



RESEARCH ARTICLE

OPEN ACCESS

SECURITY-ORIENTED CODE METRICS FOR MACHINE LEARNING– BASED VULNERABILITY PREDICTION IN SOURCE CODE

Mohammed Zagane*¹

¹LABTEC-AI, Department of Computer Science, University of Mustapha Stambouli, P.O BOX, Mascara, Algeria.

¹<https://orcid.org/0000-0003-2508-0265>

Email: *m_zagane@univ-mascara.dz

ARTICLE INFO

Article History

Received: January 29, 2026

Reviewed: March 30, 2026

Accepted: April 30, 2026

Published: May 29, 2026

Keywords:

Vulnerability prediction,
Software metrics,
Software quality,
Prediction model,
Machine learning.

ABSTRACT

Automatic Vulnerability Prediction (AVP) has emerged as a promising approach for identifying security weaknesses in source code using machine learning techniques. While classic software metrics have been widely used for this purpose, they primarily capture general code attributes such as size and complexity and do not directly quantify security-relevant characteristics. This paper proposes a novel set of security-specific code metrics explicitly designed to measure vulnerability-prone properties of C/C++ programs. The proposed metrics focus on two major sources of vulnerabilities: tainted source usage and memory management and access practices. These metrics are extracted at the code slice level through static analysis and used as features to build vulnerability prediction models. We evaluate the effectiveness of the proposed metrics using multiple machine learning classifiers on a large-scale, publicly available dataset comprising over 420,000 labeled code slices. Experimental results show that the proposed metrics achieve strong and consistent performance across multiple vulnerability types, with particularly high effectiveness for memory-related vulnerabilities. A comparative analysis with classic software metrics demonstrates that security-specific metrics significantly improve vulnerability detection for pointer- and array-related weaknesses while remaining computationally efficient. These findings highlight the value of direct security measurement for enhancing automatic vulnerability prediction.



Copyright ©2026 by authors and Galileo Institute of Technology and Education of the Amazon (ITEGAM). This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

I. INTRODUCTION

Software metrics have long been used to quantify internal attributes of software systems, such as size, complexity, and coupling. Owing to their ability to provide objective and computable measurements, software metrics have been successfully applied in a wide range of software engineering and static code analysis tasks, including development cost estimation, change-proneness analysis, change impact analysis, defect prediction, and code clone detection. In the context of Automatic Vulnerability Prediction (AVP), several studies have leveraged software metrics as predictive features to identify vulnerable code entities at different granularity levels, such as files, functions, and code slices. The motivation for using software metrics in vulnerability prediction is twofold. First, metrics offer a practical means to quantify structural and behavioral characteristics of software, including complexity and inter-module dependencies. Second, empirical evidence suggests a correlation between these characteristics and the presence of vulnerabilities. Complex and highly coupled code typically requires greater effort from developers and reviewers and poses increased challenges for static and dynamic analysis tools, thereby increasing the likelihood of implementation errors and security flaws. In addition, metric-based approaches are computationally efficient and easily integrated with traditional machine learning techniques, enabling the construction of Vulnerability Prediction Models (VPMs) without requiring sophisticated code representations or the substantial computational resources demanded by recent deep learning approaches [1]. Despite these advantages, classic software metrics remain indirect indicators of security. While they are effective at capturing general software quality attributes, they are not explicitly designed to measure security-relevant characteristics. As a result, their ability to precisely capture the root causes of vulnerabilities is inherently limited.

Several prior studies have highlighted the need for security-specific code metrics that directly quantify aspects of code related to security weaknesses, such as unsafe data handling and improper memory usage. These works emphasize that tailored metrics are essential for capturing vulnerability-prone patterns that are not adequately reflected by traditional measures of complexity or size. To address this research gap, this paper proposes a novel set of security-specific code metrics explicitly designed to quantify critical code characteristics that influence software security. The proposed metrics focus on two fundamental and widely recognized sources of vulnerabilities in C/C++ programs: tainted source usage and memory management and access practices. By directly measuring properties such as unsanitized tainted variables, unsafe pointer operations, and improper memory allocation and access patterns, the proposed metrics provide a more targeted and interpretable assessment of vulnerability-prone code. Furthermore, this study investigates the effectiveness of these security-specific metrics as features for building vulnerability prediction models using machine learning algorithms. The proposed approach is evaluated on a large-scale, publicly available dataset of labeled code slices, using multiple classifiers and standard evaluation metrics. Experimental results demonstrate that the proposed metrics are particularly effective for predicting memory-related vulnerabilities and complement classic software metrics in other vulnerability categories.

Overall, this work contributes a practical and computationally efficient methodology for enhancing automatic vulnerability prediction and offers new insights into the role of security-specific measurement in software security analysis. The remainder of this paper is organized as follows. Section II reviews related work on vulnerability prediction, including the use of traditional software metrics and fine-grained code representations. Section III presents the proposed security-specific code metrics, detailing their design for capturing tainted source usage and memory management practices, as well as the methodology for extracting these metrics from source code. Section IV describes the experimental setup, including dataset preparation, model training, and evaluation protocols. Section V presents and discusses the results, including classifier performance, evaluation across different vulnerability types, and a comparison with classic metrics, followed by a discussion of the approach's limitations. Finally, Section VI concludes the paper and outlines directions for future research aimed at extending and generalizing the proposed metrics for broader applications in source code vulnerability prediction.

II. RELATED WORK

AVP has been extensively studied as an effective means to reduce the cost and effort associated with manual security analysis. Inspired by defect prediction research [2-11], many studies have adopted traditional software metrics to quantify internal code attributes such as size, complexity, coupling, and cohesion, and to use them as predictors in machine learning-based Vulnerability Prediction Models (VPMs). Early work by Shin and Williams [12-14] demonstrated that complexity metrics can distinguish vulnerable from non-vulnerable functions, providing initial empirical evidence of the relationship between code complexity and security flaws. Subsequent studies [15-20] investigated different families of metrics, including complexity, coupling, cohesion, code churn, and developer activity metrics, across various software systems and vulnerability datasets. Despite these promising results, the performance of metric-based VPMs has remained inconsistent, often suffering from high false-positive or false-negative rates. According to [21] identified several fundamental challenges in applying VPMs, including coarse-grained vulnerability labels, data imbalance, and the lack of security-aware features, which collectively limit the practical effectiveness of these approaches. One of the main limitations of early metric-based VPMs is their reliance on coarse-grained program entities, such as files or functions, which restricts the ability to precisely localize vulnerabilities. To overcome this limitation, researchers have explored fine-grained code representations, including program slices and vulnerability-related code fragments.

Program slicing, originally introduced by [22], has been successfully applied to bug classification and vulnerability analysis by isolating statements that are semantically related to a given computation or security-sensitive operation. Building on this concept, introduced Code Gadgets (CGs) [23], Syntax-based Vulnerability Candidates (SyVCs), and Semantics-based Vulnerability Candidates (SeVCs) [24] as fine-grained representations explicitly tailored for vulnerability detection. These representations capture compact sets of statements that preserve the syntactic or semantic characteristics of vulnerable code. Deep learning-based approaches such as VulDeePecker [23] and SySeVR [24] demonstrated that leveraging such vulnerability-aware representations significantly improves detection accuracy while enabling vulnerability localization. Motivated by the effectiveness of fine-grained representations, by [1], [25] investigated whether code metrics could be applied at a finer level of granularity to improve vulnerability prediction. Instead of computing metrics over entire files or functions, metrics were extracted directly from Code Gadgets and SeVCs, which preserve the semantic context of vulnerabilities. The experimental results showed that traditional metrics, including Lines of Code (LOC), McCabe complexity, and Halstead metrics, can effectively characterize vulnerable code fragments when applied at this level. Moreover, these metric-based VPMs achieved competitive precision and recall while enabling accurate localization of vulnerable code.

Despite the demonstrated effectiveness of traditional software metrics, researchers [17], [21] have emphasized that these metrics are inherently indirect indicators of vulnerabilities. They primarily capture structural and syntactic properties of code rather than explicitly modeling security-relevant behaviors. As a result, researchers in [26] have proposed vulnerability-oriented metrics, to better capture security concerns. Despite this effort, there remains a lack of a comprehensive and structured set of metrics explicitly designed to quantify multiple security dimensions, such as tainted data propagation, improper memory management, and misuse of security-sensitive constructs. This limitation constrains the ability of VPMs to fully capture the semantics of vulnerabilities, even when fine-grained code representations are employed. The present study extends this body of work by introducing a novel set of security-specific code metrics explicitly designed to quantify critical security-related properties of the code. Unlike traditional metrics, which serve as indirect proxies for vulnerability-prone code, the proposed metrics directly model security-relevant behaviors, including tainted sources, memory management practices, and other vulnerability-inducing patterns. By leveraging these metrics as features in machine learning-based VPMs and evaluating them on publicly available labeled datasets, this work bridges the gap between structural software analysis and security-aware vulnerability prediction. In doing so, it builds upon prior metric-based and fine-grained approaches while advancing toward a more precise and semantically grounded assessment of software security.

III. MATERIALS AND METHODS

III.1 APPROACH OVERVIEW

The problem of vulnerability prediction is formulated as a supervised binary classification task. Let $C = \{c_1, c_2, \dots, c_n\}$ represent a corpus of code entities (e.g., code slices, functions, or files), where each code entity c_i is associated with a set of security-specific metrics and a binary label indicating its vulnerability status. For each code entity c_i , we extract a feature vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{im}) \in \mathbb{R}^m$, where m represents the number of proposed metrics. Each feature x_{ij} quantifies a particular security aspect of the code entity, such as the number of unsanitized tainted variables or the count of uninitialized pointers. The vulnerability status of each code entity is represented by a binary label $y_i \in \{0,1\}$, where:

- $y_i = 1$ indicates that the code entity is vulnerable
- $y_i = 0$ indicates that the code entity is clean (non-vulnerable)

The objective is to learn a classification function $f: \mathbb{R}^m \rightarrow \{0,1\}$ that maps the feature vector $\mathbf{x}_i$ to a predicted vulnerability label $\hat{y}_i: \hat{y}_i = f(\mathbf{x}_i; \theta)$. Where θ denotes the parameters of the classification model learned from the training dataset $D_{\text{train}} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$. In this study, we evaluate three classification algorithms, including XGBoost, Random Forest, and Logistic Regression. The proposed security-specific metrics are computed at the program slice granularity level, where each slice represents a semantically meaningful code fragment associated with a potential vulnerability. Model performance is evaluated using Recall (R), Precision (P), False Positive Rate (FPR), and False Negative Rate (FNR), as described in Section 4.3.

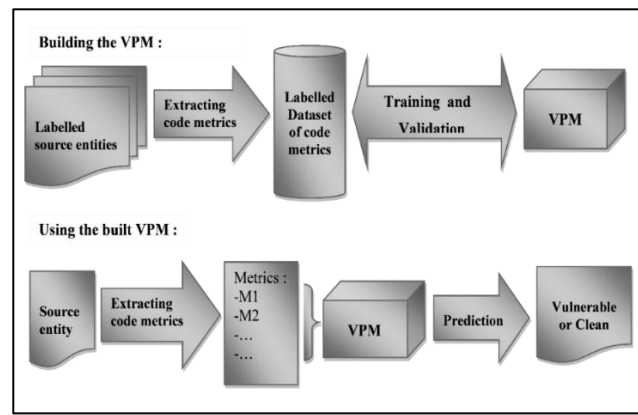


Figure 1: Approach overview.
Source: Author, (2026).

III.2 PROPOSED METRICS

Our goal is to address AVP in a direct manner by proposing a set of metrics that quantitatively express critical security-related properties of the code. In this initial study, we focus on vulnerability prediction for C/C++ programs. Two major security aspects are considered: use of tainted sources and memory management and access. The proposed metrics are detailed in the following subsections.

III.2.1 Use of Tainted Sources

A tainted source represents any external source of untrusted data. In C/C++ programs, tainted values may originate from command-line arguments, user input functions (e.g., `fgetc`, `getc`, `getchar`), file input, or environment variables. The use of tainted values without proper sanitization may lead to critical vulnerabilities such as buffer overflows, command injection, and other memory-related flaws, particularly when such values are passed to unsafe library functions (e.g., `strcpy`, `strcat`). Table 2 presents the proposed metrics for quantifying the use of tainted sources.

Table 1: Use of tainted sources metrics.

Metric	Description
#Tainted_Var	Total number of tainted variables
#San_Tainted_Var	Number of sanitized tainted variables
#Unsan_Tainted_Var	Number of unsanitized tainted variables
#San_Passed_To_Unsec_Funcs	Number of sanitized tainted variables passed to unsafe library functions
#Unsan_Passed_To_Unsec_Funcs	Number of unsanitized tainted variables passed to unsafe library functions

Source: Author, (2026).

III.2.2 Memory Management and Access

Memory management encompasses dynamic allocation, memory access, and deallocation. In C/C++, dynamic memory allocation is performed using functions such as `malloc` or `new`. Improper memory allocation or deallocation may lead to memory leaks, dangling pointers, or use-after-free vulnerabilities. Memory access relies heavily on pointers and arrays. Unsafe pointer operations (e.g., uninitialized pointers, pointer arithmetic, type casting) and improper array access (e.g., out-of-bounds access) are common sources of vulnerabilities due to the lack of inherent bounds checking in C/C++. Table 2 summarizes the proposed metrics related to memory management and access.

Table 2: Memory management and access metrics.

Metric	Description
#Allocation_Reallocation	Number of dynamic memory allocation and reallocation operations
#Deallocation	Number of memory deallocation operations
#Total_of_Pointers	Total number of pointer variables
#Uninitialized_Pointers	Number of pointers not initialized before use
#Initialized_Pointers	Number of properly initialized pointers
#Pointers_Arithmetic	Number of pointer arithmetic operations
#Casted_Pointers	Number of pointer type casting operations
#Total_Arrays	Total number of array declarations
#Fixed_Size_Arrays	Number of arrays with fixed size
#Variable_Size_Arrays	Number of arrays with variable size
#Use_of_Size_T_As_Array_Size	Number of arrays using size_t for size specification
#Array_Boundary_Check	Number of array accesses with explicit boundary checking

Source: Author, (2026).

IV. EXPERIMENTAL EVALUATION

IV.1 DATA PPREPARATION

Since vulnerability prediction is treated as a supervised binary classification problem, labeled datasets are required. Based on granularity level, dataset quality, size, and vulnerability type coverage, we selected the Semantic Vulnerability Candidates (SeVCs) dataset proposed by [24]. The dataset contains 420,627 code entities extracted from 1,591 open-source projects from the NVD [27] and 14,000 programs from SARD [28]. It includes 56,395 vulnerable and 364,232 clean code entities. The dataset is divided into four parts according to four vulnerability types [1]:

1. Library/API Function Call: vulnerabilities that are related to library/API function calls.
2. Array Usage: vulnerabilities that are related to arrays (e.g., improper use in array element access, array address arithmetic, address transfer as a function parameter).
3. Pointer Usage: vulnerabilities that are related to pointers (e.g., improper use in pointer arithmetic, reference, address transfer as a function parameter).
4. Arithmetic Expression: vulnerabilities that are related to improper arithmetic expressions (e.g., an integer overflow).

Table 3 summarizes statistics of the used dataset.

Table 3: Descriptive statistics about the CGs dataset.

	Number of vulnerable slices	Number of clean slices	Total
Library/API Function Call (FC)	13603	50800	64403
Array Usage (AU)	10926	31303	42229
Pointer Usage (PU)	28391	263450	291841
Arithmetic Expression (PU)	3475	18679	22154

Source: Author, (2026).

To prepare the metrics dataset that we used to train and validate VPMs, we implemented a custom static analysis tool in Java. As a first step, the source code of each entity of the SeVCs dataset is transformed into the srcML [29], [30] representation, which provides a lightweight XML-based format that preserves both the syntactic structure and lexical information of the code, thereby facilitating accurate and language-aware parsing. The tool operates at the program slice granularity level, where each slice represents a semantic code fragment associated with potential vulnerabilities. For each slice, the corresponding srcML representation is parsed to construct an abstract view of the code structure, including declarations, statements, expressions, and control constructs. Based on this representation, the proposed metrics are computed by traversing the srcML tree and identifying security-relevant constructs. Each instance in the generated dataset consists of the features vector and the class label as described in the approach overview sub-section.

IV.2 MODEL TRAINING AND EVALUATION

We used the metric dataset described in the previous subsection to train and evaluate the classification models for each vulnerability types: FC, AU, PU and AE. We evaluated three widely used classification algorithms that have demonstrated strong performance in prior vulnerability prediction research: XGBoost, Random Forest, and Logistic Regression. These classifiers were selected to cover complementary learning paradigms, including ensemble boosting, ensemble bagging, and linear classification. To mitigate the impact of class imbalance, the Synthetic Minority Over-sampling Technique (SMOTE) was applied to the training data only, ensuring that no information from the test folds was leaked into the training process. Model hyperparameters were optimized using grid search combined with cross-validation, following the experimental protocol adopted in related work. To ensure robust and unbiased evaluation, we employed 10-fold cross-validation, where the dataset is partitioned into ten disjoint folds. In each iteration, nine folds are used for training and one fold for testing, and the process is repeated until each fold has served as the test set exactly once. The final performance results correspond to the average values obtained across all folds. Model performance is assessed using four standard evaluation metrics widely used in vulnerability prediction research: Recall (R), Precision (P), False Positive Rate (FPR), and False Negative Rate (FNR).

Recall measures the ability of the model to correctly identify vulnerable code slices, while Precision quantifies the proportion of predicted vulnerable slices that are truly vulnerable. High values of Recall and Precision indicate better model performance. In contrast, FPR measures the proportion of clean code slices incorrectly classified as vulnerable, and FNR measures the proportion of vulnerable slices that are incorrectly classified as clean; therefore, lower values of FPR and FNR indicate better performance. These metrics are formally defined as follows:

$$\text{Recall} \\ R = \frac{TP}{TP + FN} \quad (1)$$

$$\text{Precision} \\ P = \frac{TP}{TP + FP} \quad (2)$$

$$\text{False Positive Rate} \\ FPR = \frac{FP}{FP + TN} \quad (3)$$

$$\text{False Negative Rate} \\ FNR = \frac{FN}{TP + FN} \quad (4)$$

In the context of vulnerability prediction, the output of a Vulnerability Prediction Model (VPM) can be interpreted using four possible outcomes. A True Positive (TP) corresponds to a vulnerable code slice that is correctly classified as vulnerable. A True Negative (TN) refers to a clean (non-vulnerable) code slice that is correctly classified as clean. A False Positive (FP) represents a clean code slice that is incorrectly classified as vulnerable, leading to a false alarm. Conversely, a False Negative (FN) denotes a vulnerable code slice that is incorrectly classified as clean, which is particularly critical in security contexts as it results in missed vulnerabilities. This evaluation framework provides a balanced assessment of the models by jointly considering vulnerability detection capability and false alarm reduction, which is essential for practical deployment in security analysis and software quality assurance contexts.

V. RESULTS AND DISCUSSIONS

This section presents and discusses the experimental results obtained using the proposed security-specific metrics. The analysis is structured around four key aspects: the impact of the classification algorithm on VPMs performance, the effectiveness of the proposed metrics across different vulnerability types, a comparative evaluation against classic software metrics used in prior work, and the limitations of the proposed approach.

V.1 EVALUATING DIFFERENT CLASSIFIERS

To evaluate the impact of the classification algorithm on vulnerability prediction performance, we fixed the dataset to the FC vulnerability category and varied the learning algorithm. This vulnerability type was selected because it involves security-sensitive function usage combined with control-flow and data-flow complexity, making it a representative case for classifier comparison. We evaluated three widely used machine learning algorithms: XGBoost, Random Forest, and Logistic Regression, under identical experimental settings, including the same feature set, SMOTE-based data balancing, and 10-fold cross-validation. The performance comparison is reported in Table 4.

Table 4: Performances using different classifiers.

Classifier	R (%)	P (%)	FPR (%)	FNR (%)
XGBoost	0.7306	0.8050	0.2759	0.2694
Logistic Regression	0.6637	0.7425	0.4318	0.3363
Random Forest	0.6980	0.8017	0.2522	0.3020

Source: Author, (2026).

The results show that XGBoost achieves the best overall performance, with a recall of (73.06%) and a precision of (80.50%), indicating a strong ability to detect vulnerable code slices while maintaining a reasonable false alarm rate. In contrast, Logistic Regression exhibits the weakest performance, with the lowest recall (66.37%) and the highest false positive and false negative rates. This confirms that linear models struggle to capture the complex and non-linear relationships between security-specific metrics and vulnerability presence. Random Forest achieves competitive precision (80.17%) and a lower false positive rate compared to XGBoost, but at the cost of a higher false negative rate. This indicates that Random Forest is more conservative and misses a larger proportion of vulnerable code entities. The superior performance of XGBoost can be attributed to its gradient boosting mechanism, which incrementally focuses on difficult instances and incorporates regularization to mitigate overfitting. Based on these results, XGBoost was selected as the primary classifier for the subsequent experiments.

V.2 EVALUATING ACROSS VULNERABILITY TYPES

After selecting XGBoost as the most effective classifier, we evaluated its performance across four vulnerability types: Library/API Function Call, Arithmetic Expression, Array Usage, and Pointer Usage. This analysis assesses both the generalizability of the proposed metrics and their effectiveness for different classes of vulnerabilities. The results summarized in Table 5 demonstrate that the proposed security-specific metrics achieve consistently strong performance across all vulnerability categories, with particularly high effectiveness for memory-related vulnerabilities. Pointer Usage vulnerabilities achieve the highest recall (83.94%) and precision (90.82%), as well as the lowest false negative rate. This performance can be directly attributed to the relevance of the proposed memory-related metrics, which explicitly capture risky pointer operations such as uninitialized pointers, pointer arithmetic, and unsafe casting.

Table 5: Performance across vulnerability types.

Vulnerability Type	R (%)	P (%)	FPR (%)	FNR (%)
Library/API Function Call	0.7306	0.8050	0.2759	0.2694
Arithmetic Expression	0.7827	0.8600	0.2389	0.2173
Array Usage	0.8106	0.7841	0.3660	0.1894
Pointer Usage	0.8394	0.9082	0.2494	0.1606

Source: Author, (2026).

Array Usage vulnerabilities also show high recall (81.06%), indicating effective detection of array-related vulnerabilities. However, precision is slightly lower, resulting in a higher false positive rate. This behavior can be explained by the overlap between safe and unsafe array usage patterns in C/C++, where benign code may exhibit metric values similar to vulnerable code. Arithmetic Expression vulnerabilities exhibit balanced performance, with a recall of (78.27%) and a precision of (86.00%). These results suggest that the proposed metrics capture relevant patterns associated with integer overflows and unsafe arithmetic operations, although their effectiveness is somewhat lower than for memory-related vulnerabilities. Library/API Function Call vulnerabilities achieve the lowest performance among the four categories. While the results remain acceptable, they suggest that such vulnerabilities may depend more heavily on the semantic behavior of external functions and APIs, which is only partially captured by the current metric set. Overall, this evaluation confirms that the proposed metrics are particularly well-suited for detecting memory-related vulnerabilities, which are among the most critical and prevalent security issues in C/C++ software.

V.3 COMPARISON WITH CLASSIC CODE METRICS

To assess the added value of the proposed security-specific metrics, we conducted a comparative evaluation against classic code metrics used in related work that investigated the same granularity level (slice) [1], [25]. The classic metrics include complexity, size, and structural properties. Both metric sets were evaluated under identical conditions using the XGBoost classifier, SMOTE balancing, and 10-fold cross-validation to ensure a fair comparison. Results presented in Table 6 reveal that security-specific metrics significantly outperform classic metrics for memory-related vulnerabilities, particularly for pointer and array usage. For pointer usage vulnerabilities, security-specific metrics achieve substantially higher recall and lower false negative rates while maintaining comparable precision. This indicates that metrics explicitly designed to quantify pointer initialization, pointer arithmetic, and memory manipulation are more effective at identifying pointer-related vulnerabilities than general-purpose complexity metrics.

Table 6: Comparison with classic code metrics.

Vulnerability Type	Metrics Type	R (%)	P (%)	FPR (%)	FNR (%)
Library/API Function Call	Security-Oriented	0.7306	0.8050	0.2759	0.2694
	Classic	0.7056	0.8373	0.1555	0.2944
Arithmetic Expression	Security-Oriented	0.7827	0.8600	0.2389	0.2173
	Classic	0.8106	0.9287	0.0534	0.1894
Array Usage	Security-Oriented	0.8106	0.7841	0.3660	0.1894
	Classic	0.7758	0.8386	0.1650	0.2242
Pointer Usage	Security-Oriented	0.8394	0.9082	0.2494	0.1606
	Classic	0.7565	0.9074	0.1399	0.2435

Source: Author, (2026).

For array usage vulnerabilities, security-specific metrics also achieve higher recall, detecting more vulnerable code entities, although this improvement comes at the cost of a higher false positive rate. In security-critical contexts, this trade-off is often acceptable, as missing vulnerabilities is typically more costly than investigating false alarms. In contrast, classic metrics remain competitive or superior for arithmetic expression vulnerabilities, achieving higher precision and lower false positive rates. This suggests that traditional complexity measures may better capture the characteristics of arithmetic-heavy code where vulnerabilities such as integer overflows often occur. These findings highlight the complementary nature of security-specific and classic metrics. While security-specific metrics excel at capturing low-level, security-relevant code properties, classic metrics remain useful for vulnerability types that correlate strongly with code complexity.

V.4 LIMITATIONS

Despite the promising results, several limitations of the proposed approach should be acknowledged, which also highlight interesting open research directions for future work.

First, this study focuses exclusively on C/C++ programs, and the proposed metrics are specifically designed for memory-unsafe languages; consequently, their direct applicability to memory-safe languages such as Java or Python is limited. Extending the approach to other programming languages would require the design of language-specific security metrics. Second, the current metric set primarily targets tainted source usage and memory management practices. Other important security aspects, including concurrency-related vulnerabilities (e.g., race conditions and deadlocks), cryptographic misuse, and authentication or authorization flaws, are not addressed and represent promising directions for future metric design. Third, the proposed approach relies on static code analysis, which inherently cannot capture runtime-dependent behaviors or vulnerabilities that only manifest under specific execution conditions; as a result, certain classes of vulnerabilities may remain undetected. Finally, although SMOTE is employed to mitigate class imbalance, synthetic oversampling may not fully capture the diversity and complexity of real-world vulnerable code patterns, potentially affecting model generalization. Investigating alternative imbalance-handling strategies, such as cost-sensitive learning or advanced ensemble techniques, constitutes another avenue for future research.

VI. CONCLUSION

This paper presented a novel set of security-specific code metrics designed to directly quantify security-relevant characteristics of C/C++ programs for automatic vulnerability prediction. Unlike traditional software metrics that focus on general attributes such as size and complexity, the proposed metrics explicitly capture tainted source usage and memory management behaviors known to cause security vulnerabilities. Experimental evaluation on a large-scale, real-world dataset demonstrates that the proposed metrics achieve strong and consistent performance, particularly for memory-related vulnerabilities such as pointer and array misuse. The results further show that XGBoost is the most effective classifier for exploiting these metrics, highlighting the importance of using non-linear learning models. A comparative analysis with classic software metrics confirms that security-specific and classic metrics are complementary: security-specific metrics excel at detecting memory-related vulnerabilities, while classic metrics remain effective for arithmetic expression vulnerabilities. These findings suggest that hybrid approaches combining both metric families represent a promising direction for future research. Overall, this work provides strong empirical evidence that security-specific metrics constitute a practical, interpretable, and computationally efficient solution for vulnerability prediction, offering a compelling alternative to complex deep learning approaches. Future work will focus on extending the metric set, exploring hybrid models, and evaluating the approach across additional programming languages and industrial settings.

VII. AUTHOR'S CONTRIBUTION

Conceptualization: Mohammed Zagane.

Methodology: Mohammed Zagane.

Investigation: Mohammed Zagane.

Discussion of results: Mohammed Zagane.

Writing – Original Draft: Mohammed Zagane.

Writing – Review and Editing: Mohammed Zagane.

Resources: Mohammed Zagane.

Supervision: Mohammed Zagane.

Approval of the final text: Mohammed Zagane.

VIII. REFERENCES

- [1] M. Zagane, M. K. Abdi, and M. Alenezi, "A New Approach to Locate Software Vulnerabilities Using Code Metrics," *International Journal of Software Innovation*, vol. 8, no. 3, pp. 82–95, Jul. 2020, doi: 10.4018/IJSI.2020070106.
- [2] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–14, 2007, doi: 10.1109/TSE.2007.10.
- [3] T. Zimmermann and N. Nagappan, "Predicting Subsystem Defects using Dependency Graph Complexities," 2007.
- [4] V. Singh, V. Bhattacharjee, and S. Bhattacharjee, "An analysis of dependency of coupling on software defects," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 1, p. 1, Jan. 2012, doi: 10.1145/2088883.2088899.
- [5] B. Turhan, "Software Defect Prediction Modeling," *Proceedings of the 2nd International Doctoral Symposium on Empirical Software Engineering (IDoESE 2007)*, pp. 90–95, 2007.
- [6] B. Turhan, A. Bener, and T. Menzies, "Nearest neighbor sampling for cross company defect predictors," in *Proceedings of the 1st International Workshop on Defects in Large Software Systems (DEFACTS'08)*, 2008, p. 26. doi: 10.1145/1390817.1390824.
- [7] E. Sreedevi, Y. Prasanth, E. Sreedevi, and Y. Prasanth, "A Novel Multi-Ensemble based Feature Selection and Defect Prediction Model on Software Defect Projects," vol. 9, no. 40, pp. 827–836, 2016.
- [8] C. Tantithamthavorn et al., "A review of process metrics in defect prediction studies," *Methods of Applied Computer Science*, vol. 1, no. 5, pp. 133–145, 2015, doi: 10.1007/978-3-319-26285-7.
- [9] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Softw Pract Exp*, vol. 41, no. 5, pp. 579–606, Apr. 2011, doi: 10.1002/spe.1043.
- [10] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Inf Softw Technol*, vol. 59, pp. 170–190, 2015, doi: 10.1016/j.infsof.2014.11.006.

- [11] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput*, vol. 22, no. S4, pp. 9847–9863, Jul. 2019, doi: 10.1007/s10586-018-1696-z.
- [12] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in *Proceedings of the 4th ACM workshop on Quality of protection - QoP '08*, New York, New York, USA: ACM Press, 2008, p. 47. doi: 10.1145/1456362.1456372.
- [13] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*, New York, New York, USA: ACM Press, 2008, p. 315. doi: 10.1145/1414004.1414065.
- [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, 2011, doi: 10.1109/TSE.2010.81.
- [15] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista," in *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, 2010, pp. 421–428. doi: 10.1109/ICST.2010.32.
- [16] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?," in *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, New York, New York, USA: ACM Press, 2010, p. 1963. doi: 10.1145/1774088.1774504.
- [17] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 23–33, 2014, doi: 10.1109/ISSRE.2014.32.
- [18] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*, New York, New York, USA: ACM Press, 2016, pp. 1415–1421. doi: 10.1145/2851613.2851777.
- [19] S. Moshtari, A. Sami, and M. Azimi, "Using Complexity Metrics For Within-Project And Cross-Project Software Vulnerability Prediction," wpengine.netdna-cdn.com, pp. 1–30, [Online]. Available: http://1jaw8k1wf8wx44wvqz4aeb5b2pw.wpengine.netdna-cdn.com/kasperskyacademy/files/2013/06/Sara_Moshtari.pdf
- [20] M. Zagane and M. K. Abdi, "Evaluating and Comparing Size, Complexity and Coupling Metrics as Web Applications Vulnerabilities Predictors," *International Journal of Information Technology and Computer Science(IJITCS)*, vol. 11, no. 7, pp. 35–42, 2019, doi: 10.5815/ijitcs.2019.07.05.
- [21] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security - HotSoS '15*, New York, New York, USA: ACM Press, 2015, pp. 1–9. doi: 10.1145/2746194.2746198.
- [22] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984, doi: 10.1109/TSE.1984.5010248.
- [23] Z. Li et al., "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA: Internet Society, 2018. doi: 10.14722/ndss.2018.23158.
- [24] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," pp. 1–13, Jul. 2018, doi: 10.21227/fhg0-1b35.
- [25] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep Learning for Software Vulnerabilities Detection Using Code Metrics," *IEEE Access*, vol. 8, pp. 74562–74570, 2020, doi: 10.1109/ACCESS.2020.2988557.
- [26] X. Du et al., "LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 60–71, Jan. 2019, doi: 10.1109/ICSE.2019.00024.
- [27] National Institute of Standards and T. (NIST), "National Vulnerability Database (NVD)," 2026.
- [28] National Institute of Standards and T. (NIST), "Software Assurance Reference Dataset (SARD)," 2026.
- [29] M. L. Collard and J. I. Maletic, "srcML 1.0: Explore, Analyze, and Manipulate Source Code."
- [30] H. Alomari et al., "srcSlice: A Tool for Efficient Static Forward Slicing," 2016, doi: 10.1145/2889160.2889173.