# A Comprehensive Software Vulnerability Dataset Based on OWASP Top Ten Standard

Moses Ndebugre*, Mahmoud Nabil*, Ahmad Patooghy*, and Abdolhossein Sarrafzadeh*
*Department of Electrical and Computer Engineering, North Carolina A&T State University, Greensboro NC, USA 27411.

2025 Silicon Valley Cybersecurity Conference (SVCC) | 979-8-3315-3429-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/SVCC65277.2025.11133646

***Abstract*—Software vulnerabilities pose a critical threat to the security and reliability of modern systems, with the increasing sophistication of cyberattacks necessitating robust detection and mitigation mechanisms. This work presents a novel, publicly available dataset collected to study and enhance software vulnerability, aligned with the OWASP Top Ten security standard. We systematically collect and differentiate vulnerable and fixed code samples from GitHub commits across multiple programming languages by leveraging an automated web crawling pipeline. Our methodology integrates keyword mapping, graph-based representation using Code Property Graphs (CPGs), and automated parsing with Joern to provide textual and structural insights into code vulnerabilities. Unlike existing datasets, our resource includes one-to-one mappings between vulnerable and fixed code versions, addressing scale, diversity, and structure limitations. We evaluate the dataset using five machine learning models: Naive Bayes, Random Forest, Support Vector Machine, Gradient Boosting, and Multilayer Perceptron, which demonstrate its utility for both binary and multiclass classification tasks. Our results reveal that Gradient Boosting and Random Forest outperform other models, highlighting the dataset's efficacy for real-world vulnerability study. This work establishes a foundation for further research into the automated collection of labeled vulnerability datasets and their application in improving cybersecurity tools and techniques.**

***Index Terms*—CWE, OWASP Top Ten, Web crawling, Machine learning, Code property graph**

## I. Introduction

From personal devices to critical infrastructure, software has been deeply integrated into various aspects of human lives [1], making software vulnerabilities one of the biggest cybersecurity challenges. A software vulnerability describes a weakness, flaw, or bug in an entity that can be exploited to compromise data security, system reliability, and privacy [2]. As software environments are becoming ever more complex and dependent on open-source components and third-party libraries, most of the software vulnerabilities stem from coding mistakes, integration of older codes, and inadequate testing [3]. Common programming errors, such as buffer overflows and injection flaws, allow points of entry into the system through which unauthorized access and manipulations can take place.

Addressing software vulnerabilities has resulted in extensive research and the development of detection and mitigation techniques [4]. Projects from the MITRE Corporation [5] such as the Common Vulnerabilities and Exposure (CVE) program offers a standardized approach to identifying vulnerabilities by providing unique identifiers to publicly known security vulnerabilities. Another standard maintained by the MITRE Corporation is the Common Weakness Enumerations (CWEs). CWE is a formal list of software and hardware weaknesses that are identified as vulnerabilities or flaws in system design, implementation, configuration, or in coding. It is one of the lists that are widely leveraged in identifying and mitigating vulnerabilities in applications and systems by security professionals, developers, and organizations. It aims to enhance the quality of software systems based on the potential offered by the detection, management, and resolution of security-related issues at different phases of the software development lifecycle.

Current vulnerability datasets mainly explore CVEs, which focus on specific vulnerabilities and can help with the immediate identification and remediation of vulnerabilities. However, we believe that creating vulnerability datasets based on CWEs is a must as it will help prevent vulnerabilities from arising in the first place by targeting the root causes. Existing benchmark datasets have a significant imbalance between the number of vulnerable data points and neutral data, with many more neutral data points than vulnerable ones. To address these issues, in this paper, we introduce a balanced vulnerability dataset based on CWE vulnerabilities borrowed of the Open Worldwide Application Security Project (OWASP) [6]. Our dataset covers the following vulnerabilities: Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures, and Server-Side Request Forgery.

Since each CVE can often be traced back to one or more CWEs, we hypothesize that a CWE-based dataset will help generalize on vulnerabilities that have yet to be identified. Accordingly, this paper tries to answer the following research questions (RQs):

- **RQ1:** Can we construct a dataset for CWE without supervised human annotation?
- **RQ2:** How can inter-procedural vulnerabilities be addressed in a dataset?
- **RQ3:** How does multiclass classification in vulnerability detection enhance our understanding of specific vulnerability types?

The collection of our dataset and its preprocessing is entirely automated, and thus, we do not experience inter-rater errors as seen in previous works [7]. We analyze our dataset by comparing it with benchmark datasets and perform several evaluations. Our contributions are listed as follows:

- **Public Availability and Accessibility:** Unlike previous datasets, which are often proprietary or limited in scope [8], our dataset is publicly available on GitHub and can be accessed at https://shorturl.at/7tb5Z. All scripts and code used in the dataset generation and preprocessing are also provided, ensuring full transparency and reproducibility of our results.
- **Detailed Mapping of Vulnerable and Fixed Code Versions:** In contrast to [9], which has very few vulnerable codes compared to neutral codes, our dataset offers a precise, organized mapping for vulnerable codes and their fixes. This provides an enhanced resource for training and evaluating models that require both vulnerable and fixed versions of code snippets.
- **Graph-Based Representation of Code for Enhanced Analysis:** We include a graph representation of all code files in our dataset, facilitating more complex, structure-based analyses not possible with text-only datasets like [10]. This approach supports graph-based machine learning models, which have shown promise in identifying structural vulnerabilities in code.

The remainder of this paper is structured as follows: Section II reviews related work on vulnerability detection approaches and datasets. Section III details the methodology used, including keyword mapping, web crawling, and dataset construction. Section IV characterizes the dataset, discussing its format and characteristics. Section V outlines the evaluation process, covering the experimental setup and metrics. Section VI presents the results of machine learning experiments, and Section VII offers concluding remarks and explores potential future research directions.

## II. Related Work

In this section, we cover existing approaches to vulnerability detection, tools, and techniques applied, and the datasets supporting research. We start by reviewing the main classes of tools and methods in use: static analysis and machine learning models, which have been applied to identify vulnerabilities in code. Afterwards, we review datasets supporting research by offering structured sources of vulnerable code; we assess the design, diversity, and limitations. We finally conclude by pointing out the gaps in current approaches and how our work tries to fill these gaps.

### A. Vulnerability Detection Tools and Techniques

Previous works have looked at identifying vulnerabilities in code using static analysis tools. While SAST tools are great, they often have high false positives [16]. Also, previous works focus on a binary solution to identifying vulnerabilities, without specifying the type of vulnerability found.

Static analysis tools use predetermined rules often written as regular expression (regex) to find where certain patterns appear in code. Many works have used these tools to evaluate their dataset [17], [18] and also analyze AI-generated dataset [19]. Bhuiyan et al. [7] identifies patterns from the abstract syntax tree representation of codes to develop a tool that trace potential exploits and analyze attacks once they have occurred. [20] uses data flow analysis to track how data moves through a program, which helps detect issues like taint tracking [21], where untrusted data flows into sensitive areas.

Empirical studies have been conducted to assess frequently occurring security weaknesses and the most impactful ones in supervised learning projects. It was discovered that security issues often co-occur in scripts, implying that certain vulnerabilities tend to appear together [22]. It was also discovered that training datasets often include duplicated samples, class imbalances, and simplified vulnerability patterns, which fail to represent real-world complexities. For potential vulnerabilities imported from third-party libraries, Zhao et. al [23] proposed an evaluation study on Software Composition Analysis (SCA) tools to resolve dependencies found in scripts.

There have been studies of deep learning models for vulnerability detection. [11] presents current limits regarding the detection of vulnerabilities using deep learning methods and states that there is an important margin for improvement at the model design level. There is also high variability across model outputs, where different models have low agreement in their predictions, and performance does not always improve substantially with increased dataset size [24]. Steenhoe et al. [25] introduced an abstract dataflow embedding to encode information about variable definitions and their dataflow within a program's Control flow graph (CFG) as a means to effectively identify real-word vulnerabilities across unseen data. In retrospect, many studies that have proposed a solution to improving vulnerability detection at the model design level have used some variation of the Graph Neural Network (GNN) [14], [26]. Vera et al. [27] specifically addresses the challenges of identifying vulnerabilities and assessing the impact of dependency upgrades in open-source packages.

### B. Vulnerability Detection Datasets

Identification and analysis of security vulnerabilities first begins with its collection. Several studies have focused on creating structured, accessible datasets to help researchers explore patterns in vulnerable code, evaluate detection algorithms, and understand trends in vulnerability emergence and remediation. Other datasets capture real-world vulnerabilities by extracting code from open-source projects or using commits associated with fixing a vulnerability. For example, the Devign dataset [14] labeled vulnerabilities through manual annotation of vulnerability-fixing commits, primarily covering FFmpeg and QEMU. The problem is that manual labeling is extremely resource-intensive, hence constraining dataset size and diversity. One prominent example is REVEAL [11], a comprehensive framework that aggregates real-world vulnerabilities and their fixes from diverse open-source projects. REVEAL organizes data to support fine-grained analysis and incorporates metadata such as vulnerability type and patch information. The recently released DiverseVul dataset [15], stands out in its size and diversity but only covers vulnerable functions in 150 CWEs from over 295 new projects not covered by the prior datasets.

TABLE I
COMPARISON OF VULNERABILITY DETECTION FRAMEWORKS AND THEIR DATASETS

| Dataset | Vulnerability Ratio | Multiclass Vulnerability | Multi Language | Inter-procedural Vulnerability | Total files | Detection method |
|---|---|---|---|---|---|---|
| REVEAL [11] | 9.16% | ✗ | ✗ | ✗ | 18,169 | Graph-based |
| SySeVR [12] | 94.80% | ✗ | ✗ | ✓ | 15,591 | Syntax and Semantics-based |
| D2A ♦ [9] | 1.44% | ✗ | ✗ | ✓ | 1,295,623 | Differential analysis-based |
| FUNDED ♦ [13] | 50.00% | ✗ | ✓ | ✓ | 150,950 | Graph-based |
| DEVIGN [14] | 46.90% | ✗ | ✗ | ✗ | 58,965 | Graph-based |
| DIVERSEVUL ♦ [15] | 5.42% | ✗ | ✗ | ✗ | 349,437 | Function-based |
| This work | 42.17% | ✓ | ✓ | ✓ | 9,771 | Graph-based |

♦ The framework combines data from standard vulnerability databases such as NVD, CVE, Sard, httpd and libav.

Despite advances in both tools and datasets, existing approaches still face limitations. Tools often produce high false positives and are limited in detecting specific vulnerability types, hindering nuanced detection. Additionally, current datasets may not adequately represent the complexity of real-world vulnerabilities, often suffering from limited diversity and scalability. As shown in Table I, our dataset effectively addresses the limitations of existing solutions by providing ten different categories of vulnerabilities considering the various CWEs, vulnerable codes with their fixes covering multiple languages, and considered inter-procedural vulnerability. Vulnerabilities are inter-procedural when the statements that induce the vulnerability are in different functions. This positions our dataset as better at capturing the complexity and variability of vulnerabilities across different contexts.

## III. METHODOLOGY

Fig. 1 illustrates the pipeline utilized to gather both vulnerable and neutral (fixed) code samples for our dataset, ensuring a comprehensive coverage of critical security vulnerabilities as outlined in the OWASP Top Ten. This process begins by identifying all the categories listed in the OWASP Top Ten, which offers an effective roadmap to address the most pressing security risks impacting web applications today. Focusing on this standard allows our dataset to provide a foundation for detecting vulnerabilities that align with the most widely acknowledged security risks, fostering tools and insights for web application resilience.

### A. Keyword and Vulnerability Mapping

To align vulnerabilities with specific OWASP Top Ten categories, we created a targeted keyword list from each category's nomenclature and mapped Common Weakness Enumerations (CWEs). For instance, within the Broken Access Control category, CWE-275 (Permission Issues) represents a well-defined vulnerability, prompting the creation of keywords such as "Fixed CWE-275" and "Fixed Permission Issues" to ensure that all relevant instances are identified during our search. We relied on ChatGPT to generate these keywords by using the prompt :

> "If I want to create an English search query to search for all the CWEs under the OWASP Top 10: Broken Access Control category that have been fixed on GitHub with the aim of creating a dataset, what are the keywords to include in my query when searching on Github? Each query should be for a particular CWE."

We repeated this prompt for the other OWASP Top Ten categories. This method systematically standardizes search terms across categories and guarantees that only relevant vulnerabilities are targeted.

### B. GitHub Repository Search, Web Crawling and Data Collection

Using our structured keywords, we performed an exhaustive search on GitHub, which yielded two types of repositories: Type (1): Repositories containing isolated instances of a vulnerable code snippet paired with its respective fix. Type (2): Repositories containing multiple interdependent vulnerable code sections, each with corresponding fixes. This differentiation aligns with the methodology in Devign [14], where dependency-aware models show increased accuracy in capturing vulnerability patterns due to their ability to understand interrelations within code structures. A similar dependency-aware approach here enhances the dataset's ability to represent complex code interactions, allowing models trained on this dataset to learn nuanced patterns of vulnerability and mitigation.

We fetched the Commit IDs from GitHub and gave certain sets of search queries. The script uses the requests library to fetch GitHub's search results using our generated keywords. We used Selenium [28] and BeautifulSoup [29] to automate the collection and differentiation of vulnerable and fixed code from GitHub commit pages. Selenium enables automated navigation and interaction with web elements, allowing access to and expansion of all code changes on a commit page. After expanding the code, the script captures the page source and pass it to BeautifulSoup, which parses the HTML to locate specific code lines marked as either additions or deletions in the diff view.

As an example, Figs 2 and 3 are a pair of codes that resulted from the search for CWE-20, which is mapped to the OWASP Top Ten category: Injection. The code counts how many times a specific letter appears in a given text string by prompting the user for input. "gets(texto)" is an unsafe function because it does not perform bounds checking. This absence can result in

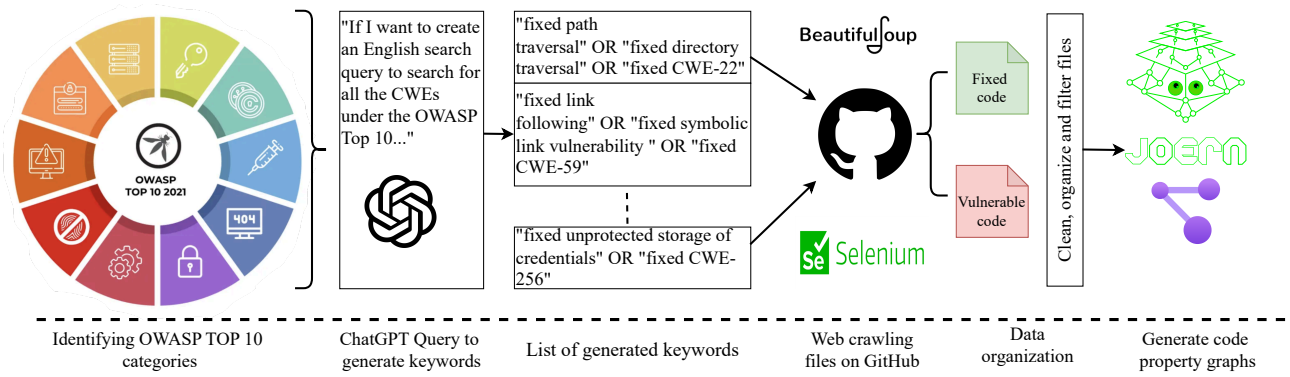


Fig. 1. The pipeline illustrates the stages for gathering vulnerable and fixed code sample datasets of the same vulnerability type. It results in a structured dataset for vulnerability analysis.

buffer overflow vulnerabilities, allowing an attacker to inject malicious data into memory beyond the allocated space for texto. The function "fgets()" reads at most 'n-1' characters, where 'n' is the size specified as the second argument (in this case, 80). This restriction ensures that the input does not exceed the designated buffer size, preventing overflow into adjacent memory.

```
#include <stdio.h>
main()
{
    char texto[80], letra;

    printf("TEXTO: ");
        gets ( texto ) ;
    printf("\nLETRA: ");
        scanf("%c", &letra);
    printf("\n\nO TEXTO POSSUI %d LETRAS %c",
        qtdletras(texto, letra, 0), letra);
}
```

Fig. 2. The red marking indicates the line of code that causes the vulnerability, thereby making the entire code vulnerable.

### C. Programming Language Detection

The crawled files from GitHub are stored as text files. To identify the programming language of these files, we used the Guesslang library [30], which detects programming languages in code snippets. Guesslang is a machine learning-based tool that determines the programming language by utilizing a pre-trained neural network model capable of recognizing language-specific features and patterns. This library has been trained on a large, labeled dataset of code snippets from various programming languages, allowing it to learn the distinctive characteristics of each language.

```
#include <stdio.h>
main()
{
    char texto[80], letra;

    printf("TEXTO: ");
        fgets ( texto , 80, stdin ) ;
    printf("\nLETRA: ");
        scanf("%c", &letra);
    printf("\n\nO TEXTO POSSUI %d LETRAS %c",
        qtdletras(texto, letra, 0), letra);
}
```

Fig. 3. The green marking indicates the line of code that resolves this type of vulnerability, making the entire code neutral.

### D. Joern Parsing and CPG Exporting

Joern [31] is employed in the creation of CPGs for our dataset. It is a well-known source code translation tool that creates graph-based representations for the effective analysis of code. This process allowed us to capture comprehensive program semantics in syntax, control flow, and data dependencies, which are important features in accurate vulnerability detection. The steps involved are detailed below.

❍ **Parsing Source Code:** Joern parses the source code in each sample into basic entities using language-specific syntax. Currently, Joern supports these languages: C/C++, Java, JavaScript, Python, x86/x64, JVM Bytecode, Kotlin, PHP, Go, Ruby, Swift, and C#. Each sample code is parsed and the elements are transformed into various generations of graphs, each mapping to a different aspect of the program.

❍ **Combining Graphs into a Composite CPG:** Joern combines several graphs, such as the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Control Flow Graph (CFG), into a single composite CPG by keeping the AST nodes as the primary structure, but this is enriched with edges coming from both the CFG and DFG.

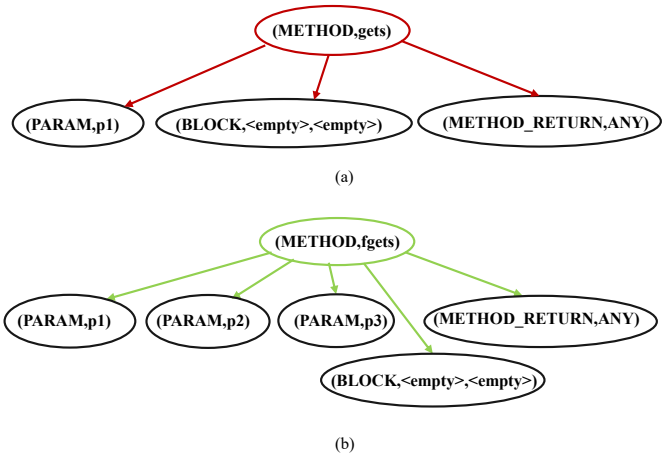


Fig. 4. The abstract syntax tree showing the data structure of the methods, ”gets” and ”fgets” within the codes

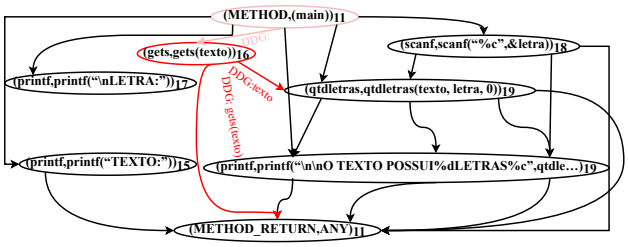


Fig. 5. The data flow graph showing how data moves through the method, ”main()” within the vulnerable code.

❍ **Exporting and Storing CPGs:** We then exported the CPGs in DOT format consumable by graph-based analytics tools, such as GNN and other machine learning frameworks. As an example, the graphs for the vulnerable code from *part C* are displayed in this section. Due to how large the CPG gets, we show some portions of the graph. Figs. 4 (a) and (b) are the AST of the vulnerability-causing method, ”gets()” and its fix ”fgets()”, respectively, illustrating the differences in their structure. The root node, representing the function, branches out to include its parameters, body (block), and return type. P1, P2, and P3 are the parameters accepted by the ”fgets” function, while the ”gets” function only takes P1. P1 is the buffer where the string is stored (char *str), P2 is the maximum number of characters to read (int n), and P3 is the input stream to read from. The $block(< empty >, < empty >)$ represents the body of the function. Since ”gets and fgets” are typically standard library functions, their actual implementation may not be directly visible in user code, and the AST shows it as empty or abstracted. Method_return indicates that ”fgets and gets” returns a value, which is a pointer to the character buffer (char *).

We also show the DFG of the main() method for the vulnerable code in Fig. 5. The root node representing the main function acts as the starting point. Nodes like printf, gets, scanf, and qtdletras are part of the Abstract Syntax Tree. These nodes are structured hierarchically to show how the function operates step-by-step. The Data Flow (DDG) captures how data flows between variables and function calls, specifically focusing on the ”gets” functions. For example, the function outputs to texto, which is consumed by qtdletras and that is also consumed by printf. This data flow shows how the ”gets” function corrupts other nodes before finally leading to the final node (method_return).

## IV. Dataset

The dataset aims to provide a comprehensive collection of categorized code vulnerabilities and their fixes, structured in alignment with the OWASP Top Ten security risks. During the initial web crawl, we extracted 26,823 GitHub Commit IDs, however, many commit IDs were found to be duplicates after cleaning the dataset, which reduced to 7,552. These commit IDs generate 59,334 code files across multiple languages, with 9,771 supported by Joern.

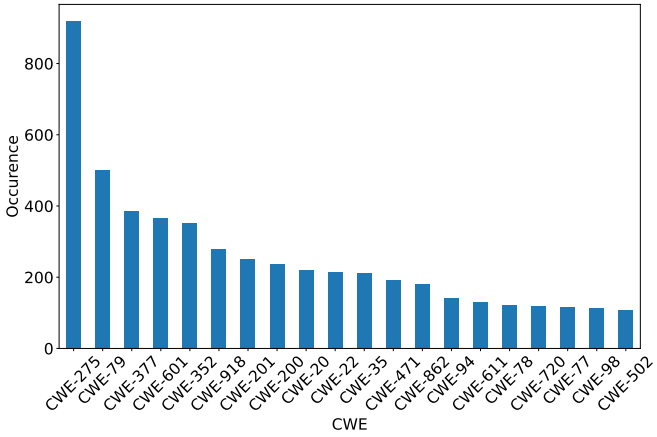


Fig. 6. The distribution chart of the top 20 CWEs shows the most common vulnerability types in the dataset, highlighting critical areas such as CWE-275 Permission Issues and CWE-79 Cross-site Scripting, aligning with real-world security concerns.

Each category of this dataset corresponds to a common security risk, enabling precise focus on specific vulnerabilities and their remediation. The inclusion of both vulnerable and fixed code samples makes this dataset particularly valuable for training models not only to identify vulnerabilities but also to understand the nature of security fixes. This dataset is not limited to one programming language or file type, making it versatile for broad applications in cybersecurity. There is a total of 156 CWEs across all vulnerability categories in our dataset and the top 20 most occuring CWEs are shown in Fig. 6. Table II summarizes the distribution of vulnerabilities across different OWASP categories, highlighting the most affected programming languages.

Common software vulnerabilities, such as CWE-352 (Cross-Site Request Forgery) and CWE-79 (Cross-Site Scripting), are

TABLE II
DISTRIBUTION OF SOFTWARE VULNERABILITIES ACROSS OWASP CATEGORIES

| OWASP Category | Most Affected Languages | Description of Vulnerabilities |
|---|---|---|
| Broken Access Control | Python, C, Java, JavaScript | Enterprise applications in Java tend to have complex role-based access control; hence, the likelihood of a misconfiguration occurring is higher. In the same vein, frameworks built on top of PHP, like Laravel or WordPress, could have insufficient access control implemented due to a poorly designed plugin or a deprecated API. Also, JavaScript may be exposed to broken access control in single-page applications (SPAs) if authorization is applied only on the client side |
| Cryptographic Failures | Java, JavaScript, Python, C++ | Many legacy applications still rely on obsolete cryptographic libraries like MD5 or SHA-1 in older versions of Java and PHP. Such reliance poses significant security risks. Also, Python's simplicity encourages fast prototyping but can also result in inconsistent implementations. Best practices in cryptography are generally overlooked, and developers tend to misuse cryptographic primitives either because of the complexity of the APIs or simply for lack of understanding, such as not using salting when encrypting data. |
| Injection | PHP, JavaScript, Python, Java | Applications that involve many databases will typically use PHP and Python. This can be dangerous for query safety because they predominantly use dynamic SQL queries. What's more, with languages like JavaScript and PHP, there is a lot of dynamic string manipulation via string concatenation. If not taken care of, this can open avenues to injection issues. Also, incorrect usage of Object-Relational Mapping (ORM) tools, such as Java's Hibernate or Python's SQLAlchemy, may not prevent injection attacks properly if they are not configured correctly. |
| Insecure Design | Python, JavaScript, PHP, Java | Frameworks like Flask and Django in Python focus on being easy to use, but this can lead to less secure settings, causing insecure designs. Likewise, large systems made in Java might focus more on adding features than on secure design. |
| Security Misconfiguration | Java, JavaScript, C, PHP, C++ | Older versions of PHP were often criticized for their insecure default configurations. Misconfigurations in widely used Java frameworks, such as Spring or Struts, also frequently contribute to security vulnerabilities. |
| Vulnerable and Outdated Components | Java, Python, Ruby, JavaScript, Go, C, PHP, C++ | Languages like Python and Ruby, which depend heavily on third-party libraries, may face vulnerabilities due to outdated dependencies, a phenomenon commonly referred to as "dependency hell." |
| Identification and Authentication Failures | Java, Python, JavaScript, C#, Go | Session management flaws can arise in Java's enterprise applications when sessions or cookies are not properly handled. Similarly, PHP-based login systems frequently exhibit weak authentication flows, often lacking robust password policies and multi-factor authentication. JavaScript applications may face issues with token handling, particularly when JWT tokens are not stored securely, leading to potential token theft. |
| Software and Data Integrity Failures | JavaScript, Java, Python, Ruby, C# | Java applications frequently encounter code signing issues, resulting in difficulties when verifying the integrity of JAR files. Additionally, the package managers used in Python and Ruby, such as pip and Bundler, pose dependency risks by potentially installing malicious or outdated packages. |
| Security Logging and Monitoring Failures | Java, C#, Python, Go, JavaScript | Java enterprise systems often suffer from inadequate logging practices, which can create significant gaps in threat detection. Similarly, Python's versatile logging module may be misused, leading to improper sanitization of logs. Additionally, JavaScript applications tend to log client data in an insecure manner, making it vulnerable to interception. |
| Server-Side Request Forgery (SSRF) | JavaScript, Python, Ruby, Go, Java, PHP | Improper URL sanitization in languages like Python, with its urllib, and Java, using HttpURLConnection, can result in unintended requests being made. In cloud environments, applications developed in Java or JavaScript frequently query internal APIs, which exposes them to server-side request forgery (SSRF) vulnerabilities. |

prevalent because they often target widely used web technologies like JavaScript, PHP, and Python that dominate web development. Certain CWEs, like CWE-89 (SQL Injection), have been studied extensively due to their historical impact, leading to more frequent identification and reporting by tools and researchers.

Other issues, such as CWE-200 (Exposure of Sensitive Information) and CWE-502 (Deserialization of Untrusted Data), arise from common patterns in programming practices that are frequently misused. Vulnerabilities that are easier to exploit, such as CWE-22 (Path Traversal) and CWE-79 (XSS), tend to be more common because attackers can capitalize on them using relatively simple techniques.

Some CWEs, like CWE-20 (Improper Input Validation) and CWE-78 (Command Injection), are linked to errors in coding. These vulnerabilities typically occur when developers fail to adhere to secure coding guidelines or underestimate the risks associated with unchecked input.

Static and dynamic analysis tools are often more effective at detecting specific types of vulnerabilities, such as CWE-22 or CWE-89, which can skew the reported occurrence rates in vulnerability datasets. CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) frequently arises in scenarios where authorization and access control measures are insufficiently enforced. This vulnerability is particularly common in systems that handle sensitive data, such as those in healthcare, finance, or government applications, where stringent operational controls are essential.

## V. Evaluation

To tokenize the cleaned dataset, a pre-trained tokenizer from the transformers library, specifically bert-base-uncased [32], is utilized. This tokenizer is chosen for its ability to convert textual input into numerical representations that are suitable for transformer-based models. It truncates or pads the input to a maximum sequence length of 512 tokens, ensuring consistency across samples.

The tokenizer generates two outputs: one representing the numerical encoding of the code, and the other, indicating which tokens are valid versus those that are padding. These outputs are crucial for preparing the code as input for transformer-based models, particularly for downstream tasks such as classification. During this step, the vulnerability status of each file is also assessed. Files that include "Vulnerable" in their name are flagged as vulnerable, while others are categorized as fixed or non-vulnerable. The numeric category associated with the parent folder is adjusted based on this vulnerability status, adding an extra layer of metadata. Finally, the processed data, which includes both the metadata and the encoded representations, is saved into a CSV file. The metadata contains the file name, subfolder name, vulnerability status, numeric category, and the tokenized outputs.

A central aspect of vulnerability detection is balancing identifying true vulnerabilities (recall) with minimizing false alarms (precision). Precision measures the proportion of correctly identified vulnerabilities out of all instances flagged, reducing the risk of "false positives" that could result in unnecessary mitigation efforts. On the other hand, recall reflects the model's ability to identify all true vulnerabilities, an extremely important characteristic in reducing the risk of "false negatives" that would leave systems open to attack. The F1 score, as a harmonic mean of precision and recall, gives a single measure of overall effectiveness in the face of this trade-off.

## VI. Results

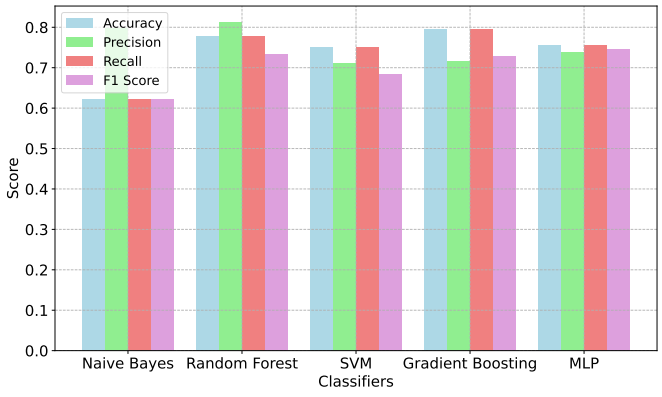


Fig. 7. Evaluation results of our dataset using five machine learning classifiers.

Five machine learning models: Naive Bayes [33], Random Forest [34], Support Vector Machine (SVM) [35], Gradient Boosting [36], and Multilayer Perceptron (MLP) [37] were used for evaluating the performance metrics on the dataset Fig. 7. The dataset was divided into training and test sets, with 80% for training and 20% for testing, ensuring consistency across models. Each model's performance metrics are as follows:

The Naive Bayes classifier achieved an accuracy of 62.15%, precision of 79.79%, recall of 62.15%, and an F1 score of 62.26%. While the model shows reasonable performance in detecting true vulnerabilities, the moderate recall indicates room for improvement in identifying all instances of vulnerabilities. The Random Forest classifier delivered strong performance with an accuracy of 77.80%, precision of 81.19%, recall of 77.80%, and an F1 score of 73.40%. These results highlight the model's robustness in capturing patterns in the dataset, though the F1 score indicates slightly lower balance in precision and recall compared to its accuracy. The SVM model performed moderately, with an accuracy of 75.04%, precision of 71.12%, recall of 75.04%, and an F1 score of 68.45%. This indicates the model struggles to balance between precision and recall effectively, potentially due to the dataset's complexity and the need for more advanced kernel functions. The Gradient Boosting classifier showed the best overall performance, with an accuracy of 79.44%, precision of 71.57%, recall of 79.44%, and an F1 score of 72.74%. The sequential learning method appears to capture intricate relationships within the data effectively, resulting in a balanced performance across all metrics. Finally, the MLP classifier displayed notable improvement compared to prior evaluations, achieving an accuracy of 75.50%, precision of 73.93%, recall of 75.50%, and an F1 score of 74.54%. The results suggest that the adjustments in architecture or hyperparameters contributed to its increased effectiveness.

Given the remarkable results shown by our dataset when tested with traditional machine learning classifiers, along with the availability of its graph-based representation through Code Property Graphs (CPGs), we anticipate it to be greatly compatible with Graph Neural Networks (GNNs).

## VII. Conclusion and Future Work

This paper presents a new, comprehensive dataset that is specifically curated to support the detection and classification of software vulnerabilities in multiple OWASP Top Ten security risks. Our approach, including elaborate keyword mapping, automated crawling of the web, and graph-based representation of code, results in a structured and accessible resource for researchers and developers. This dataset allows multiclass classification, which enables models to identify subtle vulnerabilities, at the same time helping more in the exact design of security mitigations.

For future work, we plan to expand the dataset by incorporating additional sources beyond GitHub to ensure better representation of under-sampled OWASP categories. Additionally, given the structured graph representation of our dataset, we will explore Graph Neural Networks (GNNs) for designing an advanced vulnerability detection system.

## VIII. Acknowledgment

The work under this project is supported by the National Science Foundation Grants number 2319934.

## References

[1] V. Terragni, P. Roop, and K. Blincoe, "The future of software engineering in an ai-driven world," 2024. [Online]. Available: https://arxiv.org/abs/2406.07737

[2] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garceran, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2020.

[3] G. George, J. Kotey, M. Ripley, K. Z. Sultana, and Z. Codabux, "A preliminary study on common programming mistakes that lead to buffer overflow vulnerability," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 1375–1380.

[4] N. S. Harzevili, A. B. Belle, J. Wang, S. Wang, Z. Ming, Jiang, and N. Nagappan, "A survey on automated software vulnerability detection using machine learning and deep learning," 2023. [Online]. Available: https://arxiv.org/abs/2306.11673

[5] MITRE Corporation, "Mitre corporation - solving problems for a safer world," 2024, accessed: 2024-11-10. [Online]. Available: https://www.mitre.org/

[6] OWASP Foundation, "Owasp top 10," 2024, accessed: 2024-10-28. [Online]. Available: https://owasp.org/Top10/

[7] F. A. Bhuiyan and A. Rahman, "Log-related coding patterns to conduct postmortems of attacks in supervised learning-based projects," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3568020

[8] Y. Guo, S. Bettaieb, and F. Casino, "A comprehensive analysis on software vulnerability detection datasets: trends, challenges, and road ahead," *International Journal of Information Security*, vol. 23, no. 5, pp. 3311–3327, Oct 2024. [Online]. Available: https://doi.org/10.1007/s10207-024-00888-y

[9] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 111–120.

[10] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, ser. NDSS 2018. Internet Society, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" 2020. [Online]. Available: https://arxiv.org/abs/2009.07235

[12] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2022.

[13] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2021.

[14] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1909.03496

[15] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," 2023. [Online]. Available: https://arxiv.org/abs/2304.00409

[16] F. Cheirdari and G. Karabatis, "Analyzing false positive source code vulnerabilities using static analysis tools," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 4782–4788.

[17] C. Wang, Z. Li, Y. Peng, S. Gao, S. Chen, S. Wang, C. Gao, and M. R. Lyu, "Reef: A framework for collecting real-world vulnerabilities and fixes," 2023. [Online]. Available: https://arxiv.org/abs/2309.08115

[18] A. Kathikar, A. Nair, B. Lazarine, A. Sachdeva, and S. Samtani, "Assessing the vulnerabilities of the open-source artificial intelligence (ai) landscape: A large-scale analysis of the hugging face platform," in *2023 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2023, pp. 1–6.

[19] D. Cotroneo, R. De Luca, and P. Liguori, "Devaic: A tool for security assessment of ai-generated code," *Information and Software Technology*, vol. 177, p. 107572, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001770

[20] P.-F. Mihancea and R. Scott, "Codesonar (r) extension for copy-paste-(mis) adapt error detection," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 386–389.

[21] A. Rahman and C. Parnin, "Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3536–3553, 2023.

[22] F. A. Bhuiyan, S. Prowell, H. Shahriar, F. Wu, and A. Rahman, "Shifting left for machine learning: An empirical study of security weaknesses in supervised learning-based projects," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022, pp. 798–808.

[23] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, and Y. Liu, "Software composition analysis for vulnerability detection: An empirical study on java projects," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 960–972. [Online]. Available: https://doi.org/10.1145/3611643.3616299

[24] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," 2023. [Online]. Available: https://arxiv.org/abs/2212.08109

[25] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," 2023. [Online]. Available: https://arxiv.org/abs/2212.08108

[26] S. Liu, W. Ma, J. Wang, X. Xie, R. Feng, and Y. Liu, "Enhancing code vulnerability detection via vulnerability-preserving data augmentation," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 166–177. [Online]. Available: https://doi.org/10.1145/3652032.3657564

[27] F. Vera, P. Pauliuchenka, E. Oh, B. C. Kao, L. DiValentin, and D. A. Bader, "Profile of vulnerability remediations in dependencies using graph analysis," 2024. [Online]. Available: https://arxiv.org/abs/2403.04989

[28] Selenium Developers, "Selenium WebDriver: Browser Automation Framework," 2024, accessed: 2025-03-09. [Online]. Available: https://www.selenium.dev/

[29] Leonard Richardson, "Beautiful Soup: A Python Library for Web Scraping," 2025, accessed: 2025-03-09. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/

[30] Yoeo, "Guesslang: Programming language detection," https://github.com/yoeo/guesslang, accessed: 2024-11-09.

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[32] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[33] F.-J. Yang, "An implementation of naive bayes classifier," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2018, pp. 301–306.

[34] Y. Liu, Y. Wang, and J. Zhang, "New machine learning algorithm: Random forest," in *Information Computing and Applications*, B. Liu, M. Ma, and J. Chang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 246–252.

[35] S. Suthaharan, *Support Vector Machine*. Boston, MA: Springer US, 2016, pp. 207–235. [Online]. Available: https://doi.org/10.1007/978-1-4899-7641-3_9

[36] A. V. Konstantinov and L. V. Utkin, "Interpretable machine learning with an ensemble of gradient boosting machines," *Knowledge-Based Systems*, vol. 222, p. 106993, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705121002562

[37] D. Morariu, R. Cretulescu, and M. Breazu, "The weka multilayer perceptron classifier," *International Journal of Advanced Statistics and IT&C for Economics and Life Sciences*, vol. 7, no. 1, 2017.