# An XAI-based Framework for Software Vulnerability Contributing Factors Assessment

**Ding Li**

**A Thesis**

**in**

**The Department**

**of**

**Electrical and Computer Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Master of Applied Science (Electrical and Computer Engineering) at**

**Concordia University**

**Montréal, Québec, Canada**

**October 2023**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ding Li**

Entitled: **An XAI-based Framework for Software Vulnerability Contributing Factors Assessment**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science (Electrical and Computer Engineering)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
*Dr. Abdelwahab Hamou-Lhadj*

_____ External Examiner
*Dr. Peter Rigby (CSE)*

_____ Examiner
*Dr. Abdelwahab Hamou-Lhadj*

_____ Supervisor
*Dr. Yan Liu*

Approved by _____
, Chair
Department of Electrical and Computer Engineering

_____ 2023

_____
, Dean
Faculty of Engineering and Computer Science

# Abstract

An XAI-based Framework for Software Vulnerability Contributing Factors Assessment

Ding Li

Software vulnerability detection plays a proactive role in reducing risks to software security and reliability. Despite advancements in deep learning-based detection, a semantic gap persists between model-learned features and human-interpretable vulnerability semantics. The challenge lies in the absence of a systematic approach to assess feature importance, capable of explaining the relationship between these two elements. Explainable Artificial Intelligence (XAI) techniques become indispensable in offering comprehensive explanations of features learned by AI models, emphasizing their applicability in software vulnerability detection.

This research introduces an XAI-based framework to systematically evaluate XAI techniques and apply them for assessing the contributing factors of feature representations in classifying software code into Common Weakness Enumeration (CWE) types. The focus is on applying XAI methods to examine the importance of features underlying vulnerability detection. An additional challenge arises from the lack of a systematic evaluation to ensure consistent explanation results during the selection of state-of-the-art XAI methods.

To address this, this thesis defines three evaluation metrics for XAI: consistency, stability, and efficiency. A novel XAI method, named Mean-Centroid PredDiff, is introduced to strike a balance among these three metrics, significantly enhancing the framework's efficacy. This method, along with SHAP, are integrated into the framework based on their well-performance across the evaluation in three domain case studies.

Findings from this work reveal that the proposed framework enables the summarization of the importance of 40 syntactic constructs and the similarities among 20 CWEs based on graph-embedded semantic features. The study results align closely with expert knowledge from the CWE

community, achieving approximately 77.8% Top1, 89% Top5 similarity hit rates and mean average precision of 0.70 in CWE classification. The study validates the significance of attention values of transformer-based models in representing the importance of code tokens.

Overall, this thesis contributes a new XAI method to the open-source community, achieving a trade-off of efficiency with consistency and stability. In addition, the XAI-based framework successfully assesses the nine meta syntactic constructs importance across 20 CWE types and evaluate their similarity. The dataset and the code of framework have been made publicly available on GitHub[1].

[1] https://github.com/DataCentricClassificationofSmartCity/XAI-based-Software-Vulnerability-Detection.git

# Acknowledgments

I'm profoundly thankful to the many people who have played a part in the work encapsulated in this thesis. My deepest gratitude goes to my supervisor, Prof. Yan Liu, whose consistent support, valuable guidance, and extensive knowledge have been crucial throughout my Master's journey. I'm also thankful to Prof. Liu for her financial support, which has greatly enabled my research.

I want to extend my thanks to all the co-authors of my papers, as their joint efforts have significantly enhanced this thesis. On a personal level, I owe much to my family, particularly my mother, and my friends. Their love, support, and encouragement have been my pillar of strength. To all these individuals, I extend my heartfelt appreciation.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software vulnerability signifies the existence of flaws, weaknesses or faults in a software system. These could be within the system's internal controls, security procedures, or even the implementation that may potentially be manipulated by threat sources [1]. Such vulnerabilities often emerge from design errors, poor coding practices, or insufficient security testing. Detecting such vulnerabilities in large-scale software systems poses significant challenges regarding accuracy and transparency, in both academic and industrial settings [2, 3, 4, 5, 6, 7]. Implementing vulnerability analysis and detection during the early stages of software development can provide a proactive means of mitigating potential threats [8].

The methods for software vulnerability detection have evolved over time, transitioning from static code analysis techniques to machine learning approaches. Static code analysis tools, which rely on pattern matching techniques [9, 10] using predefined rules to identify bugs, are often plagued by high false-positive rates [11]. The emergence of machine learning-based methods, utilizing source code, software complexity metrics, and version control system data to predict vulnerabilities, have gained significant attention [12, 13, 14]. These methods automate the process of feature extraction and enable the learning of complex patterns, which improves the need for extensive expert-driven feature engineering [3, 15, 16, 17]. The application of these techniques has led to enhancements in detection accuracy [18, 19].

## 1.1 Problem Statement

Despite significant advancements, machine learning based software vulnerability detection approaches possess discernible limitations [20]. These include underperformance in real-world applications, the learning of irrelevant features, and issues related to data duplication and imbalance [20]. Such challenges raise questions about the models' effective and reliable transferability to different datasets, as well as their transparency in operation [21, 22, 8].

The opacity of these models prompts inquiries such as: (1) To what extent can the signatures of vulnerable artifacts, learned from one set of software projects, be transferred to others [21]? (2) What factors most significantly influence representation learning? (3) How do these factors cause variance in detection results across different learning methods [22]? Addressing these questions requires an assessment of the importance of code features to the semantics of vulnerability detection. Such an assessment should be model agnostic, focusing solely on the inputs (code features) and the outputs (vulnerability classification) of the model.

The domain of software vulnerability detection has seen a development in the application of eXplainable Artificial Intelligence (XAI) methodologies. XAI serves as an essential component of responsible AI, fostering transparency by unraveling the intricate decision-making processes within complex AI models. Offering insights into the association between specific inputs and outputs, XAI enables a deeper understanding of AI systems, augmenting accountability and understandability [23]. XAI utilizes various methods including SHAP (SHapley Additive exPlanations) [24], LIME [25], and Lemna [26], to detect the relevance of features within program code representation.

One of the key concerns when implementing XAI in a specific domain, such as software vulnerability detection, lies in ensuring that the methods deliver consistent and stable explanations, thus promoting trustworthiness [27]. It is imperative to maintain the robustness of XAI techniques, which hinges on their ability to provide coherent explanations for similar inputs. Inconsistencies in the results can instigate ambiguity, potentially undermining the reliability of AI predictions. This poses the first core problem : **Are the explanations derived from varying XAI methods trustworthy?**

Addressing this issue involves two potential solutions. The first involves evaluating existing

XAI methods via defined metrics, which aspire to capture facets of feature contributions overlooked by current methods. The second involves the development of a novel XAI algorithm, with the aim of striking a balance between computational efficiency and the delivery of high-quality, consistent, and stable explanations.

The goal of implementing XAI in software vulnerability analysis revolves around evaluating the contribution of features under various program artifacts, encompassing code textual tokens, abstract syntax trees, and other graph-based code representations. This applies to diverse machine learning models, including natural language processing and graph-based models. By permuting the feature input and summarizing from the outputs, model agnostic XAI methods can potentially achieve this explanation goal.

However, existing works which leveraging XAI for software vulnerability analysis face set of challenges. Some studies employ attention values from transformer-based deep learning models to signify the relevance of code semantics [28, 29]. However, the correlation between high attention values and code feature importance remains a contentious issue [8, 30, 31]. Cross-validation with feature value interpretation tools such as SHAP [24], coupled with a manual examination of individual code blocks, is imperative.

Moreover, syntactic constructs such as operators, operands, and control flows are crucial elements of code semantics. Nevertheless, studies investigating the correlation between these constructs and the significance of code semantics in software vulnerability detection are sparse [29]. Moreover, there is a notable absence of systematic methodologies to relate these constructs with common attributes across various vulnerability types as defined by the Common Weakness Enumeration (CWE) [32]. These list another core problem: **How to develop an XAI-based framework to reliably measure the various types of feature's contribution and correlate them with common attributes across various software vulnerability types?**

## 1.2 Objective

This study aims to address the existing limitations within software vulnerability detection by developing an XAI-based framework. The primary objective involves a comprehensive examination of existing XAI methods that explain on feature importance. Additionally, this thesis introduce a novel approach, named Mean-Centroid PredDiff, designed to enrich the feature explanation taxonomy within XAI. This endeavor focuses on improving the reliability and trustworthiness of XAI methods, thereby ensuring the quality of explanations they provide in terms of consistency, stability, and efficiency.

By leveraging XAI methods to evaluate contributing factors in software vulnerability detection analysis, the subsequent goal is to enhance the accuracy and transparency of machine learning-based approaches. This objective is achieved by rendering the underlying learning models more interpretable and trustworthy. Committed to identifying the contributing factors from both textual-based and graph-based code feature types in software vulnerability, this work provides a systematic explanation of how these elements shape the outcomes of software vulnerability detection.

## 1.3 Contribution

The **contributions** of this thesis can be summarized into four key points:

(1) This study proposes three evaluation metrics addressing the trustworthiness of XAI methods concerning consistency, stability, and efficiency. Furthermore, it enriches the feature explanation XAI taxonomy by introducing a novel XAI method, Mean-Centroid PredDiff, which achieves a balance between consistency, stability, and efficiency, compared to the state-of-the-art feature-based XAI methods.

(2) This study presents a XAI-based framework to assess contributing factors in software vulnerability analysis. To identify these factors, this thesis builds a taxonomy of code representation techniques and extend it to the feature factor level.

(3) This study provides a comprehensive summary of feature importance explanations for syntactic constructs in Abstract Syntax Trees (AST) at the vulnerability type level. This thesis

assesses nine meta-data syntactic constructs (with forty-three detailed constructs) for their contributions across twenty CWE types. Additionally, this thesis analyzes CWE similarity using ranking distance and validate our explanation results against an expert-defined baseline, thereby demonstrating the effectiveness of the proposed approach. This knowledge can potentially be used for IDE programming prompts, allowing the IDE to leverage the syntactic construct priority to provide warnings for potentially vulnerable code.

(4) This study examines into the influence of code token length, code type, and attention values on the model, particularly focusing on the relationship between code tokens' attention values and their significance in a model's decision-making process. To achieve this, it utilizes attention-based models and XAI methods.

In summary, this study begins by evaluating XAI methods to address trustworthiness, laying the foundation for subsequent application in software vulnerability analysis. The taxonomy presented in relation to code feature representation offers a comprehensive perspective on code feature types, aiding practitioners in understanding the contributing factors in software developments. By applying XAI techniques, evidence is presented that attention values from transformer-based models can indicate token importance. However, it's noted that these models also assign weight to separators like commas, which might not hold semantic relevance in code. Furthermore, the XAI-based findings on the importance of syntactic constructs reveal crucial insights into inner syntactic structures. Such insights can guide practitioners in detecting similar CWE vulnerabilities during software development. This research underscores the utility of XAI techniques in software vulnerability detection, addressing concerns of weak transferability in real-world scenarios. The code and dataset for this study are available as open-source resources[1].

## 1.4 Outline

The organization of this thesis is as follows:

- **Chapter 2** reviews the background of XAI, concepts of software vulnerabilities, and the CWE (Common Weakness Enumeration) type.

[1] https://github.com/DataCentricClassificationofSmartCity/XAI-based-Software-Vulnerability-Detection.git

- **Chapter 3** discusses related work in XAI techniques and their applications in software vulnerability domains.
- **Chapter 4** presents evaluation metrics for XAI methods and introduces a novel approach called Mean-Centroid PredDiff.
- **Chapter 5** introduces our XAI-based framework for assessing software vulnerability contribution factors and provides an in-depth analysis of the experimental results related to this assessment.
- **Chapter 6** addresses the limitations of our approach and discusses potential threats to validity.
- **Chapter 7** offers a conclusion and summarizes the thesis.

## 1.5 Publications

The research presented in this thesis has been accepted for publication in the following:

(1) D. Li, Y. Liu, J. Huang, and Z. Wang, "A Trustworthy View on Explainable Artificial Intelligence Method Evaluation," *Computer*, vol. 56, no. 4, pp. 50-60, 2023. This publication proposed the XAI evaluation metrics and the newly Mean-Centroid PredDiff XAI method in Section 4.2 and 4.3 in Chapter 4.

(2) J. Huang, Z. Wang, D. Li, and Y. Liu, "The Analysis and Development of an XAI Process on Feature Contribution Explanation," in *2022 IEEE International Conference on Big Data (Big Data)*, pp. 5039-5048, December 2022. IEEE. This publication developed a general XAI process including XAI taxonomy (Section 3.1), the XAI goal, and XAI criterion to select different XAI methods (Section A.1).

A journal paper is currently under review, focusing on applications assessing contribution factors of software vulnerability detection, as discussed in Chapter 5.

(1) D. Li, Y. Liu, J. Huang, "Assessment of Software Vulnerability Contributing Factors using Model-Agnostic eXplainable AI Techniques" *IEEE Transactions on Software Engineering*, submitted September 2023.

# Chapter 2

# Background

This chapter introduces key terms of the research, including the concept of eXplainable Artificial Intelligence (XAI), software vulnerabilities, and the Common Weakness Enumeration (CWE) type. It also traces the evolution of software vulnerability detection practices, from rule-based tools to deep learning-based methodologies. The significance of the CWE type in classifying software vulnerabilities is also covered.

## 2.1 Explainable AI (XAI) Techniques

The growing importance of black-box AI models in generating critical context outgrowths an increasing demand for transparency from various AI stakeholders [33, 23]. The threat lies in the creation and usage of decision-making processes that are unexplainable, unjustifiable, or illegitimate [34]. In this light, explanations supporting the model's outcoming become crucial. EXplainable AI (XAI) techniques suggests developing a suite of machine learning techniques that 1) create more explainable models without compromising learning performance (such as prediction accuracy), and 2) enable humans to understand, trust appropriately, and effectively manage the rising generation of AI partners [23]. XAI also considers the psychology of explanation, drawing insights from Social Sciences [35].

Generally, XAI techniques can be broadly classified into two main categories: methods for

building inherently interpretable AI models and post-hoc methods for explaining existing models [36]. Inherently interpretable AI models often utilize algorithms such as linear regression, logistic regression, and decision tree models, which offer built-in interpretability for specific domains [37]. However, as deep learning models become increasingly prevalent across various domains, their inherent lack of explainability has prompted the development of post-hoc methods. Post-hoc methods are further divided into two subcategories: model-specific and model-agnostic methods. Model-specific methods, as the name suggests, are tailored to explain specific types of models, while model-agnostic methods treat the model as a black-box.

## 2.2 Software Vulnerabilities Detection

Software vulnerability detection is the practice of identifying and characterizing weaknesses, flaws, or vulnerabilities in a software system that could potentially be exploited by malicious entities [1]. This practice has seen a significant evolution over time, transitioning from traditional rule-based tools to more advanced machine learning-based methods, especially those that incorporate deep learning models.

**Rule-based Vulnerability Detection Tools** Rule-based tools for vulnerability detection typically involve static code analysis [38] or dynamic code analysis [39]. Static code analysis examines the source code without executing the program. Each type of vulnerability is associated with a predefined rule, and rule violations signal potential vulnerabilities. Tools like Flawfinder [40], RATS [41] and ITS4 [42] exemplify rule-based static analyzers. Although effective in identifying known vulnerabilities, these tools often struggle with detecting novel or complex vulnerabilities. Dynamic analyzers evaluate potential vulnerabilities in relation to the program's runtime behavior [39]. They detect vulnerabilities by interpreting user inputs or by generating meaningful program inputs that could cause the program to crash. While both static and dynamic analyzers offer valuable insights, they also present limitations. These tools often struggle with high false positive and false negative rates. Furthermore, their efficacy is confined to the scope of their existing rules, making them less capable of identifying novel or unique vulnerabilities [43].

**Deep Learning-based Vulnerability Detection Approaches** Deep learning-based vulnerability detection demonstrates promising results when compared with rule-based tools in recent years. In this approach, the code is initially embedded as a feature representation. This includes text token-based, graph-based, or binary-based representations. Following this, the model classifies the code into either a binary label, indicating whether the code is vulnerable or not, or into multi-classification labels, highlighting specific vulnerability types.

Several models have been designed to learn these code representations and conduct downstream tasks like vulnerability classification. Early models, such as Code2Vec [44], leverage an attention mechanism to learn distributed representations of code partitions. Following the success of transformer models, CodeBERT [45] introduces pre-trained models based on BERT [46], which are specifically tailored for programming languages. Other transformer-based models, such as XL-Net [47], Longformer [48], and Bigbird [49], incorporate more complex architectural designs to address issues related to the independence assumption of masked tokens and limitations in handling extended code sequences.

Furthermore, Graph Neural Networks (GNNs) have also shown their effectiveness in code vulnerability detection tasks. Models such as VulBERTa [50], Devign [18], GraphCodeBERT [51], and GraphCodeVec [52], as well as VulDeeLocator [53] and REVEAL [20], utilize the ability of GNNs to capture intricate relationships between code entities. This is achieved by representing program structures as graphs and propagating information through graph nodes. The further taxonomy introduction of code feature code representations is presented in Section 5.1.

## 2.3 Common Weakness Enumeration (CWE) Type

The Common Weakness Enumeration (CWE) [54] is a community-developed list of common software security weaknesses. It serves as a common language for describing these vulnerabilities, a standard for measuring software security tools, and as a baseline for identifying, mitigating, and preventing issues. It also commonly used as labels for supervised learning in vulnerability detection.

Different CWE type often share similarities. These commonalities may be derived from their impact on system functionality, the system components they affect, or the degree of access they

grant an attacker. The following presents an example of a vulnerability category along with similar CWE types under that category.

Improper Input Handling: This category captures vulnerabilities that arise when a software system fails to correctly validate input or mistakenly validates improper input. This could potentially manipulate the control or data flow of the program. The related CWE types under this category are CWE-78, 79, 89, and 90. CWE-78 (OS Command Injection): This weakness emerges when software does not appropriately neutralize special elements that could alter an intended OS command when sent to a downstream component. CWE-79 (Cross-site Scripting): Here, the software does not adequately neutralize, or incorrectly neutralizes, user-controllable input before it is used in a web page, which may lead to cross-site scripting (XSS) attacks. CWE-89 (SQL Injection): This vulnerability occurs when the software forms an SQL command using externally influenced input from an upstream component. If it does not correctly neutralize special elements, the intended SQL command may be modified. CWE-90 (LDAP Injection): This flaw arises when the software constructs an LDAP (Lightweight Directory Access Protocol) query using externally influenced input from an upstream component. If it fails to neutralize special elements properly, the intended LDAP query could be modified.

# Chapter 3

# Related Work

This chapter provides a comprehensive review of XAI taxonomy and the existing XAI techniques, discussing their strengths and limitations. It also introduces critical evaluation metrics that measure the effectiveness and reliability of these XAI methods as part of the framework. Finally, the chapter delves into the application of XAI methods in the software vulnerability domain.

## 3.1 Model-Agnostic XAI Taxonomy

Model-agnostic Explainable AI (XAI) methods investigate the connections between features and prediction outcomes. Their aim is to provide understandable insights into model decision-making processes. These methods can be primarily classified into three categories [55] based on how they present their explanations as Figure 3.1.

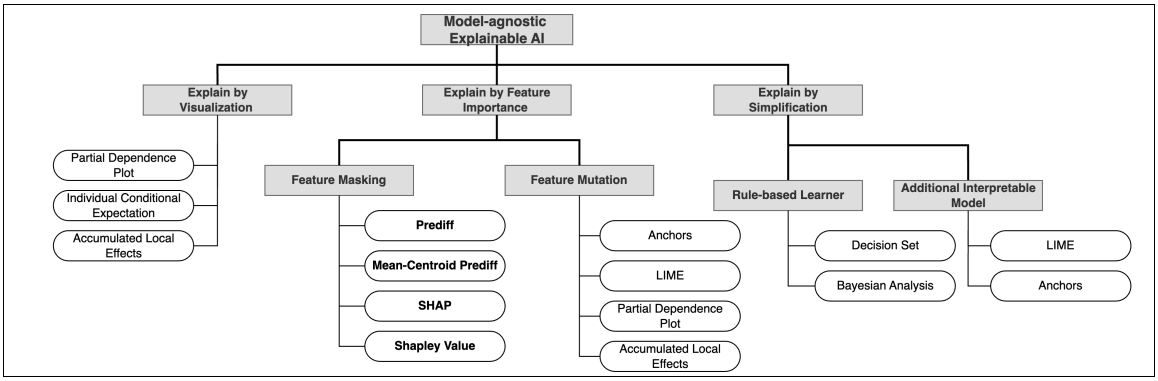



Figure 3.1: Taxonomy of model-agnostic explainable AI methods.

**Explain by Visualization [56]**: These methods use visual tools to help illustrate model behaviors, often by highlighting key features in the data.

**Explain by Feature Importance [57]**: This group of methods determines the importance of features on predictions, with a specific focus on feature masking and feature mutation. **Feature Masking**: These techniques operate by removing or replacing certain input features to examine how the model's predictions change. This allows for an understanding of how individual features contribute to the prediction. **Feature Mutation**: These techniques change one or a few feature values to see how the individual prediction shifts, providing an understanding of the feature's impact on model predictions.

**Explain by Simplification [25]**: These methods attempt to create a simpler, more interpretable model to emulate the complex black-box model. They include Rule-based Learners, which provide decision rules to clarify prediction paths, and Additional Interpretable Models, which train another interpretable model to explain the original one.

## 3.2 XAI Feature Importance Explanation Methods

Explainable AI (XAI) is an emerging research topic in recent years, aiming to explain AI models' logic and decision-making processes for users in the goodness of safety and fairness. Conventionally, post-hoc XAI methods are categorized as model-agnostic and model-specific [23]. Model-specific methods probe and extract the model gradients or neuron activation states from the neural network models. Examples are the family of XAI methods based on Class Activation Mapping (CAM)[58] including EigenCAM[59], GradCAMElementWise [60], Grad-CAM++[61], XGrad-CAM[62], and HiResCAM [63]. They have been applied to explain feature contributions to image classification algorithms and tasks.

Model-agnostic methods are black-box based and non-intrusive to specific machine learning algorithms [64]. PredDiff [65] quantifies the impact of each feature on a model's prediction by evaluating the changes in prediction scores after the perturbation of a particular feature. Formally,

the importance of feature $j$ is given as follows:

$$\phi_j^{\text{PredDiff}} = |\hat{f}(x) - \hat{f}(x_{-j})| \tag{1}$$

Here, $\hat{f}(x)$ denotes the model's prediction with all features, while $\hat{f}(x_{-j})$ represents the model's prediction after the perturbation or removal of feature $j$. The PredDiff method assumes that features are independent, which might not hold true for many real-world applications [66]. LIME (Local Interpretable Model-agnostic Explanations) [25] is an XAI method that explains the predictions of any classifier or regression model by approximating it locally with an interpretable model. The approximation is driven by the assumption that every complex model is linear at a local scale. The explanation provided by LIME for instance $x$ is formally given by:

$$\xi(x) = argmin_{g \in G}[L(\hat{f}, g, \pi_x) + \Omega(g)] \tag{2}$$

Here, $L$ is a loss function that measures how well the explanation $g$ approximates the prediction function $\hat{f}$ in the locality defined by $\pi_x$, and $\Omega(g)$ is a measure of the complexity of the explanation. The local linear approximations of LIME may be inaccurate for non-linear models, and the explanations depend heavily on the quality of perturbations. LIME can also be computationally expensive. SHAP (SHapley Additive exPlanations) [24] assigns each feature an importance value for a particular prediction based on the concept of Shapley values [67] from cooperative game theory. It quantifies the contribution of each feature to the prediction for a specific instance. The SHAP value, $\phi$, is defined as:

$$\phi_j^{\text{SHAP}} = \sum_{S \subseteq P \setminus j} \frac{w(S)}{|S|!(|P| - |S| - 1)!}(\hat{f}_{S \cup j}(x) - \hat{f}_S(x)) \tag{3}$$

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 $\hat{f}_S(x)$ 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. SHAP can be computationally taxing for models with large scale features. CXPlain [68] utilizes causal inference

to provide feature importance explanations, CXPlain can be computationally burdensome, especially for tasks with numerous features or complex causal structures. ALE (Feature Importance by Accumulated Local Effects) [69] calculates the cumulative effect of varying feature values on model predictions, its assumption of feature independence may hinder its effectiveness with highly correlated features. PDP (Partial Dependence Plot) [70] provides a global view of feature importance, PDP visualizes the marginal effect of a feature on predicted outcomes. However, its assumption of independent features may not hold true in many practical applications. ICE (Individual Conditional Expectation) [71] is an extension of PDP providing local explanations of feature importance, ICE may produce complex and dense plots that can be difficult to interpret when dealing with high-dimensional data.

In summary, while these methods offer different perspectives for interpreting AI models, they also encounter challenges concerning computational efficiency, feature correlation, and interpretability, underscoring the need for a balanced XAI method.

## 3.3 Evaluation of XAI Methods

The evaluation of XAI methods is crucial for establishing their trustworthiness, particularly because XAI methods highlight the transparency and understanding of AI models, as a part of building responsible AI [72]. This trust is often evaluated at the level of individual predictions by users. Correspondingly, the robustness of the explanation for each data sample prediction becomes essential in shaping this trust. Subsequently, users may elevate their trust to the model level. In this context, the consistency across multiple XAI methods applied to the same model becomes critical to the AI model's accountability. As XAI methods bloom, it becomes increasingly challenging to identify the most reliable ones, particularly given the variability in the explanations generated by different methods. Therefore, as emphasized in previous studies [73], well-defined quantifiable metrics are important in measuring the results of XAI explanations, helping build trustworthy AI systems.

Having robust evaluation metrics is thus indispensable for assessing the quality of XAI methods. These metrics evaluate factors like soundness, completeness, and trustworthiness of the explanations generated by these methods. In turn, this facilitates the selection of suitable XAI methods for specific applications and enhances the overall trust in deployed AI systems. For instance, a study [74] introduced soundness and completeness as XAI evaluation metrics. Soundness measures the correctness of the explanation but often requires access to a model's ground truth, which might not always be available. Completeness, on the other hand, estimates how thoroughly an explanation covers the entire task model, being more relevant for global than for local explanations.

Another aspect of evaluating XAI methods is trustworthiness, as highlighted in a study on an explainable book search system [75]. Here, trustworthiness evaluation was tied to retrieval system performance, using ranking by user clicks, user responses to questionnaires, and user eye-tracking data. The comparison of these results helped evaluate the trustworthiness of the system's explanations. In summary, the evaluation of XAI methods not only determines their performance but also bolsters the trust in AI systems by ensuring the reliability of the explanations they provide.

## 3.4 XAI-based Vulnerability Analysis

Tanwar et al.[28] provide an interpretable instance for the Juliet test suite, where they visualize attention values for each code line. VulANalyzeR[29] utilizes attention mechanisms to pinpoint crucial instructions and basic blocks that lead to vulnerabilities. However, most studies concentrate on visualizing attention values for individual code snippets and lack a method for further probing the status of AST syntactic constructs and their linked code tokens. Code2Vec [44] and Multiple Instance Learning (MIL) techniques [76] provide explainability at the individual AST path level. Furthermore, several studies [77, 78, 79, 80] underscore the importance of syntactic identifiers like name, literal, type, and parameter for vulnerability classification. Notably, a study by Sotgiu et al.[81] uses SHAP to analyze the importance of code tokens. Duan et al.[82] investigate buffer error vulnerability cases and find that names, conditions, and parameters are key features. Despite these insights, a comprehensive evaluation of the importance of all syntactic constructs across multiple vulnerability types is lacking, which forms the motivation for the investigation in this study.

# Chapter 4

# Feature Importance XAI Explanation

This chapter first defines the objective of feature importance explanation. It then illustrates XAI engineering by evaluating various XAI methods using well-defined metrics, including consistency, stability, and runtime efficiency. It introduces a novel model-agnostic XAI method, termed Mean-Centroid PredDiff, which summarizes the explanation by employing the clustering centroid of the prediction difference. This enriches the field of trustworthy XAI by offering an evaluation of existing methods, coupled with the introduction of a new technique for trustworthy explanation generation.

## 4.1 Explanation on Features

The analysis of feature importance explanation is key to discerning how individual features influence a model's predictions. This section initially introduces three fundamental terms - *feature*, *feature contribution value*, and *feature importance order*.

At its core, a ***feature*** $j$ is an individual measurable property or characteristic of a phenomenon being observed. In the context of machine learning, features are used to represent the patterns that the algorithm learns from and, thus, significantly contribute to the model's predictive or classification power. Features can span a wide range, for text tokens, token type [80], token frequency [83], token attention value [84] in natural language processing tasks, pixel intensities in image recognition tasks [85], or even more complex constructs depending on the domain and the specific task.

The whole feature space is donated as $P$.

***Feature contribution value***, denoted as $\phi_j$, represents the quantitative impact of a specific feature $j$ in driving the model's predictions [25]. Higher feature contribution values indicate more substantial influences, while lower values suggest less impact.

***Feature importance order***, denoted as a sort vector $sort(v)$ that $v \leftarrow \{\forall j \in P, \phi_j\}$, refers to the ranking of features based on their feature contribution values. This ranking enables an intuitive understanding of the relative importance of each feature in the model's decision-making process. By examining the ranking distances between different entities' feature importance orders, it can infers the similarity of these entities based on their feature characteristics.

## 4.2 XAI Evaluation Metrics

This section discuses three measurement metrics regarding the view of trustworthy XAI methods, consistency, stability and time complexity. Consistency assesses the agreement among different XAI methods, ensuring they yield similar explanation results. Stability evaluates the agreement of feature importance explanation results across similar data input in an inner-XAI method view. Time Complexity examines the computational efficiency of XAI methods, providing insight into their practicality for use in large datasets.

The three metrics provide a measure in relation to the XAI's criteria [55]. For instance, a lower value in the consistency metric indicates a higher degree of consistency across different XAI methods. Conversely, a higher value in the consistency metric demonstrates a greater diversity among the different XAI methods in achieving the explanation results.

### 4.2.1 Define Explanation Consistency

The goal of consistency metrics in the context of Explainable AI (XAI) is to quantitatively evaluate the agreement between different XAI methods when explaining the same dataset and model. Consider $\hat{f}(x_i)$ is the model prediction on instance $x_i < x_i^1, x_i^2, ...x_i^p >$, where $p$ is the number of features. Suppose $S$ is the subset of all the features by masking or removing a feature $j$ that is

$S \subseteq \{1, 2, 3, ..., p\} \backslash \{j\}$ and $P$ contains the whole features, $P = S \cup \{j\}$. Under feature masking, the prediction on the masked feature set $S$ and on the whole feature set $P$ for each instance $x$ has the difference as $\delta_j^{x_i} = \hat{f}_S(x_i) - \hat{f}_P(x_i)$. Hence the feature contribution to the payout by masking feature $j$ on the prediction of instance $x_i$ is defined as a function as $\phi_j(\delta_j^{x_i})$. An XAI method develops the aggregation of $\phi_j(\delta_j^X)$ on all the data samples differently. Finally, by masking the features one by one, the feature importance order is derived by ranking the feature contribution values. After the transformation from feature contribution values to the feature importance order by descending the contribution values, Kendall Tau Ranking Distance [86] is applied to measure the distance $d_{ktrd}$ of any two pairs of the XAI method's feature importance order explanation results, $d_{ktrd}(sort(v^{XAI_m}), sort(v^{XAI_M}))$, where $m$ and $M$ are two different XAI methods.

### 4.2.2 Define Explanation Stability

Explanation stability refers to the degree of agreement within the explanations generated by a single XAI method when applied to multiple datasets. In other words, it quantifies the variation in the explanations for different datasets given by the same XAI method. Given multiple datasets with each producing an explanation summary through the same XAI method, it calculates the distance between every pair of explanation summaries. It quantifies the stability of an XAI method by calculating the mean of these distances across all pairs of datasets. This is expressed as $Average(d_{ktrd}(sort(v^{dataset_n}), sort(v^{dataset_N})))$, where $n$ and $N$ are two different datasets.

### 4.2.3 Analyze Time Complexity

Asymptotic analysis for $\phi_j(\delta_j^X)$ depends on the size of data instances number $N$ and the number of features $P$. Shapley value computes the feature value difference under feature masking $\delta_j^X$ for the whole data set for each masked feature. Shapley value considers the permutation when selecting one feature to mask and makes the reverse value of permutation as the weight to sum the feature contribution value$\phi_j$. Overall, this work derives that the Shapley value has the complexity as $\Theta(N \times P \times 2^P)$. KernelSHAP [24] uses the linear LIME explanation model and the classical Shapley value. According to the definition, KernelSHAP depends on the LIME loss function [25], weighting Kernel and the regularization term. Therefore, Kernel SHAP has the complexity of $\Theta(N \times (2^P + P^3))$.

PredDiff removes each feature individually and measures the difference between each instance's prediction and the feature removal prediction. The time complexity of PredDiff is $\Theta(N \times P)$. Section 4.3 presents a newly proposed XAI method with the time complexity of $\Theta(N \times P^2)$.

## 4.3 Developing a New XAI Method - Mean-Centroid PredDiff

The objective is to elucidate the impacts of feature masking by considering the relative difference in the ratio to the prediction made without feature masking. While state-of-the-art methods [65] mainly focus on the absolute prediction difference, the proposed XAI method aims to offer a comparable consistency in explanation summary to these advanced techniques, while simultaneously reducing computational time. Figure 4.1 illustrates the key tasks involved in computing the prediction difference under feature masking and the feature contribution value for each masked feature, divided into three distinct phases.

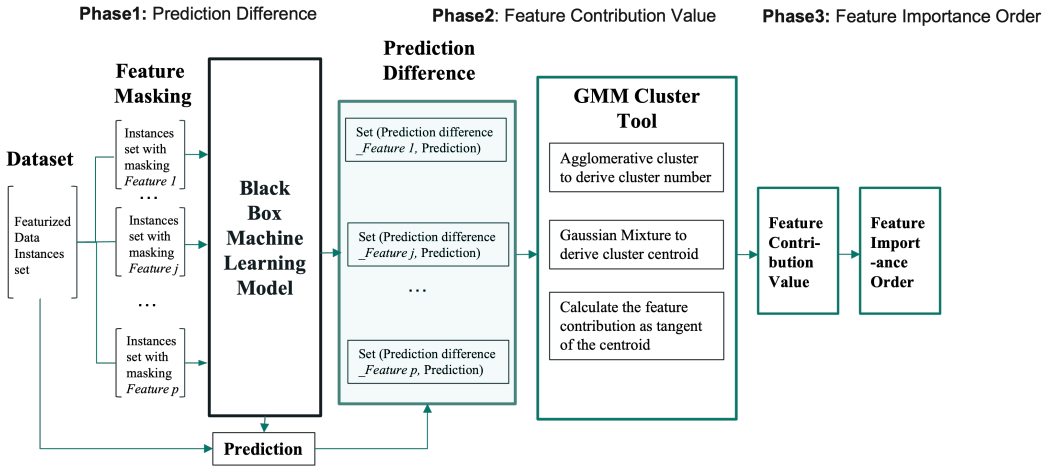



Figure 4.1: The dataflow of Mean-Centroid PredDiff (Prediction Difference) explanation summary.

### 4.3.1 Phase 1: Compute Prediction Difference under Feature Masking

Algorithm 1 presents that the prediction difference $\delta_j^{x_i}$ (as x-coordinate) and its corresponding prediction $\hat{f}_P(x_i)$ (as y-coordinate) form a data point in two dimensional Euclid plane. Hence, $N$ numbers of two-dimensional points are created for each masking feature $j$.

**Algorithm 1** Mean-Centroid Prediction Difference (PredDiff) Explanation

**Require:** Input data set $X$, full feature set $P$, masking feature set $S$, model prediction $\hat{f}(x_i)$

/***Phase 1:** compute the difference of prediction under feature*/
**for all** $j \in P$ **do** ▷ $P$,the whole features, $P = S \cup \{j\}$
  **for all** $x_i \in X$ **do** ▷ $S \subseteq \{1, 2, 3, ..., p\} \backslash \{j\}$, the subset of all the features by absent feature $j$
    $\delta_j^{x_i} \leftarrow |\hat{f}_S(x_i) - \hat{f}_P(x_i)|$
    $\nu_j^i \leftarrow < \delta_j^{x_i}, \hat{f}_P(x_i) >$
  **end for**
**end for**
$V_j \leftarrow \{\nu_j^{[1]}, \nu_j^{[2]}, ..., \nu_j^{[N]}\}$
/* **Phase 2:** compute the feature contribution value
/* group $V_j$ to $k_j$ clusters */ ▷ $k_j$, the number of clusters under feature masking j
$k_j \leftarrow \hat{f}_{agg}(V_j)$ ▷ $\hat{f}_{agg}$, agglomerative clustering algorithm [87]
$centroid_j \leftarrow \hat{f}_{gmm}(k_j, V_j)$ ▷ $\hat{f}_{gmm}$, gaussian mixture model [88]
$\phi_j(\delta_j^X) = \tanh(centroid_j)$ ▷ $Centroid$ as cluster centroid point
/***Phase 3:** convert the contribution values to the feature importance orders */
$order = sort(abs(\phi_j(\delta_j^X)))$

**Ensure:** $\phi_j(\delta_j^X), order$

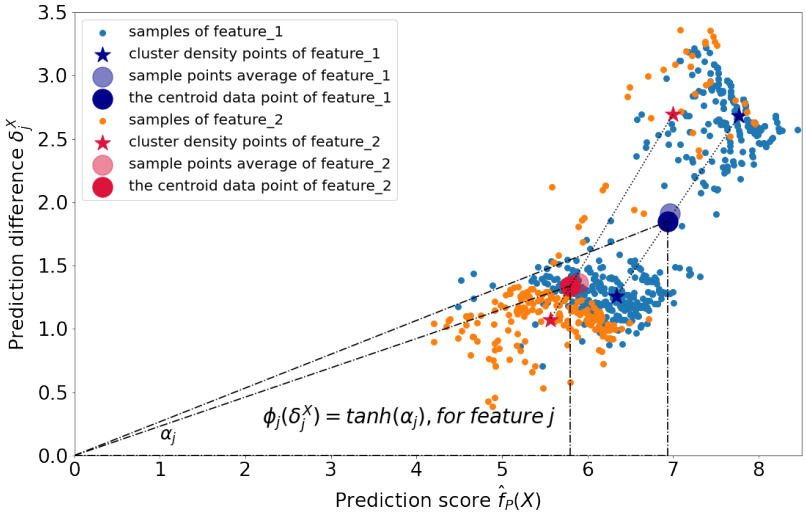



Figure 4.2: An example of deriving two features' contribution values by Gaussian Mixture clusters.

### 4.3.2 Phase 2: Compute Feature Contribution Values

The observation from Phase 1 output is the data points form clusters. It further groups the data points into $k_j$ numbers of clusters by agglomerative clustering algorithm [87]. It then estimates the centroid data point of these clusters using the Gaussian mixture model [88]. For each masked feature $j$, it defines its feature contribution value $\phi_j$ aggregated for all the input data samples as the slope or tangent of the centroid data point to the origin point in a two-dimensional plane.

An example in Figure 4.2 depicts how the Gaussian mixture clusters aggregate the contribution values of two feature markings. Data points are grouped into two clusters for each feature. The centroid data point is derived as the weighted average by the clusters' density points generated by the Gaussian Mixture model. This algorithm has considered the distribution density of the prediction changes of the whole data samples.

### 4.3.3 Phase 3: Convert to Feature Importance Order

The conversion is simply ranking the features in descending order according to their feature contribution value. The consistency of the two explanations is then measured as the distance between two orders.

### 4.3.4 Asymptotic Analysis on Time Complexity

Given the number of features $P$ and the number of instances $N$, computing the prediction difference is of the time complexity $\Theta(N \times P)$ in phase one. In phase two, computing the clusters takes $\Theta(N \times P^2)$. Overall, the time complexity is $\Theta(N \times P^2)$.

# Chapter 5

# The Framework's Application: Assessing Contributing Factors of Code Vulnerability Classification

This chapter applies the XAI based framework into the deep learning-based software vulnerability detection domain. It aims to enhance the interpretability of the model's decision-making process in a model-agnostic perspective and provide valuable insights into the critical factors contributing to software vulnerabilities by presenting an XAI-based framework for software feature contribution assessment.

The vulnerability detection task is considered as a multi-class classification problem where the source code is embedded and then categorized into different vulnerability types. The code program is first presented as graph-based contexts including code tokens and the abstract syntax trees paths with syntactic constructs. This work hence evaluates the importance of forty syntactic constructs importance in terms of influencing the model predictions. These constructs importance ranking summarize are also connected to the Common Weakness Enumerations (CWEs) types and used for identifying CWE similarity.

On the other hand, up-to-date research has modeled software vulnerability detection as a natural language processing (NLP) task by state-of-the-art attention-based models [89, 45] and provides

explanation from attention values. However, studies argue that attention values may not always align with token importance [31]. This motivates this XAI-based framework to be extended to examine whether attention values align the explanation results of the token importance from XAI techniques.

This chapter is structured into seven sections. First the taxonomy of related work introduce the existing research on software representations and vulnerability detection 5.1. The application of XAI-based framework is introduced in Section 5.2, and it is applied to assessing contribution factors 5.2, summarizing CWE similarity 5.3, and extending to textual based features assessment 5.4. The research questions and the following experiments and analysis are in Section 5.5. This chapter also presents a retrospective analysis of a pair of similar CWE siblings code samples in Section 5.6, examining how contributing factors are reflected in a detailed case study using both attention-based and graph-based models. Finally, section 5.7 carries out a comparative review of the findings with existing work, highlighting the contribution.

## 5.1 Taxonomy of Related Work

In exploring how source code is represented and transformed into a format that can be processed by deep-learning models, this study establishes a taxonomy of code representation techniques and their feature types. This taxonomy is structured around four primary representation techniques: text-based, graph-based, code binary, and a mixed feature representation [90, 91, 92] as demonstrated in **Figure 5.1**.

### 5.1.1 Text-based Code Representation and Feature Types

**Text-based Code Representation Techniques.** Text-based code representation approaches treat source code similarly to natural languages, directly embedding existing word embedding techniques into code [93, 94, 95]. Here, the code content is treated as plain text with no consideration given to structural nuances like data flow and function call flow. With the advancement in the field of natural language processing, representation techniques have progressed from static embeddings such as word2vec [96] and fastText [93] to self-attention transfer learning-based models. These

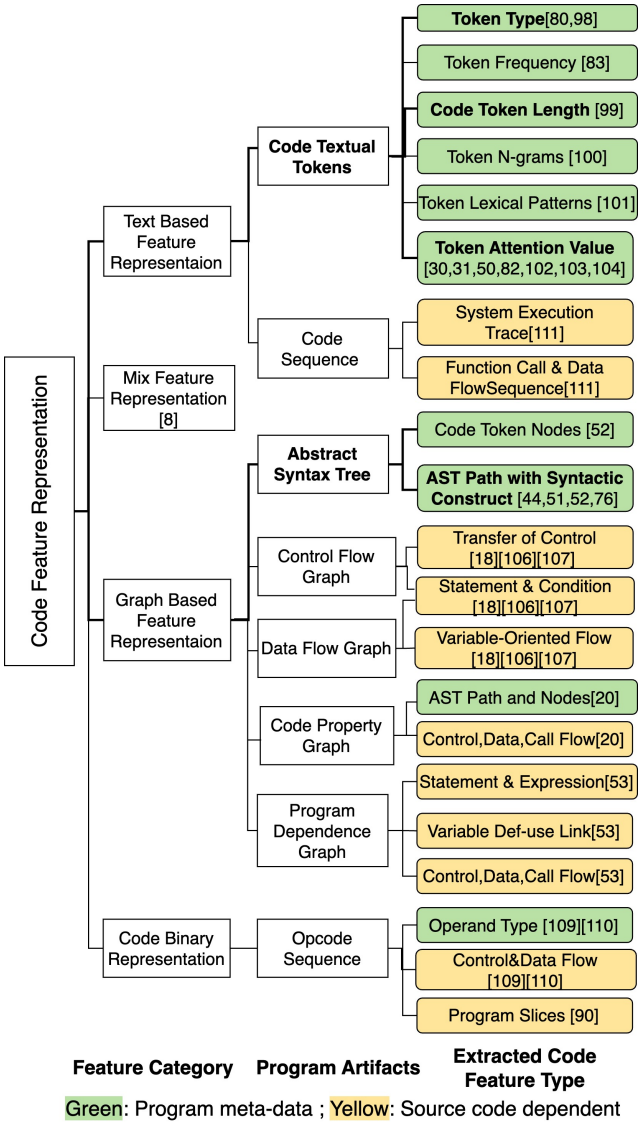



Figure 5.1: Taxonomy of factors under various code feature representation techniques (bold: assessed in this study).

include large corpus embedding models trained on code such as codeBERT [45], XLNet [47], Longformer [48], BigBird [49], and GPT [97]. These models utilize pre-trained contextualized embeddings, exhibiting a higher expressive capacity compared to static embeddings.

The codeBERT model [45] employs embeddings that utilize a dual-transformer architecture, effectively merging the advantages of masked language modeling and code summarization. This model handle with the challenges presented by long code sequences. XLNet embeddings [47] adopt

a permutation-based approach, recognizing inter-token dependencies and accommodating bidirectional context for a comprehensive token representation. The BigBird [49] and Longformer [48] models are specifically tailored for long token sequences, enabling extended input token lengths. Longformer uses a sliding window-based local attention mechanism for proximate tokens and a global attention mechanism for distant tokens. Conversely, BigBird amalgamates dense and sparse attention patterns, adeptly managing lengthy text sequences while maintaining its capacity to model long-range dependencies.

**Text-based Feature Types.** Within the framework of text-based code representation, a variety of feature types have been distinguished. These features can influence the behavior of the model when it processes source code. These features include token type [98, 80], token length [99], token frequency [83], token n-grams [100], token lexical patterns [101], and token attention values.

*Token Type*: Tokens can be classified into comments and code. The previous work [98] has indicated that comment tokens supplied by programmers can enhance the comprehension of code semantics and structure for learning models. Another study [80] reveals the significant role of separator symbols when models make predictions, as demonstrated by an attention-based model assessment. Thus, token types can also be subdivided into textual tokens and symbol tokens.

*Token Length*: Constraining the length of code tokens can lead to information loss and have a negatively influence on the model's performance, as demonstrated by Yuan et al. [99]. Their analysis restrict to a maximum sequence length of 512 tokens.

*Token Frequency*: As a prominent feature type in static text-based representation techniques, token frequency can influence model performance. Zeng et al. [83] concluded that preserving code frequency information results in superior model performance.

*Token n-grams*: These fixed-size, contiguous sequences of tokens capture the local context within a set window [100]. However, their utility may be constrained for longer code sequences and transformer models.

*Token Lexical Patterns*: Lexical patterns, which represent recurring structures in the code [101], can facilitate comprehension of the basic logic and structure of the code. However, they may have limited effectiveness in capturing more complex, high-level semantic information and dependencies across distant tokens.

*Token Attention Values*: Attention values act as a feature type in transformer-based models that can identify key tokens or content contributing to natural language processing tasks [84]. The attention mechanism can adaptively learn the significance of even distant parts of the input code sequence, better understanding the code's contextual information and demonstrating effectiveness in software vulnerability detection tasks [50, 102, 103, 104, 82]. Some researchers have suggested that attention values can serve as a surrogate for token importance [30]. However, it's important to interpret this with caution as high attention values may not always align with high token importance [31].

### 5.1.2 Graph-based Code Representation and Feature Types

Graph-based code representations are widely employed in various studies. Common representations include Abstract Syntax Tree (AST), Program Dependence Graph (PDG), Control Flow Graph (CFG), Data Flow Graph (DFG), and approaches that combine these graphs [20]. As outlined by Zeng et al.[22], among these graph-based methods, AST-based approaches take the majority account of existing works. AST nodes signify the syntactic constructs of code, such as loops, declarations, and expressions, making them intuitive to developers[105].

**Graph-based Code Representation Techniques.** *AST-based methods:* Code2Vec [44] introduces a method to learn vector representations from the AST's path context, thereby emphasizing their importance in predicting code semantic properties. Hariharan M. et al.[76] implement a Multiple Instance Learning (MIL) technique that treats each AST path as a distinct instance for supervised learning. GraphCodeBERT[51] combines the AST's graph structure information with transformer-based techniques to represent code structure. Recently, GraphCodeVec [52] learned generalizable code embeddings from code tokens and AST structures, demonstrating state-of-the-art performance in six code downstream tasks, including vulnerability detection.

*Other graph-based methods:* VulDeeLocator [53] employs PDG and integrates AST information to learn discriminative vulnerable features. Devign [18] assembles a hybrid graph representation incorporating AST, CFG, and data dependence graph to capture complex code structural information more effectively, albeit at a higher computational cost. REVEAL [20] extracts syntax and semantic features from the Code Property Graph (CPG), which includes elements from the

data-flow, control-flow, AST nodes, and program dependency.

**Graph-based Feature Types.** The feature types in graph-based code representations rely on specific graph structures, nodes, edges, and their definitions. In the case of the AST, leaf nodes denote code tokens affiliated with specific syntactic constructs [105], rendering code token nodes as one type of feature. Moreover, path-based representations, which include branch nodes as syntax, can effectively capture code contextual semantics and are extensively used in leading-edge approaches [44, 76, 52]. Features based on CFG and DFG [106, 107, 18], focus primarily on program flows, such as data and control flow through variables, statements, and conditions. Lastly, PDG-based features incorporate both control and data flow dependencies within a program, capturing statements, expressions, variable def-use links, and function call flow [107, 108]. These features represent individual programs more than entire software projects.

### 5.1.3 Other Code Representations and Their Feature Types

A number of studies have investigated *binary code representation* for vulnerability detection. BVDetector [90] combines program slicing with a BGRU network for intricate vulnerability detection. HAN-BSVD [109] adopts a hierarchical attention network for preserving context and emphasizing key regions. BinVulDet [110] harnesses decompiled pseudo-code and BiLSTM-attention for robust vulnerability pattern extraction. VulANalyzeR [29], on the other hand, introduces an explainable approach that employs multi-task learning and attentional graph convolution. Feature types under binary representations typically include operand types, control flows, and program slicing, among others. *Code sequence representation*, under text-based approaches, has been explored in different contexts, such as function call sequences, data flow sequences, and system execution traces. DeepTriage [111] is an example of a method that analyzes system execution traces for software defect prediction.

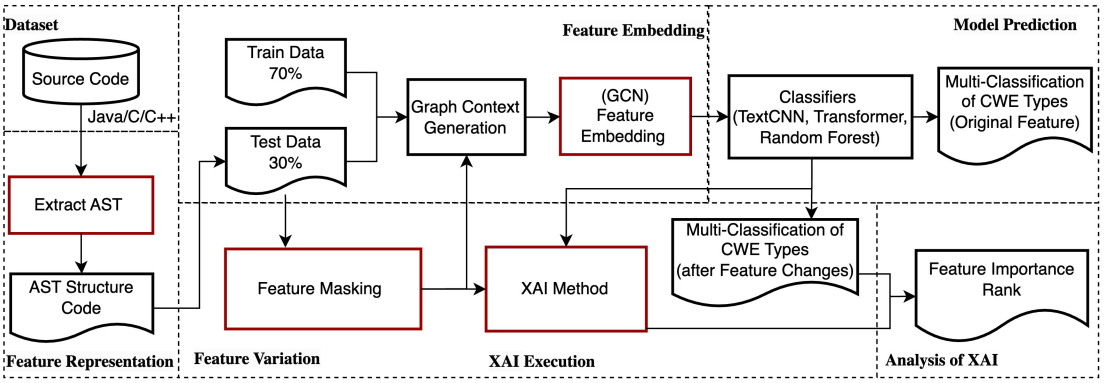



Figure 5.2: The assessment of feature contribution by XAI explanations. The main components include feature representation, feature variation, XAI method, pre-trained model and analysis of XAI results.

## 5.2 An XAI-based Framework for Software Vulnerability Contribution Factor Assessment

This these proposes a workflow that leverages XAI techniques to extract mean feature contribution values and analyze XAI explanation summaries. This approach enables us to quantitatively evaluate the contributions of different features to the multi-classification of code vulnerabilities categorized as CWE types. As discussed in Section 2.3, CWE types are expert-defined classifications based on extensive real-world samples. Their inherent similarities subtly impact the learning tasks associated with vulnerability classification. Therefore, the workflow deploys XAI methods to probe into the high-dimensional space of code features and correlate feature variations to classification results.

As depicted in Figure 5.2, the workflow incorporates several key components distinct from traditional code vulnerability classification solutions. These include feature variation, XAI method application, and XAI output analysis. The workflow employs post-hoc, model-agnostic XAI methods to calculate feature contribution values under various feature variations, such as feature mutation, masking, and removal. The outputs from XAI methods are subsequently analyzed to pinpoint a set of code features with high contribution rankings.

Table 5.1: Syntactic constructs in AST (Abstract Syntax Tree)

| **Meta Syntactic Constructs [112]** | **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>,... |

### 5.2.1 The Graph Context Extraction of Program Code

To capture the connections between semantic meanings and syntactic constructs, it first extracts program paths from the AST (Abstract Syntax Tree) of input source code. These paths consist of leaf nodes representing code tokens and non-leaf nodes denoting syntactic constructs. Figure 5.3 presents the full set of syntactic constructs for a code example with CWE23 Relative Path Traversal Weakness.

```
1 public void action(String data) throws Throwable {
2     String root;
3     /* POTENTIAL FLAW: no validation of concatenated value */
4     root = "/home/user/uploads/";
5     if (data != null) {
6         File file = new File(root + data);
7         FileInputStream streamFileInputSink = null;
8         ...
9 }
```

Listing 5.1: Code snippet from Juliet dataset CWE23 relative path traversal weakness.

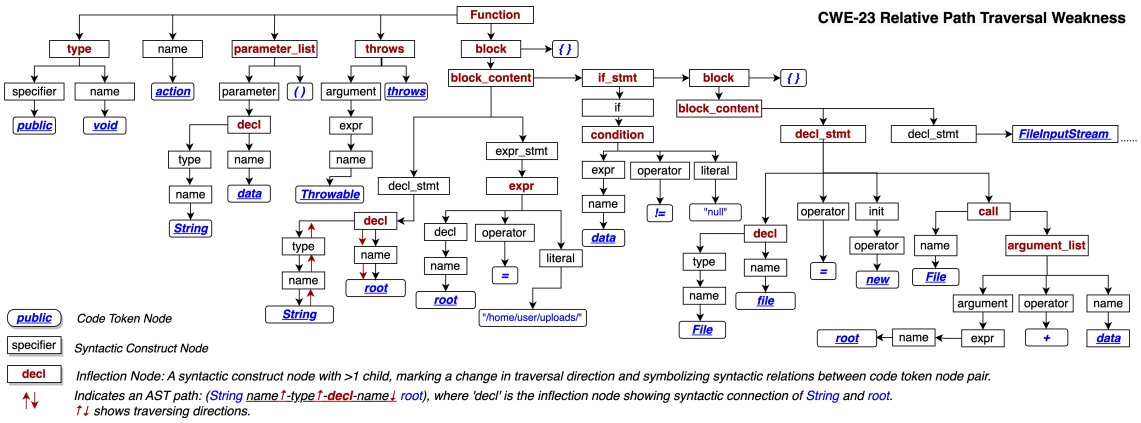



Figure 5.3: The Abstract Syntax Tree of the code snippet from Listing 5.1.
*Note:* The leaf nodes of the tree are code tokens, while the non-leaf nodes are syntactic constructs that provide the syntax structure of the code

Syntactic constructs serve as the fundamental building blocks of program syntax, including elements such as loops, conditionals, declarations, and expressions. Table 5.1 provides a summary of these constructs, including higher-level meta syntactic constructs as defined in previous work [112]. These meta constructs retain the semantic roles within a program. For instance, the *Declarations, Definitions, Initialization* meta category includes syntactic constructs related to defining and initializing variables, functions, and objects.

The focus then turns to the paths connecting code tokens, which preserve functional meanings. As shown in Figure 5.3, a syntactic construct tree represents the code listed in List 5.1, which exhibits a CWE23 relative path 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`. The symbols ↑ and ↓ indicate the traversal directions. In this example, `decl` alters the traversal direction, switching from upward to downward—making it an *inflection node*. By traversing through an inflection node, a pair of code tokens become linked, forming the *shortest path* that includes the nearest inflection node. All nodes on this path then become neighbor nodes of the target code token, including the source code token. As a result, it creates a **graph context** for the target token by extracting the path that links the pair of source and target tokens.

Therefore, it extracts the program's source code into distinct paths, with each path representing the shortest traversal between pairs of code tokens. The graph context of a target node is composed

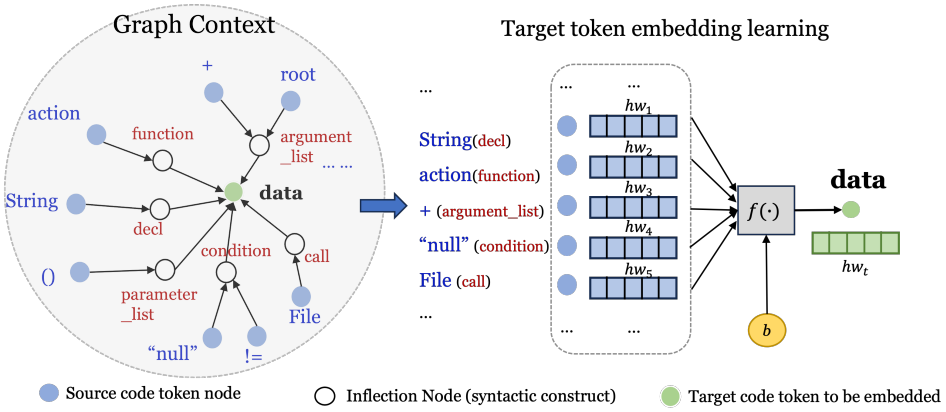



Figure 5.4: The overview of embedding learning. The distributed representations of target code token *data* is learnt from the relevant context tokens (blue nodes) that are fed into a one layer GCN (Graph Convolutional Network). $h_{w_i}$, $h_{w_t}$ are hidden representations of context token and target token, and $b$ is the added bias.


Figure 5.5: An example of how the window size restricts the selection of neighboring nodes as source code node for the target code node `data`, considering both upwards and downwards directions.

of all the paths originating from source leaf nodes and leading to the target node. All the source leaf nodes in this context are considered neighbor nodes of the target code token. Syntactic constructs are situated along these paths, serving as edges that link source code tokens with the target code token. For instance, Figure 5.4 depicts the graph context obtained from the complete syntactic paths covering the CWE23 vulnerability sample code provided in Listing 5.1.

The shaping of the graph context during generation involves two key configurations: *path length* and *window*. *Path length* pertains to the length of the shortest AST path linking two leaf nodes, or code tokens. The *window*, on the other hand, is the maximum distance allowable between the target code token and its neighboring tokens within a code function, applicable in both upwards and downwards directions (see Figure 5.5). When it analyzes a target code token, only the neighboring tokens situated within this *window*, from either direction, are considered as source code token nodes in the graph context. Hence, these two parameters, *path length* and *window*, crucially influence the structure of the graph context.

### 5.2.2 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 5.4 shows the graph context vector is input to a one-layer Graph Convolutional Network (GCN) from a state-of-the-art approach GraphCodeVec [52]. It adopts the GCN model developed in [113] which has demonstrated the usage for six software repository analysis tasks including code classification. The output embedding for each code tokens are 128-dimension vectors containing both code token and syntactic constructs information.

### 5.2.3 Multi-Classification of CWE Types

The GraphCodeVec produces a $128-$dimensional embedding vector for each code token, which can be input to models like textCNN [114], Transformer [84] , and Random Forests [115], similar to the traditional NLP embeddings like Word2Vec [116]. These vectors are directly fed into traditional machine learning models like Random Forests. For textCNN and Transformer models, the embeddings form the initial input layer, which is fine-tuned during training to reduce prediction errors and optimize the representation of tokens for predicting multi-classification CWE types.

**TextCNN** [114] (Text Convolutional Neural Network) is a deep learning model specifically designed for natural language processing tasks. After extracting code representations with graph structural information from GraphCodeVec, it adopts TextCNN to capture the dependencies between the code tokens (also considering the AST path information) and assigning CWE labels from the fully connected layer. This downstream classifier has proven its effectiveness in the original work of GraphCodeVec [52]. **Transformer** [84] is based on self-attention mechanisms, which allow the model to weigh the importance of different tokens in the input sequence. It can effectively capture long-range dependencies within the code, making it a suitable choice for vulnerability detection. **Random Forest** [115] is another option for testing a simple tree-based classifier's ability to efficiently capture the structural information from GraphCodeVec.

By combining GraphCodeVec based embedding learning with these three classifiers, it aims to evaluate the effectiveness of AST based code representation learning and classification in the

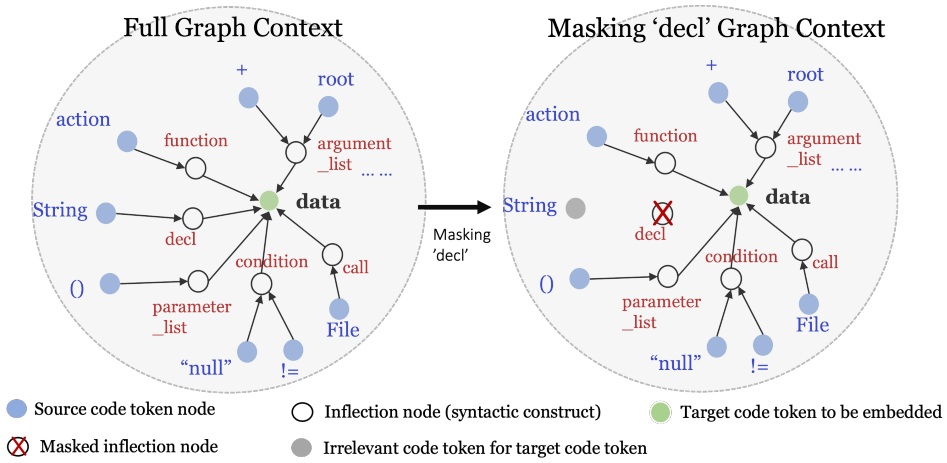



Figure 5.6: After masking syntactic constructs `decl`, the target `data` embedding will not learn the information from AST paths and related source nodes with inflection node `decl`.

context of software vulnerability detection. Subsequently, it selects the best solution to perform downstream XAI feature importance explanation.

### 5.2.4 Feature Masking

In the approach, it regards each syntactic construct as a feature, with the goal of measuring its influence on the model's predictions. This method does this by altering the feature space; specifically, it masks paths in the AST where the construct in the AST as an inflection node, effectively removing some AST paths. For example, when examining the construct `declaration`, it masks paths such as $\texttt{String}^{\uparrow}\texttt{-name}^{\uparrow}\texttt{-type}^{\uparrow}\texttt{-decl}_{\downarrow}\texttt{-name}_{\downarrow}\texttt{-data}$, where it serves as the inflection node. This operation yields a graph context that lacks the `declaration` linking the source code node `String` with the target node `data`. As a result, the `data` embedding lacks the information from its neighboring node `String` and the syntactic meaning of `declaration`, as depicted in Figure 5.6. It then retrieves the embeddings from this modified graph context data and use them to generate predictions with the classification models, which allows us to evaluate the impact on the model's performance.

### 5.2.5 Integrating XAI methods in Multi-Classification

Feature explanation XAI methods require the predictions made by the model with and without a specific feature. In the scenario, this involves the full graph context as well as the context with a masked syntactic construct. Both serve as prediction outcomes for the downstream classifier which predicts the logit of the ground truth label, given the complete feature results and results lacking a particular feature.

```
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),...]
```

Listing 5.2: CWE23 vector featuring syntactic constructs and their corresponding contribution values

As a result, the XAI method produces outputs as CWE vectors containing the features (i.e., syntactic constructs) and their associated contribution values, as shown in List 5.2. Each CWE vector consolidates feature contribution values from data instances with the same ground-truth CWE label. It sorts the features by their contribution values in descending order and establish the feature importance order for each CWE vector, as outlined in **Algorithm. 2**. Each XAI method generates a set of CWE vectors. It compiles these results by averaging the contribution values across different XAI methods, thereby obtaining the final CWE feature importance order. This outcome facilitates the understanding of how features contribute to model predictions and enables us to discern the similarities and differences among various CWEs by comparing their feature importance orders.

We analyze the complexity of our algorithms. Given the size of the dataset samples $N$, the number of feature number $P$ and the CWE label number $K$, the complexity of Algorithm 2 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 2** Compute the CWE vector of each syntactic construct's contribution value

**Require:** The input dataset $X$;

The full AST constructs 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 data set 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**

**Ensure:** CWE vector of each CWE label $cwe_k \in K, V^K$

## 5.3 CWE Similarity Summary and Validation

The approach determines the similarity between CWEs based on the explanation of feature importance related to syntactic constructs provided by XAI. This method allows us to numerically express the similarity between CWE pairs by analyzing the distance between their corresponding feature importance orders. To evaluate the effectiveness of the XAI-based CWE similarity results, it compares them to an expert-defined baseline using four metrics: Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score.

### 5.3.1 Summary of XAI-based CWE Similarity

It denotes the similarity score between CWEs as $\rho$. This score, which represents the relationship between two CWEs, is derived from the normalized ranking distance [117] of their respective feature importance orders, as described in Algorithm 2. A smaller $\rho$ value implies a higher degree of similarity between a CWE pair. The $\rho$ value ranges from 1, indicating total dissimilarity, to 0, which signifies identical CWE pairs. For ease of interpretation, it sorts the $\rho$ values in ascending order, which then serve as the similarity rankings for a given CWE. The specific steps for this process are detailed in **Algorithm 3**. The complexity of Algorithm 3 depends on the number of CWE pair combinations. The complexity is $\Theta(K^2)$, where $K$ is the number of CWE types.

### 5.3.2 CWE Similarity Validation

Table 5.2 shows the baseline CWE similarity that contributed by the community. The CWEs sharing a common characteristic is categorized into tree leaves under a more abstract weaknesses type. In the datasets being examined, the CWEs belong to seven different branches. For example, CWE22, 23, 36 are under path traversal weakness. It can considers CWE23, CWE36 are two siblings of CWE22.

To validate the CWE similarity from XAI explanation, we apply four metrics to compare with the baseline, namely Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score. $B^{cwe_i}$ is the set that contains all the

**Algorithm 3** Create CWE similarity vector for CWE types

**Require:** Sorted vectors of feature importance for each CWE label $V^{cwe_k}$ output from Algorithm 2;

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**

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

CWE types that are sibling to $cwe_i$ defined in the baseline. $W^{cwe_k}$ is the set of CWE types derived from Algorithm 3. Given an example CWE23, $B^{cwe_{23}} = \{cwe_{22}, cwe_{36}\}$ and $W^{cwe_{23}} = \{cwe_{22}, cwe_{79}, ..., cwe_{36}\}$.

(1) **Top-N Similarity Hit** is defined as the boolean value. For example Top-1 Similarity Hit of CWE22 equals one.

$$H_N^{cwe_k} = \begin{cases} 0 & \text{if } B^{cwe_k} \cap W^{cwe_k} \equiv \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

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

$$MRR^{cwe_i} = \frac{1}{||B^{cwe_i}||} \sum_{1}^{||B^{cwe_i}||} \frac{1}{rank_{cwe_j}}, \forall cwe_j \in B^{cwe_i} \tag{5}$$

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 $k = 14$. $MRR^{cwe_{23}} = \frac{1}{2} \times (1 + \frac{1}{k}) =$

Table 5.2: CWE categorized by baseline similarities

| **Category** | **Similar CWEs [32]** |
|---|---|
| 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 |

$\frac{1}{2} \times (1 + \frac{1}{14}) = 0.5357$.

(3) **Mean Average Precision (MAP)** is a metric 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 represent a cut-off rank. For a given CWE type $cwe_i$, Average Precision (AP) is calculated as mean precision value at each rank:

$$AP^{cwe_i} = \frac{1}{||B^{cwe_i}||} \sum_{\kappa=1}^{N} \frac{||B^{cwe_i} \cap W_\kappa^{cwe_i}||}{||W_\kappa^{cwe_i}||} \cdot rel(\kappa) \tag{6}$$

where $rel(\kappa)$ is an indicator function equaling one if the item at rank $\kappa$ is a ground truth sibling CWE type of $cwe_i$ , that is $W^{cwe_i}[\kappa] \in B^{cwe_i}$; 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 over all $Q$ number of CWE types:

$$MAP^{\Phi} = \frac{1}{Q} \sum_{q=1}^{Q} AP^{cwe_q} \tag{7}$$

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

$$\overline{S} = \frac{\sum_{cwe_k \in B^{cwe_i}}^{||B^{cwe_i}||} \sum_{cwe_j \in W^{cwe_i}}^{||W^{cwe_i}||} (1 - \rho(\text{cwe}_k, \text{cwe}_j))}{||B^{cwe_i}|| \cdot ||W^{cwe_i}||} \tag{8}$$

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 by an XAI method. $\rho(\text{cwe}_k, \text{cwe}_j)$ is calculated in Algorithm 3.

Top-N Similarity Hit and Average Normalized Similarity Score help to observe the distance between the baseline and XAI explanation derived CWE types' similarity. Mean Reciprocal Rank and Mean Average Precision measure the accuracy of the CWE similarity derived from the XAI methods.

## 5.4 Extending the XAI-based Framework for Textual-based Feature Contribution Assessment

This section is dedicated to the evaluation of three textual code token features - code token length, code token type, and code token attention values - in the context of text-based explainable code vulnerability learning. The extended framework from previous XAI-based feature contribution assessment in Section 5.2.1 is illustrated in Figure 5.7. The first step involves using tokenization to capture features in the source code while retaining critical symbols and comments. Following this, it separates the dataset into training and testing subsets. Next, it employs three transformer-based models - XLNet [47], Longformer [48], and BigBird [49] - to train on these token representations. In the third phase, it modifies the features extracted from the test dataset, namely the code token length and attention values, to immediately discern their impact on model performance. Afterward, it integrates XAI techniques to further explore the influence of these feature adjustments. By juxtaposing each token's contribution value ascertained by XAI with the results of feature permutation from performance difference, it is able to cross-validate the potency of the approach.

### 5.4.1 Feature Representation

**Code Token Length:** Retaining comments from the original source code leads to longer sequences, often reaching the input limitations of several models. Initially, it constrains the token

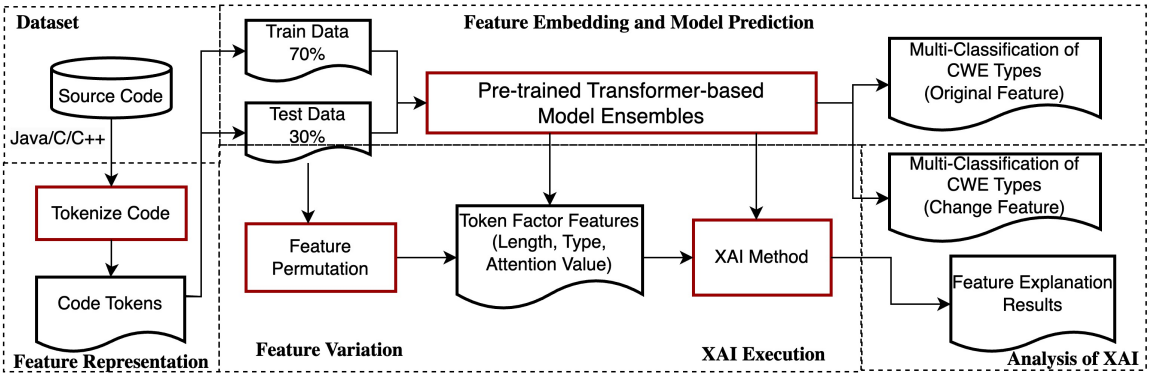



Figure 5.7: Framework of explainable text-based factors assessment.

length to 1,024 tokens per individual program, and subsequently extend this maximum length to 4,096 tokens for the Longformer and BigBird models. This adjustment permits us to directly observe if increased token length yields performance improvements.

**Code Token Type:** The previous research indicates that code comments significantly impact the detection of code vulnerabilities [98], with models learning from comment content to guide their decisions. In this study, the main focus lies in determining the influence of separator symbols in code on transformer-based models. It observes the high importance tokens and their belonging construct types.

**Code Token Attention Value:** The attention mechanism, specifically the scaled dot-product attention, is designed to calculate the importance of input tokens for a particular output token. These attention values help to understand the relationships between different input tokens and assist the model in focusing on the most relevant parts of the input sequence when generating the output. The scaled dot-product attention mechanism computes attention values for a code token $j$ using the following formula [84]:

$$a_t = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{9}$$

In this equation, $Q$, $K$, and $V$ denote the query, key, and value matrices, respectively, while $d_k$ represents the dimension of the key vector. The query matrix represents the current input token, whereas the key and value matrices represent all tokens in the input sequence. The dot product between the query and key matrices captures the similarity between tokens.

### 5.4.2 Multi-Classification of CWE Types

This research employs three state-of-the-arts transformer-based learning models for the task of text-based code vulnerability detection: XLNet, Longformer, and Bigbird.

**XLNet:** XLNet [47] is an autoregressive pre-training transformer model, engineered to capture bidirectional context via a permutation-based training strategy. Having proven its effectiveness across a variety of natural language processing tasks, including text-based code classification [118], XLNet does have limitations in terms of maximum input sequence tokens—1,024 for the XLNet-large and 512 for the XLNet-base pre-training frameworks.

**Longformer:** The Longformer model [48] is designed specifically to process long input sequences more efficiently. It leverages a combination of global and sliding window-based self-attention mechanisms, thus enabling it to handle input sequences of up to 4,096 tokens. Given that the dataset it examines often contains long sequences of code, Longformer's capacity to handle such sequences is notably beneficial for the vulnerability detection task.

**Bigbird:** Bigbird [49], another transformer-based model, has been designed to handle lengthy input sequences. It employs a sparse attention mechanism, known as block-sparse attention, which enables it to scale linearly with sequence length. This unique feature renders Bigbird a suitable choice for large-scale code vulnerability detection tasks. Additionally, it extends the maximum input sequences to 4,096 tokens.

### 5.4.3 Feature Variation

This method first assesses the influence of the three features - code token length, code token type, and code attention values - by observing performance differences after permuting these features.

For **Code token length**, this approach initially limits the token length to 1,024 tokens for an individual program due to the constraints of certain models. It then extends the maximum length to 4,096 tokens for the Longformer and BigBird models that can handle longer sequences. This modification allows us to directly observe whether longer token lengths lead to performance improvements. For **code attention values**, it aims to assess whether the attention value of a token correlates with its importance in the model's performance. It accomplishes this by masking tokens

within different attention value percentile ranges and measuring the resulting performance difference from model. As for **Code token type**, it assesses the construct types from high attention values token results. This study statistically analyzes these types, relating their frequency to their associated attention values.

**Algorithm 4** Assessment influence of token attention values

**Require:** Input data set $X$, full token set $P$,
$S \subseteq \{1, 2, 3, ..., p\} \backslash \{j\}$, the subset of all the tokens by masking a token $j$, model prediction $\hat{f}(X)$, feature $j$ contribution value $\phi_j = \Phi(P, S, j, \hat{f}(X)))$
/*Compute attention values to define masked token set*/
**for all** $j \in P$ **do**
    Calculate attention value $a_j$ (Eq. 9) for token $j$
    **for all** $x_i \in X$ **do**
        **if** $a_j$ is in a certain percentile range **then**
            Add $t$ to the masked token set $A_n$
        **end if**
    **end for**
    $A \leftarrow \{A_1, ...A_n, ...A_N\}$
**end for**
/*Compute the prediction difference under attention value masking*/
**for all** $A_n \in A$ **do**
    $\delta_n \leftarrow |\hat{f}_P(X) - \hat{f}_{A_n}(X)|$
**end for**
/*Compute token contribution values by XAI methods*/
$\phi_j = \Phi(P, S, j, \hat{f}(X))$
**Ensure:** Set of $\{\delta_n\}$, Set of $\{a_j, \phi_j\}$

### 5.4.4 Execute XAI Methods

XAI techniques are integrated to further cross-validate the findings of the previous steps. Following the extraction of token's attention value, it obtains these tokens' contribution values via XAI methods. Comparing these two indicators of importance helps deepen the understanding of the relationship between the model's attention mechanism and XAI methods' explainability. For code token length, it adjusts the max token length parameters in the models to observe the performance difference. For **Code token type**, it presents statistics derived from high-importance tokens according to both attention value and contribution score determined by XAI methods. This part is formulated as Algorithm 4.

## 5.5 Experiments and Analysis

This section designs experiments to systematically examine feature's contribution that deep learning models learn from, associating these with human-interpretative semantic meanings of vulnerable artifacts. The taxonomy, delineated in Section 5.1, identifies nine feature types from a high-level perspective, thereby ensuring transferability across both text-based and graph-based code representation methods.

### 5.5.1 Research Questions

In an Abstract Syntax Tree (AST), code token nodes are interconnected through inflection nodes, representing syntactic constructs that convey higher-level abstract semantic meanings [119]. Using XAI methods, it aggregates the importance of these syntactic constructs and emphasize the commonness of CWE labels based on their input data features. Additionally, it devises experiments to derive and compare the similarity results with knowledge-based baselines. Furthermore, to determine if the attention value can serve as a proxy for the importance of code tokens, to understand how the length of code tokens affects model prediction, and to identify high-importance code token types, it is motivated to evaluate text-based feature types and their influence. From the above rationale, it derives the following research questions:

**RQ1. What are the top-ranking syntactic constructs in Abstract Syntax Trees (AST) based code representation relative to software vulnerability types?** This question focuses on applying XAI techniques to rank the importance of syntactic constructs in the prediction process of a machine learning model across various vulnerability types.

**RQ2. How does the CWE similarity, as summarized by the importance explanations of syntactic constructs, align with expert-defined similarity?** This question aims to validate whether the patterns discerned via XAI-driven feature importance explanations align with expert-determined baseline similarities. In doing so, it seeks to corroborate the effectiveness of the approach in quantifying the similarity of vulnerability types against the baselines.

**RQ3. How do text-based code features influence code vulnerability detection tasks?** This study assesses the impact of three text-based code features - code token type, code token length, and

code token attention value - and cross-validate the performance variance results using XAI methods.

These three research questions are refer to the second core question presented in Section 1.1. To address research questions, this study divides the research into three primary sections, each correlating with a specific task: 1) ranking the importance of syntactic constructs in the abstract syntax tree; 2) validating CWE similarity against an expert-defined baseline; 3) assessing the influence of text-based feature types. The dataset is first introduced. Each section comprises a step-by-step approach, results, and conclusions.

### 5.5.2 Dataset

The investigation includes an examination of three benchmark software vulnerability datasets at the method/function level: Juliet Test Suite (Java), OWASP Benchmark (Java), and the Draper dataset (C/C++). Each of these datasets varies based on the method of collection and annotation of code samples and can be classified into three distinct categories: synthetic, semi-synthetic, and real data.

Synthetic data involves both the vulnerability code examples and their annotations being artificially constructed. For instance, the Juliet Test Suite, developed by the National Security Agency's Center for Assured Software, falls into this category. This dataset is composed of 217 vulnerable methods (accounting for 42%) and 297 non-vulnerable methods (constituting 58%), offering a balanced distribution of method-level examples, all of which are constructed based on recognized vulnerability patterns.

Semi-synthetic data, on the other hand, pertains to either the code or its annotation being artificially derived. The OWASP Benchmark dataset, which is also based on Java, serves as an instance of semi-synthetic data. It includes 1,415 vulnerable methods (52%) and 1,325 non-vulnerable methods (48%).

Finally, real data consists of code and corresponding vulnerability annotations sourced directly from real-world repositories. The Draper dataset is an example of this category. The functions in this dataset were gathered from open-source repositories and annotated using static analyzers. Despite the original dataset featuring an imbalanced distribution, it is been processed into a balanced dataset for practical purposes while preserving all comments and code. As a result, this dataset encompasses

43,506 vulnerable functions, which constitutes 50.1% of the dataset.

Table 5.3 provides a summary of the vulnerability types and their respective distributions in each of these datasets.

Table 5.3: 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% |
| | CWEOther | Other | 2.9% |
| **Draper** | CWE119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 28.4% |
| | CWE120 | Classic Buffer Overflow | 26.9% |
| | CWEOther | Other | 26.7% |
| | CWE476 | NULL Pointer Dereference | 11.9% |
| | CWE469 | Use of Pointer Subtraction to Determine Size | 6.1% |

### 5.5.3 Selecting XAI Methods

The selection of XAI methods is based on the findings from the experiments detailed in the appendix A. Our goal is to achieve higher consistency and stability in the explanation results, such that the selected XAI methods could agree and provide consistent syntactic constructs ranking outcomes for observation. Additionally, it is important that the time required to obtain these results remained reasonable.

Based on these criteria, we select SHAP due to its better performance in consistency evaluations, and Mean-Centroid PredDiff, which has an acceptable runtime and also performs well in terms of consistency and stability.

### 5.5.4 Ranking the Importance of Syntactic Constructs in AST (RQ1)

The approach to evaluating the importance of syntactic constructs consists of three main steps: (1) code transformation and classifier application, (2) syntactic construct masking and prediction difference calculation, and (3) the use of XAI methods and cross-validation for construct importance determination.

#### Experiment Design

***Step 1:*** Code Transformation and Classifier Application. It leverages the srcML tool[1] to convert method-level programs into Abstract Syntax Tree (AST) structures, discarding code comments but retaining mathematical and logical operators. The resulting XML-based content, which encompasses both code tokens (AST leaf nodes) and AST paths, is converted into a graph context as described in section 5.2.1. it maintains the default ***edge length*** of 8 and ***window*** size of 10 as per [52].

***Step 2:*** Code Token Embedding Learning. It employs a graph convolutional network-based embedding model to convert the graph context into a 128-dimensional vector representation of each code token. This model can be enhanced with a classifier layer for downstream tasks. The hyperparameters are kept at default settings: one layer, a batch size of 64, and a dropout rate of 0.

***Step 3:*** Syntactic Construct Masking and Importance Analysis. After generating full graph context embeddings for all program code tokens, it masks each syntactic construct, yielding altered embedding sets devoid of the masked constructs' syntactic meanings. The XAI methods SHAP and Mean-Centroid PredDiff utilize these prediction result differences as input, derived from classifications based on the embeddings. It then averages the contribution values across different XAI methods to compile the results.

#### Experiment Results

Table 5.4 ***(for step 2)*** presents the performance of three classifiers augmented with GraphCodeVec embeddings on the Juliet, OWASP, and Draper datasets. From the results, it is evident that

[1] https://www.srcml.org/

the TextCNN classifier significantly outperforms RandomForest and Transformer in terms of both accuracy and F1-score across all three datasets. It then choses TextCNN as the classifier with GraphCodeVec to perform the following XAI tasks.

Table 5.4: Performance of classifiers augmented with GraphCodeVec embeddings

| Model | Metric | **Juliet** | **OWASP** | **Draper** |
|---|---|---|---|---|
| **RandomForest** | 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 |

Figure 5.8 ***(for step 3)*** shows the meta syntactic constructs (categorized the syntactic constructs in Table 5.1 ) importance ranking for each CWE type. It observes that **1)** different CWEs have varying importance orders of constructs, indicating that each vulnerability is affected differently by the code syntax content. **2)** Certain constructs, such as *statement_subelements, parameters, name, statement* consistently rank high across multiple CWEs, suggesting their general impact on code vulnerabilities. In contrast, constructs like *specifier, classes_etc* have lower importance across CWEs. **3)** Some vulnerabilities share common top-ranked constructs, which may be indicative of similar code patterns. For instance, CWE78, CWE79, and CWE89 share similar top-ranked constructs such as *statement_subelements, name, decl_def_init, and operators*.

**Conclusion, Answering RQ1**

The significance of syntactic constructs varies across different CWEs and datasets, thus suggesting a diverse role of these constructs in different contexts. Certain constructs, such as *statemen_subelements, statement, name,* and *parameters*, consistently feature high in the rankings across sixteen CWEs approximate 80% of all CWE types. Moreover, certain CWE types share similar top-ranking constructs, potentially indicating a commonality in the code patterns contributing to these vulnerabilities.

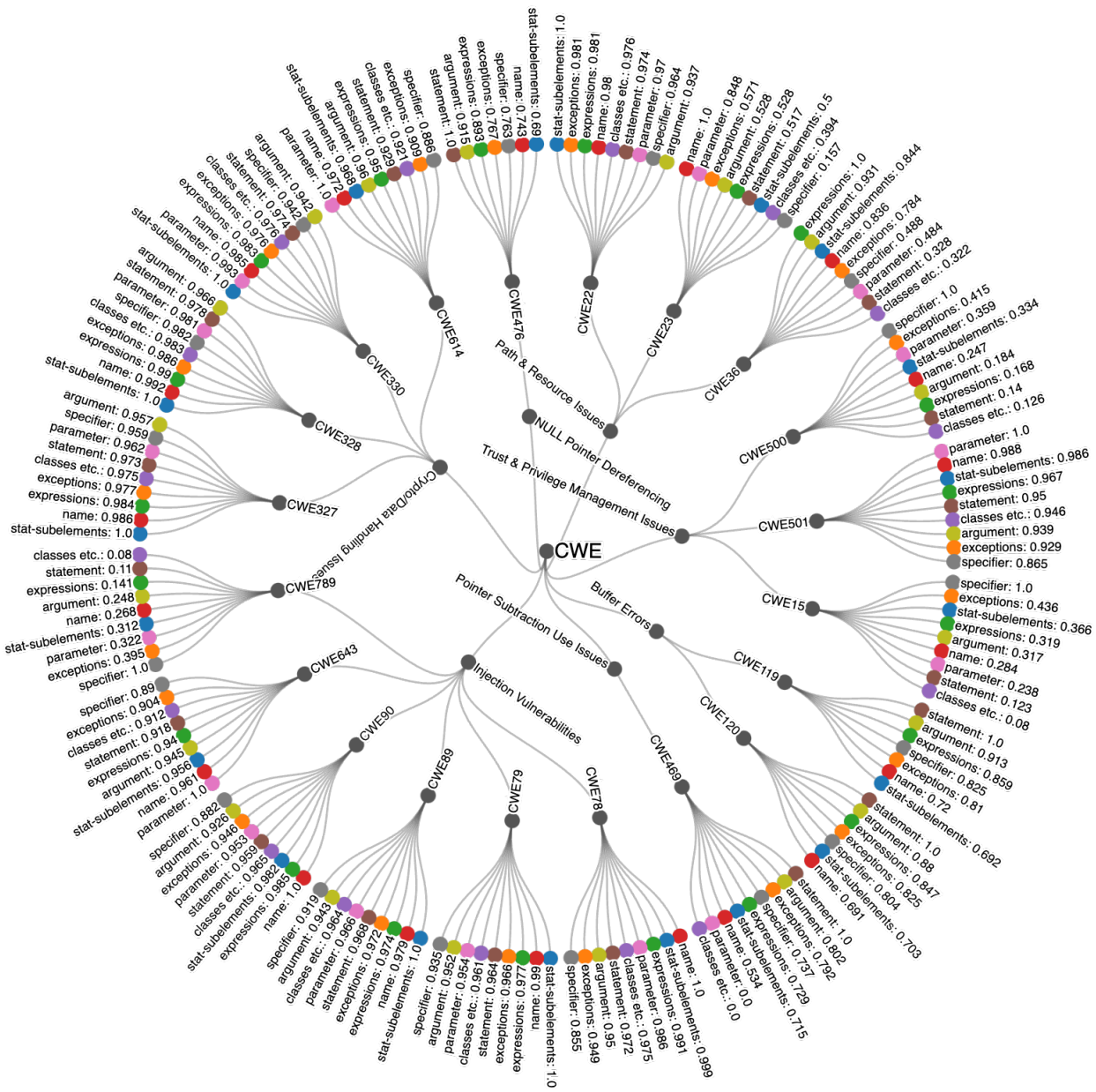



Figure 5.8: Feature importance of meta syntactic constructs per CWE type, represented in descending order clockwise.

*Note:* Importance is quantified as normalized feature contribution value from XAI method, shown in the leaves nodes after contracts name. CWEs that describe similar vulnerability issue [32] are also categorized in the dendrogram.

While these prior studies [80, 82] identified *statement_subelements* and *name* as key factors influencing code vulnerability, they fell short of providing a comprehensive assessment of the role of different syntactic constructs. The study extend these findings by offering a comprehensive view of the role of syntactic constructs in contributing to code vulnerabilities.

### 5.5.5 Validating CWE Similarity against Expert-defined Baseline (RQ2)

Guided by the observation of syntactic similarities among certain CWE types, it boards on quantifying CWE similarity based on feature importance order distance, comparing these results with an expert defined CWE similarity baseline. While the baseline is from the understanding of domain experts, the method leverages XAI techniques to extract insights directly from a data-driven model.

#### Experiment Design

The experiment design comprises two steps: ***Step 1:*** This experiment calculates CWE similarity based on the feature importance order of syntactic constructs, as demonstrated in Figure 5.8. Instead of considering ten meta constructs, it focuses on forty more specific syntactic constructs' importance orders for a detailed comparison. Through the CWE similarity Algorithm 3, it can measures the proximity between different CWEs based on their syntactic construct importance orders, as derived from XAI explanations.

***Step 2:*** It then validates the XAI-based CWE similarity against an expert-defined baseline [32]. The similar sibling set for each CWE (required in Algorithm 3) is detailed in Table 5.2. To evaluate the results, it uses three metrics: Top-N Similarity Hit, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Average Normalized Similarity Score (Subsection 5.3.2). Each CWE type yields a score for these three metrics, and it obtains a final score by averaging across all CWE types. This process quantifies the effectiveness of the XAI-based method.

#### Experiment Results

Figure 5.9 ***(for step 1)*** presents the CWE similarity $\rho$ across the CWEs examined in three datasets based on the importance of syntactic constructs from the XAI approach. For example,

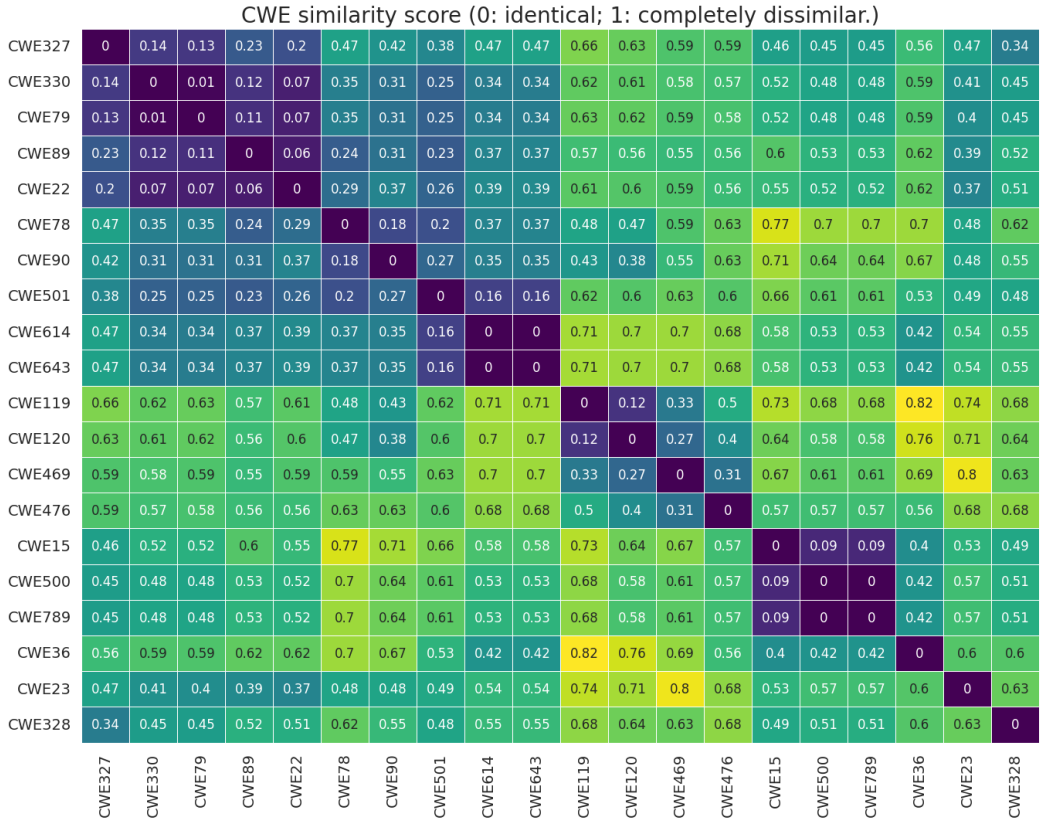



Figure 5.9: CWE similarity score $\rho$ for CWE pair from syntactic construct feature importance based on XAI approach.

CWE23 and CWE22 display a strong similarity, indicated by a low distance value, suggesting that they share similar syntactic constructs. In contrast, CWE23 and CWE328 have a high distance value in the matrix, indicating a low degree of similarity between them in terms of their syntax information.

The results displayed in Table 5.5 ***(for step 2)*** highlight the effectiveness of the approach in deriving CWE similarity based on XAI methods. A Top-1 Hit rate of 75% in alignment with the expert-defined baseline shows robust agreement. Moreover, extending this to the Top-5 similar CWEs, the alignment with the baseline increases further, with more than 87% of CWEs in the same category as defined by the experts. These findings are reinforced by a Mean Average Precision (MAP) score of 0.696, affirming that the approach tends to rank similar CWEs, as per the baseline, higher in the list.

Table 5.5: 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, that each row is the AP (Average Precision) of a CWE, and $\overline{S}$ represents the Average Normalized Similarity Score. CWE469 and CWE476 do not have a similar CWE in the datasets scope.

**Conclusion, Answering RQ2**

The XAI-based method for evaluating CWE similarity has demonstrated its effectiveness in identifying related CWEs, capitalizing on shared syntactic construct characteristics. With a Top-1 Similarity Hit of 77.8%, a Top-5 Similarity Hit of 88.9%, and a MAP score of 0.696, the method exhibits efficacy with expert-derived CWE similarity rankings. By emphasizing feature importance and elucidating the reasons for CWE similarity, it provides an effective validation for expert summaries built upon experiential knowledge.

### 5.5.6 Assessing the Influence of Textual-based Features (RQ3)

#### Experiment Design

The experiment encompasses **five steps**. Initially, it pre-processes the datasets and evaluate the efficacy of three models on the code vulnerability detection task. It also examines their performance as token length increases. Subsequently, it obtains the token attention values for the test dataset, mask various attention percentiles of tokens, and execute the prediction task. It also employs two XAI techniques, SHAP and Mean-Centroid PredDiff to determine code tokens' contribution values and juxtapose them with their attention values. Finally, it performs statistical analysis on the types of high attention value tokens pertaining to the syntactic constructs displayed in Table 5.1, as well as their frequency in each program within the dataset.

The raw code data encompasses several types of information: 1) code, 2) special symbols which include punctuation characters (for example, . ,: ;? () *][ '"*), 3) mathematical and logical operators (for instance, +-=/∗! & |<>), 4) miscellaneous symbols (such as, @ˆ///**/), and 5) code comment text. It performs data preprocessing by discarding miscellaneous symbols (category 4) utilizing regular expressions. In the Juliet dataset, it is found that the CWE information is directly incorporated into the code. To avert data leakage, it extracts this content. For example, *CWE89_SQL_Injection* is replaced by an empty string as depicted in Listing 5.3. This precaution ensures that the model does not have access to explicit CWE labels within the code during its training and evaluation phase, thereby enabling a fair performance assessment.

```
1 # Before processing
2 package testcases.CWE89_SQL_Injection.s01;
3 public class CWE89_SQL_Injection__connect_tcp_execute_01 extends
      AbstractTestCase
4 # After processing
5 package testcases s01
6 public class 01 extends AbstractTestCase
```

Listing 5.3: Remove CWE label content to avoid data leakage (from Juliet dataset)

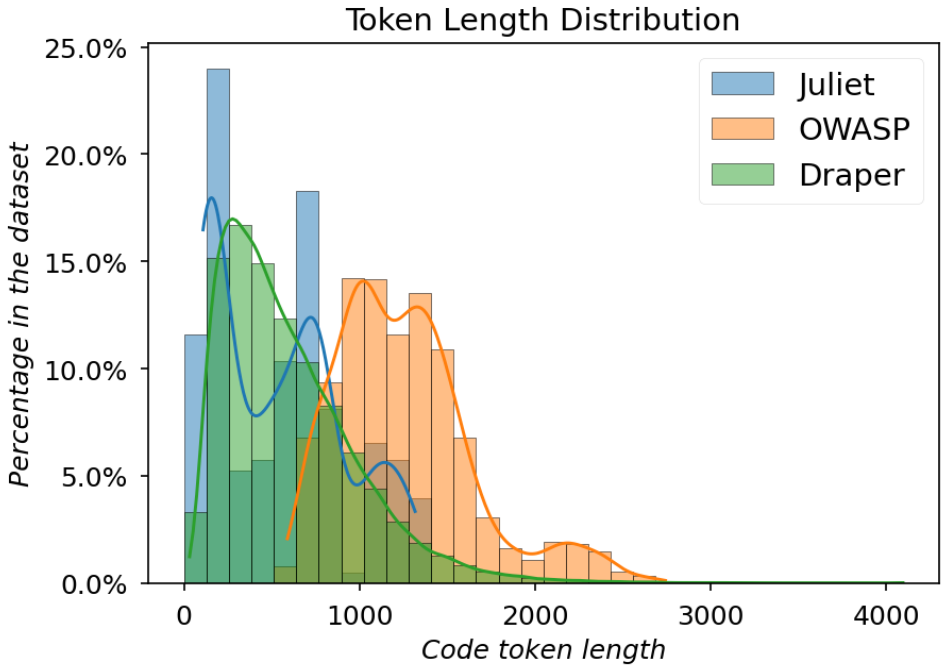



Figure 5.10: Code token length distribution.
*Note:* Bars: Percentage of code token length in each dataset, Curves: kernel density estimation smoothed

***Step 2:*** Model Performance Evaluation. Even though the interpretations of a model with a lower performance might be beneficial for a particular task, it is essential to accurately measure the predictive proficiency. Hence, it tests the performance of three models: XLNet, Longformer, and Bigbird, on a multi-class classification problem. It notes from Figure 5.10 that the token length varies in each dataset. OWASP presents the longest code length exceeding 1024 tokens. While XLNet has a maximum token length limitation of 1024, Longformer and BigBird can handle longer documents and sequences, extending the max token length limit to 4096. It then evaluates the efficacy of Longformer and BigBird using both token lengths, 1024 and 4096, to discern the optimal configuration for the vulnerability detection task. This process enables us to identify the most successful model by considering various metrics such as accuracy and F1-score.

***Step 3:*** Analysis of Attention Value Ranges on Model Performance. After selecting the optimal model, it extracts each token's attention value within the source code using the model-specific attention mechanism. It then establishes ten attention value ranges, like the top 90th percentile, 80th to 90th percentile, and so on. For each range, it masks the tokens within that attention range by

replacing them with a null string. Afterwards, it employs the pre-trained model to evaluate the performance on the altered corpus. This helps us analyze the impact of masking tokens within different attention value ranges on the model's performance.

***Step 4:*** Correlation Validation between Token Importance by XAI Methods and Token Attention Value. This step derives the importance of a token from feature importance explanation methods such as SHAP, which quantify the contribution value of a code token. It applies two XAI techniques, SHAP and Mean-Centroid PredDiff, to the best-performing model to ascertain each token's contribution to the vulnerability detection task. Due to the computational complexity of SHAP, it concentrates on the top 1000 most frequent tokens. It then probes into the correlation between the attention values of tokens and their contribution values, as determined by the XAI techniques.

***Step 5:*** Analysing High Attention Value Tokens. To decipher which types of tokens the model focuses on, it analyzes the top 20 tokens with attention values over the 90th percentile, based on their frequency within each program. First, this approach compiles all tokens within each dataset that exhibited attention values over the 90th percentile, creating a unique set. Subsequently, it computes the frequency of these tokens by dividing the number of their appearances in this unique set by the total number of programs. A frequency value of 100% indicates that the token appears in every program. Additionally, it scrutinizes the construct types of these tokens, as detailed in Table 5.1, to further comprehend their specific classifications.

**Experiment Results**

Table 5.6 ***(for step 2)*** illustrates that increasing token length enhances model performance especially for the OWASP dataset with longer code contents. Longformer with a 4096-token length achieves the best performance on the Juliet dataset (F1-score: 0.8454) and the Draper dataset (F1-score: 0.7595). However, XLNet has a limitation on token length and generally underperforms compared to Longformer and BigBird on all datasets. Considering these results, Longformer is chosen for the following validation steps due to its superior performance on two datasets.

Figure 5.11 ***(for step 3)*** shows the change in model performance when masking code tokens with varying attention value percentiles. It observes that removing tokens with lower attention values has a relatively minor impact on performance. However, when more than 60 percentiles of attention

Table 5.6: Code token lengths effects: the performance comparison of increasing token length across multiple models and datasets

| Model | Metric | **Juliet** | | **OWASP** | | **Draper** | |
|---|---|---|---|---|---|---|---|
| | | Len=1024 | Len=4096 | Len=1024 | Len=4096 | Len=1024 | Len=4096 |
| XLNet* | F1-Score | 0.7618 | - | 0.7411 | - | 0.7311 | - |
| | Precision | 0.7739 | - | 0.7416 | - | 0.7316 | - |
| | Recall | 0.7500 | - | 0.7407 | - | 0.7307 | - |
| Longformer | F1-Score | **0.7709** | **0.8301***(+7.7%)* | **0.7486** | 0.8027*(+7.2%)* | **0.7489** | **0.7492***(+0.04%)* |
| | Precision | 0.8056 | 0.9293 | 0.6145 | 0.8452 | 0.6830 | 0.7024 |
| | Recall | 0.7391 | 0.7500 | 0.9577 | 0.7642 | 0.8288 | 0.8027 |
| BigBird | F1-Score | 0.6989 | 0.8046*(+15.1%)* | 0.7455 | **0.8380***(+12.4%)* | 0.7456 | **0.7492** *(+0.05%)* |
| | Precision | 0.8935 | 0.8378 | 0.7642 | 0.8452 | 0.7320 | 0.7406 |
| | Recall | 0.5739 | 0.7739 | 0.7277 | 0.8310 | 0.7597 | 0.7581 |
| GraphCodeVec** (with TextCNN) | F1-Score | - | **0.8358** | - | 0.6956 | - | **0.7569** |
| | Precision | - | 0.8412 | - | 0.6919 | - | 0.7470 |
| | Recall | - | 0.8305 | - | 0.6993 | - | 0.7671 |

*Note:* *: XLNet has the maximum token length 1,024. **: The performance of GraphCodeVec with TextCNN is from Table 5.4, it outperforms the textual based models on Juliet and Draper dataset. As a graph-based represented model, it is not examined by code token length. The bold is the best F1-Score in three attention-based models under the same dataset and length.

values are removed, the performance degrades significantly. It is also observed that the performance decline stabilizes after the 70th percentile of attention values. It is difficult to determine whether these attention value ranges play the same contribution role, or if it is due to the model's robustness, which allows it to maintain performance despite having incomplete or partially removed contents. To further investigate the influence of higher attention value tokens, it involves XAI methods in the next step. From the dataset perspective, the effect of masking differs across datasets. The OWASP dataset is more sensitive to masking, with a greater decline in F-1 Score compared to the Juliet dataset.

Figure 5.12 ***(for step 4)*** demonstrates that tokens with a higher attention value range (over 90 percentile) tend to be associated with larger feature contribution values. While there are variances in the contribution values across different percentile ranges, the prevailing pattern seems to be that higher attention values are aligned with significant feature contribution. This noteworthy relationship suggests that the attention value could be a proxy for the importance of a token. It notes a consistency between two XAI methods, both indicating that tokens with higher attention values contribute more significantly to the overall prediction, as evidenced by XAI-derived contribution values.

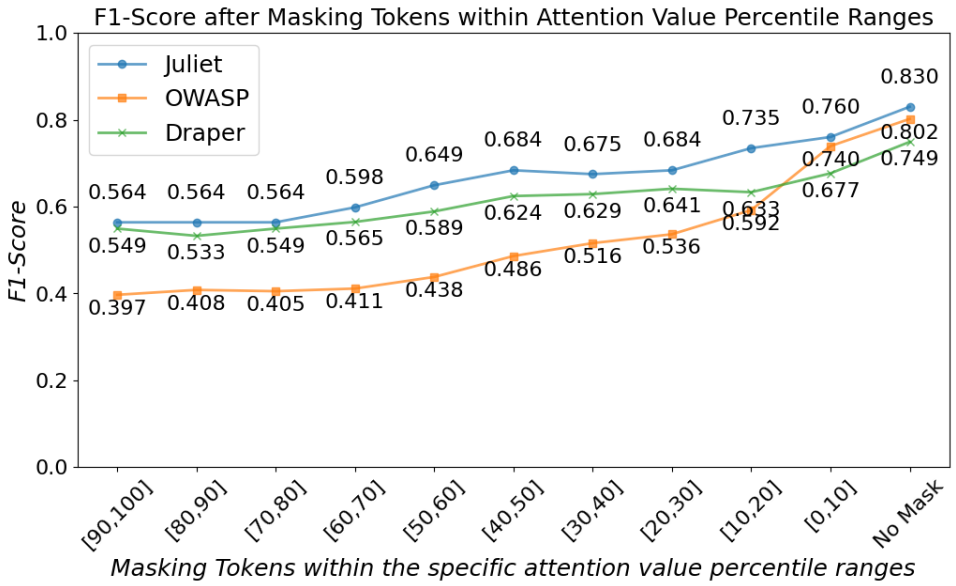
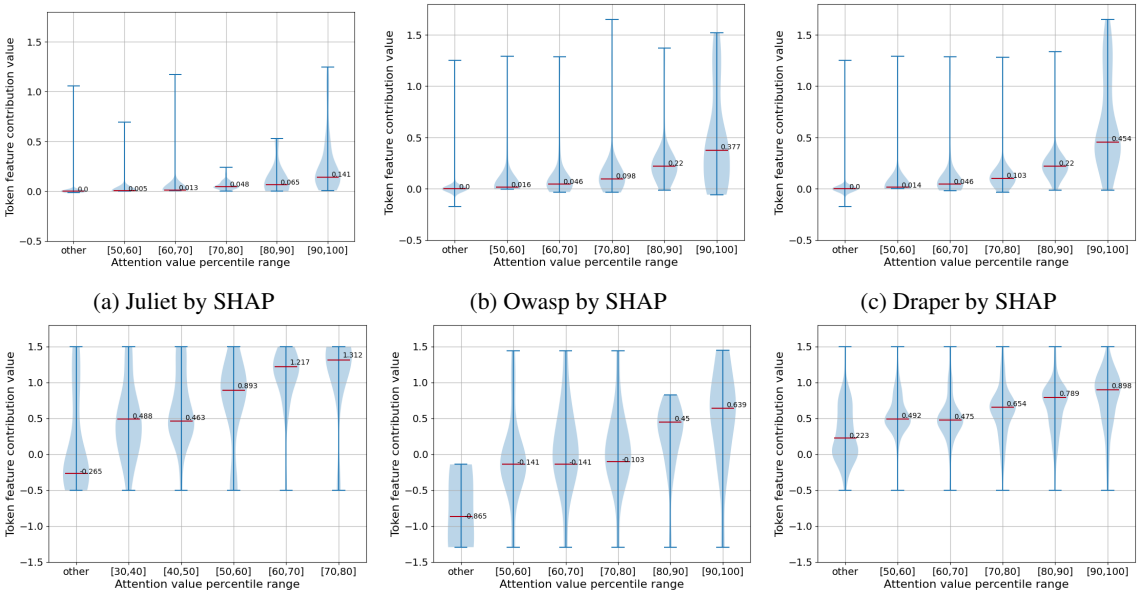



Figure 5.11: Token attention value affects: the performance comparison (F1-Score) after masking code tokens by multiple attention value percentile ranges.




(a) Juliet by SHAP (b) Owasp by SHAP (c) Draper by SHAP

(d) Juliet by MCP (e) Owasp by MCP (f) Draper by MCP

Figure 5.12: Correlation between token's feature contribution value from XAI methods (SHAP, MCP-Mean Centroid PredDiff) and token's attention values (annotated with median value).

In addition, it summarizes the token content, their syntactic constructs, and the occurrence in the dataset from tokens with an over 90 percentile range in Table 5.7 ***(for step 5)***. The occurrence with 100% means this high attention value tokens occur in every program in the dataset. It is noticeable

Table 5.7: Token type affects, top 20 tokens with high attention values (over 90 percentile) for each dataset

| Juliet | | | Owasp | | | Draper | | |
|---|---|---|---|---|---|---|---|---|
| **Token** | **Constructs** | **Occurrence*** | **Token** | **Constructs** | **Occurrence** | **Token** | **Constructs** | **Occurrence** |
| <s> | <separator>** | 100.0% | <s> | <separator>** | 100.0% | <s> | <separator>** | 100.0% |
| (){} | <block>*** | 100.0% | (){} | <block>*** | 100.0% | (){} | <block>*** | 81.07% |
| to | <comment> | 100.0% | Bench | <comment> | 100.0% | , | <separator> | 60.41% |
| Filename | <comment> | 98.26% | owasp | <name> | 100.0% | int | <type> | 52.2% |
| template | <comment> | 94.78% | value | <argument> | 100.0% | char | <type> | 40.23% |
| import | <import> | 72.17% | public | <specifier> | 98.26% | NULL | <literal> | 24.43% |
| java | <name> | 66.09% | License | <comment> | 89.53% | c | <name> | 23.49% |
| tmpl | <comment> | 65.22% | Http | <import> | 87.79% | n | <name> | 19.9% |
| support | <import> | 54.78% | Exception | <import> | 86.34% | ++ | <operator> | 18.4% |
| - | <comment> | 50.43% | class | <class> | 71.22% | s | <name> | 14.38% |
| File | <type> | 50.43% | Response | <expression> | 53.49% | != | <operator> | 11.85% |
| injection | <name> | 45.22% | type | <expression> | 25.0% | sizeof | <size_of> | 11.68% |
| ERA | <comment> | 36.52% | String | <type> | 22.38% | return | <return> | 11.46% |
| , | <separator> | 33.91% | java | <name> | 21.8% | struct | <name> | 11.19% |
| null | <literal> | 31.3% | weak | <literal> | 17.44% | *p | <modifier> | 10.74% |
| Connection | <comment> | 29.57% | age | <name> | 16.28% | & | <operator> | 10.51% |
| statement | <decl_stmt> | 26.09% | param | <declaration> | 14.24% | buf | <name> | 10.16% |
| sql | <name> | 25.22% | user | <name> | 14.24% | if | <if> | 10.08% |
| util | <import> | 25.22% | request | <name> | 13.66% | const | <specifier> | 9.79% |
| Variant | <comment> | 25.22% | IO | <import> | 13.08% | == | <operator> | 9.13% |

*Note:* *: an occurrence of 100% indicates that the token appears in every program of the dataset. **:<s> is a special token in Longformer model for separation. ***: we combine the percentage of (){} together.

that the separators, such as parentheses, commas, and special token from transformer model have a higher occurrence. This is consistent with finding in other studies [80]. In Java-based code (Juliet and Owasp), the content with a higher occurrence includes variable names, class names, and the corpus from code comments. On the other hand, in C/C++ code (Draper), there is a higher occurrence of variable type declaration, operators, such as arithmetic and logical operators. This observation highlights the differences in token distribution and importance between programming languages, suggesting that the model's attention mechanism could capture language-specific features that contribute to its performance.

### Conclusion, Answering RQ3

The model's performance is affected by the length of code tokens, with an increase in the token length boosting the efficacy of attention-based transformer models, particularly the Longformer, a finding which aligns with the results from the study by Yuan et al. [99]. When considering the type of code tokens, attention-based models tend to focus on both semantically significant tokens, such as variable names and code comments, as well as separators like commas and brackets that denote code

sections. This observation corroborates the conclusions drawn in the studies by Vashishth et al. [30], Sharma et al. [80], and Sotgiu et al. [81], and is further reinforced by a particular case study from the code vulnerability task. Transformer-based models are equipped to capture language-specific features that exhibit variation between Java and C/C++.

The importance of tokens in vulnerability detection tasks is effectively mirrored by attention values. This is evidenced by the model's performance decline when tokens within higher attention ranges are masked. There seems to be a correlation between attention values and token contribution values derived through XAI methods, where tokens with higher attention values frequently correspond to higher contribution values. The significance attributed to tokens by the attention values (obtained from deep learning models) is found to be consistent with the contribution values (sourced from backward reasoning and input-to-output tracing). Moreover, the two XAI techniques, SHAP and Mean-Centriod PredDiff, consistently indicate that tokens with higher attention values carry more importance.

## 5.6 Retrospection of Similar CWE Code Sample Siblings

### 5.6.1 A Detailed Showcase of CWE23 and CWE36

CWE23 (Relative Path Traversal Weakness) and CWE36 (Absolute Path Traversal Weakness) are both children of the same parent CWE type, "Improper Limitation of a Path Name to a Restricted Directory." These sibling CWE types share a common problem: a lack of input validation. This section reflects from experiment results to reveal how the framework analysis the similarity of the code property.

Figure 5.13 showcases a code snippet where user input `data` is used directly to access files, creating a potential security risk. The distinction between CWE23 and CWE36 lies in the way this input is utilized: in CWE23, it is appended to a root path, while in CWE36, it is used directly as the path. Mis-classification can occur when a deep-learning model mistakenly identifies a CWE36 type code as CWE23, due to the shared parent and resulting similarity between these types. The established CWE knowledge base [32] can be used to identify similarities between any pair of CWE types. Given that the CWE type is the target output of a learning model, the similarities between

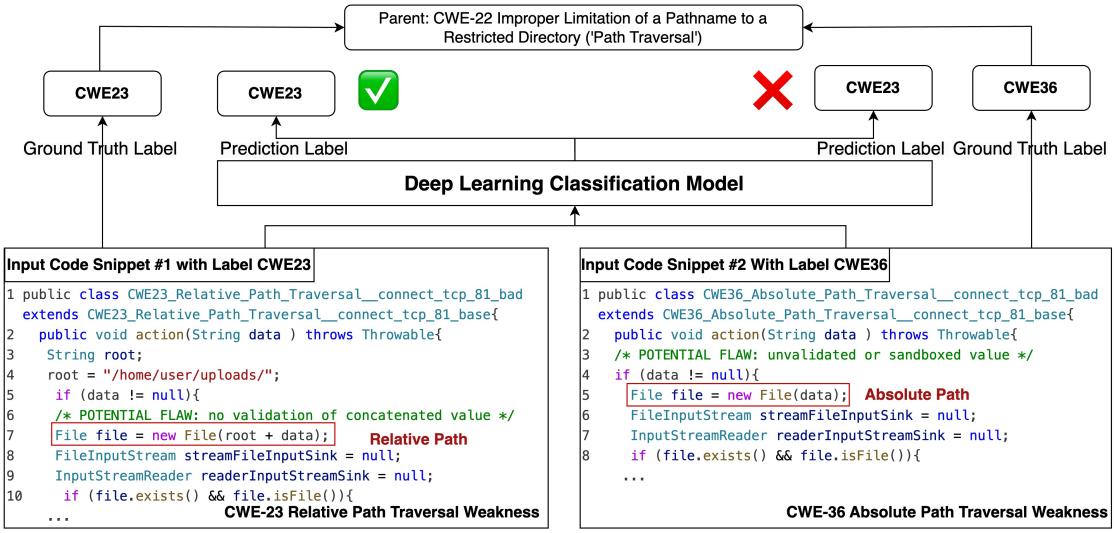



Figure 5.13: An example of deep learning model gives a incorrect prediction on CWE siblings. *Note:* CWE23 with relative path traversal weakness and CWE36 with absolute path traversal weakness. The prediction results are from GraphCodeVec [52] model in Juliet dataset [120].

sibling CWEs can provide insight into how a model determines its prediction results based on the importance of program code feature representation.

This study proposes using XAI methods to explore the high-dimensional space of program code and its association with potential vulnerability types. Given that feature importance explanation methods are post-hoc and model agnostic, they are well-suited to assessing the encoded feature representation of different types of models. The XAI methods associate outputs with changes in inputs, referring to specific metrics such as feature contribution value and feature importance rank.

In the case shown in Figure 5.13, it is found that both attention-based and graph-based models had difficulty differentiating between the similar CWE labels, CWE23 and CWE36. By examining these cases, it could be explored how models interpret vulnerability based on the similarity of CWE types in the feature representation space.

For instance, the code snippet in Figure 5.13 showcases a significant difference in the way CWE23 and CWE36 handle paths. However, attention-based models, like Longformer, often fail to focus on essential code statements and can lean towards learning irrelevant features, as shown in Figure 5.14. Therefore, attention values may not directly provide understandable visualization of vulnerable code features, emphasizing the need for cross-validation with XAI methods.

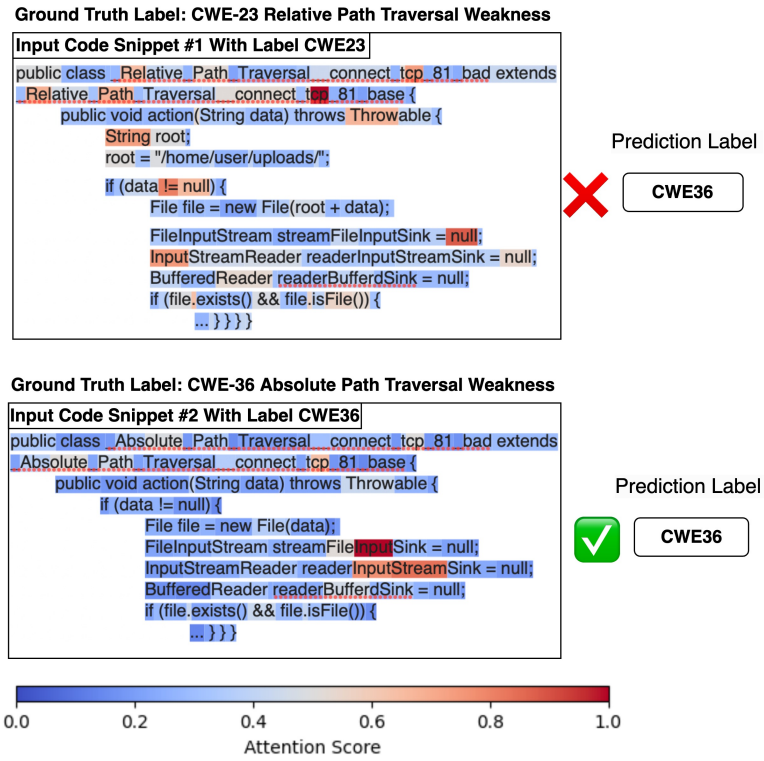



Figure 5.14: Higher attention value code tokens are not reflecting the vulnerable code lines in two code snippet with CWE23 and CWE36.

*Note:* The prediction results are from Longformer model.

In contrast, the graph-based model GraphCodeVec struggles to predict accurately in this case. However, the XAI method could offer an interpretive order of feature importance for syntactic constructs. This ability to provide a deeper understanding of how different constructs contribute to vulnerability is unique to the XAI method, making it a crucial tool in vulnerability detection. For instance, the XAI method identified `argument_list`, `argument`, and `operator` as higher-ranking constructs for CWE23 compared to their ranking in CWE36, as highlighted in Table 5.8. This finding corresponds with the unique characteristics of CWE23, where an additional argument `root` and a `+` operator are incorporated into the file, thus transforming it into a relative path. This distinct AST path difference between the two cases is depicted in Figure 5.15.

Despite the overarching similarity in feature importance orders between the two cases, with `name` and `if` leading the ranking, the XAI method offers an in-depth understanding of how distinct constructs contribute to the identification of a vulnerability. This XAI-based analysis increases the

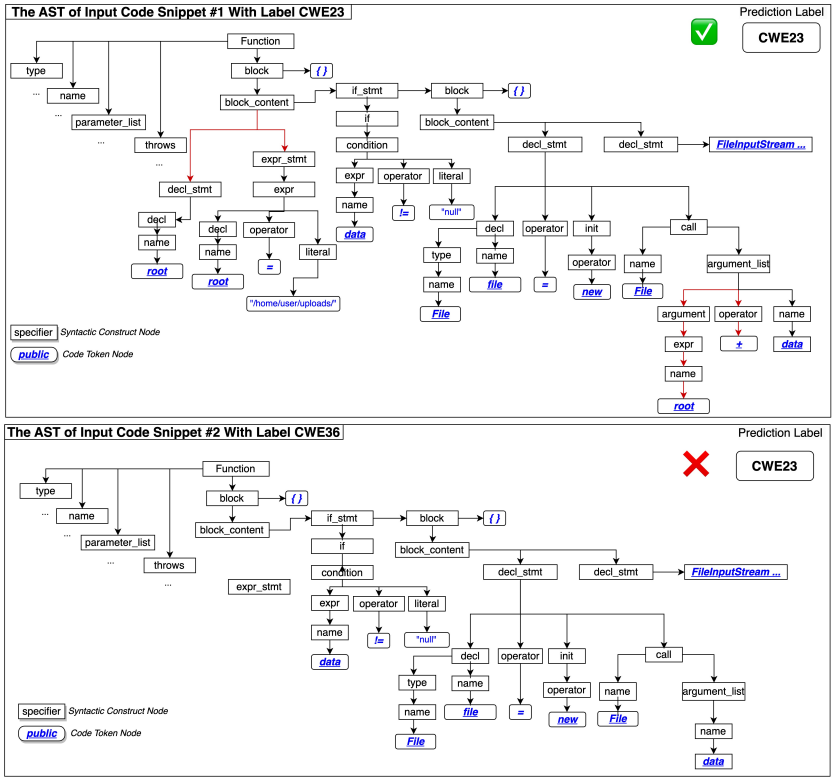



Figure 5.15: The CWE23 code snippet owns two additional AST paths (marked with red) with `argument` and `operator` to make the absolute path into a relative path, compared with CWE36. *Note:* The prediction results are from GraphCodeVec model.

transparency and interpretability in AI-driven software vulnerability detection.

### 5.6.2 Constructs Ranking Examples of Four CWE Sibling Pairs

We summarize four CWE sibling pairs, each representing a distinct CWE category in Table 5.8. This analysis encompasses the exploration of vulnerability differences and their associations with constructs derived from code snippets. Simultaneously, these differences are mirrored in the construct ranking sequences, as revealed by the XAI explanations applied to each code snippet. For detailed code snippets, as well as an in-depth analysis of similarities and differences of these CWE siblings, refer to the Appendix A.2.

Table 5.8: Constructs Ranking for CWE Sibling Pairs

| CWE | Main Difference | Construct Ranking Sequences (Listed in Descending Order of Importance) |
|---|---|---|
| **327** | In the CWE327 case, the hardcoded algorithm is called, resulting in a high ranking for `call`. | `specifier`, `call`, `argument`, `operator`, `throw`, `try`, `throws`, `type`, `break`, `decl_stmt`, ... |
| **328** | The CWE328 case allows for a configurable algorithm; besides `call`, `argument` also ranks high. | `argument`, `specifier`, `expr_stmt`, `call`, `if`, `operator`, `function`, `return`, `init`, `decl`, ... |
| **78** | CWE78 cases use `add()` to append unsanitized input to a command string. | `operator`, `specifier`, `parameter_list`, `throws`, `init`, `type`, `function`, `return`, `call`, `if_stmt`, ... |
| **79** | The CWE79 case directly writes unsanitized input into the HTTP response using `write()`. | `operator`, `specifier`, `throws`, `argument`, `call`, `parameter_list`, `try`, `throw`, `function`, `parameter`, ... |
| **119** | CWE119 cases focus on `if_stmt` and `block_content` to assign a new value to variables without proper memory bounds. | `else`, `block_content`, `if_stmt`, `sizeof`, `goto`, `argument_list`, `operator`, `argument`, `break`, `expr`, ... |
| **120** | CWE120 cases allocate memory without a preceding check, focusing on the `sizeof` constructs. | `else`, `sizeof`, `operator`, `if`, `break`, `default`, `argument`, `argument_list`, `decl`, `block_content`, ... |
| **23** | CWE23 represents a relative path traversal weakness, adding `data` into `root` via the `argument_list` construct. | `name`, `if`, `argument_list`, `finally`, `argument`, `literal`, `throws`, `decl`, `try`, `operator`, ... |
| **36** | CWE36 represents an absolute path traversal weakness. | `name`, `if`, `literal`, `finally`, `decl_stmt`, `argument_list`, `else`, `condition`, `try`, `throws`, ... |

## 5.7 Comparative Analysis of The Findings with Existing Research

This section consolidates the findings and offer a comparative analysis with existing studies. The central objective is the creation of an XAI-based framework capable of evaluating notable elements from four main feature categories for multi-classification in software vulnerability detection. Instead of a straightforward contrast of classification performance, it offers a comparative study using insights from contemporary research, highlighting both areas of agreement and the unique findings. A summary of the discoveries can be found in Table 5.9.

Table 5.9: Relative comparison of the findings with existing work

| **Assessment Type** | **The Findings** | **Existing Work** | **Contributions** |
|---|---|---|---|
| Syntactic constructs importance | Comprehensive importance ranking of nine metadata syntactic constructs and over 40 syntactic constructs across twenty CWEs presented in Figure 5.8; `statement_subelements`, `statement`, `name`, `parameters` are of high rankings across 80% CWEs. | Studies [80] [82] highlight `name` and `statement_subelements` as 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 an MAP score of 0.696 are achieved. | Community provides expert-defined baseline CWE similarity summary [32]. | The CWE similarity is derived from a data-driven and XAI-based retrospective approach. The similarity explanation provides a probing view to understand feature effects on deep learning model's classification for similar CWE types under subtle code variations. |
| Code token length | Classification performance (F1-Score) improves with increased input token length. | Same finding reported in work [99]. | It provides a specific evidence from the software vulnerability task. |
| Code token type | Attention-based models focus on semantically informative tokens and separators; Transformer-based models capture language-specific features. | Observation aligns with work [80, 81] (both claim *separator* and *identifier* are focused by model), supported by a specific case study. | Evaluation conducted using two languages: Java and C/C++. |
| Attention values | Consistency observed between the importance attributed to tokens by attention values and XAI contribution values. However, high attention values may not always reflect on vulnerable code lines. | The community debates whether attention value reflects token importance. Supporting studies [30, 44, 76] use attention values to reflect code token importance, but caution is advised in other studies [31]. | Providing evidence that overall attention values are consistent with token importance from XAI, but caution should be exercised as they may focus on separators and not reflect vulnerable code lines. |

# Chapter 6

# Threats to validity

This research's validity may be affected by several potential obstacles, including threats to internal and external validity.

Threats to internal validity might stem from model evaluation and the choice of datasets for XAI. The XAI evaluation in this study is confined to a single case study — the arXiv scholarly paper ranking — which utilizes forty datasets directly for consistency and stability assessment. Assuming their similarity without a data shifting test might introduce limitations. For software vulnerability detection, this research leverages three datasets: Juliet, OWASP, and Draper. Both Juliet and OWASP are characterized by synthetic samples with artificially constructed annotations, potentially constraining their generalizability to real-world scenarios. While Draper comprises samples from actual source code, it lacks overlapping CWE types with both Juliet and OWASP. This disparity means it does not have cross-validation of top-ranking sequences of syntactic constructs for common CWE types across multiple datasets. Consequently, this study can't further validate XAI explanation consistency across datasets. The explanation consistency metrics, defined in Section 4.2, aim to measure explanations across various datasets.

Regarding external validity, the primary concern is the generalizability of our findings. Although the GraphCodeVec model is adept at managing graph-based feature embeddings, it focuses mainly on AST-based code representation. This scope falls short of providing a holistic view of code graph representation, unlike the study [20] which gleans syntax and semantic features from the Code Property Graph (CPG) by harnessing a complete graph context inclusive of data-flow, control-flow,

AST nodes, and program dependencies. Similarly, this thesis adopts a single model, S2Search, for academic paper ranking, and ImageNet for image classification. While these models stand out in their respective domains, an expanded evaluation encompassing more models would offer a more robust analysis.

This research includes twenty CWE types from three datasets, six of which appear in the "Top 25 Most Dangerous Software Weaknesses" list by the CWE community [121]. However, the absence of balanced data samples for all CWEs and the lack of data labels compatible with deep learning classification models present significant challenges. The findings' transferability might be limited, particularly given that the evaluation is primarily focused on Java (Juliet, Owasp) and C/C++ (Draper) codes. A broader, language-agnostic assessment would enhance the study's external validity.

# Chapter 7

# Conclusion

This study highlights the potential application of eXplainable AI (XAI) methods for software vulnerability detection, particularly in assessing contributing factors. The evaluation of existing XAI methods focuses on scrutinizing their trustworthiness from three perspectives: consistency, stability, and efficiency. Acknowledging the challenge that current XAI methods encounter in achieving a balance among these evaluation points, this research introduces a novel approach named Mean-Centroid PredDiff, devised specifically for this objective.

This thesis first establishes the explanation between program code in a graph context as features and semantics of vulnerability types collectively defined by open community experts. The study begins with defining a feature type taxonomy of code representations, subsequently progressing to analyze syntactic constructs within abstract syntax tree-based graph code representations. It develops an XAI-based framework to explain the relation among the combination of 20 code vulnerability types and over 40 syntactic constructs from three Java and C++ datasets. It is observed that the variation of syntactic construct importance ranking relates to intrinsic similarities amongst certain CWEs that share common characteristics of vulnerability. This work thus derives the CWE similarity based on the XAI explanation summary and validated it by the expert-defined baseline. This work further extends the XAI-based framework to assess three textual factors: code token length, code type, and token attention value, and their influence on the model's predictions.

In summary, this research advances the assessment of explanation consistency, stability, and

efficiency among existing XAI methods and offers guidance in the development of new ones, forming the foundation for the proposed XAI service pipeline. This study provides valuable insights into the diverse impacts of code syntax on CWE vulnerability types, links the comprehension of code semantics and syntactic feature representation learned by deep learning models for vulnerability classification. This knowledge can enhance IDE programming prompts, allowing for the early detection of potentially vulnerable code through prioritized syntactic constructs.

# Appendix A

# My Appendix

## A.1 Evaluating XAI Methods Through Three Case Studies

The previous sections introduces three metrics—explanation consistency, stability, and efficiency—to evaluate the trustworthiness of Explainable AI (XAI). This study also propose a new addition to the XAI feature explanation branch for this evaluation. Explanation consistency reflects the level of agreement among multiple XAI methods when explaining the same dataset and the same model. In contrast, explanation stability represents the level of agreement within the explanations provided by a single XAI method applied to different datasets. The focus of this chapter lies in understanding the ability of these XAI methods to ensure stability and consistency in their explanations and their efficiency in reaching an explanation. Given this focus, this section proposes two research questions (**RQ**) to guide the evaluation of XAI methods:

**RQ-I. How consistent and stable are the explanations generated by different XAI methods across various case studies?**

**RQ-II. How does the Mean-Centroid PredDiff method balance computational efficiency with explanation consistency and stability?**

These two research questions are refer to the first core question presented in Section 1.1. To address research questions, three case studies are selected, each representing a different data structure and machine learning task. The first case study focuses on a regression problem involving tabular data, specifically an academic paper ranking. The second case study explores an NLP (Natural

Language Processing) multi-label classification problem - code vulnerability detection. The final case study examines the use of XAI methods in an image classification problem, which involves a masking-type classification task.

This chapter begins by selecting the XAI methods based on their XAI goals. It then proceeds with evaluation of the three case studies, providing detailed descriptions of the datasets and models used, the experiment settings, observations, results, and conclusions. Each case study contributes to answering the research questions, providing insights into the stability, consistency, and efficiency of various XAI methods.

### A.1.1 Select XAI Method Based on XAI Goal

This work considers the goal of explanation as a criterion to select the most suitable XAI methods, based on the study [55]. For case studies such as academic paper ranking and code vulnerability detection, where the aim is to sort features by their contribution values, it selects XAI methods that explain feature importance through feature masking, namely PredDiff [65], Shapley Value [67], Mean-Centroid PredDiff [98], and SHAP [24]. On the other hand, the Grad-CAM family of XAI methods are dedicated to image explanation via saliency maps. A saliency map in image explanation works by highlighting the areas of the image that a neural network model finds most relevant for making a prediction. In essence, it provides a visual understanding of which parts of the input image were significant in influencing the model's decision, thereby helping to interpret the model's reasoning process. Therefore, this work selects six state-of-the-art model-specific methods from the Grad-CAM family, namely Grad-CAM [122], EigenCAM [59], GradCAMElementWise [60], Grad-CAM++ [61], XGrad-CAM [62], and HiResCAM [63], to compare with the proposed Mean-Centroid PredDiff.

### A.1.2 Case Study I, Academic Paper Ranking

This case study aims to evaluate four XAI methods, namely PredDiff [65], Shapley Value [67], Mean-Centroid PredDiff [98], and SHAP [24], within the context of an academic paper ranking model. The model under examination is the open-source Semantic Scholar Search ranking model (S2Search) [123], which predicts the ranking score for each scholarly article given a query keyword

and various distinctive features. The chosen XAI methods are employed to discern the overall importance order of these features and subsequently assessed based on explanation consistency and stability.

### Model and Dataset

The target model for this evaluation is S2Search [123], an open-source machine learning ranking model developed by Semantic Scholar. This model ranks papers based on user behavior data gathered from search logs and user clicks.

The dataset employed for this study is derived from the arXiv metadata collection available on Kaggle [1], specifically focusing on entries within the field of Computer Science. This dataset encompasses 542,877 academic papers, each assigned one or more meta topics. For the purposes of the study, each paper's secondary categories within the Computer Science field, as provided by arXiv, are utilized as individual datasets, culminating in forty datasets in total.

Each paper within the dataset is characterized by six key features: title, abstract, venue, authors, publication year, and the number of citations (n_citations). These features form a tabular type of data. The objective of the study is to ascertain the importance order of these six features.

### Experiment Setting

In the experimental setup, this study applies each of the selected XAI methods to the S2Search model in conjunction with the arXiv dataset. The performance of these methods is evaluated based on their ability to accurately ascertain the importance order of the six features, paying particular attention to the stability and consistency of explanations each method provides. Furthermore, it also assesses the computational efficiency required by each method to achieve their results. The experiment is executed via the Google Colab Pro [2] machine learning services platform, leveraging the NVIDIA Tesla T4 GPU for computation.

[1] https://www.kaggle.com/datasets/Cornell-University/arxiv
[2] https://colab.research.google.com/

### Experiment Results

**Feature Importance Order Summary.** The explanatory results are obtained through the feature importance order defined in Section 4.1. For the forty datasets, it derives forty sets of feature contribution values for each method. It calculates the mean value of these groups to obtain an overall feature contribution value, which in turn helps us to determine the overall feature importance order as explained by each method. These results are detailed in Table A.1.

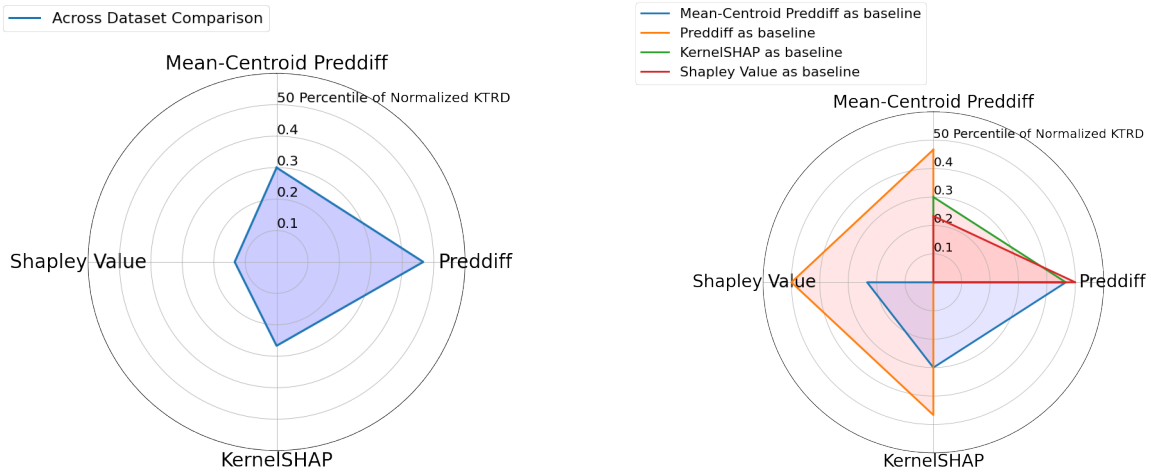
Table A.1: Feature importance order summary of academic paper ranking case study

| Method | Feature Importance Order (With Contribution Value) | | | | | |
|---|---|---|---|---|---|---|
| PredDiff | *abstract(0.938)* | *year(0.306)* | *title(0.127)* | *n_citations(0.073)* | *venue(0.068)* | *authors(0.033)* |
| Mean-Centroid PredDiff | *abstract(0.987)* | *title(0.107)* | *year(0.094)* | *venue(0.066)* | *n_citations(0.022)* | *authors(0.008)* |
| Shapley Value | *abstract(0.960)* | *title(0.219)* | *venue(0.156)* | *year(0.069)* | *n_citations(0.027)* | *authors(0.016)* |
| SHAP | *abstract(0.965)* | *title(0.197)* | *year(0.159)* | *venue(0.065)* | *n_citations(0.027)* | *authors(0.017)* |

**Observe Explanation Stability.** The median contribution values help us to determine the feature importance order of an XAI method across the forty datasets. It computes the KTRD distance between this aggregated feature importance order and the orders from the forty datasets to compare them. The median value of KTRD distances across datasets is depicted in Figure A.1a, indicating that Mean-Centroid PredDiff, Shapley Value, and KernelSHAP offer more stability than Prediff.

**Observe Explanation Consistency.** This study rotational selects a baseline method from the four methods under examination. Then the Kendall Tau Ranking Distance (KTRD) distance is computed between the feature importance order from two XAI methods pair. The 50th percentile of KTRD distances is plotted in Figure A.1b. It shows that Mean-Centroid PredDiff has greater consistency than PredDiff, although less than the other two methods.

**Analysis of Computation Time Consumption** The curve illustrating time consumption, as shown in Figure A.2, ascends with the growth in the number of data samples. PredDiff and Mean-Centroid PredDiff consume less time compared to KernelSHAP and Shapley Value. The Mean-Centroid PredDiff method consumes approximately $10\%$ more time than Prediff due to the need for cluster computation.




(a) Explanation Stability

(b) Explanation Consistency

Figure A.1: The consistency and stability evaluation of four XAI methods in academic paper ranking case study.

*Note:* A shorter link edge indicates a more consistent or stable XAI method.

### A.1.3 Case Study II, Code Vulnerability Classification

This case study focuses on evaluating four XAI methods - PredDiff [65], Mean-Centroid Pred-Diff [98], SHAP [24], and Shapley Value [67] - in the context of a natural language processing (NLP)-based classification problem. Specifically, this case examines software code vulnerability detection based on three features: code comments, code body, and import packages. It use the state-of-the-art NLP model XLNet [47] to classify vulnerable software code into different Common Weakness Enumeration (CWE) types at the method level. The aim is to understand the contribution and importance of each feature to code vulnerability classification using XAI methods and evaluate their performance based on explanation consistency and accuracy of feature importance ranking.

#### Model and Dataset

The datasets used in this study come from the Open Web Application Security Project (OWASP) Benchmark [124], the Juliet test suite [120], and the Draper dataset [125]. These resources provide a robust corpus of method-level software code files, each labeled with CWE types. The datasets comprise the code body, comments, and import packages. For instance, the OWASP Benchmark includes 2,740 test cases, with 52% of the files indicating vulnerable code mapped to one of 11 CWE

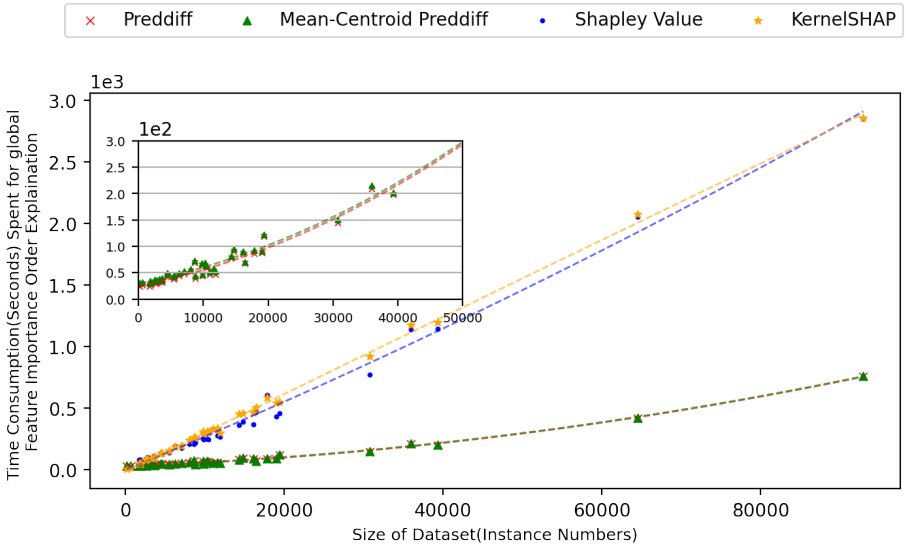



Figure A.2: Time consumption between XAI methods along with the data set size increasing in academic paper ranking case study.

labels. In comparison, the Juliet test suite contains a total of 514 files, 217 of which are vulnerable. While Draper dataset contains 86,839 methods with full code and comments information that with 50.1% vulnerable functions.

The model used for this study is XLNet, a state-of-the-art NLP model renowned for its ability to capture bidirectional text information and outperform other top-tier NLP models. Each piece of text content in a code file that contains a method with a CWE label is considered a data instance and paired with a label. To determine which of these contexts have the most significant impact on the machine learning classifier, it identifies three features within the data: comments, code body, and import packages.

**Experiment Setting**

Each dataset is shuffled and split into a training set and a testing set, using a 70:30 ratio. The training set is used to fine-tune the XLNet model, while the testing set is employed for masking features and gathering predictions for XAI processing. To prevent unintentional data leakage that might artificially boost model accuracy, it removes any direct mentions of CWE types both from code and comment. Moreover, labels accounting for less than ten percent of the total size are

Table A.2: Feature importance order summary for code vulnerability classification case study

| XAI Methods | Juliet | OWASP | Draper |
|---|---|---|---|
| PredDiff | $comment > code > import$ | $code > import > comment$ | $code > comment$ |
| Mean-Centroid PredDiff | $comment > code > import$ | $code > import > comment$ | $code > comment$ |
| Shapley value | $comment > code > import$ | $comment > code > import$ | $code > comment$ |
| SHAP | $comment > code > import$ | $comment > code > import$ | $code > comment$ |

combined. To compute the log-odd probability of the ground truth CWE label, features are removed before being input into the XLNet model. The experiment is conducted using the Google Colab Pro platform.

### Experiment Results

Following a similar approach to the previous case study, it derives the feature importance order for the three data sets, as shown in Table A.2. The Draper dataset lacks the ”import” feature, so it considers only two features for its importance.

**Observe Explanation Stability.** In this case, as only selecting three data sets, the explanation stability metric value for the Shapley Value and SHAP methods is 0, given that their feature importance orders across the three data sets are identical. For PredDiff and Mean-Centroid PredDiff, the value is 0.223. These results suggest that the Shapley Value and SHAP methods exhibit greater stability compared to the PredDiff and Mean-Centroid PredDiff methods.

**Observe Explanation Consistency.** The feature importance order results of the four XAI methods are consistent for the Juliet test cases, with ”comment” being more important than ”code” and ”import”. Each method has a zero Kendall tau distance with the others, indicating perfect agreement. Similar results are found in the Draper dataset.

However, for the OWASP Benchmark dataset, PredDiff and Mean-Centriod PredDiff offer differing insights on features. When it measures by Kendall tau distance, the average feature importance order distances for PredDiff and Mean-Centroid PredDiff from the other two methods are found to be 0.33. This suggests a higher level of consistency achieved by the Shapley Value and SHAP methods.

### A.1.4 Case Study III, Image Classification

The third case study explores the potential applicability of the Mean-Centroid PredDiff method to image classification, specifically focusing on face mask detection. Unlike traditional feature importance methods that rank feature contributions, image explanations concentrate on pixel attribution and saliency maps, offering insight into the active areas of an image that influence model predictions.

Although the Mean-Centroid PredDiff method is model-agnostic, it will be cross-validated with six state-of-the-art model-specific XAI methods that specialize in image models. These methods include Grad-CAM [122], EigenCAM [59], GradCAMElementWise [60], Grad-CAM++ [61], XGrad-CAM [62], and HiResCAM [63]. This case study will showcase how the MCP method, despite its model-agnostic properties, can effectively be used in model-specific scenarios such as image Convolutional Neural Networks (CNN).

#### Applying Mean-Centroid PredDiff to Image Explanation

As depicted in Figure A.3, the MCP method is applied to generate a kernel masking matrix. This matrix is used to iteratively mask pixels in the image by filling in zeros. Consequently, it gets a set of $(l \times l)/(n \times n)$ masked images for model prediction, where the image size is $(l \times l)$, and the kernel masking matrix has a size of $(n \times n)$. The MCP method summarizes pixel feature contributions based on the prediction difference between the original and masked images. In the experiment, it uses an image size $l$ of 256 and a kernel masking matrix size $n$ of 8.

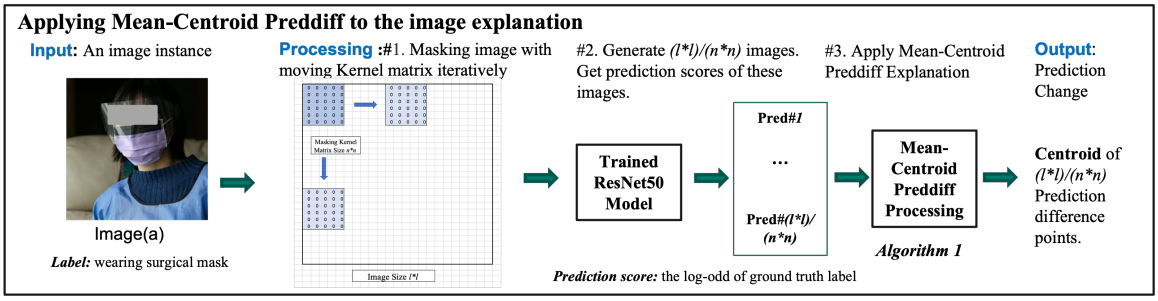



Figure A.3: The process of Mean-Centroid PredDiff on image explanation.

### Model and Dataset

For this case study, it utilizes a pre-trained ResNet50 [126] model for image classification. The data set[3] contains 2,630 images with five different labels, *'wearing N95 mask', 'wearing cloth mask', 'wearing a surgical mask', 'mask worn incorrectly'* and *'no mask'*.

### Experiment Setting

In the experimental setup, the pre-trained ResNet50 model is used to classify images from the selected dataset. Using the Mean-Centroid PredDiff method, it generates a kernel masking matrix enabling iterative pixel masking. This procedure aids us in summarizing pixel feature contributions by comparing prediction differences between original and masked images.

Six Grad-CAM familiy XAI methods are also used to elucidate the saliency map of input images. These saliency map explanations help highlight the active regions within images that significantly contribute to the model's prediction. This approach offers a comparative view of how these methods perform against the Mean-Centroid PredDiff method in a model-specific setting. The experiment is executed by the Google Colab Pro platform as well.

### Experimental Results

**Summary of Prediction Change Aggregation.** In the image classification task, it initially extracts a saliency map using an XAI method. The saliency map indicates the impactful areas of the image contributing to the model's prediction. It denotes the model's prediction on the original image $x_i$ as $\hat{f}_P(x_i)$, and $x_i^+$ as the masked image derived from the saliency map $\rho(x_i)$. The model's prediction on this masked image is $\hat{f}_S(x_i^+)$. Hence, the prediction changes for the data sample $x_i$ is formulated as:

$$\left| \frac{\hat{f}_P(x_i) - \hat{f}_S(x_i^+)}{\hat{f}_P(x_i)} \right| \times 100, \quad \text{where } x_i^+ \leftarrow \rho(x_i). \tag{10}$$

This implies that even after masking the image based on the saliency map, the model should correctly classify the image. Figure A.4 showcases an example of a saliency map generated by the XAI method (Mean-Centroid PredDiff), along with an image masked by this saliency map. In this

[3] https://github.com/youyinnn/ai_face_mask_detection_project.git

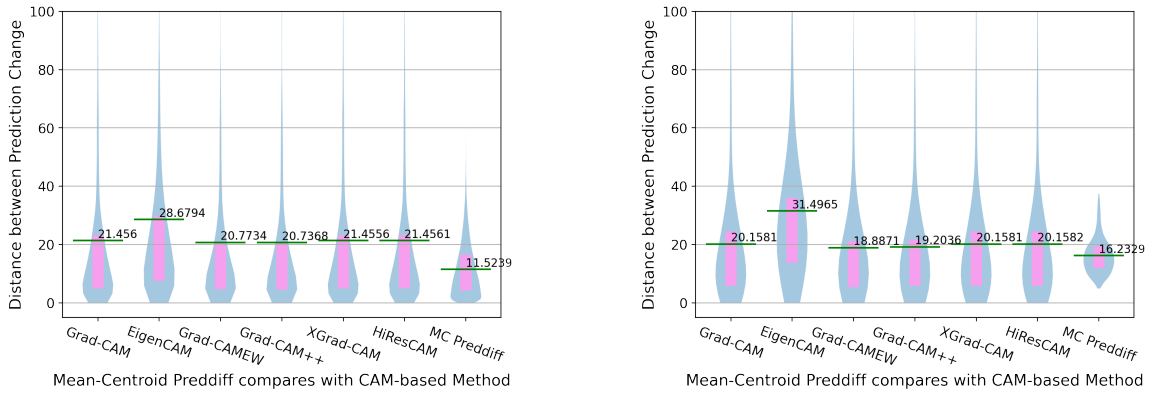
scenario, the change in prediction represents the difference in prediction logits for the ground truth label *'wearing a surgical mask'* between two images: a) the unmasked image, and c) the masked image."


Figure A.4: Example of the original image, saliency map generated by XAI Method (Mean-Centroid PredDiff), and masked image.

**Observing Explanation Stability.** The explanation stability results are summarized by considering each prediction change on a data sample as a single summary. It calculates the distance between any two pairs of these summaries and plot the distribution of these distances in Figure A.5a. It observes that Mean-Centroid PredDiff has the lowest mean value on the stability metric, while EigenCAM is on the opposite end.

**Observing Explanation Consistency.** The distribution of prediction change distances for 2,630 images is compared across different XAI methods in Figure A.5b. EigenCAM has the widest range, suggesting that the explanations of feature contributions by EigenCAM vary significantly across images. On the other hand, the Mean-Centroid PredDiff plot has the narrowest range, indicating consistent explanations across all images.

**Time Complexity Analysis.** Mean-Centroid PredDiff has a time complexity of $\Theta(N \times P^2)$, with $N$ as the number of images and $P$ as the number of features. In this context, an image with a masking kernel matrix is counted as one feature. Therefore, $P = (l \times l)/(n \times n)$ is the number of features. Since Grad-CAM family methods derive the saliency map directly from the model, they are faster than post-processing the features and re-running the model prediction.




(a) Explanation Stability

(b) Explanation Consistency

Figure A.5: The consistency and stability evaluation of four XAI methods in image classification case study.

### A.1.5 Evaluation Conclusion

**Answering RQ-I (How consistent and stable are the explanations generated by different XAI methods across various case studies?).** This experiment examines ten XAI methods, group into image and non-image case studies. Among the feature masking based XAI methods, named SHAP, Shapley Value, PredDiff and Mean-Centroid PredDiff, SHAP achieves the best performance in terms of stability and consistency, followed by Mean-Centroid PredDiff and Shapley Value. In the image classification case study, Mean-Centroid PredDiff achieves the smallest distance of prediction difference compared to methods from the Grad-CAM family. EigenCAM performs the least effectively within the Grad-CAM family.

**Answering RQ-II (How does the Mean-Centroid PredDiff method balance computational efficiency with explanation consistency and stability?).** The examination in the academic paper ranking case study shows that Shapley Value and SHAP are two low-efficiency XAI methods, with efficiency decreasing as the number of data samples increases. Mean-Centroid PredDiff is 10% less efficient than PredDiff, but much quicker than SHAP and Shapley Value, while providing similar stability and closer consistency with SHAP and Shapley Value. Compared with model-specific XAI methods from the Grad-CAM family that directly derive the explanation during the model prediction process, Mean-Centroid PredDiff is less efficient.

In conclusion, Mean-Centroid PredDiff achieves a trade-off of consistency, stability, and efficiency compared to other XAI methods in different case studies. Mean-Centroid PredDiff, SHAP achieves the highest consistency in non-image case studies, while Grad-CAMEW and Grad-CAM++ are the top two consistent and stable methods.

## A.2 Analysis of CWE Sibling Pairs with Constructs Ranking

We summarize another three CWE sibling pairs, each from one CWE category in Table 5.2, analyzing the differences in code snippets, the similarities, and the constructs ranking from XAI explanations for each pair.

**CWE327 and CWE328:**

CWE327 showcases the usage of the Data Encryption Standard (DES) encryption algorithm, which is considered weak and outdated, making the encrypted data susceptible to decryption by attackers. CWE328 case represents the usage of a hashing algorithm (potentially Secure Hash Algorithm (SHA-512), which is strong, but the exact algorithm can be changed based on properties). Both vulnerabilities are centered around cryptographic operations, with one focusing on encryption and the other on hashing.

**Vulnerable code line:**

```
1 // CWE327
2 ...
3 javax.crypto.Cipher c = javax.crypto.Cipher.getInstance("DES/CBC
    /PKCS5Padding"); // Vulnerable code line
4 ...
5
6 // CWE328
7 ...
8 String algorithm = benchmarkprops.getProperty("hashAlg1", "
    SHA512");
```

```
9  java.security.MessageDigest md = java.security.MessageDigest.
       getInstance(algorithm);
10 md.update(input); // Vulnerable code line
11 ...
```

**Construct difference:** Algorithm specification: CWE327 case has a hardcoded algorithm; CWE328 case potentially allows configurable algorithms. From List A.1 we note that the construct *argument* is ranked foremost in the CWE-328 case, highlighting the significance of algorithm configuration in this vulnerability. Additionally, the construct *call* occupies a high rank in both cases, underscoring the prominence of invoking the algorithm's object in both of these vulnerabilities.

```
1 Code snippet #1 with CWE327:
2 {specifier > call > argument > operator > throw > try > throws >
      type > break > decl_stmt >  ...}
3 Code snippet #2 with CWE328:
4 {argument > specifier > expr_stmt > call > if > operator >
     function > return > init > decl > ...}
```

Listing A.1: Ranking of syntactic constructs' feature importance for CWE327 and CWE328.

### CWE78 and CWE79

CWE78 case is concerned with unsanitized input being used in system command execution, potentially allowing malicious command injection. CWE79 case deals with unsensitized input being reflected back to the user, which could lead to Cross-Site Scripting (XSS) attacks. Both vulnerabilities arise from the lack of input sanitization and validation.

**Vulnerable code line:**

```
1 // CWE78
2 ...
3 argList.add("echo" + bar); // Vulnerable code line
4 ...
```

```
5 // CWE79
6 ...
7 response.getWriter().write("Parameter value:" + bar); //
    Vulnerable code line
8 ...
```

**Construct difference:** CWE78 case involves argument constructs where unsanitized input is appended to a command string. CWE79 case involves output constructs where unsanitized input is written directly to the HTTP response. We observe from List A.2 that in both cases, the operator + holds significance, followed by the specifier. However, the primary difference between CWE78 and CWE79 cases arises from the way they handle the *add()* method; one appends via *add()* while the other directly employs *write()* to the response. It is hard to observe these subtle differences from the syntactic constructs.

```
1 Code snippet #1 with CWE79:
2 {operator > specifier > throws > argument > call >
    parameter_list > try > throw > function > parameter > ...}
3 Code snippet #2 with CWE78:
4 {operator > specifier > parameter_list > throws > init > type >
    function > return > call > if_stmt > ...}
```

Listing A.2: Ranking of syntactic constructs' feature importance for CWE79 and CWE78.

**CWE119 and CWE120:**

CWE119 case pertains to the improper restriction of memory buffer operations, while CWE120 case deals with a classic case of buffer overflow due to the absence of a size check. Both vulnerabilities arise from inadequate management or verification of memory buffers.

**Vulnerable code line:**

```
1 // CWE119
2 ...
```

```
3 if (value && (newvariable = ast_var_assign(name, value))) {
    AST_LIST_INSERT_HEAD(headp, newvariable, entries); }
4 ...
5 // CWE120
6 ...
7 iflist->data = malloc(sizeof(*(iflist->data)) * iflist->size);
8 ...
```

```
1 Code snippet #1 with CWE119:
2 {else > block_content > if_stmt > sizeof > goto > argument_list
    > operator > argument > break > expr > ...}
3
4 Code snippet #2 with CWE120:
5 {else > sizeof > operator > if > break > default > argument >
    argument_list > decl > block_content > ...}
```

Listing A.3: Ranking of syntactic constructs' feature importance for CWE119 and CWE120.

**Construct difference:** CWE119 case: assigning a new value to a variable and inserting a new entry in a list without proper memory bounds. CWE120 case: allocating memory based on a particular size without a preceding check to ensure the size of input does not exceed the allocated memory. In the case of CWE119, three constructs are relevant to the areas of vulnerability. `if_stmt`: employing conditional statements is crucial for enforcing boundary checks prior to executing memory operations. `block_content`: capturing critical code segments within blocks is key to ensuring localized effect and facilitating easier error handling. `argument_list`: it is essential to validate arguments passed to functions adequately, especially when these arguments pertain to memory operations. For CWE120, the emphasis shifts to: `sizeof`: the utilization of the `sizeof` operator is instrumental in determining the size of data types and structures, thereby assisting in accurate memory allocation. `if` and `else`: utilization of conditional statements is essential for performing size checks and addressing any discrepancies in a suitable manner. The importance of these constructs as portrayed in the Listing A.3 aptly reflects the differences between CWE119 and CWE120.

# Bibliography


[1] National Institute of Standards and Technology. Vulnerability definition. Computer Security Resource Center. URL https://csrc.nist.gov/glossary/term/vulnerability. Accessed: 2023-03-15.

[2] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2019.

[3] Zhen Li, Deqing Zou, Shouhuai Xu, Xinming Ou, Hai Jin, Suo Wang, Zhongyang Deng, and Yang Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[4] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.

[5] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2010.

[6] Robert Russell, Leanna Kim, Lois Hamilton, Tom Lazovich, John Harer, Onur Ozdemir, Paul Ellingwood, and Mark McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[7] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010.

[8] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10): 1825–1848, 2020.

[9] David A. Wheeler. Flawfinder. https://github.com/david-a-wheeler/flawfinder, 2021.

[10] Checkmarx. Checkmarx software security platform. https://www.checkmarx.com, 2021.

[11] Muhammad Nadeem, Byron J Williams, and Edward B Allen. High false positive detection of security vulnerabilities: a case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 359–360, 2012.

[12] Y Shin, A Meneely, L Williams, and J A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, Nov 2011.

[13] Y Shin and L Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, pages 315–317, Oct 2008.

[14] Y Shin and L Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Softw. Eng.*, 18(1):25–59, Feb 2013.

[15] C D Sestili, W S Snavely, and N M VanHoudnos. Towards security defect prediction with ai. *arXiv preprint arXiv:1808.09897*, 2018.

[16] G Lin, M Tang, Y Wang, W Luo, X Luo, and X Liao. Cross-project transfer representation

learning for vulnerable function discovery. *IEEE Trans. Ind. Informat.*, 14(7):3289–3297, Jul 2018.

[17] J Jiang, S Wen, S Yu, Y Xiang, and W Zhou. Identifying propagation sources in networks: State-of-the-art and comparative studies. *IEEE Commun. Surveys Tuts.*, 19(1), 2017.

[18] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[19] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[20] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022. doi: 10.1109/TSE.2021.3087402.

[21] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2469–2485, 2019.

[22] Peng Zeng, Guanjun Lin, Lei Pan, Yonghang Tai, and Jun Zhang. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, 8:197158–197172, 2020.

[23] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information fusion*, 58:82–115, 2020.

[24] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[25] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " 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*, pages 1135–1144, 2016.

[26] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 364–379, 2018.

[27] Boris Babic, Sara Gerke, Theodoros Evgeniou, and I Glenn Cohen. Beware explanations from ai in health care. *Science*, 373(6552):284–286, 2021.

[28] Anshul Tanwar, Hariharan Manikandan, Krishna Sundaresan, Prasanna Ganesan, Sathish Kumar Chandrasekaran, and Sriram Ravi. Multi-context attention fusion neural network for software vulnerability identification. *arXiv preprint arXiv:2104.09225*, 2021.

[29] Litao Li, Steven HH Ding, Yuan Tian, Benjamin CM Fung, Philippe Charland, Weihan Ou, Leo Song, and Congwei Chen. Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. *ACM Transactions on Privacy and Security*, 2023.

[30] Shikhar Vashishth, Shyam Upadhyay, Gaurav Singh Tomar, and Manaal Faruqui. Attention interpretability across nlp tasks. *arXiv preprint arXiv:1909.11218*, 2019.

[31] Sarthak Jain and Byron C Wallace. 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)*, pages 3543–3556, 2019.

[32] MITRE Corporation. CWE-1000: Research concepts, 2022. URL https://cwe.mitre.org/data/definitions/1000.html. Accessed: 2022-09-01.

[33] Alun Preece, Dan Harborne, Dave Braines, Richard Tomsett, and Supriyo Chakraborty. Stakeholders in explainable ai. *arXiv preprint arXiv:1810.00184*, 2018.

[34] David Gunning. Explainable artificial intelligence (xai): technical report defense advanced research projects agency darpa-baa-16-53. *DARPA, Arlington, USA*, 2016.

[35] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence*, 267:1–38, 2019.

[36] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[37] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[38] L. Developers. Clang, 2019. [Online]. Available: clang.llvm. org.

[39] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 475–485, 2018.

[40] Oliver Ferschke, Iryna Gurevych, and Marc Rittberger. Flawfinder: A modular system for predicting quality flaws in wikipedia. In *CLEF (Online Working Notes/Labs/Workshop)*, pages 1–10, 2012.

[41] Daniel Persson and Dejan Baca. Software security analysis: Managing source code audit, 2004.

[42] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000.

[43] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

[44] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–29, 2019.

[45] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[47] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[48] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[49] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2021.

[50] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[51] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[52] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Chen. Towards learning generalizable code embeddings using task-agnostic graph convolutional networks. *ACM Transactions on Software Engineering and Methodology*, 2022.

[53] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2821–2837, 2021.

[54] MITRE Corporation. The common weakness enumeration community, 2006. URL https://cwe.mitre.org/community/.

[55] Jun Huang, Zerui Wang, Ding Li, and Yan Liu. The analysis and development of an xai process on feature contribution explanation. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 5039–5048, 2022. doi: 10.1109/BigData55660.2022.10020313.

[56] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2014.

[57] Ian Covert, Scott Lundberg, and Su-In Lee. Explaining by removing: A unified framework for model explanation. *Journal of Machine Learning Research*, 22(209):1–90, 2021.

[58] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[59] Mohammed Bany Muhammad and Mohammed Yeasin. Eigen-cam: Class activation map using principal components. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2020.

[60] Jacobgil. Jacobgil/pytorch-grad-cam: Advanced ai explainability for computer vision. support for cnns, vision transformers, classification, object detection, segmentation, image similarity and more. URL https://github.com/jacobgil/pytorch-grad-cam.

[61] Aditya Chattopadhay, Anirban Sarkar, Prantik Howlader, and Vineeth N Balasubramanian. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 839–847. IEEE, 2018.

[62] Ruigang Fu, Qingyong Hu, Xiaohu Dong, Yulan Guo, Yinghui Gao, and Biao Li. Axiom-based grad-cam: Towards accurate visualization and explanation of cnns. *arXiv preprint arXiv:2008.02312*, 2020.

[63] Rachel Lea Draelos and Lawrence Carin. Hirescam: Faithful location representation in visual attention for explainable 3d medical image classification. *arXiv preprint arXiv:2011.08891*, 2020.

[64] Jaspreet Singh, Megha Khosla, Wang Zhenye, and Avishek Anand. Extracting per query valid explanations for blackbox learning-to-rank models. In *Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval*, pages 203–210, 2021.

[65] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.

[66] Stefan Blücher, Johanna Vielhaben, and Nils Strodthoff. Preddiff: Explanations and interactions from conditional expectations. *Artificial Intelligence*, 312:103774, 2022.

[67] Harold William Kuhn and Albert William Tucker. *Contributions to the Theory of Games*, volume 2. Princeton University Press, 1953.

[68] Patrick Schwab and Walter Karlen. Cxplain: Causal explanations for model interpretation under uncertainty. *Advances in Neural Information Processing Systems*, 32, 2019.

[69] Daniel W. Apley and Jingyu Zhu. Visualizing the effects of predictor variables in black box supervised learning models, 2016. URL https://arxiv.org/abs/1612.08468.

[70] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001. doi: 10.1214/aos/1013203451. URL https://doi.org/10.1214/aos/1013203451.

[71] Alex Goldstein, Adam Kapelner, Justin Bleich, and Emil Pitkin. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional. page 23.

[72] Qinghua Lu, Liming Zhu, Jon Whittle, and James Bret Michael. Software engineering for responsible ai. *Computer*, 56(4):13–16, 2023. doi: 10.1109/MC.2023.3242055.

[73] Jianlong Zhou, Amir H Gandomi, Fang Chen, and Andreas Holzinger. Evaluating the quality of machine learning explanations: A survey on methods and metrics. *Electronics*, 10(5):593, 2021.

[74] Todd Kulesza, Simone Stumpf, Margaret Burnett, Sherry Yang, Irwin Kwan, and Weng-Keen Wong. Too much, too little, or just right? ways explanations impact end users' mental models. In *2013 IEEE Symposium on visual languages and human centric computing*, pages 3–10. IEEE, 2013.

[75] Sayantan Polley, Rashmi Raju Koparde, Akshaya Bindu Gowri, Maneendra Perera, and Andreas Nuernberger. Towards trustworthiness in the context of explainable search. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2580–2584, 2021.

[76] M Hariharan, Anshul Tanwar, Krishna Sundaresan, Prasanna Ganesan, Sriram Ravi, R Karthik, et al. Proximal instance aggregator networks for explainable security vulnerability detection. *Future Generation Computer Systems*, 134:303–318, 2022.

[77] Leonhard Applis, Annibale Panichella, and Arie van Deursen. Assessing robustness of ml-based program analysis tools using metamorphic program transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1377–1381. IEEE, 2021.

[78] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135: 106552, 2021.

[79] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022.

[80] Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. An exploratory study on code attention in bert. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 437–448, 2022.

[81] Angelo Sotgiu, Maura Pintor, and Battista Biggio. Explainability-based debugging of machine learning for vulnerability discovery. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–8, 2022.

[82] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *IJCAI*, pages 4665–4671, 2019.

[83] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Yuxing Xun, Guoliang Liu, and Xiang Chen. An empirical study of high-impact factors for machine learning-based vulnerability detection. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 26–34, 2020. doi: 10.1109/IBF50092.2020.9034888.

[84] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

[85] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[86] Ravi Kumar and Sergei Vassilvitskii. Generalized distances between rankings. In *Proceedings of the 19th international conference on World wide web*, pages 571–580, 2010.

[87] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

[88] Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

[89] Charles Jin and Martin Rinard. Evidence of meaning in language models trained on programs. *arXiv preprint arXiv:2305.11169*, 2023.

[90] Junfeng Tian, Wenjing Xing, and Zhen Li. Bvdetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology*, 123: 106289, 2020.

[91] El Habib Boudjema, Sergey Verlan, Lynda Mokdad, and Christèle Faure. Vyper: Vulnerability detection in binary code. *Security and Privacy*, 3(2):e100, 2020.

[92] Sean Heelan and Agustin Gianni. Augmenting vulnerability analysis of binary code. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 199–208, 2012.

[93] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[94] Alexey Svyatkovskiy, Vadim Zaytsev, and Neel Sundaresan. Semantic source code models using identifier embeddings. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 554–565. IEEE, 2019.

[95] Pablo Loyola, Bartosz Matzger, and Gregor Schiele. Import2vec learning embeddings for software libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1106–1108. IEEE, 2019.

[96] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[97] OpenAI. Gpt-4 technical report, 2023.

[98] Ding Li, Yan Liu, Jun Huang, and Zerui Wang. A trustworthy view on explainable artificial intelligence method evaluation. *Computer*, 56(4):50–60, 2023. doi: 10.1109/MC.2022.3233806.

[99] Xue Yuan, Guanjun Lin, Yonghang Tai, and Jun Zhang. Deep neural embedding for software vulnerability discovery: Comparison and optimization. *Security and Communication Networks*, 2022:1–12, 2022.

[100] Mamdouh Alenezi, Mohammed Zagane, and Yasir Javed. Efficient deep features learning for

vulnerability detection using character n-gram embedding. *Jordanian Journal of Computers and Information Technology (JJCIT)*, 7(01), 2021.

[101] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. Survey on software vulnerability analysis method based on machine learning. In *2016 IEEE first international conference on data science in cyberspace (DSC)*, pages 642–647. IEEE, 2016.

[102] Zhou Zhou, Lili Bo, Xiaoxue Wu, Xiaobing Sun, Tao Zhang, Bin Li, Jiale Zhang, and Sicong Cao. Spvf: security property assisted vulnerability fixing via attention-based models. *Empirical Software Engineering*, 27(7):171, 2022.

[103] Junae Kim, David Hubczenko, and Paul Montague. Towards attention based vulnerability discovery using source code representation. In *Artificial Neural Networks and Machine Learning–ICANN 2019: Text and Time Series: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part IV 28*, pages 731–746. Springer, 2019.

[104] Yi Mao, Yun Li, Jiatai Sun, and Yixin Chen. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 4651–4656. IEEE, 2020.

[105] Terence Parr. The definitive antlr 4 reference. [sl]: Pragmatic bookshelf, 2013. *9781934356999. Citado na*, page 22.

[106] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–30, 1970.

[107] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[108] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 178–182, 2022. doi: 10.1145/3510454.3516865.

[109] Han Yan, Senlin Luo, Limin Pan, and Yifei Zhang. Han-bsvd: a hierarchical attention network for binary software vulnerability detection. *Computers & Security*, 108:102286, 2021.

[110] Yan Wang, Peng Jia, Xi Peng, Cheng Huang, and Jiayong Liu. Binvuldet: Detecting vulnerability in binary program via decompiled pseudo code and bilstm-attention. *Computers & Security*, 125:103023, 2023.

[111] Senthil Mani, Anush Sankaran, and Rahul Aralikatte. 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*, pages 171–179, 2019.

[112] Michael L Collard, Michael John Decker, and Jonathan I Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*, pages 516–519. IEEE, 2013.

[113] Shikhar Vashishth, Manik Bhandari, Prateek Yadav, Piyush Rai, Chiranjib Bhattacharyya, and Partha Talukdar. 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*, pages 3308–3318, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1320. URL https://aclanthology.org/P19-1320.

[114] Yahui Chen. Convolutional neural network for sentence classification. Master's thesis, University of Waterloo, 2015.

[115] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[116] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

[117] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[118] Yao Li, Tao Zhang, Xiapu Luo, Haipeng Cai, Sen Fang, and Dawei Yuan. Do pre-trained language models indeed understand software engineering tasks? *arXiv preprint arXiv:2211.10623*, 2022.

[119] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

[120] Software Assurance Reference Dataset (SARD). Juliet test suite for C/C++ and Java. Technical report, National Institute of Standards and Technology (NIST), 2019.

[121] MITRE Corporation. Cwe top 25 list 2023, 2023. URL https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. Accessed: [insert date you accessed the link].

[122] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[123] Allen Institute for AI. Github - allenai/s2search: The semantic scholar search reranker., 2020. URL https://github.com/allenai/s2search.

[124] Jeff Williams and Dave Wichers. The OWASP benchmark project. In *Open Web Application Security Project (OWASP)*, 2019.

[125] Gili Draper, Jonah Buchanan, Brett Hutchinson, Feiyi Liu, and Erica Dietrich. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[126] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.