*machine learning & knowledge extraction*

MDPI

*Article*

# Assessment of Software Vulnerability Contributing Factors by Model-Agnostic Explainable AI

**Ding Li, Yan Liu * and Jun Huang**

Department of Electrical and Computer Engineering, Concordia University, Montréal, QC H4B 1R6, Canada; ding.li@mail.concordia.ca (D.L.); jun.huang@mail.concordia.ca (J.H.)
* Correspondence: yan.liu@concordia.ca

**Abstract:** Software vulnerability detection aims to proactively reduce the risk to software security and reliability. Despite advancements in deep-learning-based detection, a semantic gap still remains between learned features and human-understandable vulnerability semantics. In this paper, we present an XAI-based framework to assess program code in a graph context as feature representations and their effect on code vulnerability classification into multiple Common Weakness Enumeration (CWE) types. Our XAI framework is deep-learning-model-agnostic and programming-language-neutral. We rank the feature importance of 40 syntactic constructs for each of the top 20 distributed CWE types from three datasets in Java and C++. By means of four metrics of information retrieval, we measure the similarity of human-understandable CWE types using each CWE type's feature contribution ranking learned from XAI methods. We observe that the subtle semantic difference between CWE types occurs after the variation in neighboring features' contribution rankings. Our study shows that the XAI explanation results have approximately 78% Top-1 to 89% Top-5 similarity hit rates and a mean average precision of 0.70 compared with the baseline of CWE similarity identified by the open community experts. Our framework allows for code vulnerability patterns to be learned and contributing factors to be assessed at the same stage.

**Keywords:** explainable AI; graph learning; software code vulnerability; feature representation



**Citation:** Li, D.; Liu, Y.; Huang, J. Assessment of Software Vulnerability Contributing Factors by Model-Agnostic Explainable AI. *Mach. Learn. Knowl. Extr.* **2024**, *6*, 1087–1113. https://doi.org/10.3390/make6020050

Academic Editor: Luca Longo

Received: 29 February 2024
Revised: 28 April 2024
Accepted: 1 May 2024
Published: 16 May 2024



## 1. Introduction

Software vulnerability refers to weaknesses within an information system, its internal controls, its system security procedures, or its implementation that could be exploited by a threat source [1]. These vulnerabilities often arise from design errors, poor coding practices, or inadequate security testing. In large-scale software systems, detecting vulnerabilities presents challenges in terms of the accuracy and transparency of both research [2–5] and industrial [6,7] practices. Applying vulnerability analyses and detection at the early stage of the software process, prior to deployment, is a proactive attack mitigation solution [8]. The analysis involves learning existing patterns of vulnerability types and analyzing the underlying factors in the code structure that may contribute to the weakness [9]. Vulnerability detection is the process that identifies, classifies, remediates, and mitigates code vulnerabilities.

Research in software vulnerability detection has progressed from static code analysis techniques to machine learning approaches. Static code analysis tools, such as security scanners, employ pattern matching [10,11] based on well-defined rules to identify bugs or flaws in the software [12–14]. However, these tools suffer from high false-positive rates [15].

Machine-learning-based approaches utilize source code, software complexity metrics, and version control system data to predict vulnerabilities [5,16,17]. These approaches enable automatic feature extraction and the learning of complex patterns, reducing the need for expert-driven feature engineering [3,18–20]. Data-driven software vulnerability detection has been reported to improve the detection accuracy in practice [21,22].

A comprehensive study [23] has identified the common limitations of six deep learning models in producing realistic code vulnerability detection. The main limitation is inadequate models that reduce their learning performance when transferred to real-world settings. Further reasons include the learning of irrelevant features, data duplication, and data imbalances. All these aspects impose limitations on model-specific approaches' ability to provide transferable patterns beyond the training datasets. Questions remain regarding the scope of interpretability and explainability of AI, what kind of features these models are learning, and whether they can be effectively and reliably transferred to other datasets [23].

One limitation is that practitioners cannot understand the features learned by a deep learning model without mapping the semantic meanings of vulnerable artifacts [8]. The opacity leads to questions, such as the following: (1) How transferable are the signatures of vulnerable artifacts learned from one set of software projects to others [24]? (2) What factors are mostly involved in the representation learning? (3) What variance is caused by factors from (2) in the classification results among different learning methods [25]? A key to bridging the gap between the learned feature representations and human-understandable vulnerability semantics is to assess the importance of code features to the semantics of the vulnerability classification. Such an assessment necessitates that the techniques are model-agnostic, emphasizing only the code features as inputs and the resulting vulnerability classification as outputs.

EXplainable Artificial Intelligence (XAI) is an emerging research field that aims to enhance AI models as trustworthy and transparent [26]. XAI encompasses diverse techniques, methods, and models to explain how the learning models reach their predictions. For instance, model-agnostic attribute-based XAI methods focus on identifying attributes that contribute the most to the model's prediction. A manifesto of XAI was thoroughly defined based on a set of XAI survey papers and shared visions by scholars [27]. Applications of XAI covered in the XAI manifesto [27] include healthcare, medicine, bio-informatics, finance, environmental science, agriculture, and education. Additionally, the software development and software system domains have a large amount of code, documentation and diverse scenarios that require AI learning and explanation. In the context of code vulnerability learning, SHapley Additive exPlanations (SHAP) [28], LIME [29], Lemna [30] and Mean-Centroid PredDiff [31] have been applied to measure the feature contribution values of program code feature representation.

The current application of XAI techniques in software vulnerability analysis faces the issue that the covered attributes cannot be extrapolated beyond the domain of the input data. Thus the explanation is (1) limited to a few attributes and (2) disconnected from the semantically defined relations among CWE types. Such a semantic relation is embedded in the definition of CWE types accumulated over years of practice in the community. We consider explanations with a link to CWE semantics as human-understandable explanations. This type of research challenge has been identified and defined as one of the nine aspects of the XAI manifesto [27].

Several studies have attempted to explain the importance of Abstract Syntax Tree (AST) path content [32,33] or individual code tokens [34] using XAI methods such as SHAP [28]. However, only limited syntactic constructs such as `name`, `parameters`, `statements` are investigated, rather than the whole set of syntactic constructs. Moreover, there is a lack of studies that relate learned features' representation to the semantic similarity collectively described by security experts for a variety of types of software vulnerabilities [33]. In summary, a systematic method to correlate these meta syntactic constructs with common characteristics across multiple vulnerability types is the purpose of this study.

The work from CSAIL MIT [35] has demonstrated, with a novel experiment design, that program synthesis trained as program corpus in textual input–output is well-suited for characterizing the meaning in language models. We are informed by the evidence in [23] of the limitations of the token sequences at the program level to reveal the semantic meanings of feature contributions. We further consider the code tokens at the syntactic

construct level to assess the feature contributions to vulnerability classification through XAI probing techniques.

In this paper, our method first studies the inputs as code features by defining the taxonomy of the state-of-the-art works into four types, namely, text-based, graph-based, code binary, and mixed representation [25], as discussed in Section 3. For each type of code feature representation, we further categorize works into eight groups of program artifacts and twenty-one extracted code feature types. We mainly select *Abstract Syntax Tree* out of the twenty-one extracted code feature types using the taxonomy shown in Figure 1. The abstract syntax tree contains structured meta-data of a programming language and applies this to all the programming codes of the same language. Then, graph learning models can be used to encode and decode the hierarchical structure of the abstract syntax tree.

We attempt to derive model-agnostic explanations for multi-classification in contrast to binary classification in [23]. We focus on the graph context of code tokens as features that embed code token connections through traversing the abstract syntax tree. Code property graphs such as the control flow graph, data flow graph, and program dependency graph are analyzed at the program level, which faces the challenges of maintaining balanced data samples for every vulnerability type. The sufficient and balanced samples of feature types are suitable for XAI methods such as feature masking.

Our method then assesses the output of vulnerability classification based on the Common Weakness Enumeration (CWE) [36]. CWE is a community-developed list of software and hardware security weaknesses that are commonly used as labels for supervised learning in vulnerability detection. The study [37] has identified that various vulnerability types exhibit semantic similarities. Similarly, the vulnerability similarities are represented by organizing the CWE's hierarchical structures [36] or CWE clusters [38].

We define and apply information retrieval metrics to measure the similarity between classified CWEs. Through XAI methods and techniques, we further assess the variance in CWE classification similarity under input feature changes. Thus, the machine-learned feature representations are quantitatively measured for their contributions to classifying community-defined and human-understandable code vulnerability types. Our contribution is three-fold:

1. We define the taxonomy of code representation into eight types of high-level categories, and a further twenty-one fine-grained code representations. This taxonomy distinguishes the fine-grained code representation set at the program source code level and the program meta-data level. This taxonomy clearly positions our XAI-based approach in the map of related works.
2. We design a model agnostic XAI framework that derives rankings of the feature contribution levels of a list of forty syntactic constructs in Abstract Syntax Trees (AST) across twenty CWE types for both Java and C++ datasets. This framework is applicable to different choices of classifiers and XAI methods.
3. We develop a novel feature masking technique for the graph context that varies the neighbourhood of code tokens and syntactic constructs. We define and apply information retrieval techniques to convert the change in the code token neighbourhood into the CWE type similarity.

Overall, we demonstrate that the similarity between CWE types derived from XAI explanations links subtle semantics that are understood by security experts to the learned code feature representations. Through experiments, we compare XAI-derived CWE similarities and sibling CWE types defined by security experts. Thus, our approach is able to retrospectly identify the misclassification of similar CWE types due to the variance in feature contributions. We open-sourced our code and made our dataset available on GitHub (https://github.com/DataCentricClassificationofSmartCity/XAI-based-Software-Vulnerbility-Dection, accessed on 1 May 2024).

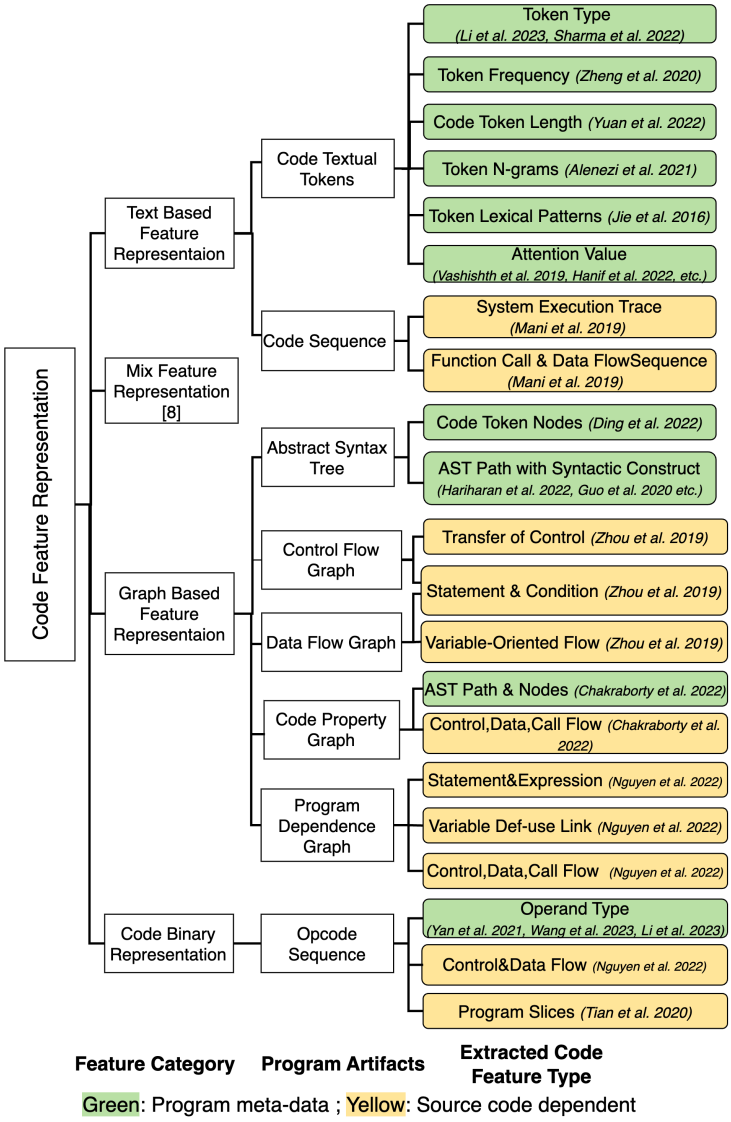


**Figure 1.** The taxonomy of factors under code feature representation techniques [21,23,31–33,39–59] .

This paper is organized starting with an overview of model-agnostic XAI methods and the motivation for adopting XAI in Section 2. Section 3 reviews the existing literature on the code feature representation and vulnerability detection domain using XAI applications. We present our main research methodology and XAI-based framework in Section 4. Then, we propose the research questions, conduct the experiments, and demonstrate our results in Section 5. We present a retrospective of the motivation case in Section 5.4. Finally, we discuss the potential threats to the validity in Section 6 and draw conclusions in Section 7.

## 2. Background and Related Work

### *2.1. Common Weakness Enumeration (CWE)*

CWE is a community-developed list of software weakness types that aims to identify and describe vulnerabilities in common programming languages. CWE vulnerabilities are normally not language-specific. A CWE type contains comprehensive information regarding description, relation to other CWE types, demonstrative examples from different programming languages, and observed examples with reference to the Common Vulnerabilities and Exposures (CVE) list.

CWE offers a standardized categorization of vulnerabilities based on the abstraction of weakness behaviors, which are organized into a tree structure [60]. For example, both CWE23 (Relative Path Traversal Weakness) and CWE36 (Absolute Path Traversal Weakness)

are children of the parent CWE type described as the "Improper Limitation of a Path Name to a Restricted Directory." CWE23 and CWE36 are siblings with commonalities. In both instances, the problem stems from a lack of input validation.

Figure 2 shows code snippets where user input (data) is used directly to access files, with the risk that an attacker can potentially access arbitrary files. In the case of CWE23, the user input is appended to a root path, while in the case of CWE36, the user input is used directly as the path. For a multi-classification code vulnerability detection problem, deep-learning models may misclassify a CWE type if the sibling CWE type shares the same CWE type parent. Figure 2 further illustrates a misclassification where a deep-learning model classifies the code in the right-hand bar of type CWE36 to CWE23. The misclassification is due to the similarity between the two vulnerability types.

The CWE similarity of any pair of CWE types can be traced by denoting them as siblings through the parent–child paths, following the community-established CWE knowledge base [60]. The CWE type is the target of classification, which is the output of a learning model. Hence, the similarity between sibling CWEs can provide clues to explain how a learning model identifies the importance of program code feature representation and determines its prediction results.

We assume that XAI-based methods can probe into the highest dimension of program code and its relation to potential vulnerability types. According to our survey and taxonomy of XAI methods [61], feature importance explanation methods are post hoc and model-agnostic, which makes them suitable for assessing feature representations encoded by different types of models and feature representations. The XAI methods relate the outputs to the changes in the inputs, which refer to specific metrics, namely, *feature contribution value* and *feature importance rank*, a vector of features sorted in descending order according to each feature's contribution value.

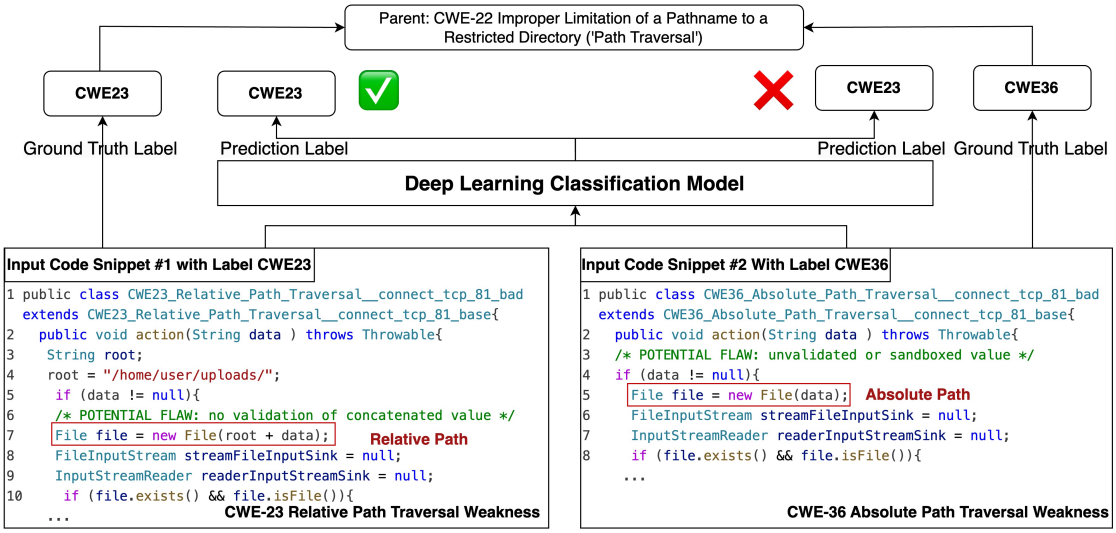


**Figure 2.** An example of a deep learning model providing an incorrect prediction for CWE siblings: CWE23 with relative path traversal weakness and CWE36 with absolute path traversal weakness. The prediction results were obtained from the GraphCodeVec [39] model in the Juliet dataset [62].

### *2.2. XAI Explanation of Feature Importance*

XAI aims to provide interpretable explanations for the complex and non-transparency machine learning models [26]. Using the XAI taxonomy proposed in the study [61], the feature importance explanation aims to quantify the contribution of individual factors to the model's prediction for a clear understanding. Various model-agnostic XAI methods have been proposed, including the following: (1) LIME [29], which provides local explanations for individual predictions; (2) Shapley values [28], a game theoretic approach used

to measure the contributions of individual features; (3) SHapley Additive exPlanations (SHAP) [28], combining Shapley values with additive feature importance measures; (4) PredDiff [63], a feature importance measurement based on the prediction difference; and (5) Mean-Centroid PredDiff [31], an extension of PredDiff that measures the prediction difference using density clustering. The XAI feature importance explanation is a rationale tool that enables observations of potential factors contributing to the detection of vulnerabilities, which, in turn, facilities practitioners' comprehension and interpretation of the learning quality [25].

We assume that XAI-based methods can probe into the high dimensions of program code and their relation to potential vulnerability types identified by human experts. According to our survey and the taxonomy of XAI methods [61], feature importance explanation methods are post hoc and model-agnostic, which makes them suitable to assess the feature representation encoded by different types of models for code tokens linked via syntactic constructs in an AST structure.

We refer to the code token feature as varied-length features while the AST syntactic constructs are fixed-length features. Fixed-length features are especially useful for providing a global summary of the CWE vector within a dataset, as they can establish a shared feature space across various code instances. This commonality allows for a more comprehensive and consistent analysis across different instances. In contrast, the variability of textual code tokens from case to case introduces challenges when aggregating a global summary of the CWE vector, as their individual uniqueness may not contribute to a broader, collective understanding. These features are inputs to learning models and XAI methods. The XAI methods relate the outputs to the changes in inputs, which refer to specific metrics, namely the *feature contribution value* and *feature importance order*.

### *2.3. Model-Agnostic XAI Methods*

**SHapley Additive exPlanations (SHAP)** [28] is a state-of-the-art XAI method that provides a unified measure of feature importance for individual predictions based on Shapley values from cooperative game theory. SHAP helps quantify the contribution of each feature to the prediction for a specific instance in model-agnostic tasks. The feature contribution value $\phi$ from the SHAP value is defined as the average marginal contribution of a feature across all possible combinations of features, as follows:

$$\phi_j^{\text{SHAP}} = \sum_{S \subseteq P \setminus j} \frac{w(S)}{|S|!(|P| - |S| - 1)!} (\hat{f}_{S \cup j}(x) - \hat{f}_S(x)) \quad (1)$$

where $P$ is the set of all features, $S$ is a subset of $P$ without feature $j$, $|S|$ denotes the size of set $S$, $|P|$ denotes the size of set $P$, $w(S)$ is the weight assigned to the subset $S$, and $\hat{f}_S(x)$ and $\hat{f}_{S \cup j}(x)$ represent the model's output with and without the feature $j$, respectively. The weights $w(S)$ are determined by a kernel function, such as the exponential kernel or the linear kernel.

**Mean-Centroid PredDiff (MCP)** [31] is based on Prediction Difference Analysis (PredDiff) [64]. PredDiff calculates feature contribution values based on the difference in the log-odds ratio of classification probabilities when individual features or groups of features are removed [65]. Despite its simplicity and effectiveness, PredDiff derives explanations for every single feature change and its corresponding prediction. Hence, PredDiff's explanation could potentially be less stable and consistent due to the variations in explanations [66].

To improve this, we developed the extension to PredDiff, creating Mean-Centroid PredDiff (MCP) [31]. MCP gathers predictions from the entire dataset and derives explanations by clustering predictions. In the previous work [31], MCP has demonstrated an improved consistency in its global explanation results compared to existing PredDiff [64]. The MCP process consists of three phases and is formulated as follows:

$$\begin{aligned} \phi_j^{\text{MCP}} &= \tanh(\mu) \\ \text{where } \delta_X^j &= |\hat{f}_{S\cup j}(X) - \hat{f}_S(X)|, \\ \mu &= \hat{f_{\text{gmm}}}(\delta_X^j, \hat{f}_S(X)) \end{aligned} \tag{2}$$

Phase 1: MCP calculates the prediction difference $\delta_X^j$ under feature masking for each masked feature $j$ across a dataset $X$ containing $N$ samples. This produces $N$ two-dimensional points, with each corresponding to a feature difference ascertained from the logit of the classification probability.

Phase 2: MCP identifies clusters from the data points generated in Phase 1 and uses a Gaussian mixture model [67] $\hat{f_{\text{gmm}}}$ to estimate each cluster's centroid. The feature contribution value $\phi_j^{\text{MCP}}$ for each masked feature $j$ is then defined as the slope from the origin point to the centroid data point in the two-dimensional plane.

Phase 3: MCP ranks the features in descending order of their contribution values to generate a feature importance vector.

### *2.4. XAI Method Selection*

The selection of XAI methods follows the taxonomy [61] that is within the group of model-agnostic and feature-changes-based XAI methods, such as Shapley Values [28], SHapley Additive exPlanations (SHAP) [28], PredDiff [63], and Mean-Centroid PredDiff [31].

In our previous study [31,61], we conducted case studies on three different applications, including NLP recommendations on tabular data, NLP multi-classification text tokens, and computer vision image classification. We evaluated twelve XAI methods based on the consistency of their explanations across multiple datasets and individual XAI methods' stability across data instances within the same dataset. The comparison also covers computing complexity and runtime costs.

We observed that SHAP and Mean-Centroid PredDiff demonstrated better consistency across the dataset explanations. Meanwhile, Mean-Centroid PredDiff reduces the runtime by approximately 17.67%. Based on these case studies, we selected SHAP and Mean-Centroid PredDiff as the preferred XAI methods in this paper to demonstrate our framework in Section 4. It should be noted that our framework applies to any XAI method in the group of feature-changes-based model-agnostic methods.

## 3. Taxonomy of Related Work

The primary aim of this study is to fill the gap between practitioners' understanding of vulnerability semantics and the code features learned by deep learning models. To achieve this, we summarized a taxonomy of code features based on four code representation techniques, which will be discussed in detail in this section along with relevant works. Different categories of code representation techniques were developed to transform the source code into a format that can be processed by machine learning models [47,68,69]. These include text-based, graph-based, and mixed-feature representations, as well as code binary representations [25] in Figure 1.

**Text-based Code Representation.** Text-based code representation approaches the treat source code similarly to natural languages, embedding code tokens as the word token embedding [70–72]. The code content is considered as plain text, disregarding structural information such as data flow and function call flow. With advancements in the natural language processing domain, representation techniques have evolved from static embeddings such as word2vec [73] and fastText [70] to self-attention transfer-learning-based models with large corpus embeddings, such as codeBERT [74], XLNet [75], Longformer [76], BigBird [77], and GPT [78]. These models use pre-trained contextualized embeddings, which are more expressive than static embeddings. CodeBERT [74] embeddings leverage a dual-

transformer architecture, combining the strengths of masked language modelling and code summarization while facing the challenges of dealing with long code sequences. XLNet embeddings [75] utilize a permutation-based approach, capturing the dependencies between tokens and allowing for bidirectional context and comprehensive token representations. BigBird [77] and Longformer [76] embeddings are specific for long token sequences, allowing for a longer input token length. Longformer uses a sliding window-based local attention mechanism for nearby tokens and a global attention mechanism for distant tokens, while BigBird combines dense and sparse attention patterns, efficiently handling long text sequences while preserving the ability to model long-range dependencies.

**Feature Types Under Text-based Code Representation.** In the context of text-based code representation, several feature types have been identified that can influence the model's behavior when processing source code, including token type [31,48], token length [50], token frequency [49], token n-grams [51], token lexical patterns [52], and token attention values [53]. Token types could be categorized as comments and code. Our previous work [31] found that comment tokens provided by programmers can improve the understanding of code semantics and structure for learning models. Another work [48] reveals that separator symbols also play an important role when the model makes a prediction by assessing the attention-based model. Hence, the token types are also categorized into textual tokens and symbol tokens. Limiting the code token length can result in information loss and negatively impact the model's performance, as Yuan et al. [50] show. However, their examination considered a maximum sequence length of 512 tokens. Serving as a key feature type for static text-based representation techniques, token frequency has been found to affect model performance. Zeng et al. [49] concluded that a better model performance is achieved when preserving code frequency information. Token n-grams are fixed-size contiguous sequences of tokens that capture local context within a fixed window [51], but their effectiveness may be limited for longer code sequences and transformer models. By representing recurring structures in the code [52] token lexical patterns can help understand the code's basic logic and structure. However, their effectiveness may be limited in capturing higher-level semantic and complex information and dependencies across distant tokens. Token attention values serve as a feature type in the transformer-based model, and are helpful in identifying key tokens or contents contributing to natural language processing tasks [79]. The attention mechanism can adaptively learn the importance of even distant parts of the input code sequence for a better understating of the code's contextual information and effectively fulfil software vulnerability detection tasks [54–58]. Some researchers found that the attention values can serve as a proxy for the importance of tokens [53]. Still, it is worth noting that this interpretation should be made with caution, as high attention values may not always correspond to high token importance [80].

**Graph-based Code Representation.** A considerable number of studies applied deep learning models to learn code structures from graph-based representations, including Abstract Syntax Tree (AST), Program Dependence Graph (PDG), Control Flow Graph (CFG), Data Flow Graph (DFG), and mixed-method approaches combining these graphs [23]. The study [25] summarizes that, among these graph-based approaches, the AST-based method is used in the majority of existing studies. The syntax nodes in an abstract syntax tree represent the syntactic constructs of the code, such as expressions, declarations, and loops, which are intuitive to practitioners [81]. AST-based methods: Code2Vec [32] presents a graph-based, continuously distributed vector learning approach, quantifying the importance of AST path context for code semantic properties' prediction tasks. Hariharan M. et al. [33] introduce a Multiple Instance Learning (MIL) technique that differentiates each AST path as an instance for supervised learning. GraphCodeBERT [40] is a hybrid approach combining the graph structure information from AST and the transformer-based techniques to represent the code structure. GraphCodeVec [39] learns more generalizable code embeddings from code tokens and AST structure and achieves state-of-the-art results in six downstream code tasks, including vulnerability detection. Other graph-based methods: VulDeeLocator [82] leverages PDG and combines the AST information to learn discriminative vulnerable

features. Devign [21] constructs a hybrid graph representation that combines AST, CFG, and the data dependence graph to enhance the ability to capture complex structural code information, but it may be computationally expensive. REVEAL [23] extract the syntax and semantics features in the Code Property Graph (CPG) that consist of the elements from the data-flow, control-flow, AST nodes, and program dependency.

**Feature Types under Graph-based Code Representation.** The feature type behind graph-based code representation depends on each specific graph structure, node, and edge, and their definitions. For feature types within the AST, leaf nodes represent code tokens belonging to particular syntactic constructs [81]. Thus, code token nodes serve as one feature type. Additionally, path-based representations with inflection nodes as syntax can effectively capture a code's contextual semantics and are widely used in state-of-the-art approaches [32,33,39]. CFG- and DFG-based factors [21,41,42], on the other hand, primarily focus on the flows in a program, such as control flow, the data flow through variables, and the statements and conditions. Lastly, PDG-based factors encompass both the control and data flow dependencies within a program, capturing statements, expressions, variable def-use links, and function call flow [42,43]. These factors are more representative of an individual program rather than the whole software project.

**Other Code Representation and Feature Types.** Several studies have focused on the use of code binary representation for vulnerability detection. BVDetector [47] uses program slices and a BGRU network for fine-grained vulnerability detection. HAN-BSVD [44] employs a hierarchical attention network for context preservation and highlighting crucial regions, while BinVulDet [45] leverages decompiled pseudo-code and BiLSTM-attention for robust vulnerability pattern extraction. Finally, VulANalyzeR [46] introduces an explainable approach with multi-task learning and attentional graph convolution. We summarized the feature types under binary features into operand types, control flows, and program slicing. Additionally, various aspects of code sequence representation in text-based systems have been explored, such as system execution traces, function call sequences, and data flow sequences. Approaches like DeepTriage [59] analyze system execution traces for software defect prediction.

## 4. An XAI-Based Framework for Feature Contribution and Vulnerability Assessment

We propose a framework that retrieves the feature contribution values utilizing XAI techniques and analyzes the XAI explanation summaries. We quantitatively assess the feature contributions to the multi-classification of the code vulnerability of CWE types to identify the factors. As discussed in Section 2, CWE types are defined and categorized by experts from many real-world samples. The similarities of CWE types have subtle effects on the learning tasks of vulnerability classification. Hence, our workflow utilizes XAI methods to probe into the high-dimension code features and relate feature variations to the classification results.

The main components of the workflow are shown in Figure 3. Compared to existing code vulnerability classification solutions, our workflow has three additional components: feature variation, XAI method, and an analysis of XAI outputs. Our workflow applies post-hoc and model-agnostic XAI methods that compute the feature contribution values under the variations in feature mutations, feature masking, and feature removal. The outputs from XAI methods are further analyzed to identify high-ranking code features.

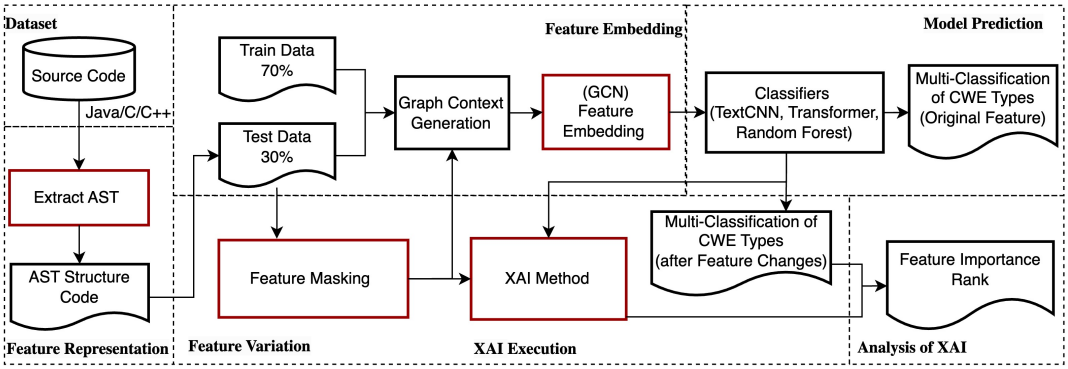


**Figure 3.** The assessment of feature contributions using XAI explanations. The main components include feature representation, feature variations, the XAI method, the pre-trained model, and an analysis of XAI results.

### *4.1. The Graph Context Extraction of Program Code*

We extracted the program paths of the input program source code derived from abstract syntax trees, which preserves the semantic properties of the program code. For example, Figure 2 illustrates the difference between two sibling CWE types that derive from the semantic meanings of arguments. One type is the relative path, and the other is the absolute path. Both are traced back to the syntax of the argument construction.

To capture links between semantic meanings and the syntax constructs, we considered extracting a path that has leaf nodes as code tokens and non-leaf nodes as the syntax constructs derived from Abstract Syntax Trees (AST). Figure 4 shows the complete syntactic constructs of the code example with the CWE23 Relative Path Traversal Weakness.

Syntactic constructs are the program syntax's building blocks, including forty constructs such as loops, conditionals, declarations, and expressions. Table 1 lists a summary of syntactic constructs and the higher-level categorized meta syntactic constructs defined in the work [83]. The meta syntactic constructs preserve the semantic roles within a program. For instance, the *Declarations, Definitions, Initializations* meta construct category consists of syntactic constructs related to defining and initializing variables, functions, and objects.

Further, traversing from one code token through the syntax paths to another shows the connection between code tokens and preserves the functional meanings. An example syntactic construct tree in Figure 4 represents the code listed in Listing 1, which contains the vulnerability type CWE23 relative path's traversal weakness. The syntactic path, `String`$^{\uparrow}$`-name`$^{\uparrow}$`-type`$^{\uparrow}$`-decl`$_{\downarrow}$`-name`$_{\downarrow}$`-root`, extends from the source code token `String` to the target code token `root`, where $\uparrow$ and $\downarrow$ are the traversing directions. In this example, `decl` changes the traversing direction from upward of the path to downward of the path. We call a node that converts the traversing directions an *inflection node*. Through an inflection node, two code tokens in a pair are linked together by the *shortest path* that traverses the nearest inflection node.

**Listing 1.** Code snippet from the Juliet dataset [62]. The code is of vulnerability type CWE23—Relative Path Traversal Weakness.

```
public void action(String data) throws Throwable {
    String root;
    /* POTENTIAL FLAW: no validation of concatenated value */
    root = "/home/user/uploads/";
    if (data != null) {
        File file = new File(root + data);
        FileInputStream streamFileInputSink = null;
        ...}
```

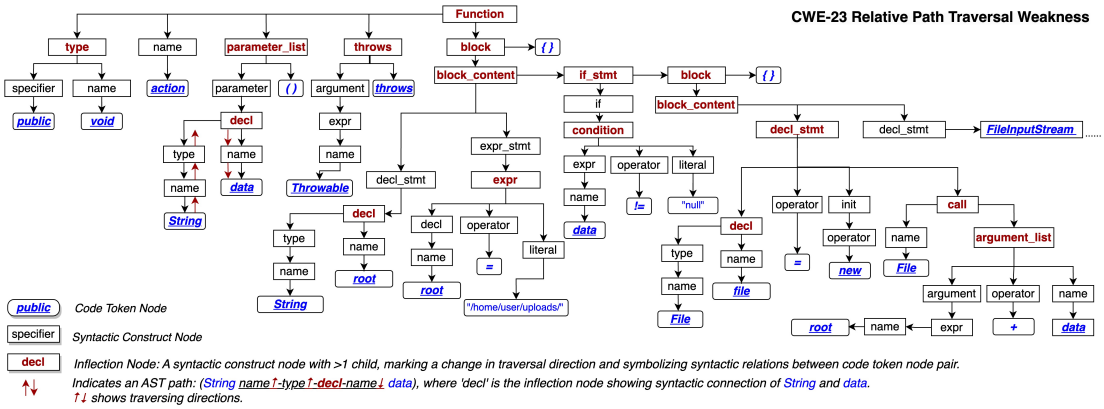


**Figure 4.** The complete paths of the syntactic construct and code tokens for Listing 1. The leaf nodes are code tokens, while the non-leaf nodes are syntactic constructs that define the syntax properties of code.

**Table 1.** Syntactic constructs in Abstract Syntax Tree.

| **Meta Syntactic Constructs [83]** | **Syntactic Constructs** |
|---|---|
| Name, Base Elements | <name>, <block_comment>, <literal>,... |
| Statements | <block>,<case>,<expr_stmt>,<for>,<do>, <if_stmt>,<return>,<switch>,<continue>, <while>,<default>,<lambda>,<function>, <decl_stmt>, <decl>,<init> ,<new>,... |
| Statement subelements | <expr>,<condition>,<block_content>,<else>, <type>,<if>,<incr>,<then>,<control>,... |
| Specifiers | <specifier>,<public>,<static>,<private>,... |
| Classes, Interfaces, Annotations, and Enums | <annotation>,<class>,<static>, <annotation_defn>, ... |
| Expressions | <call>,<this>,<super> ... |
| Arguments | <argument>, <argument_list>,... |
| Parameters | <parameter>,<parameter_list>, ... |
| Exception Handling | <finally>,<throw>,<throws>,<try> ,<catch>,... |

Therefore, the source code of programs is extracted into paths. Each path traverses pairs of code tokens through the shortest path. The set of all the paths leading from the source leaf nodes to the target source node contains the graph context of the target nodes. All the source leaf nodes become the neighbor nodes of the target code token. In the graph context, the syntactic constructs along the paths are on the edges that connect the source code tokens to the target code token. For example, Figure 5 illustrates the graph context extracted from the complete syntactic paths that cover the CWE23 vulnerability sample code in Listing 1.

Two configurations `path length` and `window` are involved in the graph context generation. The `path length` is the length of the shortest AST syntactic path connecting two leaf nodes (code tokens). The term `window` refers to the maximum distance between the target code token and its neighboring code token within a code function, considering both upward and downward directions, as illustrated in Figure 6. When learning a target code token, only neighboring tokens located within this window, either upward or downward

of it, are selected as source code token nodes in the graph context. Both *path length* and *window* are utilized to shape the scope of the graph context, as illustrated in Figure 5.

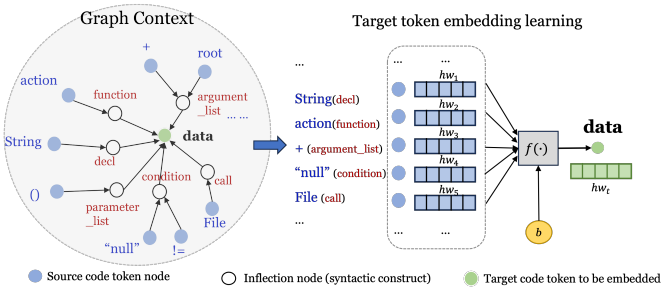


**Figure 5.** An overview of embedding learning. The distributed representations of target code token *data* are learned from the relevant context tokens (blue nodes) that are fed into a one-layer Graph Convolutional Network (GCN). $h_{w_i}$, $h_{w_t}$ are hidden representations of context token and target token, and $b$ is the added bias [39].


**Figure 6.** An example of how the window size restricts the selection of neighboring nodes as the source code node for the target code node `data`, considering both upwards and downwards directions.

### *4.2. Embedding by Graph Convolutional Networks*

The graph context of each target token is used to learn the embedding of the target token. The source tokens within the graph context form the input vector to a learning model, and the output is the target token data. Figure 5 shows that the graph context vector is input to a one-layer Graph Convolutional Network (GCN). We adopted the GCN model developed in [39,84], which demonstrated its use in six software repository analysis tasks, including code classification. The output embeddings for each code token are 128-dimensional vectors containing information about the code token and syntactic constructs.

### *4.3. Feature Masking*

XAI methods such as SHAP and Mean-Centroid PredDiff assess the feature contribution values by means of changing features and measuring the learning models' outputs. By masking a syntactic construct, the graph context of a targeted node is mutated, which results in the masking of neighboring tokens of the target token. Figure 7 shows, as an example, an inflection node `decl` (as an abbreviation for *declaration*) in the path of $\texttt{String}^{\uparrow}\texttt{-name}^{\uparrow}\texttt{-type}^{\uparrow}\texttt{-decl}_{\downarrow}\texttt{-name}_{\downarrow}\texttt{-data}$. When *declaration* is masked, the syntactic construct `decl` is not embedded in any graph context feature presentation. Correspondingly, `String` is excluded from the graph context of `data`. Through the masking of syntactic constructions, we can mutate the embedding of each target node and further assess the feature contribution values of each syntactic construct.

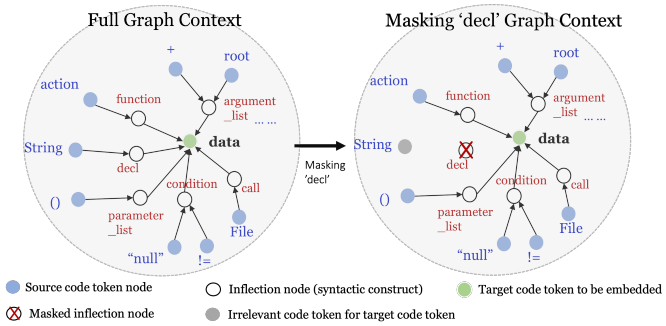


**Figure 7.** Feature masking for graph context mutation. After masking syntactic constructs such as `decl`, the target node's embedding is mutated as the graph context changes.

### *4.4. Integrating XAI Methods in Multi-Classification*

The graph context is processed through a Graph Convolutional Network (GCN) [39] model to generate code token embeddings with a 128-dimension vector. These embeddings are then classified into CWE types using various classifiers, such as the Text Convolutional Neural Network (TextCNN)[85], Random Forest[86], and Transformer [79].

An XAI method $\Phi$ works on the trained model and estimates feature contributions by masking or mutating feature representation. In Section 2, we introduced two XAI methods, SHAP and Mean-Centroid PredDiff, which are applicable as $\Phi$. The XAI outputs are vectors for each CWE type, ranked by the contribution values of each syntactic construct. An example of the XAI outputs is illustrated in Listing 2. The syntactic constructs are ranked in descending order according to the feature contribution value. Hence, we obtained a ranked sequence of syntactic constructs of CWE types that are classified for the dataset.

**Listing 2.** CWE23 vector with syntactic constructs and their feature contribution values.

```
CWE23 Vector: [("name", 0.969),("if", 0.478),("argument_list", 0.470),("
    finally", 0.349),("argument", 0.329),("literal", 0.324),("throws",
    0.301),("decl", 0.296),("try", 0.281),("operator", 0.210),...]
```

Specifically, each XAI method produces one CWE vector, as demonstrated in Algorithm 1. We aggregated and computed the average of the contribution values indexed by the syntactic constructs from different XAI methods and derived the final CWE vector. This result helps us to quantify the contributions at the level of the syntactic constructs, in addition to the code tokens, in the classification task of vulnerability code CWE types.

We analyzed the complexity of our algorithms. Given the size of the dataset samples $N$, the feature number $P$, and the CWE label number $K$, the complexity of Algorithm 1 is $\Theta(P \times K \times \Theta(\Phi))$, in which, for SHAP, $\Theta(\Phi) = \Theta(N \times (2^P + P^3))$, and for Mean-Centroid PredDiff, $\Theta(\Phi) = \Theta(N \times P^2)$.

**Algorithm 1** Compute the CWE vector of each syntactic construct's contribution value

**Input:**

- The input dataset $X$;
- The full AST construct forms the feature set $P = \{1, ..., j, ...p\}$;
- The subset $S \subseteq P \backslash \{j\}$ by masking feature $j$;
- The feature $j$ contribution value $\phi_j = \Phi(P, S, j, \hat{f}(X))$;
- The model prediction under feature $j$ masking $\hat{f}(X)$;
- The CWE label set $K = \{cwe_1, ..., cwe_k, ..., cwe_N\}$.

/* Partition dataset by ground truth CWE label */
**for all** $x_i \in X$ **do**
  **if** $x_i$ owns label $cwe_k$ **then**
    Add $x_i$ to $X^{cwe_k}$
  **end if**
**end for**
/* Compute CWE vector of feature contribution value */
**for all** $X^{cwe_k}$ **do**
  **for all** $j \in P$ **do**
    $\phi_j^{cwe_k} = \Phi(P, S, j, \hat{f}(X^{cwe_k}))$
  **end for**
  $\overline{\phi_j^{cwe_k}} = \frac{1}{||P||} \sum \phi_j^{cwe_k}$
  /* Create CWE vector */
  **for all** $j \in P$ **do**
    $V^{cwe_k} \leftarrow \langle j, \overline{\phi_j^{cwe_k}} \rangle$
  **end for**
**end for**
/* Sort elements in descending order by feature contribution values */
**for all** $cwe_k \in K$ **do**
  $V^{cwe_k} \leftarrow \{sort(V^{cwe_k})\}$
**end for**

**Output:** The CWE vector for each CWE label $cwe_k \in K, V^K$.

### *4.5. CWE Similarity Assessment*

The assessment of CWE similarity consists of two steps. The first step is to derive the CWE similarity pairs from the XAI explanation summary. The second step is to validate the CWE similarity from the baseline (ground truth) from the knowledge base of the CWE community.

**CWE Similarity Score.** We represent the similarity score between CWEs as $\rho$. It is derived from the normalized ranking distance [87] between two sorted CWE vectors. The value of $\rho$ ranges from zero, indicating identical CWE pairs, to one, indicating complete dissimilarity. A lower $\rho$ value indicates a higher similarity between a pair of CWEs. We sorted the $\rho$ values of CWEs and listed CWEs in descending order of their similarity in terms of ranking with a given CWE.

The CWE similarity assessment follows the simplified steps below, with details outlined in Algorithm 2:

1. Based on the sorted CWE vectors, we compute the similarity score between CWE types. Therefore, any pair of CWE types has a similarity score.
2. Given a CWE type, we rank the highest similarity score in all the pairs that involve this CWE type.
3. Given a CWE type, the CWE type that has the highest similarity score becomes the most similar to the given CWE type.

**Algorithm 2** Compute CWE similarity vector for CWE types

**Input:**

- Sorted vectors of feature importance for each CWE label $V^{cwe_k}$ output from Algorithm 1;
- The CWE label set $K = \{cwe_1, ..., cwe_k, ..., cwe_N\}$.

Initialize an empty array $d$ for storing ranking distances.
**for all** distinct pairs of CWE labels $\langle cwe_i, cwe_j \rangle \in K \times K$ **do**
  /*Calculate Kendall Tau ranking distance between CWE vectors*/
  $d_{ij} \leftarrow distance(V^{cwe_i}, V^{cwe_j})$.
  Store $d_{ij}$ in vector $d$.
**end for**
$d_{max} = max(d)$.
/* Compute normalized CWE similarity distance */
**for all** CWE label $cwe_j$ **do**
  $\rho(cwe_i, cwe_j) = \frac{d_{ij}}{d_{max}}$.
  $W^{cwe_i} \leftarrow \langle cwe_j, \rho(cwe_i, cwe_j) \rangle$.
**end for**
/* Sort elements in descending order by value of $\rho_{ij}$ */
**for all** $cwe_k \in K$ **do**
  $W^{cwe_k} \leftarrow \{sort(W^{cwe_k})\}$.
**end for**

**Output:** The CWE similarity vector for each CWE label $cwe_k \in K, W^K$.

These pair-wise CWE similarity results are derived from the learning process combined with XAI methods. Meanwhile, we developed the baseline similarity pairs. As an example, in Figure 2, any siblings of two CWE types in [60] from the same parent CWE type form a pair of CWE similarities. The complexity is $\Theta(K^2)$, where $K$ is the number of CWE types.

**CWE Similarity Validation.** Further, we compared the pair-wise CWE similarity derived by XAI methods with the baseline CWE similarity in terms of four metrics, namely Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score $\overline{S}$. Top-N Similarity Hit and Average Normalized Similarity Score focus on the occurrence of a certain CWE type in our explanation in the baseline. MRR and MAP focus on the occurrence ranking of a CWE type in the baseline. The better the score values, the better the explanation quality; thus, the more accurate the syntactic constructs' contribution values.

The complexity of Algorithm 2 depends on the number of CWE pair combinations. Table 2 shows the baseline of CWE similarity defined by the open community. The CWE types are classified in a tree structure. The sibling leaves share a commonality with the parent CWE type. CWE22, CWE23, and CWE36 fall under the path's traversal weakness. Then, CWE22, CWE23, and CWE36 form three CWE similarity pairs.

**Table 2.** CWE categorized by baseline similarities [60].

| Category | Similar CWEs |
|---|---|
| Path traversal and resource management issues | CWE22, CWE23, CWE36 |
| Trust boundaries and privilege management | CWE500, CWE501, CWE15 |
| Buffer errors | CWE119, CWE120 |
| Injection vulnerabilities | CWE78, CWE79, CWE89, CWE90, CWE643, CWE789 |
| Cryptographic and sensitive data handling issues | CWE327, CWE328, CWE330, CWE614 |
| Use of pointer subtraction to determine size | CWE469 |
| NULL pointer dereference | CWE476 |

To validate the CWE's similarity to the XAI explanation, we apply four metrics to compare with the baseline: Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score $\overline{S}$. $B^{cwe_i}$ is the set that contains all the CWE types that are siblings to $cwe_i$ defined in the baseline. $W^{cwe_k}$ is the set of CWE types derived from Algorithm 2. Given the example of CWE23, $B^{cwe_{23}} = \{cwe_{22}, cwe_{36}\}$ and $W^{cwe_{23}} = \{cwe_{22}, cwe_{79}, ..., cwe_{36}\}$, each metric can be illustrated as follows.

1. **Top-N Similarity Hit** is defined as a boolean value. For example, Top-1 Similarity Hit of CWE22 equals one.

$$H_N^{cwe_k} = \begin{cases} 0 & \text{if } B^{cwe_k} \cap W^{cwe_k} \equiv \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

2. **Mean Reciprocal Rank (MRR)** measures the mean reciprocal rank given a CWE type $cwe_i$.

$$MRR^{cwe_i} = \frac{1}{||B^{cwe_i}||} \sum_{1}^{||B^{cwe_i}||} \frac{1}{rank_{cwe_j}}, \forall cwe_j \in B^{cwe_i} \tag{4}$$

where $rank_{cwe_j}$ is the position index value of $cwe_j$ in $W^{cwe_j}$. In the example of $W^{cwe_{23}}$, CWE22 is ranked as one and CWE36 is ranked as $k = 14$. $MRR^{cwe_{23}} = \frac{1}{2} \times (1 + \frac{1}{k}) = \frac{1}{2} \times (1 + \frac{1}{14}) = 0.5357$.

3. **Mean Average Precision (MAP)** is a metric used to measure the XAI explanation accuracy of CWE type similarity by averaging the precision of each CWE type's similarity rank. Let $W_N^{cwe_i}$ represent the top-N subset of $W^{cwe_i}$, where N represents a cut-off rank. For a given CWE type $cwe_i$, Average Precision (AP) is calculated as the mean precision value at each rank:

$$AP^{cwe_i} = \frac{1}{||B^{cwe_i}||} \sum_{\kappa=1}^{N} \frac{||B^{cwe_i} \cap W_\kappa^{cwe_i}||}{||W_\kappa^{cwe_i}||} \cdot rel(\kappa) \tag{5}$$

where $rel(\kappa)$ is an indicator function that equals one if the item at rank $\kappa$ is a ground truth sibling CWE type of $cwe_i$ , that is, $W^{cwe_i}[\kappa] \in B^{cwe_i}$, and is zero otherwise. In the example of $W^{cwe_{23}}$, CWE22 is ranked as one and CWE36 is ranked as $k = 14$. $AP^{cwe_{23}} = \frac{1}{2} \times (1 + \frac{2}{k}) = \frac{1}{2} \times (1 + \frac{2}{14}) = 0.5714$. Finally, given an XAI explanation

method $\Phi$, MAP is calculated as the mean average precision of all $Q$ number of CWE types:

$$MAP^{\Phi} = \frac{1}{Q}\sum_{q=1}^{Q} AP^{cwe_q} \tag{6}$$

4. **Average Normalized Similarity Score ($\overline{S}$)** measures the average normalized similarity score for all CWE types in the baseline.

$$\overline{S} = \frac{\sum_{cwe_k \in B^{cwe_i}}^{||B^{cwe_i}||} \sum_{cwe_j \in W^{cwe_i}}^{||W^{cwe_i}||} (1 - \rho(\text{cwe}_k, \text{cwe}_j))}{||B^{cwe_i}|| \cdot ||W^{cwe_i}||} \tag{7}$$

where $\rho(\text{cwe}_k, \text{cwe}_j)$ represents the similarity between a CWE type $cwe_k$ in the baseline and a CWE type $cwe_j$ derived using an XAI method. $\rho(\text{cwe}_k, \text{cwe}_j)$ is calculated in Algorithm 2.

## 5. The Evaluation and Results

The evaluation aims to assess the importance of the contribution of syntactic constructs. These syntactic constructs are inflection nodes in the Abstract Syntax Tree (AST) that connect code token nodes in a path that convey semantic meanings [88]. We summarize the feature importance of syntactic constructs using XAI methods. We also validate the CWE similarity pairs from XAI explanations in comparison with the baseline from the community knowledge base. We present research experiments that could answer two specific research questions, as follows.

**RQ1. What are the top-ranking syntactic constructs that contribute most to the multi-classification of software vulnerability?** This question relies on the XAI methods to determine the importance of code tokens traversing syntactic construct paths that contribute to the deep learning model's prediction for various vulnerability types.

**RQ2. How does the CWE similarity summarized by XAI methods align with the expert-defined similarity?** This question applies the measurement of CWE similarity pairs to the baseline CWE similarity pairs to validate whether the explanations of syntactic constructs correspond to the expert-established ground truth. Thus, the explanation maps the syntactic constructs to human-understandable CWE types of semantic meanings of vulnerable artifacts.

### *5.1. Datasets*

Our experiment examined three benchmark software vulnerability datasets at the method or function level, including the Juliet Test Suite (Java) [62], OWASP Benchmark (Java) [89], and Draper (C/C++) [6]. These datasets can be sorted into three categories [23] based on the method of collection and annotation of the code samples. They represent synthetic, semi-synthetic, and real data, respectively.

Synthetic data refer to instances where both the vulnerability code example and its annotations are artificially constructed. The Juliet Test Suite, a product of the National Security Agency's Center for Assured Software, falls under this category. By comprising 217 vulnerable methods (42%) and 297 non-vulnerable methods (58%), this dataset provides a balanced distribution of method-level examples, all synthesized based on recognized vulnerable patterns.

Semi-synthetic data involve either the code or its annotation being artificially derived. The OWASP Benchmark dataset, also Java-based, is an example of semi-synthetic data. We captured 1415 vulnerable methods (52%) and 1325 non-vulnerable methods (48%) from this dataset.

Real data, on the other hand, involve code and corresponding vulnerability annotations sourced from real-world repositories. The Draper dataset fits into this category. The functions in this dataset are collected from open-source repositories and annotated using static analyzers. While the original dataset presented an imbalanced distribution, we

reprocessed it into a balanced dataset to analyze vulnerable code typed and their constructs and characters, and to preserve all comments and code. Consequently, this dataset includes 43,506 (50.1%) vulnerable functions. In Table 3, we summarize the vulnerability types and their respective distributions in each dataset. For vulnerability types that have less than 1% distribution, we group them all into a `CWE-Other` type.

**Table 3.** CWE distribution by dataset.

| Dataset | CWE | CWE Name | Percentage |
|---|---|---|---|
| **OWASP** | CWE22 | Path Traversal | 9.4% |
| | CWE78 | OS Command Injection | 8.9% |
| | CWE79 | Cross-site Scripting | 17.4% |
| | CWE89 | SQL Injection | 19.2% |
| | CWE90 | LDAP Injection | 1.9% |
| | CWE327 | Crypt. Issue | 9.2% |
| | CWE328 | Info. Leak | 9.1% |
| | CWE330 | Data Exposure | 15.4% |
| | CWE501 | Trust Boundary | 5.8% |
| | CWE614 | Sensitive Cookie | 2.5% |
| | CWE643 | XPath Injection | 1.2% |
| **Juliet** | CWE15 | External Control of System or Configuration Setting | 11.1% |
| | CWE23 | Relative Path Traversal | 6.0% |
| | CWE36 | Absolute Path Traversal | 11.1% |
| | CWE500 | Public Static Field Not Marked Final | 1% |
| | CWE643 | XPath Injection | 5.5% |
| | CWE78 | OS Command Injection | 5.5% |
| | CWE789 | Uncontrolled Memory Allocation | 25.3% |
| | CWE89 | SQL Injection | 32.3% |
| | CWE-Other | Other | 2.9% |
| **Draper** | CWE119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 28.4% |
| | CWE120 | Classic Buffer Overflow | 26.9% |
| | CWE-Other | Other | 26.7% |
| | CWE476 | NULL Pointer Dereference | 11.9% |
| | CWE469 | Use of Pointer Subtraction to Determine Size | 6.1% |

### *5.2. Assessing Contribution of Syntactic Constructs (RQ1)*

The three-step approach for assessing syntactic construct importance with settings includes the following:

***Step 1:*** Converting Graph Context. We utilized the srcML tool [83] to transform the method-level program into an AST structure. In this process, we removed code comments and retained mathematical and logical operators. The output from srcML is the XML-based content, encompassing both the code token (the leaf nodes in AST) and the AST path. This content was subsequently converted into a graph context, as introduced in Section 4.1. We retained the maximum `edge length` of eight and the `window` of ten as default values in the graph neural network model [39].

***Step 2:*** Learning Embedding for Code Tokens. The graph context of each target token was used to learn the embedding of the target token. The graph convolutional neural network model was connected with a classifier layer for downstream classification tasks. The graph convolutional neural network model has one layer with a batch size of 64 and a dropout rate of 0. The embedding vector dimension is 128, which represents each code token for the classification models.

***Step 3:*** Feature Masking and Feature Importance Ranking. We maintained the full graph context to retrieve embedding sets for the entire program's code tokens. After masking each syntactic construct, we obtained altered neighbour tokens in a graph context

as a form of feature masking to XAI methods SHAP and Mean-Centroid PredDiff. We compiled the results by averaging the contribution values across XAI methods.

***Results.*** Table 4 *(for Step 2)* presents the performance of three classifiers augmented with GCN-based embeddings on Juliet, OWASP, and Draper datasets. The TextCNN classifier outperforms Random Forest and Transformer on all three datasets. We then chose TextCNN as the classifier with GCN embeddings to perform the following XAI tasks.

Figure 8 *(for Step 3)* shows, for each CWE type, the importance ranking of the meta syntactic constructs categorized in Table 1. We observe that despite the varying importance orders of the syntactic constructs for each CWE type, certain constructs such as `statement_subelements, parameters, name, statement` are consistently ranked highly across multiple CWEs, suggesting their general impact on code vulnerabilities. For instance, CWE78, CWE79, and CWE89 share similar top-ranked constructs, such as `statement_subelements, name, decl_def_init, and operators`. On the other hand, syntactic constructs such as `specifier, classes` have a lower importance across CWEs.

**Answer summary to RQ1:** Syntactic constructs `statement_subelements,statement, name,` and `parameters` consistently rank highly across sixteen CWE types, approximately 80% of all CWE types, indicating their contribution to code vulnerability classification.

**Table 4.** Performance of classifiers augmented with GCN embeddings.

| Model | Metric | **Juliet** | **OWASP** | **Draper** |
|---|---|---|---|---|
| **Random Forest** | F1-Score | 0.8074 | 0.5826 | 0.7121 |
| | Precision | 0.8276 | 0.6031 | 0.7430 |
| | Recall | 0.7881 | 0.5634 | 0.6837 |
| **TextCNN** | F1-Score | **0.8358** | **0.6956** | **0.7569** |
| | Precision | 0.8412 | 0.6919 | 0.7470 |
| | Recall | 0.8305 | 0.6993 | 0.7671 |
| **Transformer** | F1-Score | 0.7830 | 0.6200 | 0.7383 |
| | Precision | 0.7714 | 0.6310 | 0.6983 |
| | Recall | 0.7950 | 0.6094 | 0.7831 |

*Note:* The model with the best performance, as indicated by the F1-Score, is highlighted in bold.

### *5.3. CWE Similarity Explained by XAI Methods (RQ2)*

Driven by the observations of syntactic similarities among certain CWE types, we further quantified CWE similarity based on the feature importance rank. We then compared these results with an expert-defined CWE similarity baseline.

***Step 1:*** We computed the CWE similarity based on the importance rank of syntactic constructs. Figure 8 shows the importance values of nine metadata syntactic constructs grouped by CWE type. We further expanded the assessment of forty syntactic constructs to obtain the CWE similarity distance of any two CWE pairs using the full list of syntactic construct importance ranking, following Algorithm 2.

***Step 2:*** We validated our XAI-based CWE similarity against the expert-defined baseline [60]. The similar sibling set for each CWE (required in Algorithm 2) is listed in Table 2. To assess our results, we employed four metrics to measure each CWE type, including Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score (ANSS) (Section 4.5). We obtained an aggregated score by averaging across all CWE types listed in Table 5.

***Results.*** Figure 9 *(for step 1)* presents the CWE similarity $\rho$ of three datasets. For instance, CWE23 and CWE22 show a strong similarity with a low distance value, indicating that they share similar syntactic constructs. On the other hand, the pair consisting of CWE23 and CWE328 has a high distance value in the matrix, indicating low similarity between the pair.

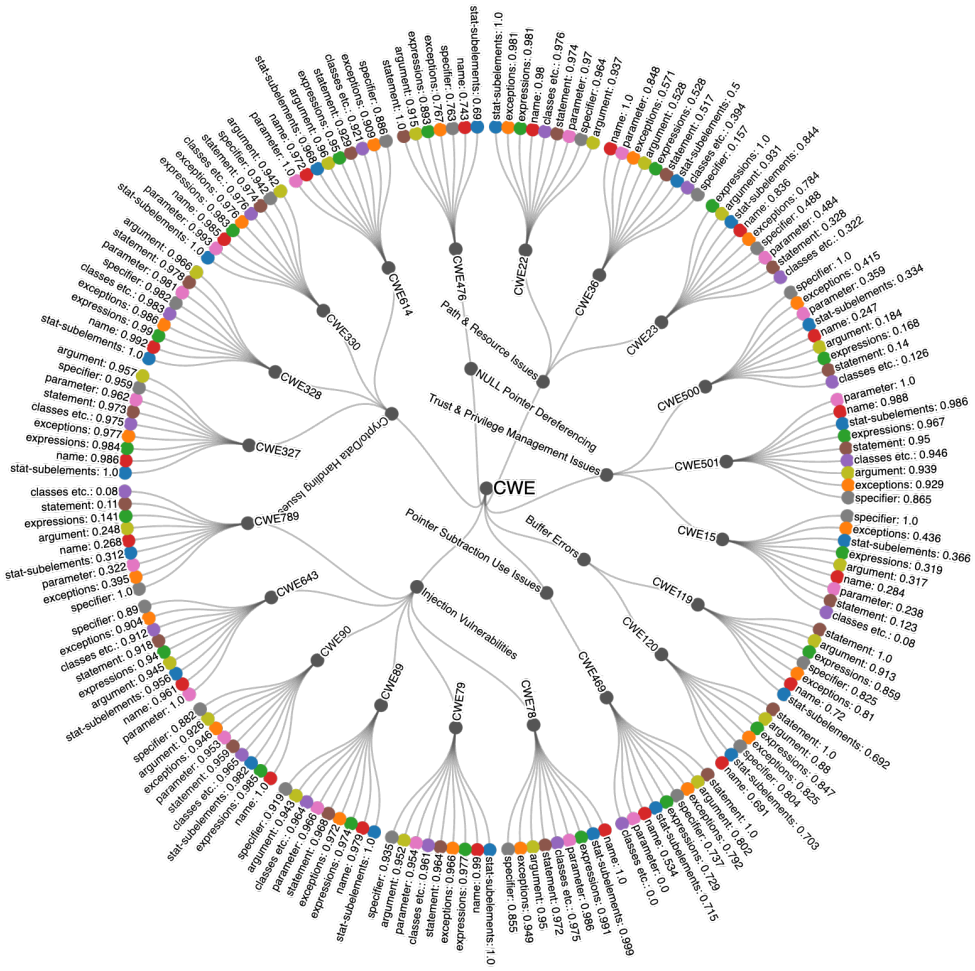


**Figure 8.** Feature importance of meta syntactic constructs per CWE type, represented in descending order clockwise. Importance is quantified as the normalized feature contribution value from the XAI method, shown in the leaf nodes after the contract's name. CWEs that describe similar vulnerability issues [60] are also categorized in the dendrogram.

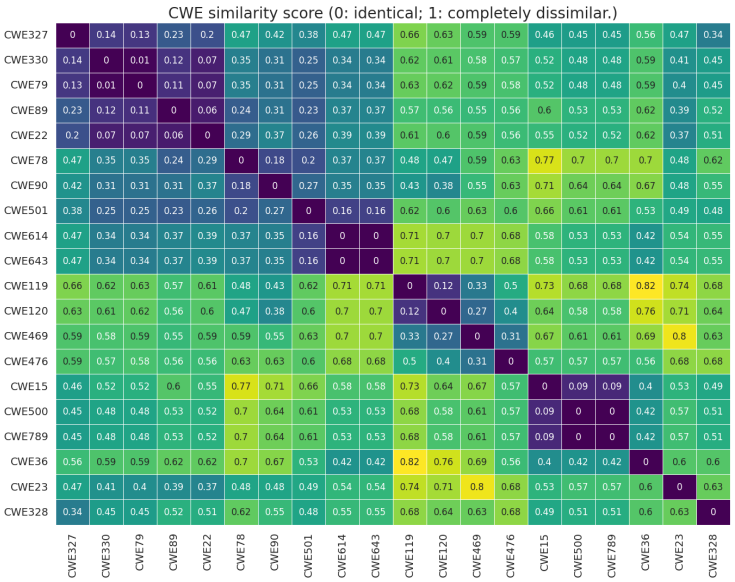


**Figure 9.** CWE similarity score $\rho(cwe_i, cwe_j)$ for CWE pair from syntactic construct feature importance based on XAI approach.

**Table 5.** CWE similarity evaluation results.

| CWE | Top1 | Top3 | Top5 | MRR | Average Precision | ANSS ($\overline{S}$) |
|---|---|---|---|---|---|---|
| CWE23 | 1 | 1 | 1 | 0.536 | 0.572 | 0.802 |
| CWE327 | 1 | 1 | 1 | 0.393 | 0.736 | 0.628 |
| CWE330 | 1 | 1 | 1 | 0.372 | 0.728 | 0.247 |
| CWE79 | 1 | 1 | 1 | 0.372 | 0.728 | 0.250 |
| CWE89 | 0 | 1 | 1 | 0.269 | 0.630 | 0.328 |
| CWE22 | 0 | 0 | 0 | 0.089 | 0.115 | 0.118 |
| CWE78 | 1 | 1 | 1 | 0.360 | 0.687 | 0.622 |
| CWE90 | 1 | 1 | 1 | 0.377 | 0.743 | 0.610 |
| CWE501 | 1 | 1 | 1 | 0.533 | 0.767 | 0.774 |
| CWE614 | 1 | 1 | 1 | 0.524 | 0.738 | 0.761 |
| CWE643 | 1 | 1 | 1 | 0.524 | 0.738 | 0.761 |
| CWE328 | 0 | 0 | 1 | 0.144 | 0.233 | 0.620 |
| CWE36 | 0 | 0 | 0 | 0.084 | 0.122 | 0.661 |
| CWE15 | 1 | 1 | 1 | 0.750 | 1 | 1 |
| CWE500 | 1 | 1 | 1 | 0.750 | 1 | 1 |
| CWE789 | 1 | 1 | 1 | 0.750 | 1 | 1 |
| CWE469 | - | - | - | - | - | - |
| CWE476 | - | - | - | - | - | - |
| CWE119 | 1 | 1 | 1 | 1 | 1 | 1 |
| CWE120 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Mean** | **0.778** | **0.833** | **0.889** | **0.491** | **0.696** | **0.677** |

*Note:* Top-1/3/5 represents the Top-N Similarity Hit, MRR represents Mean Reciprocal Rank, MAP represents Mean Average Precision, each row is the Average Precision (AP) of a CWE, and $\overline{S}$ represents the Average Normalized Similarity Score. CWE469 and CWE476 do not have a similar CWE in the datasets.

The results presented in Table 5 *(for step 2)* evaluate the similarity of CWE types, explained by XAI methods, compared to the baseline. The average Top-1 hit is approximately 78%, which means our XAI approach is able to identify accurate siblings for 78% of CWE types. Considering the Top-5 hit, the accuracy in identifying the siblings improved to 89%. Top-N metrics focus on the existence of a sibling CWE using the XAI explanation.

The Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) further consider the ranking of a sibling CWE type derived from the XAI explanation. In addition, MAP is approximately 70%, considering both the number of existing CWEs and their rankings. We revisit the example of CWE23, which has CWE22 and CWE36. The XAI methods provide the similarity assessment with CWE22 in the first position and CWE36 in the 14th position. This is the reason for CWE23's lower MRR value and MAP value.

**Answer summary to RQ2:** The XAI approach identifies the similarity between CWE types through the changes incurred in deep learning model's classification due to feature masking. We applied metrics to evaluate the alignment of the XAI-derived similarity with the expert-established baseline by measuring both the occurrence and occurrence rankings of similar CWE types. The alignment connects the deep learning feature representations with the human-understandable CWE types. In addition to deep learning's vulnerability classification, our XAI approach returns along the syntactic construct paths to locate the code that could lead to the misclassification of a similar CWE type.

### *5.4. Reflection on the Motivating Case*

Studying the case presented in Section 2.1, we observe that the graph-based learning model faces misclassification among similar CWE types, as illustrated in Figure 10. In the

example presented in Figure 2, a key difference in the processing of paths in CWE23 and CWE36 becomes evident. CWE23 employs a relative path embodied in the expression `File file = new File(root + data)`, while CWE36 uses an absolute path, represented as `File file = new File(data)`. Our XAI approach probes the feature importance in the terms of syntactic constructs to explain the potential cause of miscalculation. Both CWE23 and CWE36 cases exhibit similar overall feature importance ranking sequences, with `name` and `if` ranked in the top two syntactic constructs. However, in the case of CWE23, constructs such as `argument_list, argument` and `operator` are ranked higher than they are in the case of CWE36. This subtle feature ranking difference aligns with the unique characteristics of CWE23, which incorporates an additional argument `root` and an operator with + into the file, thereby creating the relative path.

### *5.5. Summary of Findings and Existing Research*

We analyzed our findings via a comparative summary of previous studies. We focused on the insights drawn from the current research, areas of alignment, and novel discoveries listed in Table 6.

According to the manifesto of XAI [27], our work is within the scope of the application of attribute-based XAI methods. We demonstrate that our work relates to two aspects of the manifesto, as follows: (1) evaluating XAI methods and the explanation—we integrated the SOTA XAI methods within a pipeline where the semantic relations among the CWE types defined by domain experts are encoded as the XAI output ranking and explanation evaluation; (2) supporting the human-centeredness of explanations—we were able to identify and rank the contributions of syntactic constructs across languages. Such information helps to explain the difference at the syntactic level between entities relative to the misclassification of similar CWE types in a human-understandable way.

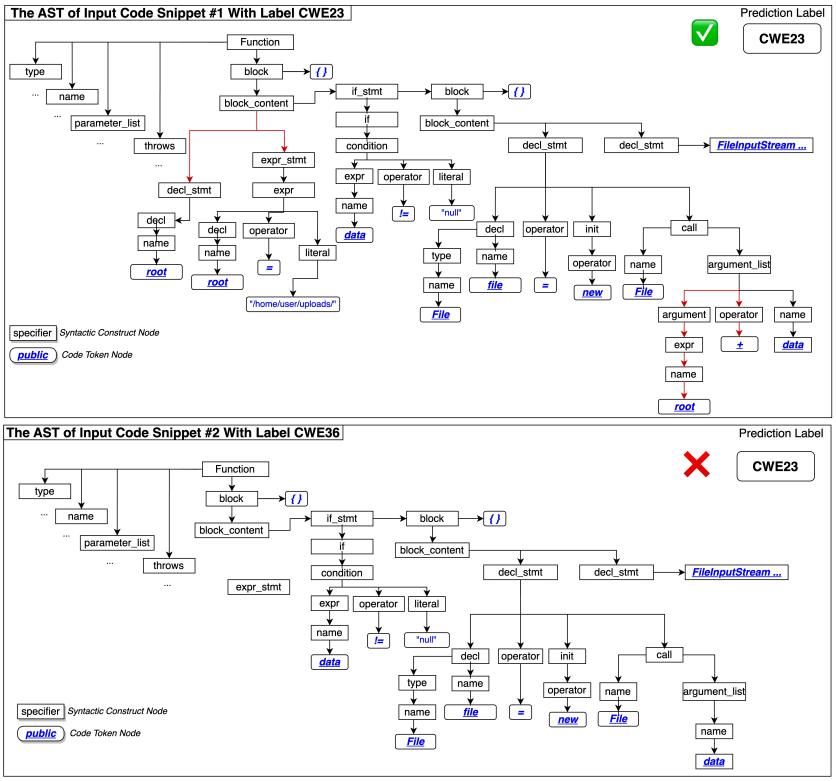


**Figure 10.** The CWE23 code snippet contains two additional AST paths (marked with red), with `argument` and `operator`, to make the absolute path into a relative path, compared with CWE36.

**Table 6.** Summary of our findings compared with existing work.

| Assessment Type | Our Findings | Existing Work | Our Contributions |
|---|---|---|---|
| Syntactic constructs importance | Comprehensive importance ranking of nine metadata syntactic constructs and over 40 syntactic constructs across 20 CWEs are presented in Figure 8; `statement_subelements`, `statement`, `name`, `parameters` are of high rankings across 80% CWEs. | Studies [32,33,39,40] propose models with code token embeddings and AST path embeddings to learn the vulnerability pattern. These models do not focus on explaining how a certain classification is produced. Studies [48,58] highlight `name` and `statement_subelements` as the most important factors without a full assessment of all syntactic constructs. | Full evaluation of syntactic constructs' importance (forty constructs; nine metadata constructs across twenty CWEs) provided. |
| CWE similarity | Comparison of CWE similarity with expert-defined baseline using multiple metrics; 77.8% Top1 similarity CWE hit rate and a MAP score of 0.696 are achieved. | The community provides an expert-defined baseline CWE similarity summary [60]. | The CWE similarity is derived from a data-driven and XAI-based retrospective approach. The similarity explanation provides a probing view for understanding the feature effects on the deep learning model's classification for similar CWE types under subtle code variations. |

## 6. Threats to Validity

The validity of our work includes the following factors: (1) internal validity stemming from limited model evaluation and the use of specific datasets for XAI and (2) external validity threats related to the transferability of the identified importance and similarity of syntactic constructs to the emerging CWE types.

**Dataset.** We employed three datasets, namely Juliet, OWASP, and Draper. Both Juliet and OWASP consist of synthetic samples with artificially constructed annotations, which may limit their generalizability to real-world data. Draper consists of samples derived from real-world source codes. However, Draper does not share overlapping CWE types with either Juliet or OWASP. The disjointed datasets in the common CWE types mean that we cannot obtain a cross-validation of the top-ranking sequences of the syntactic constructs of the common CWE types across multiple datasets. As a result, we cannot further validate the consistency of the XAI explanation across datasets. In our previous work [31,61], we defined the explanation consistency metrics to measure the explanations across multiple datasets.

**Models.** Our XAI-based framework, depicted in Figure 3, is model-agnostic, including the embedding and classifier models. We applied one graph-embedding model adopted from GraphCodeVec [39], three deep learning models to be used as classifiers, and two XAI methods. The variance incurred by different models can be further evaluated by introducing more models to assess the explanation stability [31,61] across multiple models on the same datasets.

**Transferability to Broader CWE Sets.** Our study involves 20 CWE types beyond the 1% distribution percentage from three datasets. These 20 CWE types include six of the top twenty-five most dangerous software weaknesses listed by the CWE community [90]. In the full list of CWE types, the issue is the imbalanced data samples and the lack of labelled real datasets akin to Draper. In this paper, we preprocessed the Draper dataset to ensure it was balanced, since a poor classification performance from the imbalanced dataset causes the explanation results to be meaningless. Both the explanation stability and the consistency can be further validated with larger, balanced datasets.

## 7. Conclusions

In this work, we established the explanation for the program code in a graph context as the features and semantics of vulnerability types collectively defined by open community experts. Our study begins by defining a feature type taxonomy of code representations,

and subsequently progresses to analyze syntactic constructs within abstract syntax-tree-based graph code representations. We developed an XAI-based framework to explain the relationship among the combination of 20 code vulnerability types and over 40 syntactic constructs from three Java and C++ datasets. We observed that the variation in the syntactic construct importance rankings relates to the intrinsic similarities amongst certain CWEs that share common vulnerability characteristics. We thus derived the CWE similarity based on the XAI explanation summary and validated it using the expert-defined baseline. We applied four types of information retrieval metrics to evaluate our XAI-based results. Our study links the comprehension of code semantics and syntactic feature representation learned by deep learning models for vulnerability classification.

**Author Contributions:** Conceptualization, Y.L. and D.L.; methodology, D.L. and Y.L.; software, D.L. and J.H.; validation, D.L. and Y.L.; formal analysis, D.L. and Y.L.; investigation, D.L. and Y.L.; resources, Y.L.; data curation, D.L. and J.H.; writing—original draft preparation, D.L.; writing—review and editing, D.L. and Y.L. and J.H.; visualization, D.L.; supervision, Y.L.; project administration, Y.L.; funding acquisition, Y.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research is supported by Canada Natural Sciences and Engineering Research Council Discovery Grant under RGPIN-2020-06797.

**Data Availability Statement:** The datasets used in this study are sourced from the open-source Juliet Test Suite for Java [62], the OWASP Benchmark for Java [89], and the Draper C/C++ suite [6]. The processed dataset example is available on GitHub: https://github.com/DataCentricClassificationofSmartCity/XAI-based-Software-Vulnerbility-Dection/tree/main/dataset, accessed on 1 May 2024.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. National Institute of Standards and Technology (NIST). *Vulnerability Definition*; Computer Security Resource Center: Gaithersburg, MA, USA, 2012.
2. Dam, H.K.; Tran, T.; Pham, T.; Ng, S.W.; Grundy, J.; Ghose, A. Automatic feature learning for predicting vulnerable software components. *IEEE Trans. Softw. Eng.* **2019**, *47*, 67–85. [CrossRef]
3. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. $\mu$ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2224–2236. [CrossRef]
4. Ghaffarian, S.M.; Shahriari, H.R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 56. [CrossRef]
5. Shin, Y.; Meneely, A.; Williams, L.; Osborne, J.A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.* **2010**, *37*, 772–787. [CrossRef]
6. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; pp. 757–762.
7. Zimmermann, T.; Nagappan, N.; Williams, L. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, Paris, France, 6–10 April 2010; pp. 421–428.
8. Lin, G.; Wen, S.; Han, Q.L.; Zhang, J.; Xiang, Y. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
9. Morrison, P.; Herzig, K.; Murphy, B.; Williams, L. Challenges with applying vulnerability prediction models. In Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, Urbana, IL, USA, 21–22 April 2015; pp. 1–9.
10. Wheeler, D.A. Flawfinder. 2021. Available online: https://github.com/david-a-wheeler/flawfinder (accessed on 1 May 2024).
11. Checkmarx. Checkmarx Software Security Platform. 2021. Available online: https://www.checkmarx.com (accessed on 1 May 2024).
12. Kals, S.; Kirda, E.; Krügel, C.; Jovanovic, N. SecuBat: A Web Vulnerability Scanner. In Proceedings of the 15th International Conference on World Wide Web, Edinburgh, UK, 23–26 May 2006; pp. 247–256.
13. PortSwigger. Burp Suite Web Vulnerability Scanner. 2021. Available online: https://portswigger.net/burp (accessed on 1 May 2024).
14. Acunetix. Acunetix Web Vulnerability Scanner. 2021. Available online: https://www.acunetix.com/vulnerability-scanner (accessed on 1 May 2024).
15. Nadeem, M.; Williams, B.J.; Allen, E.B. High false positive detection of security vulnerabilities: A case study. In Proceedings of the 50th Annual Southeast Regional Conference, Tuscaloosa, AL, USA, 29–31 March 2012; pp. 359–360.

16. Shin, Y.; Williams, L. An empirical model to predict security vulnerabilities using code complexity metrics. In Proceedings of the 2nd ACM-IEEE IEEE International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, 9–10 October 2008; pp. 315–317.
17. Shin, Y.; Williams, L. Can traditional fault prediction models be used for vulnerability prediction? *Empir. Softw. Eng.* **2013**, *18*, 25–59. [CrossRef]
18. Sestili, C.D.; Snavely, W.S.; VanHoudnos, N.M. Towards security defect prediction with AI. *arXiv* **2018**, arXiv:1808.09897.
19. Lin, G.; Tang, M.; Wang, Y.; Luo, W.; Luo, X.; Liao, X. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Informat.* **2018**, *14*, 3289–3297. [CrossRef]
20. Jiang, J.; Wen, S.; Yu, S.; Xiang, Y.; Zhou, W. Identifying propagation sources in networks: State-of-the-art and comparative studies. *IEEE Commun. Surveys Tuts.* **2017**, *19*, 465–481. [CrossRef]
21. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.
22. Wang, H.; Ye, G.; Tang, Z.; Tan, S.H.; Huang, S.; Fang, D.; Feng, Y.; Bian, L.; Wang, Z. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forensics Secur.* **2020**, *16*, 1943–1958. [CrossRef]
23. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Softw. Eng.* **2022**, *48*, 3280–3296. [CrossRef]
24. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; De Vel, O.; Montague, P.; Xiang, Y. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2469–2485. [CrossRef]
25. Zeng, P.; Lin, G.; Pan, L.; Tai, Y.; Zhang, J. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access* **2020**, *8*, 197158–197172. [CrossRef]
26. Gunning, D.; Stefik, M.; Choi, J.; Miller, T.; Stumpf, S.; Yang, G.Z. XAI—Explainable artificial intelligence. *Sci. Robot.* **2019**, *4*, eaay7120. [CrossRef]
27. Longo, L.; Brcic, M.; Cabitza, F.; Choi, J.; Confalonieri, R.; Del Ser, J.; Guidotti, R.; Hayashi, Y.; Herrera, F.; Holzinger, A.; et al. Explainable artificial intelligence (XAI) 2.0: A manifesto of open challenges and interdisciplinary research directions. *Inf. Fusion* **2024**, *106*, 102301. [CrossRef]
28. Lundberg, S.M.; Lee, S.I. A unified approach to interpreting model predictions. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 4768–4777.
29. Ribeiro, M.T.; Singh, S.; Guestrin, C. "Why should I trust you?" Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 1135–1144.
30. Guo, W.; Mu, D.; Xu, J.; Su, P.; Wang, G.; Xing, X. Lemna: Explaining deep learning based security applications. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 364–379.
31. Li, D.; Liu, Y.; Huang, J.; Wang, Z. A Trustworthy View on Explainable Artificial Intelligence Method Evaluation. *Computer* **2023**, *56*, 50–60. [CrossRef]
32. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
33. Hariharan, M.; Tanwar, A.; Sundaresan, K.; Ganesan, P.; Ravi, S.; Karthik, R. Proximal Instance Aggregator networks for explainable security vulnerability detection. *Future Gener. Comput. Syst.* **2022**, *134*, 303–318.
34. Sotgiu, A.; Pintor, M.; Biggio, B. Explainability-based Debugging of Machine Learning for Vulnerability Discovery. In Proceedings of the 17th International Conference on Availability, Reliability and Security, Vienna, Austria, 23–26 August 2022; pp. 1–8.
35. Jin, C.; Rinard, M. Evidence of Meaning in Language Models Trained on Programs. *arXiv* **2023**, arXiv:2305.11169.
36. Christey, S.; Kenderdine, J.; Mazella, J.; Miles, B. *Common Weakness Enumeration*; Mitre Corporation: McLean, VA, USA, 2013.
37. Hariyanti, E.; Djunaidy, A.; Siahaan, D. Information security vulnerability prediction based on business process model using machine learning approach. *Comput. Secur.* **2021**, *110*, 102422. [CrossRef]
38. Pan, S.; Bao, L.; Xia, X.; Lo, D.; Li, S. Fine-grained Commit-level Vulnerability Type Prediction by CWE Tree Structure. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 957–969.
39. Ding, Z.; Li, H.; Shang, W.; Chen, T.H. Towards Learning Generalizable Code Embeddings using Task-agnostic Graph Convolutional Networks. *ACM Trans. Softw. Eng. Methodol.* **2022**, *32*, 1–43. [CrossRef]
40. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Shujie, L.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the International Conference on Learning Representations, Addis Ababa, Ethiopia, 26–30 April 2020.
41. Allen, F.E. Control flow analysis. *ACM Sigplan Not.* **1970**, *5*, 1–30. [CrossRef]
42. Ferrante, J.; Ottenstein, K.J.; Warren, J.D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1987**, *9*, 319–349. [CrossRef]
43. Nguyen, V.A.; Nguyen, D.Q.; Nguyen, V.; Le, T.; Tran, Q.H.; Phung, D. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Pittsburgh, PA, USA, 22–24 May 2022; pp. 178–182.

44. Yan, H.; Luo, S.; Pan, L.; Zhang, Y. HAN-BSVD: A hierarchical attention network for binary software vulnerability detection. *Comput. Secur.* **2021**, *108*, 102286. [CrossRef]
45. Wang, Y.; Jia, P.; Peng, X.; Huang, C.; Liu, J. BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention. *Comput. Secur.* **2023**, *125*, 103023. [CrossRef]
46. Li, L.; Ding, S.H.; Tian, Y.; Fung, B.C.; Charland, P.; Ou, W.; Song, L.; Chen, C. VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Trans. Priv. Secur.* **2023**, *26*, 1–25. [CrossRef]
47. Tian, J.; Xing, W.; Li, Z. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Inf. Softw. Technol.* **2020**, *123*, 106289. [CrossRef]
48. Sharma, R.; Chen, F.; Fard, F.; Lo, D. An exploratory study on code attention in BERT. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 437–448.
49. Zheng, W.; Gao, J.; Wu, X.; Xun, Y.; Liu, G.; Chen, X. An Empirical Study of High-Impact Factors for Machine Learning-Based Vulnerability Detection. In Proceedings of the 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), London, ON, Canada, 18 February 2020; pp. 26–34.
50. Yuan, X.; Lin, G.; Tai, Y.; Zhang, J. Deep neural embedding for software vulnerability discovery: Comparison and optimization. *Secur. Commun. Netw.* **2022**, *2022*, 1–12. [CrossRef]
51. Alenezi, M.; Zagane, M.; Javed, Y. Efficient deep features learning for vulnerability detection using character n-gram embedding. *Jordanian J. Comput. Inf. Technol. (JJCIT)* **2021**, *7*, 25–38. [CrossRef]
52. Jie, G.; Xiao-Hui, K.; Qiang, L. Survey on software vulnerability analysis method based on machine learning. In Proceedings of the 2016 IEEE first international conference on data science in cyberspace (DSC), Changsha, China, 13–16 June 2016; pp. 642–647.
53. Vashishth, S.; Upadhyay, S.; Tomar, G.S.; Faruqui, M. Attention interpretability across nlp tasks. *arXiv* **2019**, arXiv:1909.11218.
54. Hanif, H.; Maffeis, S. Vulberta: Simplified source code pre-training for vulnerability detection. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022; pp. 1–8.
55. Zhou, Z.; Bo, L.; Wu, X.; Sun, X.; Zhang, T.; Li, B.; Zhang, J.; Cao, S. SPVF: Security property assisted vulnerability fixing via attention-based models. *Empir. Softw. Eng.* **2022**, *27*, 171. [CrossRef]
56. Kim, J.; Hubczenko, D.; Montague, P. Towards attention based vulnerability discovery using source code representation. In Proceedings of the Artificial Neural Networks and Machine Learning–ICANN 2019: Text and Time Series: 28th International Conference on Artificial Neural Networks, Munich, Germany, 17–19 September 2019; Proceedings, Part IV 28; Springer: Berlin/Heidelberg, Germany, 2019; pp. 731–746.
57. Mao, Y.; Li, Y.; Sun, J.; Chen, Y. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In Proceedings of the 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 10–13 December 2020; pp. 4651–4656.
58. Duan, X.; Wu, J.; Ji, S.; Rui, Z.; Luo, T.; Yang, M.; Wu, Y. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In Proceedings of the IJCAI, Macao, China, 10–16 August 2019; pp. 4665–4671.
59. Mani, S.; Sankaran, A.; Aralikatte, R. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, Kolkata, India, 3–5 January 2019; pp. 171–179.
60. Corporation, M. *CWE-1000: Research Concepts*; Technical report; MITRE: McLean, VA, USA, 2022. Available online: https://cwe.mitre.org/data/definitions/1000.html (accessed on 1 May 2024).
61. Huang, J.; Wang, Z.; Li, D.; Liu, Y. The Analysis and Development of an XAI Process on Feature Contribution Explanation. In Proceedings of the 2022 IEEE International Conference on Big Data (Big Data), Osaka, Japan, 17–20 December 2022; pp. 5039–5048.
62. *Juliet Test Suite for C/C++ and Java*; Technical report; National Institute of Standards and Technology (NIST): Gaithersburg, MA, USA, 2019.
63. Tamilselvam, K. Preddiff: A novel feature importance measure for machine learning models. In Proceedings of the 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA), Boca Raton, FL, USA, 16–19 December 2019; pp. 1459–1463.
64. Zintgraf, L.M.; Cohen, T.S.; Adel, T.; Welling, M. Visualizing deep neural network decisions: Prediction difference analysis. In Proceedings of the International Conference on Learning Representations (ICLR), Toulon, France, 24–26 April 2017.
65. Covert, I.C.; Lundberg, S.; Lee, S.I. Explaining by removing: A unified framework for model explanation. *J. Mach. Learn. Res.* **2021**, *22*, 9477–9566.
66. Blücher, S.; Vielhaben, J.; Strodthoff, N. PredDiff: Explanations and interactions from conditional expectations. *Artif. Intell.* **2022**, *312*, 103774. [CrossRef]
67. Reynolds, D.A. Gaussian mixture models. *Encycl. Biom.* **2009**, *741*, 659–663.
68. Boudjema, E.H.; Verlan, S.; Mokdad, L.; Faure, C. VYPER: Vulnerability detection in binary code. *Secur. Priv.* **2020**, *3*, e100. [CrossRef]
69. Heelan, S.; Gianni, A. Augmenting vulnerability analysis of binary code. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 199–208.
70. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [CrossRef]

71. Svyatkovskiy, A.; Zaytsev, V.; Sundaresan, N. Semantic Source Code Models using Identifier Embeddings. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 554–565.
72. Loyola, P.; Matzger, B.; Schiele, G. Import2vec learning embeddings for software libraries. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1106–1108.
73. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 3111–3119.
74. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 16–20 November 2020; pp. 1536–1547.
75. Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J.; Salakhutdinov, R.R.; Le, Q.V. Xlnet: Generalized autoregressive pretraining for language understanding. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 5753–5763.
76. Beltagy, I.; Peters, M.E.; Cohan, A. Longformer: The Long-Document Transformer. *arXiv* **2020**, arXiv:2004.05150.
77. Zaheer, M.; Guruganesh, G.; Dubey, K.A.; Ainslie, J.; Alberti, C.; Ontanon, S.; Pham, P.; Ravula, A.; Wang, Q.; Yang, L.; et al. Big bird: Transformers for longer sequences. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 17283–17297.
78. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
79. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.
80. Jain, S.; Wallace, B.C. Attention is not Explanation. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, MN, USA, 2–7 June 2019; pp. 3543–3556.
81. Parr, T. *The Definitive ANTLR 4 Reference*; Pragmatic Bookshelf: Raleigh, NC, USA, 2013; p. 22.
82. Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; Jin, H. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2821–2837. [CrossRef]
83. Collard, M.L.; Decker, M.J.; Maletic, J.I. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 516–519.
84. Vashishth, S.; Bhandari, M.; Yadav, P.; Rai, P.; Bhattacharyya, C.; Talukdar, P. Incorporating Syntactic and Semantic Information in Word Embeddings using Graph Convolutional Networks. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 3308–3318.
85. Chen, Y. Convolutional Neural Network for Sentence Classification. Master's Thesis, University of Waterloo, Waterloo, ON, Canada, 2015.
86. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
87. Kendall, M.G. A new measure of rank correlation. *Biometrika* **1938**, *30*, 81–93. [CrossRef]
88. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*; Pearson Education: London, UK 2006.
89. Williams, J.; Wichers, D. The OWASP Benchmark Project. In Proceedings of the Open Web Application Security Project (OWASP) Conference, Washington, DC, USA, 23 October 2019.
90. Corporation, M. CWE Top 25 List 2023. Available online: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html (accessed on 1 May 2024).