Available: https://arxiv.org/abs/1803.04497

[16] Y. Pang, X. Xue, and H. Wang, "Predicting vulnerable software components through deep neural network," in *Proceedings of the 2017 International Conference on Deep Learning Technologies*, ser. ICDLT '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 6–10. [Online]. Available: https://doi.org/10.1145/3094243.3094245

[17] Livshits, V. Benjamin, and M. S. Lam., "Finding security vulnerabilities in java applications with static analysis," *In USENIX Security Symposium*, vol. 14, pp. 18–18, 2005.

[18] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software bug prediction using machine learning approach," vol. 2. IJACSA, 2018, pp. 6–6.

[19] R. Goyal, P. Chandra, and Y. Singh, "Identifying influential metrics in the combined metrics approach of fault prediction," *SpringerPlus*, vol. 2, no. 1, pp. 1–8, 2013.

[20] D. L. Gupta and K. Saxena, "Software bug prediction using object-oriented metrics," vol. 42. Sadhana, 2018, pp. 665–669.

[21] "Ck," https://github.com/mauricioaniche/ck/, accessed: 2021-11-16.

[22] V. Luhaniwal, "Feature selection using wrapper method - python implementation," Dec 2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/10/a-comprehensive-guide-to-feature-selection-using-wrapper-methods-in-python/

[23] S. Yemulwar, "Feature selection techniques," Nov 2020. [Online]. Available: https://medium.com/analytics-vidhya/feature-selection-techniques-2614b3b7efcd

[24] "Bayes' theorem," Jul 2021. [Online]. Available: https://corporatefinanceinstitute.com/resources/knowledge/other/bayes-theorem/#: :text=Formula for Bayes Theorem,the probability of event A

[25] A. Roy, "A dive into decision trees," Nov 2020. [Online]. Available: https://towardsdatascience.com/a-dive-into-decision-trees-a128923c9298

[26] "Replication package," https://github.com/palmafr/SweDS21, accessed: 2021-10-15.

[27] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.