Software Vulnerability Detection Method Based on Abstract Syntax Tree Feature Migration (AST-FMVD)

Zi-Jun Li , Tao Li * , Hao-Dong Chen , Qin Yu , Meng-qing Qiao , Lin Li

Posted Date: 6 September 2023

doi: 10.20944/preprints202309.0374.v1

Keywords: deep learning; transfer learning; zero-shot; vulnerability detection; abstract syntax tree

Article

Software Vulnerability Detection Method Based on Abstract Syntax Tree Feature Migration (AST-FMVD)

Zi-Jun Li [1],Tao Li [2,*], Hao-Dong Chen [1], Qin Yu [1], Meng-qing Qiao [1] and Lin Li [2]

[1] School of Computer Science and Technology, Wuhan University of Science and Technology, 430065,Wuhan ,China;2731949314@qq.com

[2] Information Processing and Real Time Industrial Systems, Hubei Provincial Key Laboratory, 430065,Wuhan ,China;litaowust@163.com

* Correspondence: litaowust@163.com; Tel.: 13886155981(optional; include country code; if there are multiple corresponding authors, add author initials)

Abstract: In the broad context of vulnerability detection, deep learning has achieved considerable progress but faces generalization challenges in multilingual environments . We introduce a novel approach named AST-FMVD, which leverages transfer learning and abstract syntax trees. By employing semantic similarity clustering and context-aware technology, the method constructs node mapping relationships between different languages, enabling zero-shot learning in vulnerability detection. The method was validated by applying Java's vulnerability detection model in the Python domain, successfully demonstrating that AST-FMVD retains the original model's detection capabilities in the target domain. In conclusion, the proposed method offers a promising solution to the inherent problems in multi-language vulnerability detection, signifying a potential leap in the application of deep learning, transfer learning, and abstract syntax trees for improved cross-domain performance.

Keywords: deep learning;transfer learning;zero-shot ;vulnerability detection;abstract syntax tree

1. Introduction

Traditional vulnerability mining methods have achieved results in small-scale software, but often fail to meet the needs of large, complex software systems and diverse new vulnerabilities[1]. The high complexity of software and the diverse forms of security vulnerabilities pose severe challenges to software vulnerability research. Recently, numerous studies have attempted to apply deep learning[2] to vulnerability mining to achieve automation and intelligent vulnerability mining.

1.1. Problem and Research Background

Building high-performance vulnerability mining models often relies on extensive training sets to enhance model accuracy. Vulnerability detection models proposed by Zhen[3] and others leverage high-capacity samples composed of various vulnerability collections to enhance model precision. However, the quantity of vulnerability samples in different domains is often limited, and traditional machine learning methods do not guarantee accurate performance of models in small sample domains. In the realm of image processing, deep neural networks have demonstrated robust transferability[4–6], facilitating the transfer of pre-existing models across diverse domains. With the evolving comprehension and investigation of transfer learning tasks[7–8], prevalent transfer learning techniques mainly operate across the dimensions of samples, features, models, and relationships, progressively gaining traction in the domain of vulnerability detection. Lutz[9], operating within the knowledge framework of existing vulnerabilities, fine-tuned model parameters during the learning process for novel vulnerabilities, realizing model-level transfer learning in small sample vulnerability domains. Although model-level fine-tuning addresses data sparsity to some extent within small sample vulnerability domains by optimizing model parameters within the target domain's limited sample space, it remains constrained by the reliance of the learning model on sample data and cannot accomplish model transfer under zero-sample conditions. Zaharia[10] and others propose a core

 doi:10.20944/preprints202309.0374.v1

cross-language representation of source code to convert source code into an intermediary form that preserves security vulnerability patterns across different programming languages, reducing dependence on programming language syntax and semantic structure. Drawing inspiration from the application of transfer learning in the security domain, constructing effective feature mapping methods emerges as a pivotal research topic.

In the analysis of code vulnerability information, the corresponding code module usually focuses on specific vulnerability features and abstracts the entire code module to be detected into an abstract representation centered on vulnerability features. Selecting the appropriate code representation structure is a key step in implementing the transfer method. Feng[11] and others found that the code representation method based on the abstract syntax tree (AST) is suitable for vulnerability analysis. Extracting syntax features according to functional levels from the syntax tree can more accurately capture vulnerability information and reduce data redundancy. Focusing on vulnerability analysis, the transformation of the abstract syntax tree between different languages is different from the code language translation work based on the source code level. The transformation of the AST aims to extract key code structures and semantic information[12–13] to discover potential vulnerabilities. Although the converted syntax tree may lack some information, the mapping relationship will retain key vulnerability features and will not affect the model's vulnerability detection. Besides, the abstract syntax features captured by Word2Vec[14–17], ignoring some unimportant semantic differences, can help us ignore some unimportant semantic omissions when dealing with high-dimensional code or AST information.

1.2. Main contributions

Addressing the transfer dilemma of cross-language vulnerability detection models, we introduce a feature-level transfer method named AST-FMVD. During the process of traversing and extracting vulnerability features from syntax trees, the target language is initially parsed into syntax trees, and upon obtaining the Abstract Syntax Tree (AST) of the code, the transfer process is initiated. When constructing preliminary node mapping relationships, keyword information called "tokens" is extracted from the collected code repository, and the semantic similarity between tokens is computed, clustering tokens with similarity together. Based on clustering outcomes, nodes with analogous features are allocated to the same category. Subsequently, by virtue of feature-level semantic similarity, the mapping relationships between corresponding syntax tree nodes are established. In essence, nodes possessing similar features in distinct languages are mapped to identical categories, thereby establishing the mapping relationship between nodes across different languages. Moreover, this method focuses on the structural-level mapping relationship between ASTs in different languages, accompanied by the introduction of context-aware technology. The objective is to mitigate potential ambiguities or vague syntactic structures during the conversion process from source language to target language at the syntax tree level. This method presents a viable solution for zero-sample vulnerability detection; in the absence of annotated samples, the effective transfer of the original model's detection performance from the source domain to the target domain can be accomplished. In experiments involving the mapping of Python's syntax tree to Java, and utilizing the original Java domain's detection model for testing in the target domain, the model's outstanding detection performance in the source domain is effectively demonstrated within the target domain.

2. Related Works

Machine learning methods have offered a broader array of techniques for constructing vulnerability detection models, evolving from initial detection technologies based on expert rules. This evolution has gradually progressed towards predictive models rooted in machine learning and deep learning. In recent years, the introduction of transfer learning techniques has proven effective in mitigating the substantial demand for annotated samples inherent in traditional learning approaches.

 doi:10.20944/preprints202309.0374.v1

2.1. Vulnerability Detection Model

The core of the expert rule-based vulnerability detection model lies in collecting expertise in software security and vulnerability analysis, transforming it into detection rules, and then formulating a vulnerability detection path to construct a vulnerability detection model, known as a vulnerability detection expert system. Typical expert systems include the intrusion detection method proposed by Ilgun in 1995 [18], which uses state transition to detect intrusion. Expert-defined rules are used to simulate the normal behavior of the system, and these rules are transformed into state machines. Any behavior that deviates from normal state transitions is marked as a potential intrusion. The key to an expert system is building an expert rule knowledge base, which requires the manual compilation of a large number of expert rules and incurs high maintenance costs and limited scalability.

With the development of new technologies and the formation and application of vulnerability databases, vulnerability detection models based on statistical analysis and case analysis use data mining techniques to automatically summarize detection rules from code repositories. Lane et al. [19] used sequence matching and learning techniques to detect abnormal behavior. Statistical models like Markov chains are used to learn sequence patterns of normal behavior. This enables the assessment of new behaviors and the detection of malicious attack behaviors. These models employ data mining and statistical analysis techniques to automatically summarize vulnerability detection rules and gather statistical knowledge from software code. This approach is suitable for situations with a large amount of sample data but not for cases with sparse samples of specific vulnerabilities.

Vulnerability detection models based on machine learning formalize the vulnerability detection problem as a classification problem, extracting and selecting features from source code, binary code, or runtime data. Kruegel et al. [20] analyzed binary code to identify and prevent potential buffer overflow attacks. They proposed a machine-learning-based buffer overflow detection method. By utilizing well-constructed sample features for training machine learning classification models, vulnerability detection can be achieved using techniques such as support vector machines and random forests. Shivaji et al. [21] improved defect prediction based on code changes by using decision trees and support vector machines with dimension reduction techniques like principal component analysis. This reduces irrelevant features and enhances model performance.

Neural network technology has gradually been applied to the construction of vulnerability detection models. Vulnerability detection models based on deep learning utilize deep neural networks, such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs), to analyze code structure and semantics for vulnerability detection. Mou et al. [22] used a combination of CNNs and RNNs to capture the structure and patterns of source code, forming the basis for further code analysis and vulnerability detection. Russell et al. [23] explored how to use deep representation learning, especially Long Short-Term Memory (LSTM) networks, to detect vulnerabilities in source code. They demonstrated the effectiveness of utilizing deep representation learning, particularly LSTM, for vulnerability detection in source code. Additionally, compared to traditional rule-based and signature-based vulnerability detection techniques, this approach is more adaptable to new and unknown vulnerability patterns.

While machine learning and deep learning techniques have made progress in vulnerability detection models, the construction of learning models often relies on large-scale labeled training samples. Therefore, in the field of vulnerability mining, the construction of learning models still faces challenges of imbalanced sample spaces and limited sample quantities.

2.2. Transfer learning techniques

Transfer learning is a subfield of machine learning that aims to transfer knowledge learned from a source task to a target task. In the field of vulnerability detection, transfer learning plays a crucial role, especially in cases of imbalanced and limited sample sizes. Given the diversity and evolving nature of vulnerability types, traditional vulnerability detection methods may perform poorly on new or rare vulnerability types. The application of transfer learning can alleviate this issue, primarily operating at four levels: sample, feature, model, and relationship.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

Pan[24] introduced a sample-based transfer learning approach that balances distribution differences between the source and target domains through weighted resampling. This method holds potential for vulnerability detection, as it enables effective transfer of different types of vulnerability samples and provides a foundational overview of transfer learning in general.

Feature-level transfer learning focuses on extracting and selecting useful features for the target task. Long et al. [25] proposed a deep transfer learning method called Joint Adaptation Networks (JAN), which enhances feature alignment by maximizing the distance between the source and target domains. JAN also includes a multi-kernel selection mechanism, allowing adaptation to various domain shifts.

Transfer work can also occur from the perspectives of models or relationships [26]. Model-based transfer learning concerns how to transfer learned models or parameters from the source task to the target task. Common techniques involve fine-tuning parameters of pretrained models, where parameters learned from the source task serve as initial parameters for the target task, followed by further optimization. Relationship-based transfer learning focuses on discovering and utilizing implicit relationships between the source and target tasks. This often involves relationship mapping, finding correspondences between entities in the source and target tasks, or utilizing knowledge graphs to represent and convey relationships between tasks. This is particularly relevant in natural language processing and semantic web domains.

Through practical applications, transfer learning has been shown to significantly reduce the need for manual annotation and improve the prediction accuracy of new vulnerability types. Although transfer learning has demonstrated potential in vulnerability detection, challenges remain, such as matching differences between source and target tasks and the risk of negative transfer.

3. Model Migration Method Based on Abstract Syntax Tree Feature Mapping

As shown in Figure 1, the AST-FMVD framework is introduced. Firstly, the target language is transformed into a representation in the form of a syntax tree. Subsequently, the mapping transformation of syntax tree nodes is conducted, controlled by a predefined mapping rule dictionary. The mapping dictionary records replacement rules for different types of nodes during the cross-language syntax tree feature transformation process. These rules are based on dual mapping of code structure and code semantic information, addressing the challenge of transferring vulnerability features across languages, thus achieving the application of cross-domain knowledge transfer. The AST-FMVD technology framework enables feature transfer across different languages, and the migration process based on the feature level helps retain the minimal features of the source code, with a reduced dependence on programming language vocabulary and semantic structure.

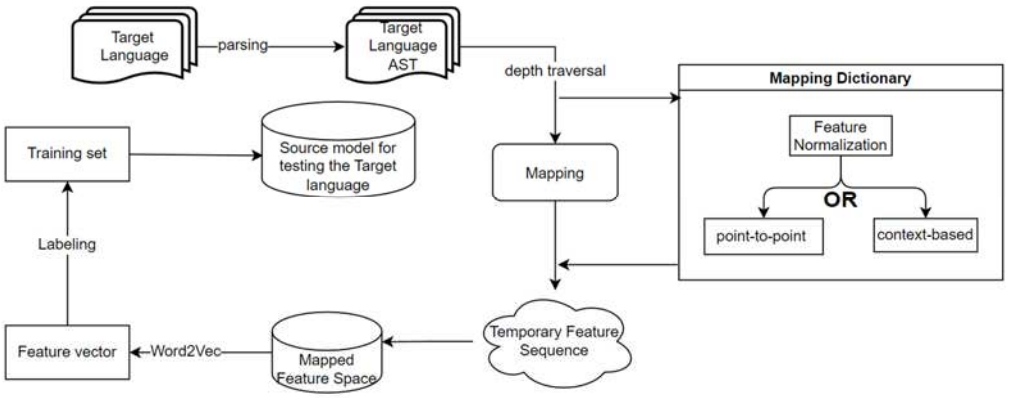

Figure 1. AST-FMVD Framework

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

3.1. Construction of Mapping Rules Dictionary

The construction of a mapping rules dictionary is a crucial step in achieving syntax transformation. This process primarily involves two steps:

- Constructing point-to-point mapping rules involves establishing rules for nodes that do not require context-aware mapping, such as declaration types, operator types, and special character types. For these nodes, it is necessary to determine the semantic similarity between different syntax tree nodes. This is achieved by applying the k-means algorithm to cluster and analyze nodes in different languages based on their similarity. This process results in the creation of point-to-point mapping rules ordered by similarity.
- Constructing point-to-point mapping rules involves establishing rules for nodes that do not require context-aware mapping, such as declaration types, operator types, and special character types. For these nodes, it is necessary to determine the semantic similarity between different syntax tree nodes. This is achieved by applying the k-means algorithm to cluster and analyze nodes in different languages based on their similarity. This process results in the creation of point-to-point mapping rules ordered by similarity.

3.1.1. Construction of Context-Aware Technology

Nodes in the syntax tree, such as variable declarations, operators, and punctuation, have clear semantics and don't require extra contextual information for feature mapping. They can be mapped using preconstructed mapping dictionaries. However, for context-sensitive structure type nodes, such as those representing control flow structures (e.g., if statements, for loops, while loops), considering the surrounding context is essential for accurate mapping.Context-aware syntax mapping using the described technique involves determining the context window size and constructing mappings based on contextual rules. This technique is applied to nodes that abstract code control flow structures, functions, and classes, which constitute the fundamental structure and semantic representation of the code

As illustrated in Figure 2, for nodes requiring context-aware mapping, a relevant code snippet is extracted from the source code as context information, based on the context window size. Following the mapping rules, this extracted context information is then mapped to the corresponding Java nodes. With the help of mapping rules, the specific node information is matched with the corresponding Java syntax tree node based on its semantic and structural attributes. This process achieves mapping for context-sensitive structural nodes. For example, for an If condition statement node in the syntax tree, the context window size is used to determine the number of code lines in the conditional expression and the code lines within the branches (if and else branches). The conditional expression and branch code lines are extracted to form the context information. After completing the structural mapping of the syntax tree, internal node information is subject to single-node mapping. This entails replacing the node names in the feature sequence according to the node replacement rules in the mapping dictionary. The obtained context information is merged with the feature information of the current node, resulting in the final feature representation.

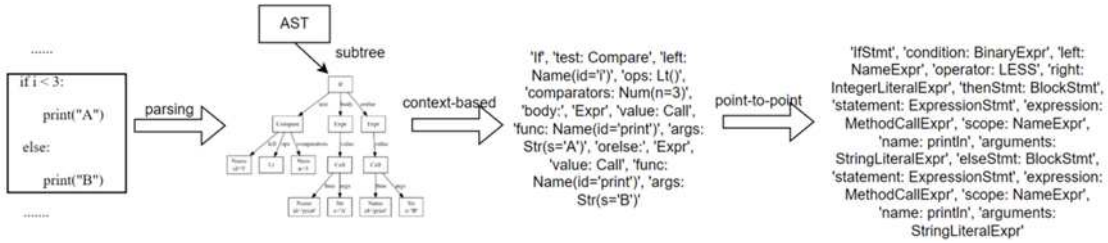

Figure 2. Context-Aware Mapping

As shown in Table 1, context-aware mapping is applied to seven types of syntax tree nodes based on their structural context. These types include if statement nodes, while loop nodes, for loop nodes, switch statement nodes, function nodes, class nodes, and try statement nodes. For these seven syntax

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

types, a custom function is employed to identify and determine code blocks, followed by extending the context window based on the respective code block. Once the context window is established, the specific information of each node requiring context-aware mapping is merged with the extracted context information to create a comprehensive feature representation that combines both code structure and semantics. Since different programming languages share similar representations for these structures, the formulation of these rules is language-agnostic. The dynamic generation of context windows, on the other hand, retains language-specific characteristics, enabling effective noise reduction in the code.

Table 1. Method for Determining Context Window Sizes

Syntax Tree Node Type	Method for Context Window Size Determination
If Statement Node	if branches and else branches, taken as context window.
While Loop Node	Loop condition expression and loop body, taken as context window.
For Loop Node	Loop condition expression and loop body, taken as context window.
Switch Statement Node	Condition expression and individual case branches, taken as context window.
Function Node	Function name, parameter list, and function body, taken as context window.
Class Node	Class name, class member variables, and class member functions, taken as context window.
Try Statement Node	Try block, catch block, and finally block of the Try statement, taken as context window.

3.1.2. Construction of Point-to-Point Mapping Method

The construction of the point-to-point mapping method involves an algorithm based on semantic similarity for node mapping. This algorithm efficiently, accurately, and with strong scalability clusters syntax tree node information using token semantic similarity for node mapping. The process is depicted in Figure 3. Initially, the syntax trees of collected target and source files are serialized to form sequences of tokens from different languages. These tokens are then subjected to Word2Vec vectorization, followed by measuring token similarity using cosine similarity. Subsequently, the k-means algorithm is applied to cluster tokens based on their similarity, grouping similar nodes into the same cluster. This process establishes relationships of semantic similarity among nodes.

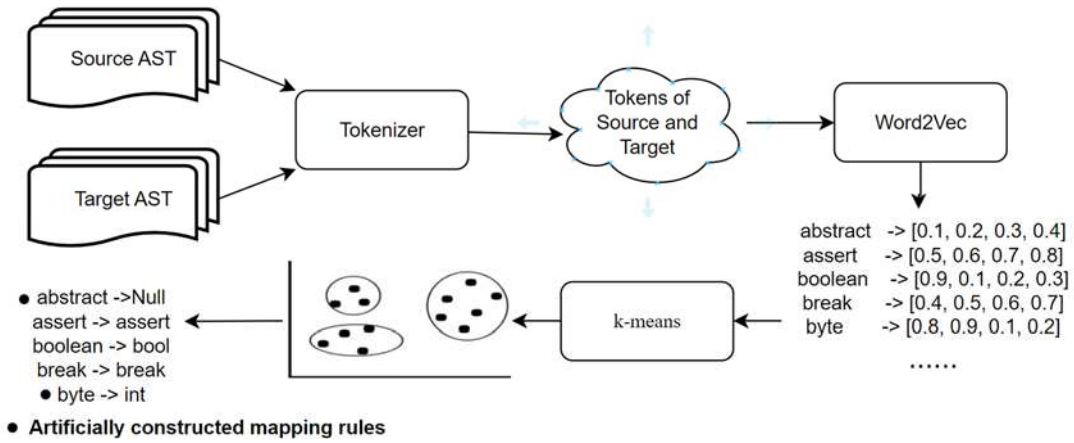

Figure 3. Construction Method Based on Semantic Similarity Mapping

To establish semantic-based mapping relationships between nodes from different programming languages, the initial step involves collecting token information from diverse programming

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

languages. The similarity between tokens corresponds to the naming of syntax tree nodes, ultimately constructing mapping rules within the mapping dictionary. Tokens represent the fundamental constituents of code, as compilers break down code into individual tokens during lexical analysis, followed by subsequent syntax analysis and execution. Tokens are categorized into ten classes based on their lexical attributes.

As depicted in Table 2, identifiers are language-specific and are not suitable for using a general mapping approach. Special characters (such as colons, commas, periods, etc.), numbers, boolean constants, and strings typically represent symbols with fixed meanings that do not involve specific code structures or semantic conversions, and they do not exhibit language variations. Additionally, syntax trees do not include comment information or code formatting details. Therefore, these types of token information retain the characteristics of the original language during the syntax tree mapping process and are not subjected to mapping treatment.

Table 2. Token Types

Token Types	
Identifiers	Keywords
Operators and Separators Literals	Literals
Comments	Strings
Numbers	Boolean Constants
Whitespace and Line Breaks	Special Characters

Furthermore, some common operators, such as addition (+) and comparison operators (e.g., ==, <, >, etc.), are similar across most programming languages. However, for operators that have different representations or functionalities in various programming languages, simple rule additions can be made in the pre-constructed dictionary. For example, the logical AND operator is represented as "&&" in the C language and "and" in Python. This requires adding rules to the dictionary accordingly. In summary, the token types that require discovering mapping rules through semantic similarity are literal types (such as int, bool, String, etc.) and keywords (such as if, else, for, etc.).

As illustrated in Figure 4, AST-FMVD employs the k-means similarity algorithm to cluster tokens based on their similarity scores. Tokens with higher semantic similarity are grouped together within the same cluster, thus establishing relationships of semantic similarity among nodes.Let X denote the set of all tokens, and $S(Xi, Xj)$ represent the cosine similarity between tokens Xi and Xj . The specific steps are outlined as follows:

1.Calculate similarity matrix: For all tokens Xi and Xj , compute the cosine similarity $S(Xi, Xj)$ between them. This can be accomplished using Word2Vec vectorized representations and the cosine similarity formula.

2.Initialize cluster assignments: Initialize a cluster assignmen $C(Xi)$ for each token Xi initially assigning each token to its own cluster.

3.Iterate to update cluster assignments: Iteratively update cluster assignments until convergence. For each token Xi ,perform the following operations based on specific conditions:

- If Xi is an outlier, assign it to the cluster of the other token that has the minimum similarity with Xi ,i.e., $C(Xi) = \underset{Xj \in X, Xj \neq Xi}{argmin} S(Xi, Xj)$;

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

- If the cluster containing Xi already includes more than two tokens from different languages, assign Xi to the cluster of the other token that has the minimum similarity with,i.e., $C(Xi) = \underset{Xj \in X, Xj \neq Xi}{argmin} S(Xi, Xj)$;
- If the cluster containing Xi includes two tokens from different languages and Xi is not an outlier, assign Xi to the cluster of the other token that has the maximum similarity with Xi, i.e., $C(Xi) = \underset{Xj \in X, Xj \neq Xi}{argmax} S(Xi, Xj)$;

4.Check for convergence: After each iteration, check if the cluster assignments have changed. If the assignments remain unchanged, the algorithm converges.

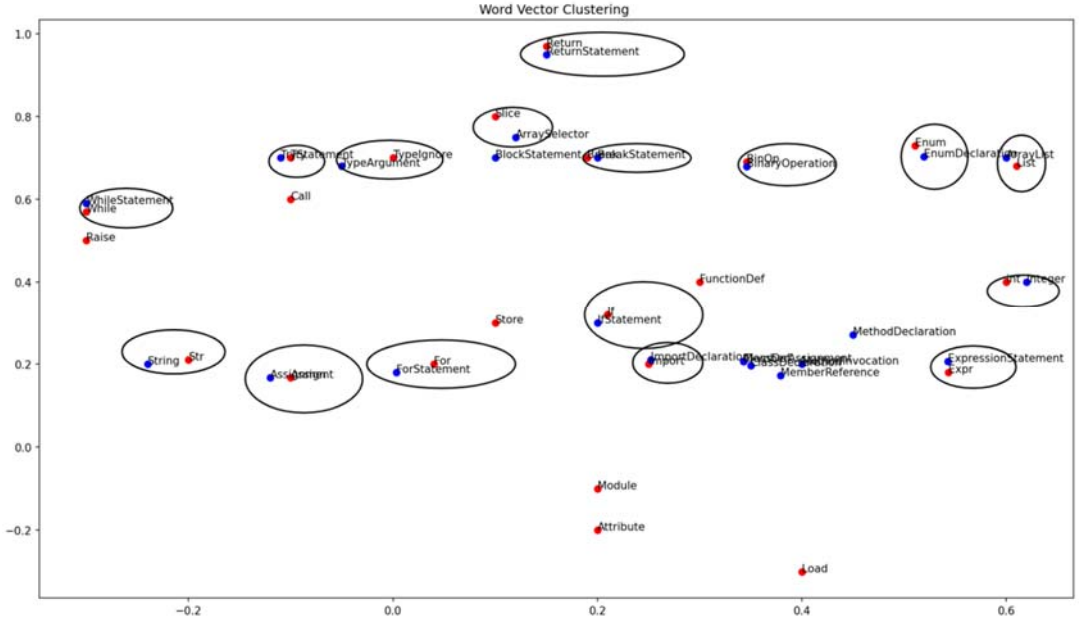

Figure 4. Bi-dimensional view of tokens'word vectors

By performing similarity-based clustering analysis on the syntax tree nodes of the two categories of tokens from different languages, direct mapping relationships among most nodes can be obtained. However, within the clustering structure, there might be outliers and clusters of nodes from the same language. For these types of nodes, we can extract and analyze them separately, and construct mapping rules tailored to their characteristics. The additional node mapping rules are presented in Table 3.

Table 3. Supplementary Mapping Rules

Python Node	Java Node	Mapping Rule
Module	CompilationUnit	Corresponds to Java's CompilationUnit
Raise	ThrowStatement	Corresponds to Java's ThrowStatement
Load	VariableReference or MethodInvocation	Possibly corresponds to Java's Variable Reference or Method Invocation

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

Attribute	MemberReference	Corresponds to Java's Member Reference, used for referring to class members or instance fields and methods
Call	MethodInvocation	Method Invocation, used for calling methods or functions
FunctionDef	MethodDeclaration	Corresponds to Java's Method Declaration, used for defining methods in classes or modules

4. Results

The current experiment utilized Java vulnerability features from the OWASP Benchmark[27]. The primary objective of OWASP Benchmark is to evaluate the capabilities of security testing tools, including accuracy, coverage, scanning speed, and more. It quantifies the scanning abilities of these tools, enabling better comparisons of their strengths and weaknesses. In this experiment, the multi-class dataset from OWASP Benchmark was fed into a Bi-LSTM model for detection.

During the process of collecting vulnerabilities from Python source code containing CWE89 vulnerabilities, a significant amount of Java source code with CWE89 vulnerabilities was already obtained from the previous Java vulnerability detection model. Therefore, this study employed the Java to Python Translator [28], a programming language translation tool, to convert the corresponding vulnerability types in Java source code to Python source code. This approach facilitated the efficient collection of Python files containing SQL injection vulnerabilities (CWE89) in a short period of time. After the Python source code underwent feature mapping processing, the node features were similarly incorporated into the test set.

Table4 outlines the types and quantities of vulnerabilities present in the test set. This methodology allowed for a streamlined and expedited process of collecting Python files with SQL injection vulnerability (CWE89) features, enhancing the efficiency of the experiment.

Table 4. The number of different types of vulnerabilities in the Benchmark Java dataset

CWE	Quantity	CWE	Quantity
0	1213	89	249
79	246	330	218
22	133	327	130
328	129	78	126
501	83	614	36
90	27	643	15

In the experiment evaluating the impact of different mapping depths on vulnerability detection, the positive samples consisted of 249 Python files without vulnerabilities, while the negative samples comprised 249 Python files containing SQL injection vulnerabilities. These samples were utilized as the test set for conducting the experiment.

4.1. Contrasting and Validating the Experiment

Performing syntax tree feature vector detection requires the conversion of each sample's corresponding features into fixed-dimensional numerical vectors. The objective is to map features represented as text into a vector space, thereby transforming them into a format suitable for deep learning models. During the process of embedding the target features, the previously constructed

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

vocabulary from Java training is leveraged. This facilitates the transition from a high-dimensional space to a lower-dimensional continuous vector space. Ultimately, these transformed feature vectors are fed into a pre-trained Java detection model.

4.1.1. Comparison of Few-Shot migration experiments based on the model level

Lutz et al. [12] conducted a study on pre-trained learning models, aiming to fine-tune model parameters during the learning process of new vulnerability knowledge while retaining expertise in existing vulnerabilities. This approach achieved model-level transfer learning in the realm of small-sample vulnerabilities. Expanding upon an existing multi-output detection model, the study extended the model's capabilities to detect new vulnerability types. The effectiveness of the model's detection performance in this context is presented in Table 5.

Table 5. Performance of Lutz Transfer Method in Different Vulnerability Spaces

Metrics	**Initial vulnerability types**						**New Vuln**	
CWE Type	cl.1	cl.2	cl.3	cl.4	cl.5	cl.6	cl.7	cl.8
Recall	0.99	0.97	0.99	0.99	1.00	0.98	0.95	0.98
F1 score	0.88	0.96	0.91	0.85	0.99	0.88	0.90	0.90
FPR	0.93	0.96	0.95	0.91	0.99	0.93	0.92	0.93
FNR	0.00	0.01	0.00	0.00	0.00	0.00	0.02	0.00
Precision	0.12	0.06	0.10	0.15	0.01	0.12	0.10	0.10

The labels cl.1-6 represent specific vulnerability types before the transfer, while cl.7-8 denote new vulnerability types detected by the model after the transfer. The detection outcomes for the new vulnerability types are presented in the last two columns of Table 5. Notably, F1 scores of 92% and 93%, along with precision values of 95% and 98%, were achieved for the two new vulnerabilities cl.7 and cl.8, respectively. This achievement underscores the model's ability to expand its capabilities through the combination of small-sample training and model fine-tuning, successfully incorporating two additional vulnerability types beyond the original six.

An analysis of the experimental results reveals that the average detection effectiveness of the original model was 98.6%, with an F1 score of 94.5% and a recall rate of 91.2%. After the transfer, the average accuracy and F1 scores for the two new vulnerability types were 96.5% and 92.5%, respectively, with a recall rate of 0.90%. Through comparative analysis, it becomes evident that the model's performance in detecting the newly expanded vulnerabilities has exhibited a marginal decline of 2.1% in accuracy, 2.1% in F1 score, and 1.3% in recall, in comparison to its proficiency in detecting the original vulnerabilities.

In this experiment, the focus was on the eleven original vulnerability types present in the Java programming language (22, 327, 328, 330, 501, 614, 643, 78, 79, 89, 90), which were denoted as cl.1-11. The AST-FMVD model transfer method was employed, where each vulnerability category was treated as both positive and negative instances. Additionally, a new language type (Python files containing CWE89 vulnerabilities) was labeled as cl.10'.To address the imbalance in the sample distribution, the dataset underwent oversampling techniques. This process ensured that each vulnerability class had an equitable representation within the dataset. Subsequently, performance metrics were calculated for each label, and the resulting averages were used as the final indicators to assess the effectiveness of the transfer approach.The experimental outcomes are presented in Table 6.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

Table 6. Transfer Performance of AST-FMVD across Different Language Spaces

Metrics	**Initial vulnerability types**					
CWE Type	cl.1	cl.2	cl.3	cl.4	cl.5	cl.6
Recall	0.91	0.97	0.96	0.99	0.97	0.94
F1 score	0.91	0.90	0.94	0.95	0.91	0.86
FPR	0.91	0.93	0.95	0.93	0.95	0.84
FNR	0.09	0.10	0.06	0.01	0.03	0.06
Precision	0.04	0.04	0.03	0.14	0.05	0.13
Metrics	**Initial vulnerability types**					**New Lang**
CWE Type	cl.7	cl.8	cl.9	cl.10	cl.11	cl.10′
Recall	0.98	0.85	0.92	0.90	0.83	0.88
F1 score	0.86	0.82	0.91	0.85	0.67	0.85
FPR	0.89	0.84	0.91	0.89	0.74	0.86
FNR	0.10	0.00	0.04	0.01	0.01	0.00
Precision	0.20	0.16	0.14	0.33	0.11	0.15

The results reveal that the average detection effectiveness for CWE89 vulnerability type code in the Java programming language is 90%, with an F1 score of 89% and a recall rate of 85%. Following the transfer to the Python programming language and targeting CWE89-type files, the average detection effectiveness remains at 88.5%, accompanied by an F1 score of 86.7% and a recall rate of 85%. Notably, the recall rate remains relatively consistent, while accuracy and F1 score experience a decline of 1.5% and 2.3% respectively.When compared to the effectiveness of Lutz et al.'s transfer learning approach, AST-FMVD exhibits a comparable proportionate decrease in accuracy and F1 score. Furthermore, the loss in recall rate is smaller than that of Lutz's proposed transfer learning method, as illustrated in Figure 5.

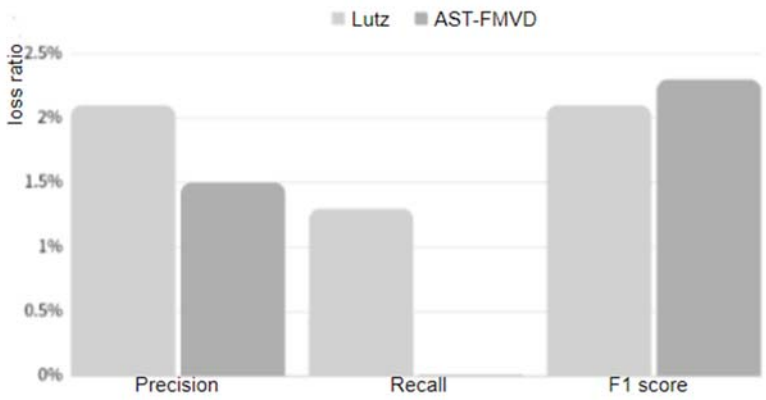

Figure 5. Loss comparison between Lutz and AST-FMVD

4.1.2. Comparison of Zero-Shot Transfer Experiments on the Feature Level

Zaharia once proposed CLaSCoRe (a cross-language representation method for source code) for implementing transfer applications of deep learning-based software vulnerability detection [10]. It's designed to convert source code into an intermediate representation, preserving vulnerability information across different programming languages, thereby reducing the dependency on the lexical and semantic structure of programming languages. Using SVM as the vulnerability detection model, it was trained on datasets in C, C++, and Java languages. Ultimately, the feature transfer constructed by CLaSCoRe [10] was used to detect security vulnerabilities in code written in C#.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 September 2023 doi:10.20944/preprints202309.0374.v1

During this process, C# code snippets were not used in training. When other languages' vulnerability detection models were transferred to C# using CLaSCoRe, the average accuracy was 0.89, with an SVM result of an average recall rate of 0.80. For native C# language model detection, the average accuracy was 0.97. From the model's recall perspective, using the CLaSCoRe-based representation and training on the C# dataset, 95% of the vulnerabilities in the C# dataset could be identified. However, the recognition rate of the detection model using existing models from other languages via feature transfer dropped to 80%. Although CLaSCoRe had a high accuracy of 0.97 in native C# language model detection, its average accuracy dropped to 0.89 when transferred from other languages, with a poor recall performance, indicating that the model's adaptability and universality are lacking when faced with different programming languages. In terms of average recall, the recall rate dropped significantly after using the CLaSCoRe method, from the original 0.90 to 0.72. In contrast, when AST-FMVD underwent cross-language transfer, the average detection performance for CWE89 type files only slightly decreased to 88.5%, with an F1 score of 86.7% and a recall rate maintained at 85%. The comparative experimental results are shown in Table 7. The results prove that AST-FMVD has excellent transferability, maintaining high detection performance even in different programming language environments.

Table 7. Comparison of Software Vulnerability Detection Results between CLaSCoRe and AST-FMVD

Metrics	**CLaSCoRe**		**AST-FMVD**	
	Results	Loss Ratio	Results	Loss Ratio
Recall	From 0.90 to 0.72	20%	From 0.85 to 0.85	0.0%
F1 score	From 0.89 to 0. 72	19%	From 0.89 to 0.86	2.3%
Precision	From 0.89 to 0.80	10%	From 0.89 to 0.88	1%
Transfer from (Languages)	C, C++, Java		Java,Python	

While CLaSCoRe exhibits high accuracy in the context of C# language model detection, its average accuracy diminishes to 0.89 when vulnerability detection models from other languages are transferred to C#. Moreover, it demonstrates poorer recall performance, indicating limited adaptability and generality when confronting diverse programming language environments.

In contrast, when employing AST-FMVD for cross-language transfer, the average detection effectiveness for CWE89-type files experiences only a marginal reduction to 88.5%, maintaining an F1 score of 86.7% and a recall rate of 85%. These results underscore the impressive transfer capabilities of AST-FMVD, which sustains high detection performance across different programming language environments.

5. Conclusions

we propose a zero-shot vulnerability detection method based on abstract syntax tree feature space mapping. This method addresses the issue of detecting CWE89-type vulnerabilities in files written in different programming languages. We achieve this by performing feature transfer and mapping while emphasizing the extraction and preservation of specific vulnerability types during the mapping process, thereby avoiding unnecessary information interference. By maximizing the retention of the original vulnerability features in the code, we maintain the detection effectiveness of the original model in its native domain. Additionally, our method leverages existing pre-trained models, reducing the time and resource costs associated with training models from scratch, rendering it practical and feasible.

Despite the merits of this zero-shot vulnerability detection method, some limitations are also apparent. Firstly, the generalization performance of the method might be limited, as it relies on the extraction and preservation of features specific to certain vulnerability types. Consequently, its effectiveness could be compromised when detecting other vulnerability types, necessitating further

refinement and optimization of the feature mapping strategy to enhance the method's adaptability and generalization performance. Furthermore, as a static code analysis method, it is unable to achieve vulnerability localization, meaning it cannot pinpoint the exact location of vulnerabilities within the source code. Addressing this limitation may require the integration of other techniques for more precise vulnerability localization and remediation.

Author Contributions: Conceptualization, Z.L., T.L. and L.L.; data curation, Z.L. and H.C.; formal analysis, Z.L.; investigation, Z.L. and T.L.; methodology, Q.Y., Z.L. and M.Q.; project administration, Z.L.; resources, Z.L.,Q.Y. and T.L; software,Z.L.; supervision, Z.L. and T.L.;validation, T.L., Z.L. and L.L.; visualization,Z.L.; writing—original draft, Z.L.; writing—reviewand editing, Z.L., T.L. and H.C. All authors have read and agreed to the published version of the manuscript.

Funding: The research was funded by the Wuhan Municipal Key R&D Program under grant number 2022012202015070, and the APC was funded by the Wuhan University of Science and Technology Graduate Teaching Reform Research Project (Yjg202111).

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: Not applicable

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

References

1. Li Yun, Huang Chenlin, Wang Zhongfeng, et al.Survey of software vulnerability mining methods based on machine learning[J]. Journal of Software, 2020, 31(7): 2040-2061.
2. Gu Mianxue, Sun Hongyu, Han Dan, et al. Software Security Vulnerability Mining Based on Deep Learning [J]. Journal of Computer Research and Development, 2021, 58(10):23. DOI:10.7544/issn1000-1239.2021.20210620.
3. Zhen L , Zou D , Xu S , et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[C]// Network and Distributed System Security Symposium. 2018.
4. Yosinski J, Clune J, Bengio Y, et al. How transferable are features in deep neural networks?[J]. MIT Press, 2014. DOI:10.48550/arXiv.1411.1792.
5. He K , Zhang X , Ren S ,et al.Deep Residual Learning for Image Recognition[J].IEEE, 2016.DOI:10.1109/CVPR.2016.90.
6. Soysal O A , Guzel M S .An Introduction to Zero-Shot Learning: An Essential Review[C]//2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA).2020.DOI:10.1109/HORA49412.2020.9152859.
7. Zhuang Fuzhen, Key Laboratory of Intelligent Information Processing, Chinese Academy of Sciences, Zhuang Fuzhen, et al. Research Progress on Transfer Learning [J]. Journal of Software, 2015, 26(1):14.
8. Yang Qiang, Tong Yongxin. Transfer Learning: Review and Progress [J]. Communications of the China Computer Federation, 2018, 014(009):36-41.
9. Lutz O , Chen H , Fereidooni H , et al. ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning[J]. 2021.
10. Zaharia S, Rebedea T, Trausan-Matu S. Detection of Software Security Weaknesses Using Cross-Language Source Code Representation (CLaSCoRe)[J]. Applied Sciences, 2023, 13(13): 7871.
11. Feng H , Fu X , Sun H , et al. Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning[C]// IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2020.
12. Wang Y S , Lee H Y , Chen Y N .Tree Transformer: Integrating Tree Structures into Self-Attention[J]. 2019.DOI:10.18653/v1/D19-1098.
13. Baars A I , Swierstra S D , Viera M .Typed transformations of typed abstract syntax[C]//Tldi09: Acm Sigplan International Workshop on Types in Languages Design & Implementation.ACM, 2009.DOI:10.1145/1481861.1481865.
14. Mikolov T,Chen K,Corrado G,et al.Efficient Estimation of Word Representations in Vector Space[J].ComputerScience,2013.DOI:10.48550/arXiv.1301.3781.

15. Mikolov T , Sutskever I , Chen K ,et al.Distributed Representations of Words and Phrases and their Compositionality.2013[2023-06-10].DOI:10.48550/arXiv.1310.4546.
16. Huang E , Socher R , Manning C ,et al.Improving word representations via global context and multiple word prototypes[C]//Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1.Association for Computational Linguistics, 2012.
17. Biesialska M , Rafieian B ,Marta R. Costa-jussà.Enhancing Word Embeddings with Knowledge Extracted from Lexical Resources[J]. 2020.DOI:10.18653/v1/2020.acl-sr
18. Ilgun K, Kemmerer R A, Porras P A. State transition analysis: A rule-based intrusion detection approach[J]. IEEE transactions on software engineering, 1995, 21(3): 181-199.
19. Lane T, Brodley C E. Sequence matching and learning in anomaly detection for computer security[C]//AAAI Workshop: AI Approaches to Fraud Detection and Risk Management. 1997: 43-49.
20. Kruegel C, Kirda E, Mutz D, et al. Automating mimicry attacks using static binary analysis[C]//USENIX Security Symposium. 2005, 14: 11-11.
21. Shivaji S, Whitehead E J, Akella R, et al. Reducing features to improve code change-based bug prediction[J]. IEEE Transactions on Software Engineering, 2012, 39(4): 552-569.
22. Mou L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing[C]//Proceedings of the AAAI conference on artificial intelligence. 2016, 30(1).
23. Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning[C]//2018 17th IEEE international conference on machine learning and applications (ICMLA). IEEE, 2018: 757-762.
24. Pan S J, Yang Q. A survey on transfer learning[J]. IEEE Transactions on knowledge and data engineering, 2009, 22(10): 1345-1359.
25. Long M, Zhu H, Wang J, et al. Deep Transfer Learning with Joint Adaptation Networks[J]. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018: 222-231.
26. Nickel M, Rosasco L, Poggio T. Holographic embeddings of knowledge graphs[C]//Proceedings of the AAAI conference on artificial intelligence. 2016, 30(1).
27. Sarpong P A , Larbi L S , Paa D P , et al. Performance Evaluation of Open Source Web Application Vulnerability Scanners based on OWASP Benchmark[J]. International Journal of Computer Applications, 2021, 174(18):15-22.
28. Coco E J , Osman H A , Osman N . JPT : A Simple Java-Python Translator[J]. 2018.