Function-Level Software Metrics for Predicting Vulnerable Code

Md Rayhan Amin*, Daniel Tanner*, Yashita Sharma*, Kollin Napier†, and Tanmay Bhowmik*
* Department of Computer Science and Engineering, Mississippi State University, USA
† Mississippi Gulf Coast Community College, USA
{ma1962, dkt91, ys460}@msstate.edu, kollin.napier@mgccc.edu, tbhowmik@cse.msstate.edu

***Abstract*—Limited experience or hard deadlines often lead developers to technical Q&A websites, e.g., Stack Overflow, to find quick answers to their coding problems. Research, however, suggests that a large share of the answers on Stack Overflow include vulnerable code and their use in a software system may cause major security implications. Researchers have suggested several approaches, e.g., leveraging software metrics, to predict vulnerability in the source code. The majority of these approaches require the whole code base or a big chunk of it to perform the prediction. The answers on those Q&A websites, however, are code snippets typically with a single function, or, at most a couple of related functions. Predicting vulnerabilities in such code snippets has received limited attention. Few examples present in the literature suggest leveraging function-level software metrics to predict vulnerable code. Such work, however, provides incomplete analysis with heavily imbalanced datasets. This paper examines whether function-level code metrics can differentiate the vulnerable and fixed versions of a given function. In particular, we develop a "near-balanced" dataset with a total of 132,754 function instances, calculate 18 function-level code metrics for both vulnerable (49.6% of the total) and fixed versions of the functions, and conduct binary logistic regression with the Wald test. The results suggest that function-level code metrics may not be useful in predicting if a given function contains vulnerability.**

***Index Terms*—software metrics; function-level software metrics; software vulnerabilities; logistic regression**

I. Introduction

Software vulnerability can be defined as a flaw or defect in a software system that can be exploited by an attacker to compromise the security of the system and cause damage or loss of data [1]. Lately, the modern software industry has become fiercely competitive. With rapid advancements in software development tools and technologies, software companies are heavily investing in developing novel and user-centric features to entice users and dominate the market [2]. However, this relentless pursuit of success comes with a trade-off. The resulting software systems, although feature rich, have become increasingly complex, making them more susceptible to vulnerabilities [3]. According to Coalition, an active cyber threat insurance provider, the total number of Common Vulnerabilities and Exposures (CVEs) for software systems is expected to increase by 25% in 2024, reaching 34,888 vulnerabilities, which is equivalent to approximately 2,900 vulnerabilities per month [4]. With the increasing number of vulnerabilities and security breaches each year, the aspect of software security and vulnerability analysis have become more critical than ever [5].

Several factors, such as coding errors, insecure design, insufficient testing, and rapid development cycles, can introduce vulnerabilities into a software at various stages of the Software Development Life Cycle (SDLC). However, research suggests that most of the software vulnerabilities stem from mistakes made during the system implementation phase [6]. These mistakes can be caused by either inexperienced and negligent professionals or by the use of legacy code. For example, inexperienced developers can rely heavily on technical Q&A websites, such as Stack Overflow or CodeRanch, to find solutions [7]. While these resources can be valuable, copying code without understanding its security implications can introduce vulnerabilities. In fact, a recent study on Stack Overflow suggests that the C/C++ code snippets posted as answers include 32 out of 89 (i.e., 36%) vulnerability types identified as Common Weakness Enumeration — CWE, a community-developed collection of common software vulnerabilities [8].

The code snippets posted as answers on technical Q&A websites are typically single functions, or, in fewer cases, a couple of related functions, performing unit tasks [8]. Due to lack of experience or the rush due to a hard deadline for time-to-market, developers often find refuge to online resources such, as Stack Overflow, and reuse the posted function without knowing its security ramification [7]. Therefore, an easy and effective way to distinguish between vulnerable and non-vulnerable functions on technical Q&A websites would further help the developers reduce security problems in their code.

To deal with software vulnerabilities, software engineers have several techniques available at their disposal. Most of the traditional vulnerability detection techniques follow a reactive approach. In that case, a vulnerability is detected after it has been introduced into the system and potentially exploited by attackers [9]. On the other hand, proactive approaches focus on identifying and addressing vulnerabilities in the early stages of the SDLC and before the system is deployed [9]. Therefore, a proactive approach to predicting vulnerability in a given code snippet or function would be very beneficial for the developer community, especially those who are frequent users of technical Q&A websites.

Research suggests several proactive approaches to predict vulnerability in the source code leveraging the quality attributes of a software represented as code metrics [10]–[13]. Majority of these approaches, however, require the whole code base or a large chunk of it to perform the prediction [10], [11].

2835-5776/24/$31.00 ©2024 IEEE
DOI 10.1109/IRI62200.2024.00059

Whether function-level code metrics could be leveraged as a mean to distinguish between vulnerable and non-vulnerable functions [10]–[12], [14] has received very limited attention. A couple of work we find along this line include Alves *et al.'s* [14] research, which finds some correlation between the size of a function in terms of lines of code (LOC) and vulnerability. The other research, conducted by Pakshad and colleagues [10], attempts to train a machine learning (ML) model with function-level code metrics to predict vulnerable functions. Pakshad *et al.* [10], however, do not examine if there is any potential association between those metrics and the function's vulnerability aspect (i.e., vulnerable or not-vulnerable). Furthermore, both [10] and [14] work with heavily imbalanced datasets. In particular, only 0.142% and 0.366% functions were vulnerable in the datasets used in [14] and [10], respectively.

In this work, our main objective is to examine if there is an association between the function-level code metrics and the vulnerability aspect of a given function. To that end, we collect the vulnerability-fixing commits for five medium- to large-size open-source software (OSS) projects: Chromium, FFmpeg, Linux Kernel, QEMU, and Xen Hypervisor. We use these commits to calculate the function-level code metrics for non-vulnerable functions, i.e., the functions that have been changed in the vulnerability-fixing commits. Metrics for their vulnerable counterparts were calculated from the version of the functions immediately prior to the fixing commits. In this manner, we develop a "near-balanced" dataset where 49.6% data points (out of 132,754 data points) belong to vulnerable functions. (cf. Section III for details). We employ both quantitative and qualitative analysis techniques to draw conclusions.

Our quantitative analysis, involving binary logistic regression [15] with Wald test [16], suggests that there is no statistically significant association between the function-level code metrics and the vulnerability aspect of a given function. Neither the type of software shows any association in this regard. We also conduct further qualitative analysis to provide certain insights into our findings. The rest of the paper is organized as follows. Section II discusses the existing literature. Section III describes the experimental steps of our study. Section IV presents the findings from the statistical analysis performed. Section V discusses key takeaways. Finally, Section VI concludes the paper with a direction to the future work.

II. Related Work

Software metrics are considered as quality attributes of a system as they provide valuable insights about the system's reliability, maintainability, performance, and usability etc [17] [18]. Although they are not a direct indicator of the presence of vulnerability in a system, certain code metrics have been found to be useful in identifying areas of a code that might present security risks [12]. Medeiros et al. analyzed code metrics at different levels (i.e., project, file, and function) with the aim of finding the best subset of code metrics capable of distinguishing vulnerable and non-vulnerable codes [11]. They used a feature selection technique to identify the most relevant subset of metrics across different code units. To achieve this goal, they combined a random forest classifier model with a genetic algorithm to build a heuristic search technique. Though they considered vulnerable and non-vulnerable codes at function-level, they did not consider the vulnerable and non-vulnerable version of the same function.

Another study by Pakshad et al. investigated code metrics at function-level and their relationship to vulnerabilities [10]. They mainly focused on identifying vulnerable C/C++ functions based on code metrics and categorized the functions into attack types. To achieve their goal, first, they converted the code snippet of a given function into a code property graph using a fuzzy parser. The resultant graphs for all the functions were then stored in a database, and code metrics for each function were calculated by traversing the associated graph. A total of 15 function-level code metrics was considered. Subsequently, a dataset was generated by combining the code metrics for each function along with the information about that function being vulnerable and the corresponding vulnerability type. The dataset was then used to train supervised machine learning models, where the functions were used as input to the models for prediction. In this study, they only considered whether a function is vulnerable and did not consider the fixed version of the same function. Moreover, the dataset they generated suffered from class imbalance issue, resulting in biased prediction.

In a similar research, Alves et al. examined code metrics at different levels such as file, function, and class to determine whether code metrics are capable of distinguishing between vulnerable and non-vulnerable code units [14]. To answer their research questions, they developed a dataset after analyzing 2,875 security patches across five open-source C/C++ projects and calculating code metrics at different levels. Subsequently, they performed statistical analysis on the generated dataset, and the results did not show a strong correlation between code metrics and vulnerabilities. The results also suggested that vulnerable functions are prone to more vulnerabilities. Although they considered vulnerable and fixed versions of different code units, in the fixed version of the dataset, they combined the fixed code units with the rest of the code units that were not found to be either vulnerable or non-vulnerable. As a result, their dataset suffered from a class imbalance issue due to the overwhelming numbers of non-vulnerable code units.

Sultana et al. also leveraged code metrics to predict vulnerable classes and methods in Java projects [12]. They identified two sets of code metrics (one for class-level and one for method-level) using statistical analysis and employed them as features to train supervised machine learning models. The trained models for both class-level and method-level features achieved recall higher than 65% and precision higher than 75%. Similar to the previous studies, their dataset also suffers from class imbalance issue.

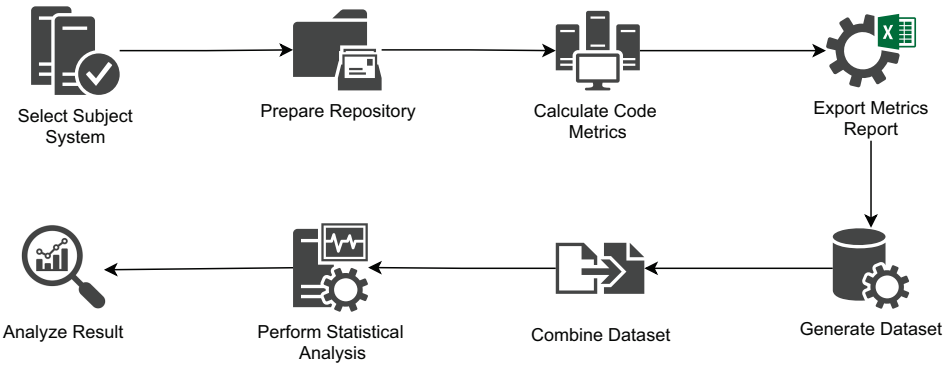

Fig. 1. Research methodology to distinguish vulnerable and non-vulnerable function using function-level code metrics.

III. Methodology

A. Research Question

As mentioned earlier, our main objective is to examine if the function-level code metrics could be used to determine whether a given function potentially contains any vulnerability. In particular, we ask the following research question.

> **RQ:** Is there any association between the function-level code metrics and the vulnerability aspect of a given function.

In order to answer the research question, we formulate the following null and alternative hypotheses.

> **Null Hypothesis,** H_0**:** There is no statistically significant association between the function-level code metrics and the vulnerability aspect of a given function.
>
> **Alternative Hypothesis,** H_1**:** There is a statistically significant association between the function-level code metrics and the vulnerability aspect of a given function.

In order to test the aforementioned hypotheses, the research activity we conduct is presented in Fig. 1. In what follows, we discuss each step of our research activity in further detail.

B. Select Subject System

The first and foremost step of our study is to select appropriate subject systems for analysis. While selecting a system, we need to consider a few aspects that are essential for the study. First, the subject system should be important from a security perspective (i.e., if a vulnerability is found and exploited, could cause damage to the system and its users). For example, if a vulnerability is found and exploited in the open-source Linux operating system, millions of users will be seriously affected. Second, the system should be an open-source project with a publicly available code repository and issue tracking system such as the Chromium project, the Xen project, etc. Finally, the associated vulnerabilities and the corresponding patches for a subject system should also be available publicly (e.g., the National Vulnerability Database (NVD) maintains information on reported vulnerabilities and relevant patches for numerous projects [19]).

To our advantage, we leverage a dataset developed by Napier et al. that included information on vulnerabilities and their associated patches for 344 open-source projects [20]. The dataset contained two separate Excel files for each project: one for the vulnerability-fixing commits and the other one for the commits preceding the fixing commits of the same file. Following [20], the second file contains the vulnerable versions of the fixed functions. Each commit within the Excel files included details like commit hash, modified file names along with file extensions, and modified function names. From this dataset, we identify five C/C++ projects (*Chromium, FFmpeg, Linux Kernel, QEMU, and Xen Hypervisor*) that meet the above mentioned criteria and also contain sufficient data points.

C. Prepare Repository

Before we can calculate the code metrics for the modified functions in a given commit, we need to reach the specific state of the repository after that commit. As all the subject systems in consideration either use `git` repositories or maintain `git` mirror of their repositories, we can achieve this by checking out the repository at the desired commit hash.

D. Calculate Code Metrics

Once we reach the specific state for a given commit, we use Understand by SciTools [21] to analyze the repository at that state and calculate code metrics for all the available code units (e.g., files, classes, functions), as Understand currently has no option to select a specific code unit for metrics calculation. It is also important to highlight that this is both an exhaustive and resource consuming task, as for each commit, we are having to deal with a complete codebase every time.

E. Export Metrics Report

Once a repository is analyzed by Understand, we export the calculated code metrics into an Excel file. For our research, we identify 18 function-level code metrics available for C/C++ languages in Understand [21]. Of the 18 code metrics, we exclude *CountLineInactive* from our statistical analysis (rationale

TABLE I
SUMMARY OF THE DATASET

Project	No. of Patches	No. of Snapshots	No. of Files		No. of Functions	
			Vul.	Fixed	Vul.	Fixed
Chromium	52	104	217	217	257	257
FFmpeg	16,391	32,782	1,599	1,599	30,847	32,174
Linux	577	1,154	857	857	1,872	1,876
QEMU	240	480	1,014	1,014	21,706	21,624
Xen	96	192	787	787	11,121	11,020
Total	17,356	34,712	4,474	4,474	65,803	66,951

explained in Section IV). A detailed explanation of the selected code metrics can be found in [22].

F. Generate Dataset

In this step, we create two Excel files: one for vulnerable commits and the other for the fixed commits. Once a report is generated for a given commit ID, we only extract the rows from the report (i.e., the Excel file) that represent the records for the modified functions in that commit. Depending on the type of commit (whether vulnerable or fixed), we append the extracted rows to the appropriate Excel file. We also add a label to each newly appended record to indicate the record type (0 for fixed functions, 1 for vulnerable functions).

G. Combine Dataset

To obtain the final curated dataset, for each subject system, we combine the two previously generated Excel files into a single file. Table I presents a summary of our dataset. We notice that the total number of fixed functions is slightly higher than that of the vulnerable functions. Such a situation is observed because, in some rare cases, new functions are introduced and old functions are deleted in the fixing commits. Although such occurrences are minimal (just 0.86% of a total of 132,754), we include them in our analysis to preserve the characteristics of real-world vulnerability fixing activities. Nevertheless, our dataset remains "near-balanced" with 65,803 vulnerable and 66,951 non-vulnerable functions (c.f. Table I). Our dataset is available at https://shorturl.at/mnEGN for replication.

H. Perform Statistical Analysis

Once we acquire the processed dataset, we perform statistical analysis using SAS to answer our research question (detailed discussion in Section IV).

I. Analyze Result

Finally, we analyze the results to draw conclusions about our research question. We also focus on identifying key nuances from the results and discuss future avenues for this research.

IV. RESULTS AND ANALYSIS

We conduct binary logistic regression [15] with Wald test [16] using SAS to examine the above hypothesis. Binary logistic regression, an extension of the "regular" linear regression, is a statistical tool to determine whether a set of independent variables (IVs) has any statistically significant association with the dependent variable (DV) [15]. In this test, the DV contains categorical data with two possible levels, whereas the IVs could be a mix of both contentious and categorical data variables. Wald test in the logistic regression model, on the other hand, helps us determine the specific IVs that significantly impact the DV [16].

TABLE II
RESULTS FROM THE TESTING OF GLOBAL NULL HYPOTHESIS

Test	**Chi-Square**	**DF**	***p*-value**
Likelihood Ratio	10.6100e	20	0.9557
Score	10.5998	20	0.9559
Wald	10.5409	20	0.9573

TABLE III
ADDITIONAL RESULTS FROM THE ANALYSIS OF EFFECTS

Effect	**DF**	**Wald Chi-Square**	***p*-value**
Software	4	1.3290	0.8564
CountInput	1	0.4001	0.5270
CountLine	1	0.1229	0.7259
CountLineBlank	1	0.1168	0.7326
CountLineCode	1	0.0114	0.9148
CountLineCodeDecl	1	0.3602	0.5484
CountLineCodeExe	1	0.1634	0.6860
CountLineComment	1	0.0462	0.8299
CountLinePreprocessor	0	.	.
CountOutput	1	1.6026	0.2055
CountSemicolon	1	0.1186	0.7305
CountStmt	1	0.6957	0.4042
CountStmtDecl	1	0.8509	0.3563
CountStmtEmpty	1	0.6969	0.4038
CountStmtExe	1	0.5799	0.4464
Cyclomatic	1	0.5498	0.4584
MaxNesting	1	1.0937	0.2957
RatioCommentToCode	1	0.2605	0.6098

Our binary logistic regression, with Wald test, includes 17[1] function-level metrics (contentious data variables) as well as the software name (categorical data variable) as the IVs and the vulnerability aspect as the DV. The DV has two possible values: 0 – not vulnerable and 1 – vulnerable. Note that we include the software name as an independent variable in case the type of software, along with the code metrics, has a significant impact on the vulnerability aspect of its functions.

Tables II and III present important highlights of our statistical analysis results. The results specifically relevant to testing our global null hypothesis (H_0H_0) are displayed in Tables II. Here, the *p*-values for "Likelihood Ratio", "Score", and "Wald" are very high (>0.9 in every case). Based on these results it is clear that we fail to reject H_0 at α = 0.05 level of significance. In other words, we find enough statistical evidence that *neither the function-level code metrics nor software type have any association with the vulnerability aspect of a given function*.

The effect of each parameter obtained from the Wald test is presented in Table III. Note that meaningful values are missing for the code metric *CountLinePreprocessor* ("0" for DF and "." for both Wald Chi-Square and *p*-value). This indicates that

[1]We discard the metric *CountLineInactive* from our analysis as we find its value being always "0" in the dataset.

SAS automatically excludes the variable *CountLinePreprocessor* from the analysis as it finds *CountLinePreprocessor* to be derivable from *CountLine, CountLineBlank, and CountLineCode*. In Table III, we clearly observe that the *p*-value for each parameter is much higher than α = 0.05, ranging from 0.8564 for *Software* to 0.2055 for *CountOutput*. Such results provide further evidence that neither any of the function-level code metrics nor the software type could be used to predict whether a given function potentially contains vulnerability.

In summary, we find the conclusion of our study inconsistent with the current knowledge presented in the literature (cf. Section II). A deeper look into our research question and the study design, however, provides further insights regarding this inconsistency, which is further elaborated in the next section.

V. Discussion

A. Inconsistency of our Findings with the Current Literature

The current literature suggests that certain function-level code metrics are capable of distinguishing between vulnerable and non-vulnerable source code [1], [14], [23], [24]. On the contrary, this study suggests that we may not use any function-level code metric to predict the vulnerability aspect of a given function. We posit the following two major reasons behind this contradiction.

The Research Objective. Existing literature mostly examines whether vulnerable source code could be predicted by machine learning models trained with datasets that include code metrics from vulnerable and non-vulnerable source code [10]–[12]. The datasets in such research are derived from the functions of different software systems where the vulnerable and non-vulnerable instances are not of the same function. Following this approach, some researchers have often observed promising results [10], [11]. In our case, however, the intended beneficiaries are the developers who directly use code snippets posted on online forums, such as Stack Overflow or CodeRanch [7]. Since such code snippets are typically standalone functions, we focus on exploring if function-level code metrics could be leveraged to predict if a given function is potentially vulnerable. Therefore, we consider the vulnerable and fixed versions of the same function and examine if we can distinguish them based on their function-level metrics. Let us take a closer look into our dataset for further explanation.

Table IV presents the code metrics for both the vulnerable and fixed versions of two randomly selected functions $AC3\_encode\_init(int*)$ from FFmpeg and $disas\_vfp\_insn(int*, int*, int)$ from QEMU. These two functions are named as $Function_A$ and $Function_B$, respectively, in Table IV. We notice that the code metrics of these functions do not change much after fixing the vulnerability in it. In fact, other functions in our dataset show a similar trend. This observation suggests that often the vulnerability in a function is fixed by making small changes that may not significantly alter the values of the code metrics. Hence, we fail to reject the null hypothesis in our experiment.

Imbalanced vs. Homogeneous Datasets. Existing research along this line has predominantly worked with imbalanced datasets. In reality, as only a small proportion of the code base include vulnerability, the number of data points associated to non-vulnerable functions have been much higher compared to that of the vulnerable functions [10]–[12], [14] in the existing research. Therefore, the findings in the current literature may include some bias due to the data imbalance problem [25]. In order to avoid this problem, our research works with dataset that is almost balanced (49.6% and 50.4% out of 132,754 data points belong to vulnerable and non-vulnerable functions, respectively). In retrospect, our data could be considered as too homogeneous as we have picked only those functions that are known to contain some vulnerable code. Such homogeneity could also be an important factor behind the findings of our work. This observation, however, opens avenue for further studies, which is discussed in Section VI.

TABLE IV
Code metrics of two random functions: Vulnerable vs. Fixed

Code Metrics	$Function_A$		$Function_B$	
	Vulnerable	**Fixed**	**Vulnerable**	**Fixed**
CountInput	0	0	0	0
CountLine	1	1	1	1
CountLineBlank	0	0	0	0
CountLineCode	1	1	1	1
CountLineCodeDecl	1	1	1	1
CountLineCodeExe	1	1	1	1
CountLineComment	0	0	0	0
CountLinePreprocessor	0	0	0	0
CountOutput	5	5	22	22
CountSemicolon	51	51	380	380
CountStmt	35	35	372	373
CountStmtDecl	10	10	57	58
CountStmtEmpty	1	1	18	18
CountStmtExe	24	24	297	297
Cyclomatic	11	11	148	148
MaxNesting	3	3	7	7
RatioCommentToCode	0	0	0	0

B. Threats to Validity

Construct validity of a study indicates that the correct operational measures are established for the concepts being studied [26]. Two main constructs in our work include the vulnerability aspect of a function, i.e., vulnerable or fixed, and the function-level code metrics. Building upon the existing research [11], [14], [20], we mine the code repository and consider the code version that fixes the vulnerability and the version immediately before that as the fixed and vulnerable versions, respectively. We calculate the function-level metrics using Understand by SciTools [21], a popular code metrics calculation tool widely used in both academia and industry [11]–[14]. Considering these aspects, our study design establishes the construct validity of the experiment. Sometimes, however, a vulnerability could be caused by interactions between multiple functions or improper dependency usage. Such situations are not taken into account in our experiment, which poses a limitation to the results.

Internal validity of an experiment suggests that conclusions are drawn accurately on the actual cause and effect [27]. Following the objective of this work, we collect data from

the functions that were known to be vulnerable and eventually fixed. Note that the function immediately prior to the fixing commit, not in the vulnerability contributing commit (VCC), is considered the vulnerable version of the function. We draw conclusions based on strong statistical evidence. Our data, however, is mostly limited to only those functions that were known to be vulnerable. It does not include the vast array of functions that are not yet found to be vulnerable. We suspect that the inclusion of such functions as the "clean code" (i.e., not-vulnerable) and the versions in VCCs as vulnerable functions may lead to different conclusions.

External validity of an empirical study establishes the domain to which the findings of the study can be generalized [26]. In this research, we consider five medium- to large-size open-source software (OSS) projects from various application domains, including video editing and OS. We have also run our analysis on individual subject systems (not reported in this paper) and the results remain the same. This work, however, includes open-source C/C++ projects only. Conducting experiment with software systems implemented in different programming languages is needed to further examine the external validity of our conclusions.

Reliability of an experiment indicates that repeated execution of the same operations will provide the same results [26]. Our data collection process is carefully curated following existing literature. We have also made the dataset available to the readers to promote replication. We believe that the findings of this research are reliable.

VI. Conclusion and Future Work

This paper conducts an empirical study to explore if the function-level code metrics can differentiate the vulnerable and fixed versions of a given function. We mine the repositories for five medium- to large-size OSS systems and collect function-level code metrics of the functions whose vulnerability was later fixed. A binary logistic regression [15] with Wald test [16] suggests that function-level code metrics may not be useful to predict if a given function contains vulnerability.

In the future, we plan to conduct further studies and uncover additional insights about any associations between functional-level software metrics and vulnerabilities. To that end, we plan to conduct experiments with vulnerability types and with balanced datasets where an equal number of data points will be collected from both the vulnerable functions and the functions that have been known to be free from vulnerabilities.

Acknowledgment

This research is partially supported by U.S. National Science Foundation (NSF) Award Number 2236953.

References

[1] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008, pp. 47–50.

[2] T. Bhowmik, A. Q. Do, H. Lam, M. R. Amin, G. L. Bradshaw, and N. Niu, "On the way to a framework for evaluating creativity in requirements engineering," in *2023 IEEE 24th International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2023, pp. 191–198.

[3] M. R. Amin and T. Bhowmik, "Information on potential vulnerabilities for new requirements: Does it help writing secure code?" in *2021 IEEE 29th International Requirements Engineering Conference (RE)*. IEEE, 2021, pp. 408–413.

[4] H. N. Security, "Cve count set to rise by 25% in 2024," https://bit.ly/4cMLrte, accessed: 2024-04-07.

[5] M. R. Amin and T. Bhowmik, "Existing vulnerability information in security requirements elicitation," in *2022 IEEE 30th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2022, pp. 220–225.

[6] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE software*, vol. 19, no. 1, pp. 42–51, 2002.

[7] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, vol. 24, pp. 637–673, 2019.

[8] H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan, "A study of c/c++ code weaknesses on stack overflow," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2359–2375, 2021.

[9] S. Imtiaz, M. R. Amin, A. Q. Do, S. Iannucci, and T. Bhowmik, "Predicting vulnerability for requirements," in *2021 IEEE 22nd International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2021, pp. 160–167.

[10] P. Pakshad, A. Shameli-Sendi, and B. Khalaji Emamzadeh Abbasi, "A security vulnerability predictor based on source code metrics," *Journal of Computer Virology and Hacking Techniques*, vol. 19, no. 4, pp. 615–633, 2023.

[11] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 216–227.

[12] K. Z. Sultana, V. Anu, and T.-Y. Chong, "Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach," *Journal of Software: Evolution and Process*, vol. 33, no. 3, p. e2303, 2021.

[13] K. Z. Sultana, C. B. Boyd, and B. J. Williams, "A software vulnerability prediction model using traceable code patterns and software metrics," *SN Computer Science*, vol. 4, no. 5, p. 599, 2023.

[14] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: dataset and exploratory study," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 37–44.

[15] F. E. Harrell, Jr and F. E. Harrell, "Binary logistic regression," *Regression modeling strategies: With applications to linear models, logistic and ordinal regression, and survival analysis*, pp. 219–274, 2015.

[16] A. A. Alkhalaf, "The impact of predictor variable (s) with skewed cell probabilities on the wald test in binary logistic regression," Ph.D. dissertation, University of British Columbia, 2017.

[17] L. H. Rosenberg and L. E. Hyatt, "Software quality metrics for object-oriented environments," *Crosstalk journal*, vol. 10, no. 4, pp. 1–6, 1997.

[18] M. R. Amin, T. Bhowmik, N. Niu, and J. Savolainen, "Environmental variations of software features: A logical test cases' perspective," in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. IEEE, 2023, pp. 192–198.

[19] NIST, "National vulnerability database," https://bit.ly/4atOvc5, accessed: 2024-04-10.

[20] K. Napier, T. Bhowmik, and S. Wang, "An empirical study of text-based machine learning models for vulnerability detection," *Empirical Software Engineering*, vol. 28, no. 2, p. 38, 2023.

[21] SciTools, "Understand: The software developer's multi-tool," https://scitools.com/, accessed: 2024-04-10.

[22] U. by SciTools, "Understand software metrics," https://documentation.scitools.com/pdf/metricsdoc.pdf, accessed: 2024-04-11.

[23] M. Y. Liu and I. Traore, "Empirical relation between coupling and attackability in software systems: a case study on dos," in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, 2006, pp. 57–64.

[24] 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.

[25] R. Longadge and S. Dongre, "Class imbalance problem in data mining review," *arXiv preprint arXiv:1305.1707*, 2013.

[26] R. K. Yin, *Case study research: Design and methods*. SAGE Publications, 2008, vol. 5.

[27] P. C. Cozby and S. C. Bates, *Methods in Behavioral Research*, 2012.