computers

MDPI

Article

MultiGLICE: Combining Graph Neural Networks and Program Slicing for Multiclass Software Vulnerability Detection †

Wesley de Kraker [1], Harald Vranken [1,2,*] and Arjen Hommersom [1,2]

[1] Department of Computer Science, Open Universiteit, 6419 AT Heerlen, The Netherlands
[2] Institute for Computing and Information Sciences, Radboud University, 6525 EC Nijmegen, The Netherlands
* Correspondence: harald.vranken@ou.nl
† This paper is an extended version of our paper published in Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Delft, The Netherlands, 3–7 July 2023, entitled GLICE: Combining Graph Neural Networks and Program Slicing to Improve Software Vulnerability Detection, by Wesley de Kraker, Harald Vranken, and Arjen Hommmersom; https://doi.org/10.1109/EuroSPW59978.2023.00009.

Abstract: This paper presents MultiGLICE (Multi class Graph Neural Network with Program Slice), a model for static code analysis to detect security vulnerabilities. MultiGLICE extends our previous GLICE model with multiclass detection for a large number of vulnerabilities across multiple programming languages. It builds upon the earlier SySeVR and FUNDED models and uniquely integrates inter-procedural program slicing with a graph neural network. Users can configure the depth of the inter-procedural analysis, which allows a trade-off between the detection performance and computational efficiency. Increasing the depth of the inter-procedural analysis improves the detection performance, at the cost of computational efficiency. We conduct experiments with MultiGLICE for the multiclass detection of 38 different CWE types in C/C++, C#, Java, and PHP code. We evaluate the trade-offs in the depth of the inter-procedural analysis and compare its vulnerability detection performance and resource usage with those of prior models. Our experimental results show that MultiGLICE improves the weighted F1-score by about 23% when compared to the FUNDED model adapted for multiclass classification. Furthermore, MultiGLICE offers a significant improvement in computational efficiency. The time required to train the MultiGLICE model is approximately 17 times less than that of FUNDED.

Keywords: code analysis; vulnerability detection; graph neural networks; program slicing; machine learning

Academic Editor: Paolo Bellavista

Received: 10 December 2024
Revised: 23 January 2025
Accepted: 28 February 2025
Published: 8 March 2025

Citation: de Kraker, W.; Vranken, H.; Hommersom, A. MultiGLICE: Combining Graph Neural Networks and Program Slicing for Multiclass Software Vulnerability Detection. *Computers* **2025**, *14*, 98. https://doi.org/10.3390/computers14030098

1. Introduction

Automated vulnerability detection by means of static code analysis has advanced significantly over the past two decades. Static code analysis implies that the source code (or bytecode) of a program is examined without executing it, whereas dynamic analysis entails running the native code and analyzing its behavior at runtime. Numerous commercial and open-source tools are available for automated vulnerability detection. Both static and dynamic analysis are applied in software development life cycle (SDLC) processes, where they are commonly known as static analysis security testing (SAST) and dynamic analysis security testing (DAST). According to the BSIMM 14 report, which evaluated the software security practices of 130 organizations in 2023, 86% of participants used automated code review tools, with mandatory code review for all projects identified as one of the top growing activities [1].

State-of-the-art tools for static code analysis can examine vast code repositories containing millions of lines of code. These tools generally depend on rule sets that define the traits of vulnerabilities. Modern query-based SAST tools, such as CodeQL, parse software code into representations that can be stored in a database [2], and the vulnerability analysis of the code is performed by searching the database using SQL-like queries that codify vulnerability knowledge [3]. These tools can only identify vulnerabilities covered by their rule sets, making them inherently limited. Additionally, they must balance false positives (where code is mistakenly flagged as vulnerable) against false negatives (where actual vulnerabilities go undetected). The resources that they require, such as memory and runtime, vary significantly based on the precision of their analysis. This analysis can be global and inter-procedural or local and intra-procedural, depending on the size of the codebase being analyzed [4]. At one end of the spectrum, some tools perform local analysis in just seconds, making them ideal for quick checks when committing code into a repository. On the other end, there are global analysis tools that run overnight to provide more comprehensive insights. Minimizing the code scanning time and reducing the manual effort in reviewing false positives are key priorities for DevSecOps, since the aim is to reduce the time between committing code modifications in the development environment and moving these to the production environment [5].

In recent years, many studies have sought to enhance static code analysis by leveraging artificial intelligence, particularly machine learning and deep learning, to identify high-quality code patterns associated with the presence or absence of vulnerabilities (e.g., [6–11]). A key advantage of deep learning is that it eliminates the need for manual feature engineering to identify relevant properties of source code. However, a major research challenge remains: how to transform source code into a model suitable for a learning algorithm while preserving the syntactic and semantic properties of the code relevant to vulnerabilities. Some studies have experimented with using sequences of tokens produced by code parsers [11]. Better results have been achieved in earlier work by Zhou et al. [8] and more recent research by Wang et al. [11] through the application of graph neural network (GNN) models [12]. These models integrate abstract syntax trees, control flow graphs, and data flow graphs to capture program syntax and semantics. However, both studies focus on function-level analysis, limiting their ability to detect vulnerabilities arising from interactions between multiple functions.

In our previous work, we developed the GLICE model, which combines inter-procedural program slicing and a GNN model [13]. GLICE applies inter-procedural program slicing to extract relevant code parts that can span multiple functions, and these code parts are subsequently analyzed using a GNN model. GLICE allows the detection of vulnerabilities caused by the interaction of multiple functions. GLICE is inspired by and builds upon the prior work by Li et al. [7,10] and Wang et al. [11]. Li et al. introduced the VulDeePecker model [7] and the SySeVR model [10], which apply program slicing with deep learning using BiLSTM models for vulnerability detection. Wang et al. [11] introduced the FUNDED model, which applies a GNN model for vulnerability detection at the function level. Our GLICE model combines inter-procedural program slicing and an improved FUNDED model. In our previous work, we applied GLICE for the binary detection of out-of-bounds write (CWE-787) and out-of-bounds read (CWE-125) vulnerabilities, which typically occur in C/C++ program code [13].

In the present paper, we introduce MultiGLICE, which is an extended version of our GLICE model. While GLICE is limited to binary classification, identifying whether code contains a vulnerability or not, MultiGLICE applies multiclass classification to also determine the specific CWE type of a detected vulnerability. Furthermore, while, in our previous work, we applied GLICE only for two types of vulnerabilities in C/C++ program

code, in the present paper, we apply MultiGLICE for 38 types of vulnerabilities in C/C++, C#, Java, and PHP program code.

Table 1 summarizes the relations and differences between the original FUNDED model, our improved FUNDED model, our GLICE model, and our MultiGLICE model.

Table 1. Summary of models.

Model	Details
Original FUNDED	The model introduced by Wang et al. [11], which applies a GNN model for vulnerability detection in single functions.
Improved FUNDED	Our improved version of the original FUNDED model, in which we resolved bugs and improved the embedding strategy [13].
GLICE	Our previous model that combines the improved FUNDED model with inter-procedural program slicing. We trained and evaluated GLICE for two types of vulnerabilities in C/C++ program code [13].
MultiGLICE	Our present model that extends GLICE with multiclass detection. We train and evaluate MultiGLICE for 38 types of vulnerabilities in C/C++, C#, Java, and PHP program code.

The contributions of this paper are as follows.

- We introduce MultiGLICE, an extended version of the GLICE model, which combines inter-procedural program slicing and a GNN model for the detection of a broad range of software vulnerabilities (38 CWE types). MultiGLICE does not only detect the presence of a vulnerability, but also identifies its specific CWE type, enhancing the usability of the model for developers and security researchers. The source code of MultiGLICE is available as open-source software at https://github.com/wesleydekraker/glice (accessed on 27 February 2025).
- In our experiments with MultiGLICE, we explore the effect of the function depth in our inter-procedural program slicing algorithm. To train and evaluate MultiGLICE, we utilize a dataset of both vulnerable and non-vulnerable samples, sourced from the Software Assurance Reference Dataset (SARD).
- We perform experiments to compare MultiGLICE with the state-of-the-art FUNDED model and demonstrate that MultiGLICE achieves significantly higher detection performance at a lower cost.

The remainder of the paper is structured as follows. In Section 2, we review related work on vulnerability detection with deep learning and program slicing. In Section 3, we introduce the MultiGLICE model and the dataset used for training and testing. In Section 4, we present and discuss the experiments that we performed. The paper concludes with Section 5, where we summarize the key findings and give directions for future work.

2. Prior Work on Vulnerability Detection

In recent years, many studies have appeared that have leveraged deep learning for software vulnerability detection. For a comprehensive overview, we refer to recent survey papers [14,15]. In this section, we focus on related work that is more closely related to the research presented in this paper, as well as highlighting some recent trends.

VulDeePecker [7] is one of the first models that applies deep learning for vulnerability detection. It focuses on vulnerabilities that are caused by improper uses of library/API function calls. First, program slices are extracted that focus on the arguments of each library/API function call in a program, using a data dependency graph. These program slices are assembled into a list of semantically related statements, called a code gadget. Subsequently, each code gadget undergoes lexical analysis to be converted into a sequence

of tokens, which are encoded into vectors using word2vec and processed by a bidirectional long short-term memory (BiLSTM) neural network.

In a subsequent study, VulDeePecker was improved as μVulDeePecker [9]. VulDeePecker employs binary classification, indicating whether a piece of code is vulnerable or not; μVulDeePecker employs multiclass vulnerability detection, enabling it to identify the specific type of vulnerability. Furthermore, μVulDeePecker refines the concept of code gadgets to capture both control and data dependencies. It is composed of three BiLSTM networks: a global features network to learn broader semantic relationships between statements from code gadgets; a local features network to capture details from individual program statements; and a feature fusion model.

The research group responsible for the development of VulDeePecker and μVulDeePecker also introduced the SySeVR model [10]. This model introduces the concept of Syntax-Based Vulnerability Candidates (SyVCs), which encapsulate vulnerability-related syntactic characteristics extracted from a program's abstract syntax tree (AST). These characteristics include patterns associated with library/API function calls, array usage, arithmetic expressions, and pointer operations. To capture both data and control dependencies, program dependency graphs (PDGs) are constructed for each function. Utilizing these PDGs and SyVCs, program slices spanning multiple functions are identified and subsequently transformed into Semantics-Based Vulnerability Candidates (SeVCs). The SeVCs are then converted into symbolic representations, tokenized into sequences via lexical analysis, and encoded as vector representations using the word2vec model. Empirical evaluations demonstrate that bidirectional recurrent neural networks (RNNs), particularly bidirectional gated recurrent units (BiGRUs), achieve superior performance compared to unidirectional RNNs and convolutional neural networks (CNNs). Both RNN and CNN architectures outperform deep belief networks (DBNs) and traditional shallow learning models, such as linear logistic regression (LR) and single-hidden-layer multi-layer perceptron (MLP).

Another approach to vulnerability detection uses graph neural networks (GNNs) [12], as exemplified by the Devign model, a binary classification framework operating at the function level [8]. In this model, functions are represented as joint graphs, where nodes correspond to elements of the function's AST and edges are derived from multiple sources, including the AST, control flow graph, data flow graph, and the natural sequence of source code statements. A node representation matrix is constructed by aggregating information from different types of neighboring nodes using a gated graph recurrent unit (GRU). Subsequently, a convolutional module employing one-dimensional convolutional layers and dense neural networks extracts features relevant for graph-level vulnerability classification.

The FUNDED model builds upon the Devign framework, enhancing function-level vulnerability detection [11]. It represents a function as a graph by integrating its abstract syntax tree (AST) with control and data flow information, along with sequential features such as token sequences. In this representation, nodes correspond to statements, code blocks, or values, while edges are encoded in an adjacency matrix that captures various types of relationships between the nodes. Leveraging these adjacency matrices and initial node embeddings, a multi-relational gated graph neural network (GGNN) learns a global embedding vector, which is subsequently processed by a fully connected neural network for classification.

The majority of prior research on deep learning-based vulnerability detection has focused on fine-grained, function-level analysis [7–9,11], enabling the precise localization of vulnerabilities. Among the existing approaches, only the SySeVR model extends its analysis beyond individual functions by incorporating inter-procedural information [10]. Inter-procedural analysis, which captures interactions across function boundaries, holds promise in detecting vulnerabilities that span multiple functions. However, the effectiveness,

computational trade-offs, and broader implications of inter-procedural analysis in deep learning-based vulnerability detection remain insufficiently explored.

Recent literature surveys highlight a significant rise in the application of machine learning and deep learning techniques for the detection of software vulnerabilities [14,16]. These surveys reveal that researchers often employ hybrid data sources, with graph-based and token-based code representations and embeddings being the most commonly used approaches. Among deep learning models, recursive neural networks (RNNs) and graph neural networks (GNNs) remain the most popular, alongside traditional machine learning models. More recently, there has been growing interest in leveraging large language models (LLMs) for vulnerability detection [17–20].

3. Methodology

In this paper, we introduce MultiGLICE (Multiclass Graph Neural Network with Program Slice), which extends the GLICE model from binary to multiclass classification, where each class corresponds to a specific CWE type. Both GLICE and MultiGLICE extend the FUNDED model with inter-procedural program slicing. This is the first approach for vulnerability detection, to the best of our knowledge, that combines inter-procedural program slicing with GNNs.

GLICE can, in principle, be applied for the detection of all kinds of vulnerabilities. In our initial work, we applied GLICE to detect out-of-bounds write (CWE-787) and out-of-bounds read (CWE-125) vulnerabilities, which typically occur in C/C++ program code [13]. In the present paper, we expand the scope with MultiGLICE to 38 vulnerability types that occur across multiple programming languages, including C/C++, C#, Java, and PHP. A complete overview of the vulnerability types is listed in Table A1 in Appendix A. We perform experiments with MultiGLICE to evaluate the trade-offs in the depth of the inter-procedural analysis and to compare MultiGLICE with FUNDED by evaluating their effectiveness for vulnerability detection and the usage of resources.

In this section, we explain program slicing (Section 3.1). We briefly describe the original FUNDED model (Section 3.2), and we outline our modifications and extensions to the original FUNDED model, which resulted in our improved FUNDED model, which we also applied in our GLICE model (Section 3.3). We describe how the GLICE model was adapted to facilitate multiclass classification in MultiGLICE (Section 3.4), and we provide details of the dataset (Section 3.5) and evaluation metrics (Section 3.6) used in our experiments.

3.1. Program Slicing

Program slicing, first introduced by Weiser in 1984, is a technique for the extraction of a subset of a program while preserving specific behaviors [21]. Weiser's algorithm formulates this process as a graph reachability problem by representing the source code as a program dependence graph—a directed graph that encodes both data and control dependencies between statements and expressions.

Program slicing has been widely applied in various domains, including debugging (identifying statements relevant to a bug), program comprehension (analyzing statements that influence a particular variable), and parallelization (determining independent statements that can be executed concurrently). Ottenstein et al. [22] and Horwitz et al. [23] extended this approach by introducing the system dependence graph, which incorporates interprocedural dependencies, including function calls.

The program slicing techniques employed by VulDeePecker and SySeVR build upon the foundational algorithms developed by Weiser [21] and Horwitz et al. [23]. Specifically, they utilize a security slicing criterion (n, V), where n represents a potentially vulnerable

statement, and V denotes the set of variables used at n. The backward program slice for (n, V) consists of all statements upon which the variables in V at n depend, whereas the forward slice comprises all statements that depend on V. MultiGLICE applies the slicing criteria specified in Table A2. Both backward and forward slicing are utilized, as their combined application is necessary to identify vulnerabilities depending on the specific Common Weakness Enumeration (CWE) category.

The example in Figure 1 shows a code sample containing a buffer overflow vulnerability at the `strcat` function at line 4, as well as the corresponding program slice. The slicing criterion here is the `strcat` function and the variables `dest` and `source` that are its arguments. The program slide is obtained by slicing backwards and including all statements that affect `dest` and `source`.

Code sample with vulnerability at line 4:

```
1   void Print_Prefix(char* source)
2   {
3      char dest[5]="A";
4      strcat(dest, source);
5      printf("%s\n", dest);
6   }
7
8   int main()}
9   {
10     char source[10]="";
11     memset(source, 'B', 9);
12     source[9]="\0";
13     Print_Prefix(source);
14  }
```

Program slice for criterion (strcat(), {dest, source}):

```
10  char source[10]="";
11  memset(source, 'B', 9);
12  source[9]="\0";
13  char dest[5]="A";
14  strcat(dest, source);
```

Figure 1. Example of program slicing with backward slicing.

The example in Figure 2 shows a code sample with vulnerability type CWE-401 (Missing Release of Memory after Effective Lifetime).

In this code sample, the slicing criterion is the `malloc` function at line 2 and the variables `users` and `userCount`. We apply forward slicing, since the vulnerability occurs at line 10 when the function `processUserData` returns without releasing the previously allocated memory for `users`. This causes a memory leak, since the allocated memory should have been freed before returning.

To identify potentially vulnerable statements, we have compiled a list of commonly misused language constructs that cover the range of vulnerability types and programming languages supported by MultiGLICE. This list moves beyond function calls and expressions used in earlier work [13] to cover a more complete set of potentially vulnerable language constructs, including different types of expressions, statements, and declarations. We generate program slices for slicing criteria that contain such a language construct.

Code sample with vulnerability at line 9:

```
1   void processUserData(int userCount) {
2       User *users = (User *)malloc(userCount * sizeof(User));
3       if (users == NULL) {
4           exit(-1);
5       }
6       for (int i = 0; i < userCount; i++) {
7           users[i] = getUserData(i);
8       }
9       // CWE-401: Memory allocated to 'users' is not deallocated
10  }
```

Figure 2. Example of program slicing with forward slicing.

We manually analyzed program slices, as generated by VulDeePecker and SySeVR, from C/C++ code samples in our dataset and observed that essential information was missing for the accurate determination of whether the code contained vulnerabilities. This missing information was related to specific language constructs: function pointers, global variables, macros, and structures. A function pointer stores the address of a function, effectively pointing to executable code rather than data. If function pointers are excluded from program slices, the corresponding function calls are not accounted for. Global variables, which are declared outside the scope of a function but can be accessed and modified within any function, should also be included in program slices to ensure completeness. Macros represent labeled code blocks that are replaced with their defined content when referenced. This substitution should occur before generating program slices to maintain code integrity. Similarly, structures (structs) define custom data types that can group related variables of different types. Since variables in V may be of such types, structures must also be considered in program slices. We observed that function pointers, global variables, macros, and structures were used in our dataset. We therefore extended the program slicing algorithm to also consider these language constructs.

We observed that, when a code sample contained multiple (say p) potentially vulnerable language constructs, SySeVR generated p slicing criteria, resulting in p corresponding program slices. Consequently, these slices may overlap, leading to duplicate entries within the overall set of program slices. This redundancy can introduce bias in the classification results, as the same program slice may appear in both the training and test datasets, potentially leading to overly optimistic performance evaluations.

An analysis of the vulnerable code samples in our dataset indicates that, although multiple lines may contain potentially vulnerable language constructs, only a single construct is typically responsible for the actual vulnerability. The corresponding code line within the sample is labeled as vulnerable. To address this, we enforce a single slicing criterion per code sample, ensuring that it corresponds to the actual vulnerable language construct. For the patched version of the sample, we apply a related slicing criterion, as the modified code often retains the same or a similar language construct.

We consider inter-procedural analysis and, therefore, generate program slices that may span multiple functions that call each other. A call tree represents the calling relationships between functions, where nodes correspond to functions and edges represent function calls. The depth of a call tree is defined as the longest path within the tree. An example is shown in Figure 3, where the slicing criterion includes function `A`. Function `B` is at depth 1, since it is called directly from `A`. Likewise, `C` and `D` are at function depth 1 since they directly call `A`. Our program slicing algorithm takes into account a target depth, which defines the maximum depth considered during analysis. Increasing the target depth expands the scope

of vulnerability detection, thereby improving the detection performance. However, this also increases the complexity of the analysis and demands more resources.

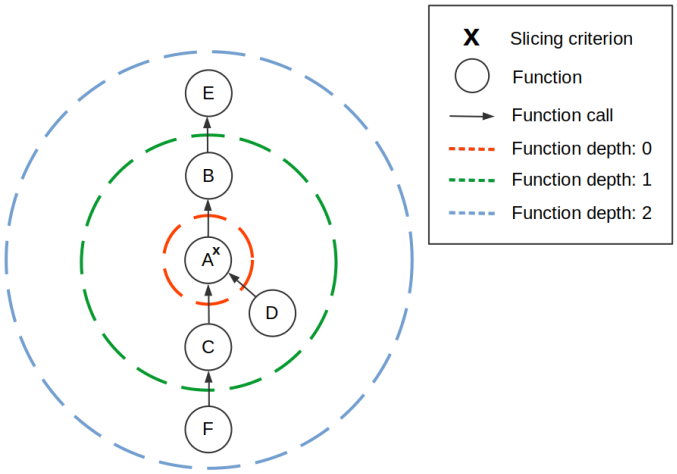

Figure 3. Example of call tree.

3.2. FUNDED

Vulnerability detection in the original FUNDED model takes as input the source code of a function. A program graph is constructed from the abstract syntax tree (AST) of the source code, which is augmented with additional syntax and semantic information such as control and data flows and sequential information. The extended AST is represented in directed relation graphs, where nodes represent tokens of the source code and edges represent relationships between the nodes. Each node is converted into a 100-dimensional vector using word2vec, which captures the semantic similarities between tokens. These vectors, together with the adjacency lists that define the edges for each graph type, are batched.

Building on prior work on gated graph neural networks (GGNNs) [24,25], an extended GGNN is used that consists of four stacked embedding models based on the gated recurrent unit (GRU) [26]. This extended GGNN learns a global embedding vector that incorporates higher-degree neighborhoods. The layers in the extended GGNN perform message passing by iteratively updating the node representations to aggregate information from neighboring nodes. Intermediate node representations from each layer are concatenated with the initial features to form the final node representations. These node-level representations are subsequently converted into graph-level representations by aggregation through the weighted average and weighted sum. The resulting vector is passed to a standard fully connected network (MLP) to perform classification using a sigmoid activation function [11].

The Python 3.7 source code of the original FUNDED model is available at https://github.com/HuantWang/FUNDED_NISL (accessed on 27 February 2025). Its GNN implementation is based on the GGNN architecture in Microsoft's GNN library using Tensorflow 2.0 (https://github.com/microsoft/tf2-gnn accessed on 27 February 2025).

3.3. Improved FUNDED and GLICE

We improved the original FUNDED model, resulting in our improved FUNDED model, in two ways: we fixed a number of bugs and software engineering issues, and we improved the embedding strategy.

We resolved the following bugs and software engineering issues.

- The FUNDED model represents source code using multiple relational graphs. For C/C++ code, it employs two external libraries to construct these graphs: CDT (https://projects.eclipse.org/projects/tools.cdt accessed on 27 February 2025) and Joern (https://github.com/joernio/joern accessed on 27 February 2025). However,

FUNDED exhibits incorrect graph aggregation. Due to a flaw in the data shuffling routine, graphs from different samples are inadvertently mixed, rendering the Joern graph ineffective. Additionally, edges in the CDT graph are incorrectly connected, as CDT utilizes one-based node numbering, whereas FUNDED adopts a zero-based numbering scheme. We have identified and resolved these issues.

- FUNDED incorrectly utilizes samples from both the training and validation sets for training. The validation set should only be used to evaluate the model and optimize the hyperparameters, such as the number of epochs, to prevent overfitting. We addressed this issue by removing the code that mistakenly adds samples from the validation set to the training set.
- As the entire dataset is loaded into the memory, exceeding the available memory limit (32 GB in our case), which results in an out-of-memory exception. To mitigate this issue, we implemented a streaming approach where data batches are loaded from the disk incrementally. While one batch is being processed during training, the subsequent batch is preloaded from the disk. To further optimize data handling, we introduced caching using the Python pickle module for the efficient serialization and deserialization of object structures. This ensures that conversions to NumPy arrays are performed only once, reducing the computational overhead.
- We modified the FUNDED implementation to ensure that evaluation metrics, including the precision, recall, F1-score, and accuracy, are computed over the entire test set, rather than being limited to the samples in the final batch.
- The FUNDED model does not perform variable renaming. However, we observed that certain variable names in our dataset, such as `dataGoodBuffer` and `dataBadBuffer`, explicitly indicate whether a buffer is sufficiently large to prevent a buffer overflow. To eliminate potential biases and prevent the model from learning patterns based on variable names rather than program semantics, we implemented a variable renaming mechanism.

FUNDED applies word2vec [27] to convert tokens to vectors. We improved the usage of word2vec in the following ways.

- Word2vec was originally developed for natural language processing, where it maps words with similar meanings to nearby vectors in a continuous vector space. The model learns these representations by analyzing the contextual relationships between words, considering the surrounding words within a sentence. In the FUNDED model, we observed that many sentences consisted of only a single token, which hinders effective learning. To address this limitation, we modified the embedding strategy so that a sentence represents an entire function rather than a single line of code.
- Additionally, we identified issues in the tokenization process used in FUNDED. Specifically, token splitting is often performed incorrectly. For example, the statement `printWLine(data);` is treated as a single token, whereas it should be split into two distinct tokens: the function name `printWLine` and the argument `data`. This error arises because tokenization considers only space characters as delimiters. Furthermore, a bug in the parser causes tokens following a closing parenthesis `)` to be removed. We addressed these issues by adopting the tokenization logic from Joern, ensuring accurate token segmentation.

GLICE combines inter-procedural program slicing (Section 3.1) with the improved FUNDED. To predict whether some target program code contains a vulnerability, GLICE first applies inter-procedural analysis to generate program slices that can span multiple functions in the program code. Next, the program slices are input to the improved FUNDED model.

3.4. MultiGLICE

MultiGLICE extends GLICE by facilitating multiclass classification. We adapted the neural network architecture of the GLICE model in three ways. First, we extended the output layer of the regression multi-layer perceptron (MLP) to contain a neuron for every class. Second, we replaced the activation function in the output layer with a softmax function, which is widely used for multiclass classification in neural networks as it ensures that it outputs a probability distribution over classes. Third, we use the categorical cross-entropy as a loss function as it takes into account the predicted probabilities over multiple classes in multiclass classification. These adaptations enable MultiGLICE to identify the specific CWE type of the detected vulnerability, enhancing the usability of the model for developers and security researchers.

3.5. Dataset

We utilized a dataset primarily sourced from the Software Assurance Reference Dataset (SARD, http://samate.nist.gov/SARD accessed on 27 February 2025). SARD is a repository of programs that contain documented software vulnerabilities. The test cases in SARD range from small, artificially created programs to large, real-world applications. From this dataset, we selected the five largest test suites, as shown in Table 2. The majority of these test cases are synthetic, specifically designed to test and evaluate the effectiveness of various software vulnerability detection tools.

Table 2. Sources of training data.

Name	Author	Language
PHP Vulnerability Test Suite	Bertrand C. Stivalet	PHP
Juliet C# 1.3	NSA Center for Assured Software	C#
Juliet Java 1.3	NSA Center for Assured Software	Java
Juliet C/C++ 1.3	NSA Center for Assured Software	C++
PHP test suite—XSS, SQLi 1.0.0	Schuckert, Langweg, and Katt	PHP

Unlike our prior work, which focused on C/C++ code samples, the dataset used in this study also includes samples written in C#, Java, and PHP. Furthermore, we have broadened the vulnerability types that we can detect, encompassing a total of 38 CWE types (see Table A1). The dataset used in this study consists of 605,264 samples, as detailed in Table 3. The distribution of programming languages and the number of vulnerable and non-vulnerable samples per CWE type can be found in Table A3 in Appendix A. This expanded dataset allows us to train a more capable model that can detect software vulnerabilities across programming languages and vulnerability types.

Table 3. Number of samples per programming language in the data.

Metric	C	C#	C++	Java	PHP	Total
Number of samples	92,822	78,834	58,462	88,750	286,396	605,264
Percentage	15%	13%	10%	15%	47%	100%

Note that the dataset used in the work by Li et al. [10] and Wang et al. [11] was also mainly based on data from SARD.

3.6. Evaluation Metrics

We evaluate the performance using the metrics outlined in Table 4. For binary classification, a true negative (TN) or true positive (TP) indicates that a code sample is correctly

classified as non-vulnerable or vulnerable, respectively. A false negative (FN) or false positive (FP) indicates incorrect classification as non-vulnerable or vulnerable, respectively. For multiclass classification, true positives and true negatives also imply that the vulnerability type is correctly identified. Precision is the ratio of vulnerable samples that are correctly classified among all samples classified as vulnerable. Recall is the ratio of vulnerable samples that are correctly classified among all actual vulnerable samples. The F1-score is the harmonic mean of the precision and recall. Accuracy is the ratio of correct classifications to the total number of classifications.

Table 4. Evaluation metrics.

Metric	Formula
Precision	$\frac{\text{TP}}{\text{TP}+\text{FP}}$
Recall	$\frac{\text{TP}}{\text{TP}+\text{FN}}$
F1-score	$\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$
Accuracy	$\frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}$

In multiclass classification, typically, three types of averaging over the classes are reported. *Micro-averaging* ignores the class distribution and treats every sample as equal. For *macro-averaging*, the metrics of each class are computed separately and averaged, thereby giving equal weight to each class regardless of the number of samples. For instance, the macro recall is computed by

$$\text{Macro recall} = \frac{\sum_{c \in Classes} \text{Recall}_c}{|Classes|}$$

In *weighted averaging*, the metric of each class is weighted by the number of samples that have this class label in the data. For instance, the weighted recall is computed by

$$\text{Weighted recall} = \frac{\sum_{c \in Classes} N_c \cdot \text{Recall}_c}{|N|}$$

where N_c is the number of samples with label c and N is the total number of samples in the data.

For multiclass classification, it holds that the overall accuracy is equal to the weighted recall and the micro-averaged metrics, i.e., the micro recall, micro precision, and micro F1-score. We will report this metric as the micro F1-score in this paper.

As a primary outcome, we will focus on the weighted F1-score. The F1-score provides a balanced view of the precision and recall, which is useful as both false positives and false negatives should be avoided. Unlike the accuracy, it is also not sensitive to the class distribution. We choose the weighted F1-score as the primary metric because it provides a balanced view of the expected F1-score across various types of CWE types that one might come across in practice. For further insights, we will also report results for other metrics in various experiments.

4. Experiments

4.1. Experimental Setup

Figure 4 outlines the approach used to perform the experiments. We compare both the original and improved FUNDED models (indicated in blue) and the GLICE and MultiGLICE models (indicated in red) for different depths of the call tree. All experiments were run on a desktop PC containing an AMD Ryzen 5 3600 CPU, 32 GB of system

memory, and an NVIDIA RTX 3060 GPU with 12 GB of dedicated memory and 3584 CUDA cores. We implemented the models using the CUDA toolkit version 12.2 and Tensorflow version 2.15.

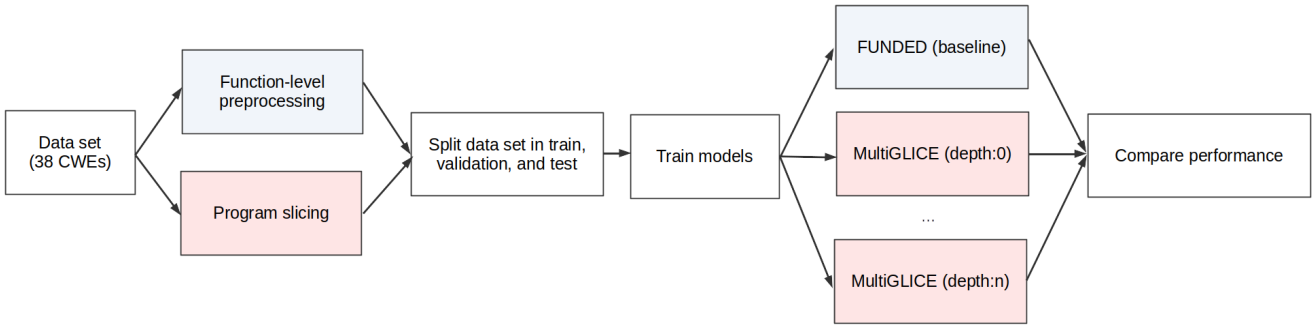

Figure 4. Summary of approach to performing the experiments, showing the flows used to evaluate the original/improved FUNDED model (indicated in blue) and the GLICE/MultiGLICE model (indicated in red).

To investigate the improvements to FUNDED, we used the same hyperparameters for FUNDED and its improvement, as discussed in Section 3.4. Hence, both models had the same number of configurable weights and the same expressive power. The hyperparameters were taken from the original FUNDED model [11], in which the hyperparameters were tuned by a Tree-Structured Parzen Estimator (TPE) [28]. Table A4 in Appendix A lists the values of the hyperparameters. For the multiclass neural network, we re-estimated the hyperparameters using the TPE because the hyperparameters that work well for the binary classification task may not work as well for the multiclass classification task. Table A5 in Appendix A shows the values of the hyperparameters used in these experiments.

We applied 10-fold cross-validation, a method that provides a comprehensive assessment of the model's performance. In this method, we repeatedly split the dataset into a training set (80%), a validation set (10%), and a test set (10%). We report the average results of 10 repetitions. This approach ensures that every sample in the data is used for both training and testing, thereby providing a more reliable estimate of the model's performance. In order to balance the training set, we oversampled the vulnerable samples to ensure that each CWE/language combination had an equal number of vulnerable and non-vulnerable samples during training. This prevented the model from being biased towards the non-vulnerable class.

We executed three sets of experiments, as outlined in the following subsections.

1. We ran experiments to compare the original FUNDED model and our improved FUNDED model including the bug fixes, as described in Section 3.3.
2. We evaluated the impact of the target depth in our program slicing algorithm.
3. We ran experiments to compare the MultiGLICE model with the original FUNDED model adapted to multiclass classification.

4.2. Improved FUNDED

We conducted experiments to compare the original FUNDED model with our improved version, in which we addressed issues in the graph aggregator, edge connector, and word2vec, as described in Section 3.3. We resolved bugs in the calculation of the evaluation metrics in both the original and improved FUNDED models to ensure a fair comparison.

In these experiments, we applied the same dataset as was used by Wang et al. [11] to evaluate the FUNDED model. This dataset is publicly available at https://github.com/HuantWang/FUNDED_NISL (accessed on 27 February 2025). We only used the code

samples written in C because only their C parser was publicly available. The dataset contains 87,802 code samples, of which 43,901 samples are vulnerable and 43,901 are non-vulnerable samples. The vulnerable samples contain vulnerabilities from 28 CWEs: CWE-020, CWE-074, CWE-077, CWE-078, CWE-119, CWE-138, CWE-190, CWE-191, CWE-200, CWE-287, CWE-362, CWE-369, CWE-400, CWE-404, CWE-467, CWE-476, CWE-573, CWE-610, CWE-665, CWE-666, CWE-668, CWE-670, CWE-676, CWE-704, CWE-754, CWE-758, CWE-770, CWE-772. We used this dataset to train and evaluate both the original FUNDED model and our improved FUNDED model. We used k-fold cross-validation with $k = 10$ for both.

The results, as shown in Table 5, indicate that our improved FUNDED model performs considerably better than the original FUNDED model. The precision, recall, F1-score, and accuracy improve by 3.74%, 1.56%, 2.84%, and 3.32% respectively.

Table 5. Results for FUNDED dataset containing 28 CWEs.

Metric	Original FUNDED	Improved FUNDED
Precision	0.7912	0.8286
Recall	0.9463	0.9619
F1-score	0.8618	0.8902
Accuracy	0.8482	0.8814

4.3. Target Depth

We evaluated the impact of the target depth in our MultiGLICE model, using the dataset as described in Section 3.5. The depth of the samples in the dataset ranges from 0 (a single function) to 4 (a call tree with 4 function calls). When the sample depth exceeds 0, we iteratively analyze the callers of a function up to the specified target depth. The overall results are presented in Table 6. We observe that the detection performance improves as the target depth increases, which is expected, as a higher target depth expands the scope of inter-procedural analysis.

Table 6. Impact of target depth on various metrics in GLICE.

Metric	0	1	2	3	4
Micro F1-score	0.8742	0.9666	0.9842	0.9886	0.9931
Weighted precision	0.9417	0.9826	0.9863	0.9900	0.9940
Weighted F1-score	0.9009	0.9733	0.9847	0.9889	0.9932
Macro recall	0.8679	0.9633	0.9850	0.9877	0.9912
Macro precision	0.7323	0.9201	0.9442	0.9632	0.9867
Macro F1-score	0.7724	0.9294	0.9612	0.9735	0.9881

The largest improvement occurs when increasing the target depth from 0 to 1. When further increasing the target depth from 1 to 4, the improvements in detection diminish. This is due to the distribution of the samples in our dataset, since 73.75% of the samples have a maximum depth of 0, 21.65% have a maximum depth of 1, and 4.70% have a maximum depth above 1, as shown in Table 7. Hence, increasing the target depth above 1 can improve the detection only in a small subset of the samples.

Table 8 shows the weighted F1-score when increasing the target depth. Each row corresponds to a set of samples with the same depth. The performance significantly increases if the target depth increases up to the maximum depth in the sample. For example, samples with depth 2 (third row) will benefit significantly from slicing up to target depth 2, increasing the F1-score from 0.3270 to 0.9943. As expected, further increasing the target depth does not alter the F1-score, except for minor rounding errors.

Table 7. Maximum depth of samples.

Maximum Depth	#Samples	%Samples
0	446,391	73.75%
1	131,094	21.65%
2	14,863	2.56%
3	6628	1.10%
4	6288	1.04%

Table 8. Weighted F1-score by depth of each sample.

Sample Depth	Target Depth 0	Target Depth 1	Target Depth 2	Target Depth 3	Target Depth 4
0	0.9960	0.9962	0.9962	0.9962	0.9962
1	0.6043	0.9838	0.9836	0.9839	0.9837
2	0.3270	0.3259	0.9943	0.9942	0.9939
3	0.5801	0.5759	0.5852	0.9869	0.9868
4	0.5620	0.5627	0.5641	0.5556	0.9872

Table 9 shows the F1-scores for each CWE type individually. Similarly to Table 8, the performance for each CWE type improves considerably if the depth of the analysis is increased. We do see some differences—for example, CWE-15 (External Control of System or Configuration Setting) has a very low F1-score for depth 0 and increases to a perfect classifier at depth 4. Remarkably, the non-vulnerable examples can be identified quite consistently at a low depth. These results suggest that there is quite some variation in the improvements that can be gained by program slicing for different types of vulnerabilities. For more detailed results, we refer to Table A6 in Appendix A, which contains the confusion matrix.

Table 9. F1-score for each CWE type (F1-score $<$ 0.90 in red, 0.90 $<$ F1-score $<$ 0.98 in yellow, F1-score $>$ 0.98 in green).

Vuln. Type	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4
CWE-015	0.0484	0.1567	0.6370	0.7692	1
CWE-023	0.6631	0.9688	0.9755	0.9823	1
CWE-036	0.6667	0.9561	0.9739	0.9854	1
CWE-078	0.7857	0.9639	0.9781	0.9869	1
CWE-079	0.9787	0.9849	0.9998	0.9998	1
CWE-080	0.7961	0.9433	0.9814	0.9925	1
CWE-089	0.9601	0.9898	0.9942	0.9955	0.9987
CWE-090	0.8225	0.9153	0.9855	0.9956	1
CWE-091	0.8649	0.8850	1	1	1
CWE-098	0.8142	0.8833	1	1	1
CWE-113	0.7818	0.9628	0.9729	0.9852	0.9915
CWE-121	0.7688	0.9688	0.9707	0.9785	1
CWE-122	0.7445	0.8554	0.8668	0.8743	0.8806
CWE-124	0.6748	0.9531	0.9412	0.9682	1
CWE-126	0.8232	0.9498	0.9860	0.9965	1
CWE-127	0.7897	0.9732	0.9852	0.9877	1
CWE-129	0.7758	0.9528	0.9668	0.9821	0.9950

Table 9. *Cont.*

Vuln. Type	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4
CWE-134	0.7119	0.9276	0.9702	0.9824	0.9974
CWE-190	0.7526	0.9014	0.9203	0.9196	0.9338
CWE-191	0.7290	0.8888	0.8927	0.9068	0.9243
CWE-194	0.7591	0.9483	0.9643	0.9821	1
CWE-195	0.7765	0.9694	0.9623	1	1
CWE-197	0.8044	0.9617	0.9751	0.9895	1
CWE-319	0.7836	0.9790	0.9459	0.9722	0.9929
CWE-369	0.7680	0.9602	0.9735	0.9814	0.9930
CWE-400	0.7835	0.9656	0.9657	0.9769	0.9941
CWE-401	0.7673	0.9527	0.9877	1	1
CWE-415	0.6351	0.9663	0.9836	1	1
CWE-457	0.9010	1	1	1	1
CWE-476	0.7886	0.9933	0.9677	0.9600	0.9799
CWE-563	0.8791	0.9888	0.9944	0.9944	1
CWE-590	0.7900	0.9729	0.9835	0.9853	1
CWE-601	0.8411	0.9198	0.9985	1	1
CWE-606	0.7541	0.9734	0.9734	0.9808	0.9922
CWE-643	0.7565	0.9425	0.9701	0.9878	0.9939
CWE-690	0.8867	0.9712	0.9643	0.9714	0.9712
CWE-762	0.5580	0.8366	0.8705	0.8967	0.9068
CWE-789	0.7862	0.9552	0.9746	0.9897	1
Non-vuln.	0.9277	0.9821	0.9915	0.9943	0.9970

4.4. MultiGLICE

We conducted experiments to compare the MultiGLICE model with the FUNDED model. The MultiGLICE model incorporates inter-procedural program slicing, as described in Section 3.1, and enhances the FUNDED model with bug fixes and our word embedding strategy, as outlined in Section 3.4. The results for the original FUNDED, our improved FUNDED model (which includes bug fixes and the improved word embedding but no program slicing), and MultiGLICE (which further improves FUNDED by adding program slicing with a target depth of 4) are shown in Table 10. Similar to what was shown Table 5, there are significant improvements in the multiclass setting. By also adding program slicing, the performance is improved further: compared to the original FUNDED model, the micro F1-score improves by 27%, the weighted F1-score improves by 23%, and the macro F1-score improves by 29%.

Table 10. Results for a wide range of vulnerability types.

Metric	FUNDED Original	FUNDED Improved	MultiGLICE Depth 4
Micro F1-score	0.7194	0.8742	0.9931
Weighted precision	0.8576	0.9417	0.9940
Weighted F1-score	0.7641	0.9009	0.9932
Macro recall	0.7944	0.8679	0.9912
Macro precision	0.6651	0.7323	0.9867
Macro F1-score	0.6940	0.7724	0.9881

We also evaluate the time required to train a model. While model training is a one-time cost, retraining may be necessary to adapt the model to changes in the security landscape. The training time depends on several factors, including the available computing resources (CPU, GPU, memory), the chosen hyperparameters, the computational complexity of each epoch, and the total number of epochs required for convergence.

Table 11 presents the number of samples processed per second during training on our desktop PC. A higher processing rate corresponds to a shorter epoch duration. Our results indicate that MultiGLICE without program slicing (target depth 0) is 89% faster than the original FUNDED model. This performance improvement is primarily due to MultiGLICE utilizing Joern for graph generation, whereas FUNDED incorporates both CDT and Joern. The rationale behind FUNDED's use of both tools remains unclear, as CDT does not introduce any additional edges beyond those already present in the Joern-generated graph.

Furthermore, we observe that MultiGLICE at depth 4 is 25% slower than MultiGLICE at depth 0, demonstrating that increasing the target depth results in a larger computational overhead. Nevertheless, even at depth 4, MultiGLICE remains 41% faster than the original FUNDED model.

Table 11. Number of samples processed per second during training.

	MultiGLICE				
FUNDED Original	**Target Depth 0**	**Target Depth 1**	**Target Depth 2**	**Target Depth 3**	**Target Depth 4**
1727.24	3263.41	2525.63	2522.94	2457.84	2442.17

The number of samples processed per second during training serves as an estimate for the duration of one epoch. However, since multiple epochs may be required for convergence, a model that processes more samples per second could still result in a longer overall training time if the optimal number of epochs is higher. To mitigate overfitting, we implement early stopping during training, halting the process whenever the model ceases to improve on the validation set.

To compare the performance of FUNDED with MultiGLICE, we calculated the average number of epochs required to reach a weighted F1-score of 0.77, which is the maximum achievable F1-score for FUNDED. On average, FUNDED requires 12.3 epochs, while MultiGLICE reaches this F1-score within a single epoch. Taking into account that MultiGLICE at depth 4 is approximately 40% faster per epoch than FUNDED, we conclude that the training of MultiGLICE is roughly 17 times faster than the training of the original FUNDED model.

5. Conclusions

The MultiGLICE model enhances and extends the previous FUNDED model, addressing a wide range of software vulnerabilities. Notably, MultiGLICE is the first model to integrate inter-procedural program slicing with a graph neural network (GNN) for vulnerability detection within a multiclass classification framework.

Our results show that vulnerability detection spanning multiple functions in a call tree using inter-procedural analysis outperforms function-level analysis. MultiGLICE can be configured with a target depth parameter, which specifies the maximum depth of the inter-procedural analysis. In our dataset, approximately one-third of the vulnerabilities span function boundaries, so the vulnerability detection improves as the target depth increases. This demonstrates that both the model's neural network architecture and the input data are crucial factors. While increasing the target depth enhances detection, it also

requires more computing resources. By adjusting the target depth, a user of MultiGLICE can balance the trade-off between the detection performance and computational efficiency.

Our experimental results obtained with code samples from the SARD dataset show that MultiGLICE outperforms FUNDED. The micro, weighted, and macro F1-scores of MultiGLICE with target depth 4 are, respectively, 27%, 23%, and 29% higher than those of FUNDED. In addition, the time required to train the MultiGLICE model is about 17 times smaller than that for FUNDED.

The research presented in this paper has several limitations. We focused on vulnerabilities in mostly synthetic samples from SARD, which are generally easier to detect than real-world vulnerabilities. Additionally, the maximum depth in these samples was limited to 4, whereas, in real-world scenarios, the maximum depth of call trees may be higher. We have shown that increasing the target depth demands more computing power. Therefore, in practice, a trade-off between the detection performance and computational efficiency may be necessary. By introducing the target depth as a parameter in MultiGLICE, users can explicitly control this trade-off.

Our results with MultiGLICE are promising. We are exploring several directions for future work. The first direction is the further validation of MultiGLICE. This implies the use of multiple datasets to validate its generalizability, as well as the use of real-world data to validate its application in practice. It is a significant challenge to construct a dataset with real-world test cases, which was beyond the scope of this paper. The second research direction involves explaining the predictions made by MultiGLICE and using this insight to further enhance the model (e.g., [29]). A related area of exploration is to investigate adversarial attacks, where a deep learning model is misled into making incorrect predictions by altering the input data. By understanding how changes to the code impact predictions, the GLICE approach can be improved further.Other research directions include exploring how LLMs can contribute to static code analysis and the automated patching of vulnerabilities.

Author Contributions: Conceptualization and methodology, W.d.K., H.V. and A.H.; software and experiments, W.d.K.; validation, W.d.K., H.V. and A.H.; writing, W.d.K., H.V. and A.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding. The APC was funded by MDPI.

Data Availability Statement: The source code of MultiGLICE and the dataset used are available at https://github.com/wesleydekraker/glice (accessed on 27 February 2025).

Acknowledgments: ChatGPT (o1-mini) was used to refine the text of this manuscript by correcting grammatical errors and suggesting clearer formulations. All content was originally written by the authors, and no new content was generated by ChatGPT.

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

Appendix A

Table A1. CWE types.

CWE Type	Description
CWE-015	External Control of System or Configuration Setting
CWE-023	Relative Path Traversal
CWE-036	Absolute Path Traversal
CWE-078	Improper Neutralization of Special Elements Used in an OS Command ('OS Command Injection')
CWE-079	Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting')
CWE-080	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
CWE-089	Improper Neutralization of Special Elements Used in an SQL Command ('SQL Injection')
CWE-090	Improper Neutralization of Special Elements Used in an LDAP Query ('LDAP Injection')
CWE-091	XML Injection (or Blind XPath Injection)
CWE-098	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')
CWE-113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Request/Response Splitting')
CWE-121	Stack-Based Buffer Overflow
CWE-122	Heap-Based Buffer Overflow
CWE-124	Buffer Underwrite ('Buffer Underflow')
CWE-126	Buffer Over-Read
CWE-127	Buffer Under-Read
CWE-129	Improper Validation of Array Index
CWE-134	Use of Externally Controlled Format String
CWE-190	Integer Overflow or Wraparound
CWE-191	Integer Underflow (Wrap or Wraparound)
CWE-194	Unexpected Sign Extension
CWE-195	Signed to Unsigned Conversion Error
CWE-197	Numeric Truncation Error
CWE-319	Cleartext Transmission of Sensitive Information
CWE-369	Divide By Zero
CWE-400	Uncontrolled Resource Consumption
CWE-401	Missing Release of Memory after Effective Lifetime
CWE-415	Double Free
CWE-457	Use of Uninitialized Variable
CWE-476	NULL Pointer Dereference
CWE-563	Assignment to Variable without Use
CWE-590	Free of Memory Not on the Heap
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
CWE-606	Unchecked Input for Loop Condition
CWE-643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
CWE-690	Unchecked Return Value to NULL Pointer Dereference
CWE-762	Mismatched Memory Management Routines
CWE-789	Memory Allocation with Excessive Size Value

Table A2. Slicing criteria of MultiGLICE.

C/C++	C#	Java	PHP
alloca function	Arithmetic operator	Arithmetic operator	db2_exec function
arithmetic operator	Array.Length property	BufferedWriter.write method	db2_prepare function
calloc function	Cast operator	Cast operator	echo statement
cast operator	DbConnection.ConnectionString property	Connection.setCatalog method	eval function
CreateFileA function	Declaration statement	Declaration statement	exit function
CreateFileW function	DirectorySearcher.FindOne method	DriverManager.getConnection method	header function
delete keyword	For statement	File constructor	http_redirect function
_execl function	HttpResponse.AddHeader method	FileOutputStream.write method	ldap_search function
_execlp function	HttpResponse.AppendCookie method	For statement	mysqli_multi_query function
_execv function	HttpResponse.Redirect method	HttpServletResponse.addCookie method	mysqli_real_query function
_execvp function	HttpResponse.Write method	HttpServletResponse.addHeader method	mysqli_stmt_execute function
fopen function	Logical AND operator	HttpServletResponse.getWriter method	mysql_query function
for statement	Math.Sqrt method	HttpServletResponse.sendRedirect method	PDO.prepare method
fprintf function	New operator	HttpServletResponse.setHeader method	PDO.query method
free function	Object.Equals method	InitialDirContext.search method	pg_query function
ifstream.open method	Object.ToString method	KerberosKey constructor	pg_send_query function
ldap_search_ext_sA function	Process.Start method	Math.sqrt method	printf function
ldap_search_ext_sW function	SecureString.AppendChar method	Method parameter	print function
LogonUserA function	SqlCommand.ExecuteNonQuery method	New operator	require statement
LogonUserW function	SqlCommand.ExecuteScalar method	Object.equals method	SimpleXMLElement.xpath method
loop statement	SqlConnection constructor	PasswordAuthentication constructor	SQLite3.query method
malloc function	SslStream.Write method	PrintWriter.println method	sqlsrv_execute function
memcpy function	StreamReader constructor	Runtime.exec method	sqlsrv_query function
memmove function	StreamWriter.WriteLine method	Statement.executeBatch method	system function
new keyword	String.Format method	Statement.execute method	trigger_error function
ofstream.open method	String.Length property	Statement.executeQuery method	user_error function
open function	String.Trim method	Statement.executeUpdate method	vprintf function
popen function	Subscript operator	String.trim method	
SetComputerNameA function	Thread.Sleep method	Subscript operator	
sizeof function	XPathNavigator.Evaluate method	System.out.format method	
sleep function		System.out.printf method	
snprintf function		Thread.sleep method	
_spawnl function		XPath.evaluate method	
_spawnlp function			
_spawnv function			
_spawnvp function			
strcat function			
strcpy function			
strncat function			
strncpy function			
subscript operator			
swprintf function			
system function			
variable declaration statement			
vfprintf function			
vsnprintf function			
wcscat function			
wcscpy function			
wcsncat function			
wcsncpy function			
_wexecl function			
_wexeclp function			
_wexecv function			
_wexecvp function			
_wspawnl function			
_wspawnlp function			
_wspawnv function			
_wspawnvp function			

Table A3. Number of samples per CWE type and programming language.

	Samples	Non-Vulnerable	Vulnerable	C	C#	C++	Java	PHP
CWE-015	2065	1205	860	91	890	16	1068	0
CWE-023	7308	4194	3114	0	890	5350	1068	0
CWE-036	7308	4194	3114	0	890	5350	1068	0
CWE-078	15,154	9116	6038	9100	890	1600	1068	2496
CWE-079	163,674	135,823	27,851	0	0	0	0	163,674
CWE-080	3204	1872	1332	0	1602	0	1602	0
CWE-089	114,403	91,391	23,012	0	3753	0	8340	102,310
CWE-090	6868	3482	3386	910	890	160	1068	3840
CWE-091	6012	4748	1264	0	0	0	0	6012
CWE-098	3264	2592	672	0	0	0	0	3264
CWE-113	8757	6426	2331	0	3753	0	5004	0
CWE-121	11,863	6997	4866	10,015	0	1848	0	0
CWE-122	14,064	8384	5680	6377	0	7687	0	0
CWE-124	4996	3000	1996	3382	0	1614	0	0
CWE-126	3690	2274	1416	2664	0	1026	0	0
CWE-127	4996	3000	1996	3382	0	1614	0	0
CWE-129	18,626	13,668	4958	0	8618	0	10,008	0
CWE-134	14,318	10,404	3914	8430	1946	1440	2502	0
CWE-190	43,822	32,052	11,770	12,780	13,761	1296	15,985	0
CWE-191	32,552	23,820	8732	9798	9174	792	12,788	0
CWE-194	2568	1464	1104	2184	0	384	0	0
CWE-195	2568	1464	1104	2184	0	384	0	0
CWE-197	16,878	9834	7044	1638	12,015	288	2937	0
CWE-319	2610	1908	702	568	556	96	1390	0
CWE-369	15,776	11,544	4232	2556	5838	432	6950	0
CWE-400	12,545	9174	3371	2130	4587	360	5468	0
CWE-401	5770	4154	1616	3134	0	2636	0	0
CWE-415	3320	2400	920	852	0	2468	0	0
CWE-457	3956	3010	946	2408	0	1548	0	0
CWE-476	2613	1869	744	963	694	262	694	0
CWE-563	2859	1961	898	1158	699	294	708	0
CWE-590	6231	3551	2680	1458	0	4773	0	0
CWE-601	6402	3144	3258	0	801	0	801	4800
CWE-606	4718	3444	1274	1420	1390	240	1668	0
CWE-643	3058	2244	814	0	1390	0	1668	0
CWE-690	3808	2444	1364	1820	556	320	1112	0
CWE-762	12,284	8880	3404	0	0	12,284	0	0
CWE-789	10,356	6516	3840	1420	3251	1900	3785	0

Table A4. Hyperparameters used in FUNDED and GLICE.

Hyperparameter	Value
Max. graphs per batch	128
Add self loop edges	True
Tie forward/backward edges	True
GNN aggregation function	sum
GNN message activation function	ReLU
GNN hidden dim	256
GNN number of edge MLP hidden layers	1
GNN initial node representation activation	tanh
GNN dense intermediate layer activation	tanh
GNN number of layers	5
GNN dense every num layers MLP hidden layers	10,000
GNN residual every number of layers	2
GNN layer input dropout rate	0.2
GNN global exchange mode	gru
GNN global exchange every num layers	10,000
GNN global exchange number of heads	4
GNN global exchange dropout rate	0.2
Optimizer	Adam
Learning rate	0.001
Graph aggregation number of heads	16
Graph aggregation hidden layers	128
Graph aggregation dropout rate	0.2

Table A5. Hyperparameters used in MultiGLICE.

Hyperparameter	Value
Max. graphs per batch	256
Add self loop edges	False
Tie forward/backward edges	False
GNN aggregation function	sum
GNN message activation function	ReLU
GNN hidden dim	64
GNN number of edge MLP hidden layers	0
GNN initial node representation activation	tanh
GNN dense intermediate layer activation	tanh
GNN number of layers	7
GNN dense every num layers MLP hidden layers	5
GNN residual every number of layers	2
GNN layer input dropout rate	0.0
GNN global exchange mode	gru
GNN global exchange every num layers	10,000
GNN global exchange number of heads	4
GNN global exchange dropout rate	0.2
Optimizer	Adam
Learning rate	0.0001
Graph aggregation number of heads	8
Graph aggregation hidden layers	[32, 32]
Graph aggregation dropout rate	0.1

Table A6. Averaged confusion matrix for MultiGLICE at depth 4 during cross-validation.

	CWE-015	CWE-023	CWE-036	CWE-078	CWE-079	CWE-080	CWE-089	CWE-090	CWE-091	CWE-098	CWE-113	CWE-121	CWE-122	CWE-124	CWE-126	CWE-127	CWE-129	CWE-134	CWE-190	CWE-191	CWE-194	CWE-195	CWE-197	CWE-319	CWE-369	CWE-400	CWE-401	CWE-415	CWE-457	CWE-476	CWE-563	CWE-590	CWE-601	CWE-606	CWE-643	CWE-690	CWE-762	CWE-789	Non-Vuln.
CWE-015	86.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-023	-	311.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-036	-	-	311.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-078	-	-	-	603.8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-079	-	-	-	-	2784.9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.2
CWE-080	-	-	-	-	-	133.2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-089	-	-	-	-	-	-	2301.2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-090	-	-	-	-	-	-	-	338.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-091	-	-	-	-	-	-	-	-	126.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-098	-	-	-	-	-	-	-	-	-	67.2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-113	-	-	-	-	-	-	-	-	-	-	233.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-121	-	-	-	-	-	-	-	-	-	-	-	486.5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.1
CWE-122	-	-	-	-	-	-	-	-	-	-	-	-	567.8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.2
CWE-124	-	-	-	-	-	-	-	-	-	-	-	-	-	199.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-126	0.1	-	-	-	-	-	-	-	-	-	-	-	-	-	141.5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-127	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	199.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-129	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	495.7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.1
CWE-134	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	391.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-190	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1071.0	106.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-191	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	32.6	840.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-194	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	110.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-195	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	110.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-197	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	704.4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-319	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	70.0	-	-	-	-	-	-	-	-	-	-	-	-	-	0.2
CWE-369	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	423.2	-	-	-	-	-	-	-	-	-	-	-	-	-
CWE-400	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	337.1	-	-	-	-	-	-	-	-	-	-	-	-
CWE-401	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	161.4	-	0.1	-	-	-	-	-	-	-	-	0.1
CWE-415	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	92.0	-	-	-	-	-	-	-	-	-	-
CWE-457	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	94.6	-	-	-	-	-	-	-	-	-
CWE-476	0.2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	73.9	-	-	-	-	-	-	-	0.3
CWE-563	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	89.3	-	-	-	-	0.1	-	0.4
CWE-590	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	268.0	-	-	-	-	-	-
CWE-601	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	325.8	-	-	-	-	-
CWE-606	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	127.4	-	-	-	-
CWE-643	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	81.4	-	-	-
CWE-690	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	135.7	-	0.7
CWE-762	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	276.7	63.7
CWE-789	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	383.9
Non-vuln.	0.1	-	-	-	-	-	3.3	-	-	-	2.7	0.2	161.5	-	-	-	6.2	1.4	9.9	6.6	-	-	-	0.4	4.2	3.3	0.1	-	0.2	0.8	0.1	-	0.1	1	1	7.8	-	0.1	44,553.7

References

1. Building Security in Maturity Model (BSIMM) Report 14. 2023. Available online: https://www.blackduck.com/resources/analyst-reports/bsimm.html (accessed on 15 December 2024) .
2. Avgustinov, P.; De Moor, O.; Jones, M.P.; Schäfer, M. QL: Object-oriented queries on relational data. In Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), Rome, Italy, 17–22 July 2016; pp. 2:1–2:26.
3. Li, Z.; Liu, Z.; Wong, W.K.; Ma, P.; Wang, S. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Trans. Dependable Secur. Comput.* **2024**, *21*, 4600–4618. [CrossRef]
4. Chess, B.; West, J. *Secure Programming with Static Analysis*; Pearson Education: London, UK, 2007.
5. Rajapakse, R.N.; Zahedi, M.; Babar, M.A.; Shen, H. Challenges and solutions when adopting DevSecOps: A systematic review. *Inf. Softw. Technol.* **2022**, *141*, 106700. [CrossRef]
6. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated Vulnerability Detection in Source Code using Deep Representation Learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 757–762.
7. Li, Z.; Zou, D.; Xux, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeepecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, USA, 18–21 February 2018 .
8. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 10197–10207.
9. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 2224–2236. [CrossRef]
10. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
11. Wang, H.; Ye, G.; Tang, Z.; Tan, S.H.; Huang, S.; Fang, D.; Feng, Y.; Bian, L.; Wang, Z. Combining Graph-based Learning with Automated Data Collection for Code Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 1943–1958. [CrossRef]

12. Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; Yu, P.S. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 4–24. [CrossRef] [PubMed]
13. de Kraker, W.; Vranken, H.; Hommersom, A. GLICE: Combining Graph Neural Networks and Program Slicing to Improve Software Vulnerability Detection. In Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Delft, The Netherlands, 3–7 July 2023; pp. 34–41.
14. Shiri Harzevili, N.; Boaye Belle, A.; Wang, J.; Wang, S.; Jiang, Z.M.; Nagappan, N. A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Comput. Surv.* **2024**, *57*, 1–36. [CrossRef]
15. Liang, C.; Wei, Q.; Du, J.; Wang, Y.; Jiang, Z. Survey of source code vulnerability analysis based on deep learning. *Comput. Secur.* **2025**, *148*, 104098. [CrossRef]
16. Sharma, T.; Kechagia, M.; Georgiou, S.; Tiwari, R.; Vats, I.; Moazen, H.; Sarro, F. A survey on machine learning techniques applied to source code. *J. Syst. Softw.* **2024**, *209*, 111934. [CrossRef]
17. Fang, C.; Miao, N.; Srivastav, S.; Liu, J.; Zhang, R.; Fang, R.; Tsang, R.; Nazari, N.; Wang, H.; Homayoun, H.; et al. Large Language Models for Code Analysis: Do {LLMs} Really Do Their Job? In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 829–846.
18. Ullah, S.; Han, M.; Pujar, S.; Pearce, H.; Coskun, A.; Stringhini, G. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 19–23 May 2024.
19. Zhang, C.; Liu, H.; Zeng, J.; Yang, K.; Li, Y.; Li, H. Prompt-enhanced software vulnerability detection using ChatGPT. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, Lisbon Portugal, 14–20 April 2024; pp. 276–277.
20. Zhou, X.; Zhang, T.; Lo, D. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In Proceedings of the IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSENIER), Lisbon Portugal, 14–20 April 2024; pp. 47–51.
21. Weiser, M. Program Slicing. *IEEE Trans. Softw. Eng.* **1984**, *SE-10*, 352–357. [CrossRef]
22. Ottenstein, K.J.; Ottenstein, L.M. The program dependence graph in a software development environment. *ACM Sigplan Not.* **1984**, *19*, 177–184. [CrossRef]
23. Horwitz, S.; Reps, T.; Binkley, D. Interprocedural Slicing using Dependence Graphs. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1990**, *12*, 26–60. [CrossRef]
24. Ye, G.; Tang, Z.; Wang, H.; Fang, D.; Fang, J.; Huang, S.; Wang, Z. Deep Program Structure Modeling Through Multi-Relational Graph-based Learning. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20), Atlanta, GA, USA, 3–7 October 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 111–123.
25. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated graph sequence neural networks. In Proceedings of the International Conference on Learning Representations, San Juan, Puerto Rico, 2–4 May 2016.
26. Cho, K.; van Merrienboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; Association for Computational Linguistics: Stroudsburg, PA, USA, 2014; pp. 1724–1734.
27. Mikolov, T.; Yih, W.t.; Zweig, G. Linguistic Regularities in Continuous Space Word Representations. In Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human language Technologies, Atlanta, GA, USA, 9–15 June 2013; pp. 746–751.
28. Bergstra, J.; Bardenet, R.; Bengio, Y.; Kégl, B. Algorithms for hyper-parameter optimization. In Proceedings of the 2011 25th International Conference on Neural Information Processing Systems (NIPS), Granada, Spain, 12–15 December 2011; Curran Associates, Inc.: Red Hook, NY, USA, 2011; pp. 2546–2554.
29. Agarwal, C.; Zitnik, M.; Lakkaraju, H. Probing GNN Explainers: A Rigorous Theoretical and Empirical Analysis of GNN Explanation Methods. In Proceedings of the 25th International Conference on Artificial Intelligence and Statistics, Valencia, Spain, 28–30 March 2022; pp. 8969–8996.