An Unbiased Transformer Source Code Learning with Semantic Vulnerability Graph

Conference Paper · April 2023

DOI: 10.1109/EuroSP57164.2023.00018

Nafis Tanveer Islam
University of Texas at San Antonio

Gonzalo De La Torre Parra
University of the Incarnate Word

Elias Bou-Harb
Louisiana State University

Paul Rad
University of Texas at San Antonio

An Unbiased Transformer Source Code Learning with Semantic Vulnerability Graph

Nafis Tanveer Islam[1], Gonzalo De La Torre Parra[2], Dylan Manuel[1]
Elias Bou-Harb[1], Peyman Najafirad[1, *]
[1]University of Texas at San Antonio, [2]University of the Incarnate Word
[1]nafistanveer.islam@utsa.edu, [2]gdparra@uiwtx.edu, [1]dylan.manuel@my.utsa.edu,
[1]elias.bouharb@utsa.edu, [1]peyman.najafirad@utsa.edu

Abstract—**Over the years, open-source software systems have become prey to threat actors. Even highly-adopted software has been crippled by unforeseeable attacks, leaving millions of devices exposed. Even as open-source communities act quickly to patch the breach, code vulnerability screening should be an integral part of agile software development from the beginning. Unfortunately, current vulnerability screening techniques are ineffective at identifying novel vulnerabilities or providing developers with code vulnerability and classification. Furthermore, the datasets used for vulnerability learning often exhibit distribution shifts from the real-world testing distribution due to novel attack strategies deployed by adversaries and as a result, the machine learning model's performance may be hindered or biased. To address these issues, we propose a joint interpolated multitasked unbiased vulnerability classifier comprising a transformer "RoBERTa" and graph convolution neural network (GCN). We present a training process utilizing a semantic vulnerability graph (SVG) representation from source code, created by integrating edges from a sequential flow, control flow, and data flow, as well as a novel flow dubbed Poacher Flow (PF). Poacher flow edges reduce the gap between dynamic and static program analysis and handle complex long-range dependencies. Moreover, our approach reduces biases of classifiers regarding unbalanced datasets by integrating Focal Loss objective function along with SVG. Remarkably, experimental results show that our classifier outperforms state-of-the-art results on vulnerability detection with fewer false negatives and false positives. After testing our model across multiple datasets, it shows an improvement of at least 2.41% and 18.75% in the best-case scenario. Evaluations using N-day program samples demonstrate that our proposed approach achieves a 93% accuracy and was able to detect 4, zero-day vulnerabilities from popular GitHub repositories. Our code and data are available at https://github.com/pial08/SemVulDet**

1. Introduction

Threat actors exploit source code vulnerabilities using some of the same sophisticated techniques used in high-level cyberattack campaigns. In many cases, vulnerabilities are introduced by software developers using code snippets from open-source solutions, such as Stack Overflow and GitHub. With insufficient vulnerabilities, new software releases can be high risk. IBM [1] findings suggest software vulnerabilities cost businesses an average of 3.9 million dollars annually; meanwhile, Common Vulnerabilities and Exposures (CVE) [2] reported that 6,015 new CVEs were added during the first quarter of 2022, a 36% increase compared to 4,415 CVEs published in the first quarter of 2021. These reports demonstrate the ubiquity of these vulnerabilities and the importance of detecting and classifying them in large-scale applications.

Detecting vulnerabilities in large-scale programs at an early stage in software development is both a challenge and priority for software developers [3]. Most software developers are experts at writing code, however, they are not always well-versed in software security [4], [5]. A recent zero-day exploit "Log4Shell" was announced against the Apache log4j library [6], the magnitude of which was unlike any other. Given the large adoption of this Java logging library, threat actors had ripe opportunity to take control of web-facing servers, and even services not connected directly to the internet, by permeating malicious code to back-end software running Apache Log4j versions. To prevent such cases, vulnerability classifiers can be enhanced to support developers, whether or not they are experts in security. Existing classifiers [7]–[9] have been proposed for vulnerability detection in code snippets at an early stage of software development. However, these detectors only provide information on whether or not a vulnerability exists in a particular code snippet, but no information is provided regarding the categories of their vulnerabilities.

The vulnerability distribution of real-world production software is imbalanced, as benign source code is released more frequently than vulnerable source code. According to Chakaraborty et al. [10], the suboptimal performance of current deep learning techniques in predicting real-world software vulnerabilities is attributed to training data imbalance and models. To address this issue in training datasets, several techniques such as data augmentation and syntactic data creation have been employed to mitigate the training data imbalance problem [11], [12]. Although balanced training datasets through syntactic data generation are crucial, but they do not typically reflect the distribution shifts that are likely to cause from real-world. According to Geirhos, distribution shifts are underrepresented in the

. * Corresponding Author

datasets widely used in the ML community today [13]. This impacts the performance of the model in predicting real-world vulnerabilities such as N-day and zero-day.

Recent graph creation techniques such as AST [14] and Code Property Graph (CPG) [7], [15], [16] are highly effective at detecting source code vulnerabilities. Even so, they have some bottlenecks. For example, neither AST nor CPG can capture the information when a variable is used out of its scope (when the variable is freed), since this is syntactically correct and only produced during the execution of a program. Similarly, when a divide-by-zero vulnerability occurs, it is exposed during program execution and is therefore also syntactically correct. In addition, current graph generation techniques [7], [14] fail to capture the long-range dependency of a variable. AST and CPG generate nodes and edges per statement, thus ignoring the long-range dependency between two faraway statements. Simply put, these graph-creation techniques fail to capture relatively simple yet high-frequency vulnerabilities, either because they occur during execution, or because of a lack of long-range dependency information.

Long-range dependency [17] [18] is a major challenge in vulnerability detection. A long-range dependency may be caused by a variable declared at the beginning of a function, but the vulnerability associated with that variable may only appear after a few hundred lines of code. Several transformer-based works were previously proposed to address the detection of vulnerabilities in source code [19] [9]. One major bottleneck observed in transformers is learning these long-range dependencies. A couple of works –namely, Longformer [20] and Linformer [21], propose approaches for long-range modeling inputs beyond this limit. Since a function could sometimes be a few hundred lines, these models still fail to capture these long-range dependencies.

In this work, we propose a semantic vulnerability graph (SVG) featuring a rich set of edges capturing semantic and syntactic information, including our novel poacher flow edges to address a variable's information and long-range dependencies during execution time. SVG integrates sequential flow for syntactic understanding of the program, data flow to capture how data flows among variables, and a control flow Graph to capture the general flow of statements. Poacher flow edges allow the integration of semantic information of source code like long-range dependencies, out-of-scope use of a variable, and divide-by-zero vulnerabilities, by generating extra edges between variables that can potentially bridge the gap between static and dynamic program analysis.We propose a transformer and graph neural network-based vulnerability classifier dubbed *Multitask RoBERTa-PFGCN* that includes a large-scale pre-trained RoBERTa [22] on C/C++ source code and utilized Focal Loss (FL) [23] to handle data imbalance issues.Moreover, we propose our new dataset with real-world example functions. Then, by jointly training RoBERTa and GCN modules, our proposed model learns node embeddings using a large-scale Multitask RoBERTa-PFGCN by propagating edge influence through a graph convolution network. The contributions of this paper can be summed up as follows:

- We propose a unique set of edges dubbed poacher flow edges, such that each of these edges is associated with a set of vulnerabilities. We defined our semantic vulnerability graph (SVG) representation of source code by unifying our introduced poacher flow edges with control, data, and sequential flow edges for vulnerability classification. To the best of our knowledge, we are the first to propose poacher Flow edges, where edges are associated with a particular set of potential vulnerabilities.
- We propose a joint interpolated multitask unbiased vulnerability classifier comprising a transformer "RoBERTa" and graph convolution neural network (GCN) with Poacher Flow (PF) edges called RoBERTa-PFGCN, trained using the Focal Loss function to address data imbalance issues. Additionally, RoBERTa-PFGCN provides description and explanation for addressing each vulnerability category. To complement this, we created a large-scale dataset called Vulnerability Finder (VulF) dataset, which contains vulnerability descriptions related to 40 CWE categories.
- We further investigated the effectiveness of our vulnerability classifier by utilizing our proposed dataset VulF and four real-world, large-scale C/C++ vulnerability datasets, including ReVEAL, FFMpeg+Qemu, D2A and MVD. Our experimental results show that our vulnerability classifier outperforms the state-of-the-art results on vulnerability detection with fewer false negatives and fewer false positives.

2. Related Work

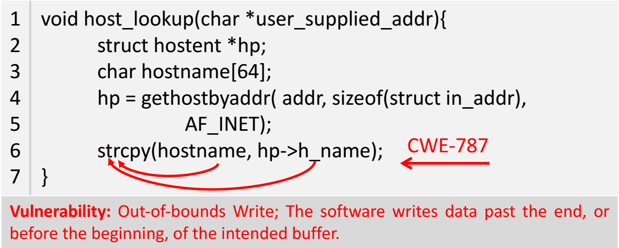
Earlier works on source code vulnerability detection prominently rely on rule-based systems. Engler et al. [24] propose a technique to automatically extract rules from source code without prior system knowledge. One such rule would be that the declaration of *spin lock* must be followed by *spin unlock* in a C/C++ code to work flawlessly. The simultaneous occurrence of these two statements takes place 99% in non-vulnerable code. If these statements do not appear in pairs, it is an indication of a security flaw. Essentially, these systems work by creating a rule template for a system. Based on this hypothesis, the authors implemented six checkers, or rules, to identify bugs in code. Founded on this idea, several static analysis based tools like *Flawfinder* [25], *RATS* [26], *Cppcheck* [27], *Coverity* [28], Infer [29] have been proposed, built on a set of predefined rules to cover a wide range of code vulnerabilities. Since these are rule-based, the rules of these static analyzers need to be updated when a new vulnerability arises and these tools are affected by high false-positive and false-negative rates [30].

The work presented by Lin et al. [31] demonstrates how traditional machine learning (ML) methods offer an alternative to automated vulnerability discovery. In contrast to ML-based vulnerability detection, deep learning-based techniques [32] offer additional possibilities and generalizability. VulDeepecker [33] proposed detecting vulnerabilities using Bi-LSTM and pre-processed source code by generating *Code Gadget*. According to the authors, a *Code Gadget* is a collection of data and control dependency statements. μVulDeepecker [34] proposed a multiclass vulnerability classification method using Bi-LSTM. They classified 40 types of vulnerability, with each type tied to a CWE [35]. Furthermore, Russell et al. [36] and Li et al. [37] proposed a TextCNN based approach to detect vulnerabilities from source code. Their proposed approach considers each token as a word embedded to feed a Convolutional Neural Network for training and inference. In recent years, machine learning and deep learning techniques have also been used to detect vulnerabilities in IoT devices [38].

Each of these works considered source code as an analog to natural language, with some limitations in capturing the correct representation of a source code. Since source code is more structured and logical, Bilgin et al. [14] proposed an AST as a representation technique to detect vulnerability using machine learning. In this approach, the code is converted to an AST. Afterward, to keep the structural information of the code intact, the original AST is converted into a binary AST. The binary AST is flattened using BFS with a CNN for feature generation and classification. Several studies [39], [40], [32] including SySeVR [8], proposed a similar AST based approach with the use of LSTM, Bi-LSTM, or BGRU based methods. VulBERTa [41] RoBERTa [42] and [43] used a transformer-based model to detect vulnerability from source code.

Although these methods consider using AST to capture the syntactical information of a programming language, these are eventually flattened to feed an encoder that yields the desired vulnerability semantic features. Thus, the original graph syntactics are suppressed. To address this issue, Devign [7] proposed using and preserving the structure of Code Property Graphs (CPGs) [15] a combination of AST, data and control flow graph, and natural code sequence by using a GGNN [44] combined with a 1d CNN layer to generate the final embeddings for classification. Chakraborty et al. [10] have proposed a similar method that makes use of CPGs as an input for training a GGNN [44]. ReGVD [45] and LineVD [46] proposed a GCN-based technique for vulnerability detection by creating a graph representation of the source code, and GraphCodeBERT [22] as a tokenizer. Moreover, VELVET [19] proposed an ensemble RoBERTa and Gated Graph Neural Network to detect vulnerabilities. Each of these techniques offer vulnerability detection at a function or file level, which is not ideal from a programmer's perspective. To address this issue, [47], [48] and [49] proposed a method that identifies statements contributing to a vulnerability in order to achieve finer granularity in locating vulnerabilities.

Generating a proper graph representation of a program is significant for program analysis [50]. Other works found in the literature have proposed an AST-based graph representation for code [51]–[53]. Allamanis et al. [51] make use of data flow edges with the original AST graph representation. Alon et al. [54] and [55] have proposed using ASTs with an attention-based network to generate program representation. TYPILUS [56] proposed a graph-based representation similar to [51], with the addition of some new edges to predict variable type in a dynamic language. CodeBERT [57] learns to represent general-purpose representations for programming languages, while GraphCodeBERT [22] and UniXcoder [58] proposes AST graph representation techniques for various programming language-related tasks like code-clone detection, code summarizing, and code translation.



```
void host_lookup(char *user_supplied_addr){
        struct hostent *hp;
        char hostname[64];
        hp = gethostbyaddr( addr, sizeof(struct in_addr),
                    AF_INET);
        strcpy(hostname, hp->h_name);     CWE-787
}
```

Vulnerability: Out-of-bounds Write; The software writes data past the end, or before the beginning, of the intended buffer.

Figure 1: Example of a vulnerability explanation as part of an automated code review process to help developers effectively resolve static software security issues.

To the best of our knowledge, the proposed RoBERTa-PFGCN is the first attempt for code graph representation using programming language structure (data flow, control flow, and sequential flow) combined with Poacher Flow edges to bridge the gap between dynamic and static analysis of a code to improve the performance of code vulnerability understanding.

3. Multitask Vulnerability Definition

Threat Model: We consider two types of software developers: (1) adversaries as potential developers who share vulnerable source code or code snippets on online developer collaboration platforms such as StackOverflow, GitHub, or SourceForge; and (2) developers who leverage vulnerable code discovered on online developer collaboration platforms and incorporate it into their software development projects with minor changes. The adversaries can share functions that have possible attack scenarios, such as Remote Code Execution, Buffer Overflow, and Information Leakage to exploit source code vulnerabilities and take control of the system/application, steal data, or launch further attacks. We aim to detect these vulnerabilities early in the development process. Therefore, our approach uses static code analysis to identify code vulnerabilities before execution. Since we are doing a static code analysis, our decision is solely based on the analysis of source code only. Moreover, our system takes external input into consideration during vulnerability analysis. The attack surface may include input function validation, access control, code injection, and configuration management.

A variety of vulnerability detectors, such as the work presented in [32], perform vulnerability classification at a file-level granularity. In contrast, other tool-based approaches, including *Cppcheck* and *Coverity*, depend on a fully compiled or syntactically correct code to run a vulnerability analysis. Despite the advancements proposed in these works, they are language-dependent approaches that cannot be generalized to other programming languages and most of them need a fully compiled program to work correctly. We are proposing a generic solution without imposing any constraints on the input language programming function. In Figure 1 we present a sample function with its output at the bottom.

A formal problem definition is presented as follows: a source code function and its label pair are defined as $\{(s_i, v_i)|s_i \in S, v_i \in V\}$ and $i \in \{1,2,3,...,n\}$, where S denotes the set of functions that may or may not be compilable as a standalone program, s_i denotes each function, n denotes total number of functions, and $V = \{0,1\}$ denotes the set of labels corresponding to each function, where the subset of vulnerable code is labeled as $V = 1$ and the subset of non-vulnerable code is labeled as $V = 0$.

We further extended the functionality of our model to provide a vulnerability classification to the developer. Thus, we define our function and classification pair as $\{(s_i, c_k)\}$ such that $c_k \in C$ and $k \in \{1,2,3,...,n\}$. $C = \{c_1, c_2, ...c_k\}$ denotes the set of description of the vulnerable function s_i, where k is the total number of descriptions that our vulnerability classifier can provide.Finally, a pre-processing pipeline converts functions s_i into embeddings using a token embedding generator $\hat{E_R}$ such that,

$$e_i = \hat{E_R}(s_i) \tag{1}$$

Thus, the set of all embedded tokens of a code is defined as $E = \{e_1, e_2, ..., e_n\}$. We map embedding E with vulnerable code samples V, and classification C such that, $h_1 : E \Rightarrow V$ and $h_2 : E \Rightarrow D$. Therefore, our vulnerability classification h_1 function with loss function $\mathcal{L}_1(.)$ is formally defined as:

$$\mathcal{L}_1 = \sum_{i=1}^{n}(h_1(\hat{E_R}(s_i), v_i|s_i)) \tag{2}$$

Similarly, we define our vulnerability description function h_2 with loss $\mathcal{L}_2(.)$:

$$\mathcal{L}_2 = \sum_{i=1}^{n}(h_2(\hat{E_R}(s_i), c_i|s_i)) \tag{3}$$

The overall loss function stands as:

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \lambda\frac{1}{2}||w_i||^2$$

Our multitask vulnerability detection function learns to detect and provide vulnerability classification by minimizing the loss $\mathcal{L}$. Here, w_i is an adjustable weight learned during training and λ is a regularization hyperparameter [59].

In our work, we broadly address the following three Research Questions (RQs):

RQ1: Based on our proposed SVG representation, can the classifier learn to identify and provide CWE Numbers of vulnerabilities in real-world source code?

RQ2: Can our classifier learn vulnerabilities in a biased setting?

RQ3: Is our classifier generalized enough to detect vulnerabilities in N-day and zero-day program samples?

4. Methodology

Current graph-based models like CPG and AST generated by tools such as Joern [60] provide a significant amount of information to detect vulnerabilities in a program. However, runtime vulnerabilities may arise due to the dynamic behavior of program during execution and assignments. Figure 1 depicts a declaration of a variable *hostname* (line 3) and its usage (line 6). Although CPGs provide sufficient information regarding the token dependencies of a graph through data flow, there is no guarantee that the hostname in this case won't be longer than 64 bytes. Furthermore, training a transformer-based model with token sequences from source code is limited given that: 1) code follows a strict syntactic structure compared to the structures found in natural languages, 2) a code's execution time output may produce different behaviors for different input and memory states, and 3) long-range dependencies are commonly found in source code.

In order to address these problems, our proposed architecture is composed of three main modules, namely: 1) Semantic Vulnerability Graph of a software program, 2) SVG Node Embedding using RoBERTa, and 3) Multitask RoBERTa-PFGCN. Figure 2 provides an overall architecture of all the mentioned components.

4.1. Semantic Vulnerability Graph of a Program

Our proposed graph representation of a program is denoted as Semantic Vulnerability Graph (SVG). Our SVG is produced via an aggregation of sequential flow edges, control flow edges, data flow edges, and poacher flow edges, a novel edge representing a vulnerability relationship that provides richer information for capturing vulnerability. Each aforementioned element is derived from the same source code. The remaining parts of this subsection provide detailed information on each component used to generate the SVG.

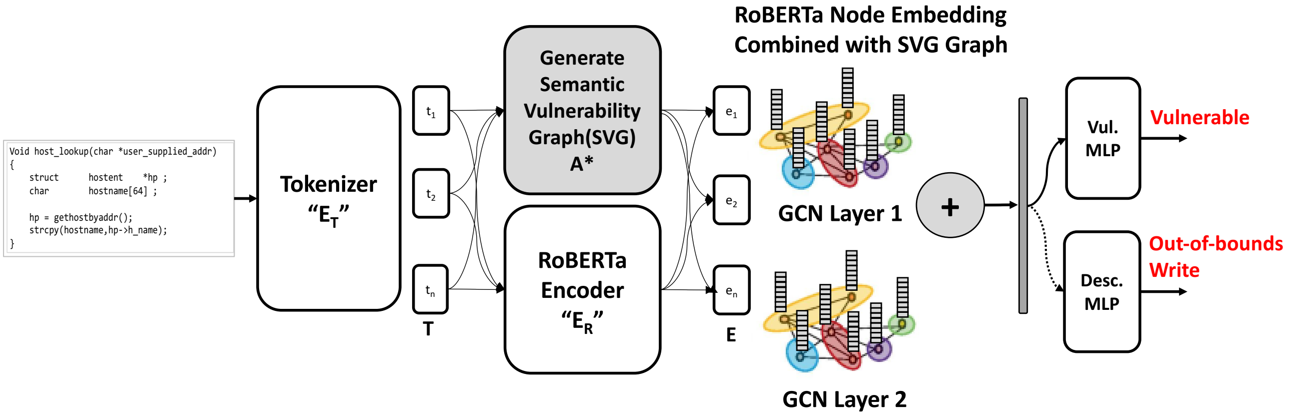
Node Generation Using Tokenizer. A token is a series of characters separated by spaces or punctuation marks generated by a tokenizer. Tokens may take the form of words, integers, real numbers, or a combination of these. However, tokens differ slightly when they are used in Programming Language Processing problem. In programming language, tokens may come in the form of *camelCasing* or *snake_casing*. Consider an example of the token *get_item*. In Natural Language Processing, the tokenizer will separate the word into two tokens, *get* and *item*. However, this combination is treated as a single token since the input is a code. Moreover, other symbols (such as parenthesis, semicolons, etc.) are considered as a single token. Each of these tokens are used as a node of our SVG. When we tokenize our code, it generates three features for each token, the original token itselt, its position, and the token type.

Adjacency Matrix Definition. Let us consider that a graph has an adjacency matrix A where m and n are some arbitrary nodes in the graph and edges are the connection between two nodes. Thus, our adjacency matrix is defined as:

$$A_{m,n} = \begin{cases} 1 \quad if \qquad edge \qquad exists \\ 0 \qquad Otherwise \end{cases}$$

where $A = 1$ indicates an edge exists between two arbitrary nodes m and n and $A = 0$ indicates otherwise.

Data Flow Edges. Show the usage and modification of a variable [7]. Data flow edges are defined as a connection between two variables dependent on each other during value assignment or modification or other usage.



```
Void host_lookup(char *user_supplied_addr)
{
    struct      hostent    *hp ;
    char        hostname[64] ;

    hp = gethostbyaddr();
    strcpy(hostname,hp->h_name);
}
```

Figure 2: Overall Architecture of our Classifier: Our classifier is divided into three parts. Initially, the input source code is pre-processed by creating an SVG. Then RoBERTa layer generates embedding for each token/node of the graph. Finally, the GCN layer takes the node embedding and adjacency matrix for feature generation. Focal Loss forces the model to learn more about the minority class. The MLP layer decides whether a function is vulnerable by leveraging the Focal Loss Function.

Some other usage of the variable may include variable definition, initialization, update, or alteration.

Control Flow Edges. Illustrate the statements or operations executed throughout the program [7]. The alternate execution of statements may be determined by conditional statements (e.g., if/while/switch).

Sequential Flow Edges. Demonstrate the syntactic relationship between the tokens of a program inspired from [61], [62]. Sequential flow edges show the connection of a token with its neighboring tokens. To generate this edge, we create an edge from a token with its subsequent neighboring tokens. The number of subsequent tokens the initial token is connected to is determined during the experiment.

Poacher Flow (PF) Edges. We defined Poacher Flow edges to bridge the gap between dynamic and static analysis of source code. As opposed to programming language structure (data flow, control flow, and sequential flow), PF edges are meant to identify program boundaries, potential corner cases, and external checkpoints. This is accomplished by considering the external environment context in which the program operates, including insecure input handling, the use of unsafe functions, SQL injection, or unauthorized code execution that have just recently been discovered by the CWE community in programs of a similar nature. Our goal is to bridge the gap between dynamic and static analysis of a program by using PF edges. Specifically, PF edges serve as a connection between the knowledge and patterns learned stochastically from known existing vulnerability patterns using labeled data by incorporating PF edges into the machine learning training procedure. We have identified three categories of PF edges: data processing edges, access control edges, and resource management edges. Each edge of these edge categories is discussed in detail in the subsections below. In addition, Algorithm 1 presents the approach for generating all the elements of Poacher Flow Edges.

Data Processing Edge: Data processing vulnerabilities are the most common types during the software development stage. For example, Out-of-Bounds Read is ranked 1 out of the top 25 vulnerabilities from 2022 [35]. Data flow edges are useful for capturing the flow of data, but may not be sufficient for capturing complex data operations, such as memory pointer arithmetic. Additionally, when data manipulation involves APIs such as (*strcpy, read, and write*), data flow edges may fail to capture this information. The data processing edge is an extension to the existing data flow graph, which estimated the potential outcome of various mathematical operations, illegal memory issues, and unsafe API execution, pointer arithmetic. For instance, estimating divide by zero, using an uninitialized variable or using unsafe APIs like *gets()* in C/C++.

Access Control Edge: According to the Open Web Application Security Project (OWASP), software and data integrity failures are ranked among the top ten web application security risks [63]. These attacks take advantage of improper neutralization of special elements in web page output. While programming language structures (data flow, control flow, and sequential flow) cannot discover these vulnerabilities, access control edges can be utilized to address this issue. These edges correspond to external program calls, including application configuration settings that may not be present in the application's source code, such as passing untrusted data as arguments. Other edges include improper control over code generation and improper neutralization of special elements used in SQL commands. By performing conditional edge checks, it is possible to prevent malicious actors from passing untrusted data as arguments.

Resource Management Edge: Software vulnerability may occur when resources are not adequately managed including when a buffer copy is executed without verifying input size, incorrect array index validation, resource exhaustion, utilization of memory after an uncontrolled allocation or incomplete cleanup, or incorrect synchronization of resources within an exclusive operation such as semaphore. These scenarios can be captured by Resource management edges, which can make the classifier aware of potential inadequate resource management operation.

Combining the Edges as SVG. Each edge type is critical for finding vulnerabilities in a function. Data flow

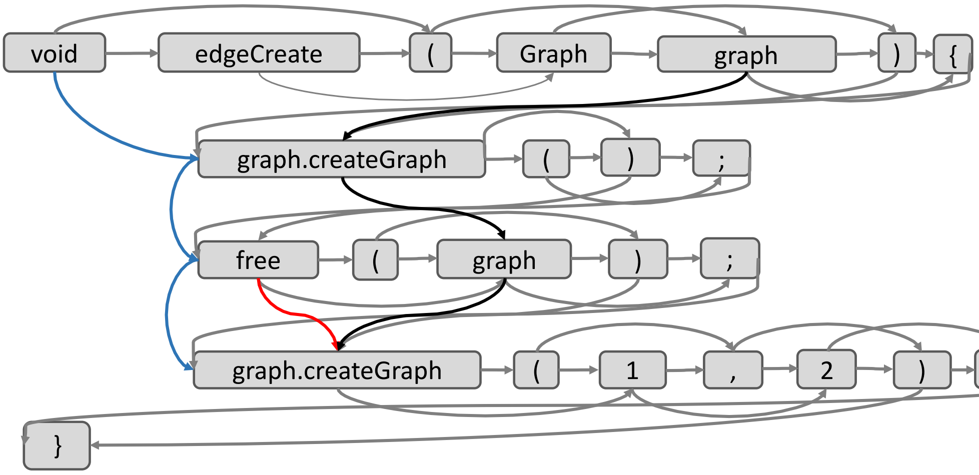


Figure 3: Depiction of our SVG. Each gray box shows individual tokens of our SVG. The red line depicts a poacher flow edge, the black line depicts data flow edges, the blue line depicts control flow edges and the gray line depicts sequential flow edges.

edges (black edges), shown in Figure 3 identify the flow of data for each variable; control flow edges (blue edges) are responsible for the overall flow of programs; sequential flow edges (gray edges) shows the syntactic relationship between the tokens of the program; lastly, Poacher Flow edges (red edges) are meant to bridge the gap between dynamic and static analysis of code by generating edges related to program boundaries, corner cases, and external checks. The SVG is constructed through the combination of data flow, control flow, sequential flow, and poacher flow edges. SVG produces richer semantic and syntactic information necessary for vulnerability detection and classification. Figure 3 presents an example of an SVG composed of 68 edges in total, including 61 sequential flow edges (gray edges), 3 data flow edges (black edges), 3 control flow edges (blue edges), and 1 Poacher Flow edge (red edge).

4.2. SVG Node Embedding using RoBERTa

SVG Node Embeddings. RoBERTa [42] is used to generate embeddings for each token in our graph. RoBERTa was built upon BERT [64], in which the system learns to predict purposefully masked text within unannotated language examples. RoBERTa modifies critical hyperparameters in BERT, such as deleting BERT's next-sentence pre-training target and training with significantly larger minibatch sizes and learning rates. This pre-training technique allows RoBERTa to outperform BERT in terms of the masked language modeling objective and improves the performance of subsequent tasks. However, to tokenize and initialize the node embeddings, a pre-trained variant of RoBERTa presented in GraphCodeBERT [22] is used for source code representation on C/C++. The classifier makes use of word embeddings generated by RoBERTa and embeddings generated by a GCN model fed with heterogeneous SVG. The embeddings E generated by the pre-trained RoBERTa encoder are as follows:

$$E = \hat{E_R}(T)$$

Here, the set of tokens T, where $t_i \in T, i = 1, 2, ..., n$, is the set of n tokens in SVG that are used as the input for the RoBERTa encoder $\hat{E_R}$.

Afterward, an adjacency matrix $A_{m,n}$ is created by using the set of tokens, T, and the connections observed between tokens following our SVG. After this step, the adjacency matrix is converted into a heterogeneous multi-edged graph $\mathbb{G}(T, E, A)$, where $E \in \mathbb{R}^d$ is the d dimensional embedding or feature vector of each token t in the graph. While different edge types compose our SVG, only a single adjacency matrix is used to represent all the edges where a value of 1 is set if any of the edges exist between two tokens and 0 if the edges do not exist.

4.3. Multitask RoBERTa-PFGCN

In an SVG, the existence of specific edges could serve as indicators of the existence of vulnerability. Graph convolution networks (GCN) are designed to comprehend the edge connection between two nodes. GCN is used to capture the relationship between the elements of $\mathbb{G}(T, E, A)$ that are essential for vulnerability detection. GCN is composed of two layers that aggregate vector representations of a node from its neighbors with a residual connection. GCN is formulated as follows:

$$H^{(\mathrm{n}+1)} = \sigma\left(W^n H^{\mathrm{n}} A^*\right) \tag{4}$$

where W^n represents the weights at n-th layer during training and $\mathrm{H^n}$ is the feature representation of nodes at n-th layer. Thus, $H^{(0)} = E$ while A^* is the normalized adjacency matrix. Matrix multiplication is done on W^n, H^n, and A^*, which goes through an activation function σ (e.g., $ReLU$).

The values in the adjacency matrix A are normalized to prevent numerical instabilities, such as vanishing or exploding gradients, that might prevent the model from converging into an optimal solution. The adjacency matrix is normalized using the method proposed by Kipf et al. [65], which performs an inverse dot product operation for normalization. Let us consider $\hat{D}$ as the diagonal node

Algorithm 1 Generating Poacher Flow Edges

```
Input: Source Code
Output: Poacher Flow Edges
procedure POACHER_EDGES($code$)
    $tokens \leftarrow Tokenizer(code)$
    $asst\_operators \leftarrow [=, + =, - =, <<=, ...]$
    $adj\_matrix \leftarrow [][]$
    $stack \leftarrow []$
  ▷ Dictionary Initialized
    $scope \leftarrow \{\}$
    for $token$ in $tokens$ do
  ▷ Data Processing
        if $token.type$ in $asst\_operators$ then
            $left \leftarrow getPrevToken(token)$
            $right \leftarrow getNextTokens(token)$
            $adj\_matrix[left][right] \leftarrow 1$
        end if
        if $token.type$ is "$API$" then
            $params \leftarrow getParameters(token)$
            $adj\_matrix[token][params] \leftarrow 1$
        end if
  ▷ Access Control
        $FunParams \leftarrow getFuncParams()$
        if $token.type$ is "$execution$" then
            if $token$ is not checked before asst then
                $next \leftarrow getNextToken(token)$
                if $next$ is subset $FunParams$ then
                    $adj\_matrix[token][next] \leftarrow 1$
                end if
            end if
        end if
  ▷ Resource Management
        if $scope[token]$ is $end$ then
            $adj\_matrix["free"][token] \leftarrow 1$
        end if
        $stack.push(token)$
        if $pairMatch(token)$ then
            $pair\_token \leftarrow stack.pop()$
        end if
        if $token$ is "$free$" then
            $next\_token \leftarrow getNextToken(token)$
            $scope[next\_token] \leftarrow end$
        end if
    end for
    return $adj\_matrix$
end procedure
```

degree matrix such that, $\hat{D}_{\text{ij}} = \sum_j A_{\text{ij}}$. The degree matrix of a graph is a diagonal matrix that records the degree of each vertex or the number of edges that connect each vertex to another vertex. $\hat{D}$ also contains information about the number of edges attached to each vertex. The normalized adjacency matrix is computed as:

$$A^* = \hat{D}^{\text{-1}}.A$$

which is equivalent to:

$$A^* = \hat{D}^{\frac{-1}{2}}.A.\hat{D}^{\frac{1}{2}}$$

According to the authors in [65], the latter formula is used for better normalization.

Residual Connection. In the work presented by He et al. [66], a residual connection is used to propagate feature representation learned from the (H^n) layer to the next layer $(H^{\text{n+1}})$ by allowing gradients to pass directly from one layer to the next without encountering a vanishing or exploding gradient problem. By adding the residual connection, our model is redefined from Equation 4 as:

$$H^{(\text{n}+1)} = H^{\text{n}} + \sigma\Bigg(W^n H^{\text{n}} A^*\Bigg) \tag{5}$$

After this, two dense parallel layers are added. The first layer consists of two neurons that provide the outcome for vulnerability detection. The second layer consists of 41 neurons that indicated the vulnerability description associated with the detected vulnerability.

Loss Function. Vulnerability in a real-world setting appears highly imbalanced. As a consequence, non-vulnerable code highly outnumbers vulnerable code, thus a classifier is always biased toward the majority class. As a result, usual loss function like CrossEntropyLoss provides higher false-positive and false-negatives. We employ *Focal Loss* [23] to rectify the class imbalances of our datasets. Without this approach, our model would learn biases towards non-vulnerable samples, drastically affecting our classification performance. The Focal Loss is denoted based on cross-entropy (CE) loss for binary classification problems as:

$$CEp, y = \begin{cases} -\log(p) & if \quad y = 1 \\ -\log(1-p) & otherwise, \end{cases}$$

where $y = \{0, 1\}$ denotes the ground truth provided to the classifier during the training process and $p = \{0, 1\}$ is the models output probability for the class $y = 1$, for binary classification. However, we expanded this for multitask classification as well. For convenience, probability distribution p_t is defined as:

$$p_t = \begin{cases} p & if \quad y = 1 \\ (1-p) & otherwise, \end{cases}$$

Focal Loss integrates a weighing factor $\alpha \in [0, 1]$ and defines the mathematical expression of Focal Loss for a binary classification problem. Thus, the balanced CE loss can be rewritten as:

$$CE(p_t) = -\alpha \log(p_t) \tag{6}$$

Vulnerability classification without the loss function shows that the classifier can be confused by the majority class, which also dominates the gradients. Although α balances majority and minority examples, it does not differentiate between easy (positives/negatives samples that are predicted as positive/negative) and hard examples (positives/negatives samples that are misclassified as negative/positive). To overcome this issue, a modulating factor δ is used with the cross-entropy loss to down-weight easy examples, which forces the model to be trained more precisely on hard negatives. By combining weight balance and Focal Loss, our final Focal Loss function from Equation 6 is defined as follows:

$$FocalLoss(p_t) = -\alpha(1-p_t)^{\delta} \log(p_t) \tag{7}$$

Where, γ is an adjustable parameter and $\gamma \geq 0$. Figure 3 shows the overall architecture of our proposed vulnerability classifier.

Complexity analysis. Algorithm 1 provides the order of logics to create our graph. Given a sample code as input, RoBERTa Transformer is used to tokenize the code snippet to generate n tokens. Thus, the time complexity to generate n tokens is $O(n)$. In order to generate each Poacher Flow edge, the tokens are iterated once and all edges are created in a single pass. We generated data flow, control flow, and sequential flow edges in a single pass by iterating over n tokens, hence the time complexity is $O(n)$. As a result, the overall time complexity to generate our complete SVG is $O(n)$. However, other graph-based analysis [58] [22] [7] consist of generating an AST, which can be very time consuming. For example, for the same program with n tokens, the time complexity to insert a single token into an AST is $O(\log n)$ on an average case when the tree is balanced. However, when the tree is imbalanced, the time complexity to insert a single element is $O(n)$. Thus, the time complexity to generate an AST by inserting n elements in a balanced tree is $O(n \log n)$ and in an imbalanced tree is $O(n^2)$, which is much higher than our proposed SVG.

5. Experiments And Discussions

Our experiments were conducted using highly balanced, mildly unbalanced, and highly unbalanced datasets, as well as real-world N-day and zero-day program samples collected from publicly available resources for evaluation purposes. Our experiment was designed to evaluate three metrics in mind: our model's ability to classify vulnerabilities with their corresponding descriptions, our model's ability to handle a biased dataset, and the importance of detecting N-day and zero-day programs.

We tested our model's vulnerability classification across various experimental settings in which each experimental subset was chosen to resolve its respective Research Question (presented in Section 3). In addition, we provided an ablation study to observe our model's performance by adding sub-components of PF edges to observe their impact on vulnerability detection. The remaining content of this section is divided into the following subsections: Datasets, Data Pre-Processing, Performance Evaluation, Time and Memory analysis, and Ablation Studies.

5.1. Datasets

We utilized different datasets that included highly balanced, mildly unbalanced, and highly unbalanced data. Particularly, we utilized the large-scale MVD [34] dataset since it includes both vulnerability data and a CWE number for each source code function. This dataset is comprised of a huge number of real-world and synthetic vulnerability samples, and it is mildly unbalanced. FFMpeg+Qemu [7] and D2A [67] are two more balanced real-world datasets we have included. We also utilized the ReVEAL [10] dataset, a highly unbalanced real-world dataset with a non-vulnerable to vulnerable data sample ratio of 9:1. Lastly, we created VulF by aggregating publicly accessible source code from GitHub and the National Vulnerability Database. In addition, we used samples from other existing datasets to build a wild dataset that accommodates the necessary subpopulation shift [13], [68] for detecting real-world vulnerabilities. Table 2 provides a brief description of the datasets.

Vulnerability Finder (VulF). We created VulF, a large-scale dataset comprised of data from multiple publicly available sources. We started Vulf from data collected from the National Vulnerability Database [69], consisting of vulnerable and non-vulnerable code samples. Each vulnerable source code was mapped to a Common Weakness Enumeration (CWE) number. A list of the most prevalent vulnerability categories was built; yet, only source code samples written in C/C++ were kept in our dataset. For example, CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer, defines a software vulnerability that occurs when software reads or writes data past the specified buffer's limit or after its specified size. CWE-020 (Improper Input Validation), another top vulnerability, refers to a program that accepts input or data but fails to validate it before use. As a result, an altered control flow, arbitrary control of a resource, or execution of arbitrary code may occur. Additionally, we combined data from μVulDeePecker's MVD dataset [34]. In order to make our dataset more robust, we implemented Code Reformatting, Beautification, Dead Code Elimination, Variable Renaming, Identifier Mangling, and Dead Code Insertion techniques presented by Jain et al. [70] to generate synthetic data.

Furthermore, we enhanced our Vulf dataset by generating descriptions linked to each CWE. To ensure the usefulness of our descriptions from developers' perspective, we engaged a group of junior programmers with limited knowledge of software security vulnerabilities. They validated the effectiveness and usefulness of our CWE descriptions for vulnerability root cause analysis and code fixes. Their feedback helped us refine our CWE descriptions to ensure they are useful and effective for developers seeking to fix source code vulnerabilities. The collected data was divided into 40 vulnerability categories and one benign category. Each vulnerable function was labeled with its corresponding CWE number obtained from [35] and description. We present further information regarding our dataset in Table 1.

ReVEAL [10]. The ReVEAL dataset was curated by Chakraborty et al. [10] by tracking vulnerabilities in two open-source projects: Chromium and Linux Debian. Chromium is an open-source project of Chrome. The authors crawled Bugzilla and Linux Debian Kernel via the Debian Security Tracker to generate their dataset. This dataset reflects the 9:1 ratio of vulnerable to benign code described in Table 2.

FFMpeg+Qemu [7]. The FFMpeg+Qemu dataset is a collection of real-world source code vulnerability detection data compiled by Devign. [7]. This dataset contains four repositories, Linux Kernel, Qemu, Wireshark and FFMpeg, but the authors only share the FFMpeg and Qemu datasets publicly. Their data annotation was conducted using the Commit Filtering approach proposed by [71], and manual verification was completed by four experienced security researchers for final verification, devoting 600 person-hours to the task.

TABLE 1: Summary of our VulF dataset with total number of functions for each CWE labels with short description.

CWE Short Description	# of Functions	CWE Short Description	# of Functions
Non-Vulnerable-N/A	115550	467 - Using of null pointer	508
020 - Process data without validation	70	469 - Incorrectly determining pointers size	1701
020, 665, 400 - Consuming resource without control	365	476 - Trying to access a dereferenced pointer	421
074 - Injection of foreign code	1640	506 - Containing malicious code	102
119 - Performing read/write outside buffer	4713	573 - Calling an API incorrectly	221
119, 666, 573 - Operation outside memory buffer	305	662, 573 - Calling API without sync.	331
138 - Unchecked use of special elements	200	573, 666 - Incorrectly following configuration	306
170 - Incorrectly terminating a string	1006	610 - Use of externally controlled resource	309
187 - Incorrect comparison of string	506	662 - Incorrect synchronization of thread	307
190 - Integer overflow by mathematical operation	326	665 - Incorrect initialization of a resource	305
191 - Integer underflow by mathemetical operation	68	666 - Performing operation on resource in wrong lifetime	734
221 - Misinterpret records	60	668 - Exposing resource to incorrect sphere	805
311 - Product does not encrypt critical data	581	670 - Malicious Incorrect Control Flow	601
327 - Use of risky algorithm or protocol	35	673 - Changing control sphere by external party	61
362 - Concurrent execution of shared resources	211	676 - Use of dangerous function	1380
369 - Division ob zero error	289	704 - Incorrectly converting type of a resource	480
400 - Consuming resource without limit	107	706 - Accessing incorrectly resolved resource	206
400, 404 - Release consuming resource	1200	754 - Improper check on unusual exceptions	78
400, 665 - Consuming uninitialized resource	1560	758 - Improperly use of API	59
404 - Release a resource incorrectly	508	834 - Iteration of a loop eccessively	210
668, 404 - Improper use of Resource	2860		
			Total = 141285

D2A [67]. IBM Research has assembled a real-world vulnerability dataset, D2A [67]. They included open-source projects like FFMpeg, OpenSSL, httpd, NGINX, libtiff, and libav in this dataset. This dataset was created using a differential analysis-based method, wherein the authors initially filtered the commit messages to sort out potentially vulnerable commits before using three static analyzer tools, *CppCheck*, *FlawFinder*, *Clang Static Analyzer*, and *Infer*, for a two-way checking.

MVD [34]. MVD curated by Zou et al. [34] is a multiclass vulnerability dataset of 40 vulnerable classes and one benign class. The datasets were collected from NIST [72], and SARD [73]. A significant number of program samples of this dataset consist of synthetic, vulnerable, and non-vulnerable code examples. Table 2 briefly describes the number of benign and vulnerable, their ratio, and the total number of code samples for each dataset.

5.2. Data Pre-Processing

Graph Data Preparation. After collecting the datasets, they were converted into an SVG for further analysis. To convert the program into a graph, we add a starting $\langle s \rangle$ and ending token $\langle \backslash s \rangle$ at the beginning and end of each program. Then we convert the code into a sequence of tokens using a RoBERTa tokenizer, pre-trained on C/C++ programs for code representation. Next, we convert the sequence of tokens into SVG which a sequential flow, data flow, control flow, and poacher flow edges are generated. Finally, Algorithm 1 creates an adjacency matrix of shape $n \times n$, where n is the total number of tokens in the graph. As a second step in creating the graph, each node of our SVG was encoded by generating a word embedding of size 768 using pretrained RoBERTa.

TABLE 2: Overview of datasets utilized for training and testing, encompassing highly balanced, mildly unbalanced, and highly unbalanced sets.

Data Source	Benign	Vulnerable	Total	Ratio
D2A [67]	3222	3506	6728	∼1:1.08
FFMpeg + Qemu [7]	14854	12460	27314	∼1.19:1
MVD [34]	138522	43119	181641	∼3.2:1
VulF (Ours)	115550	25735	141285	∼4.5:1
ReVEAL [10]	20494	2240	22734	∼9:1

SVG Analysis. Table 3 presents a comparison between the number of edges and nodes generated by SVG to those generated by an AST in three different sample codes. These code samples were selected randomly from the VulF dataset. As can be observed, SVG generates more edges than an AST. During the generation of the AST, we observed that several intermediate nodes were generated which removed parentheses. Due to this, the edges coming in and out of these intermediate nodes of AST contain no information about the code snippet's vulnerability, regarding these edges useless. In comparison, our SVG retains the parentheses, semicolons, and all other symbols of a programming language as nodes within the graph. Given this additional relationship between tokens, our classifier

TABLE 3: Three sample codes were randomly selected from our VulF dataset to show how the number of nodes and edges of AST compares with our graph.

Code Sample	SVG (Our Graph)		AST Graph		Edge Ratio
	Node	Edge	Node	Edge	
Example 1	3660	8516	2875	5748	1.5:1
Example 2	3324	6640	2415	4828	1.4:1
Example 3	4056	8114	3909	7816	1:2.5

can attain a higher comprehension of the code snippet semantically and syntactically.

5.3. Performance Evaluation

We randomly split all datasets with a ratio of 80:10:10 for training, validation, and testing. We implemented RoBERTA-PFGCN with a 12-layer RoBERTa encoder to generate the token embeddings. We also completed a time and memory analysis using RoBERTa-AST, in which we generated an AST instead of SVG for our GCN. To generate features, the tokens were converted into their equivalent SVG representation. We created a two-layer graph convolutional neural network with a residual connection from the first layer's input to the second layer's input. The dimensionality of the hidden layer is set to 128, with a learning rate of 5e-4 when training a balanced dataset. When we used ReVEAL, the highly unbalanced dataset, we used 1e-5 as the learning rate and trained the model for 100 epochs with a batch size of 512, while the maximum token length was set to 400. We trained our model on 8 DGX-A100 NVIDIA GPU, wherein each model training and testing session took 4-6 hours to complete due to the data size.

Evaluation Metrics.. Our work was evaluated using four metrics: Accuracy, Precision, Recall, and F1. Our model's predictions were categorized as True Negative (TN), True Positive (TP), False Negative (FN), and False Positive (FP). TP refers to samples correctly classified as vulnerable, FP to samples incorrectly classified as vulnerable, TN to samples correctly classified as benign, and FN to samples incorrectly classified as benign. Using these statistics, we compute the Precision as $P = \frac{TP}{TP+FP}$, Recall as, $Recall = \frac{TP}{TP+FN}$, and the F1 score as, $F1 = 2 \times \frac{Precision*Recall}{Precision+Recall}$.

The remainder of this section will outline experiments designed and performed to explore each research question.

RQ1: Based on our proposed SVG representation, can the classifier learn to identify and provide CWE Numbers of vulnerabilities in real-world source code?

A vulnerability classification system should generate semantic features to detect vulnerable patterns from source code and classify vulnerabilities with higher accuracy and lower false positive and false negative rates. To test the effectiveness of our classifier, it was trained and evaluated with the five datasets mentioned earlier.

In order to test how our vulnerability model detect and classify each type of vulnerability, our model was tasked with providing a CWE number and a CWE description for each type of vulnerability. As a result, the software developer can properly understand the detected type of vulnerability. For the multiclass vulnerability classification task, we used our curated dataset VulF and MVD. We also tested our classifier's ability to separate vulnerable and non-vulnerable classes as a binary classification task. We tested the performance of binary classification using three datasets; namely, FFMpeg+Qemu [7], ReVEAL [10], and D2A [67].

TABLE 4: Comparison of our proposed RoBERTa-PFGCN vulnerability detection model against top recent models, including μVulDeePecker, BiLSTM, TextCNN, RoBERTA, CodeBERT, Devign, and VELVET, with a focus on their respective datasets.

Data	Model	Acc.	Prec	Recall	F1
VulF	μVulDeePecker	78.35	78.94	78.30	77.10
	Devign	84.55	83.94	83.15	84.30
	VELVET	84.45	85.12	85.20	84.46
	RoBERTa-PFGCN	**96.24**	**96.18**	**95.15**	**95.85**
MVD	μVulDeePecker [34]	-	-	-	94.22
	RoBERTa-PFGCN(Ours)	98.23	98.28	98.23	**98.01**
FFMpeg + Qemu	BiLSTM [74]	59.37	-	-	-
	TextCNN [74]	60.69	-	-	-
	RoBERTa [74]	61.05	-	-	-
	CodeBERT [74]	62.08	-	-	-
	Devign [10]	58.57	53.60	62.73	57.18
	RoBERTa-PFGCN(Ours)	**63.29**	**63.08**	**62.97**	**62.99**
D2A	VELVET [19]	59.3	**70.5**	50.4	58.8
	RoBERTA-PFGCN(Ours)	**61.2**	61.97	**62.07**	**61.21**

Discussion. In order to analyze the performance of our multitask vulnerability description model, our model was tasked with classifying 40 categories of vulnerability from VulF dataset where each category is associated with a CWE number and a description. For multiclass vulnerability classification, the goal of the classifier initially is to detect whether vulnerability exists in the code, and if a vulnerability exists, provide the CWE number as depicted in Figure 1. This experiment tests our classifier's ability to detect and classify vulnerable code samples . Table 4 shows a comparison of the classification performance of various models against ours when tested with different datasets. Table 1 shows that our VulF dataset is mildly imbalanced. For example, the number of vulnerable codes for CWE-676, CWE-362, and CWE-662 is deficient, less than 300. On the other hand, there are 4713 code samples for CWE-119 and 1380 for CWE-704. We prevented training a biased model by using the Focal Loss function during the model's training.

We tested VulF dataset with our proposed RoBERTa-PFGCN and two other publicly available models, Devign and VELVET. Moreover, we also implemented μVulDeePecker [34] ourselves based on their proposed architecture since their model is not publicly available. Our experimental results show that our model achieves the highest accuracy compared to Devign, VELVET, and μVulDeePecker. Table 4 shows that our model improved accuracy, precision, recall, and F1 score by 11.79%, 11.06%, 9.98%, and 11.39%, respectively, which is at least 11% higher than other models.

Our work was also compared with the results from μVulDeePecker [34], which evaluated 40 classes. Table 4 shows that our model outperforms μVulDeePecker on their MVD dataset by almost 3.80% on F1 score. We provide an F1 score comparison for all 40 classes by our model against μVulDeePecker with the MVD dataset in Figure 4. This bar chart shows a comparative analysis of the F1 score with our model vs. μVulDeePecker, and we

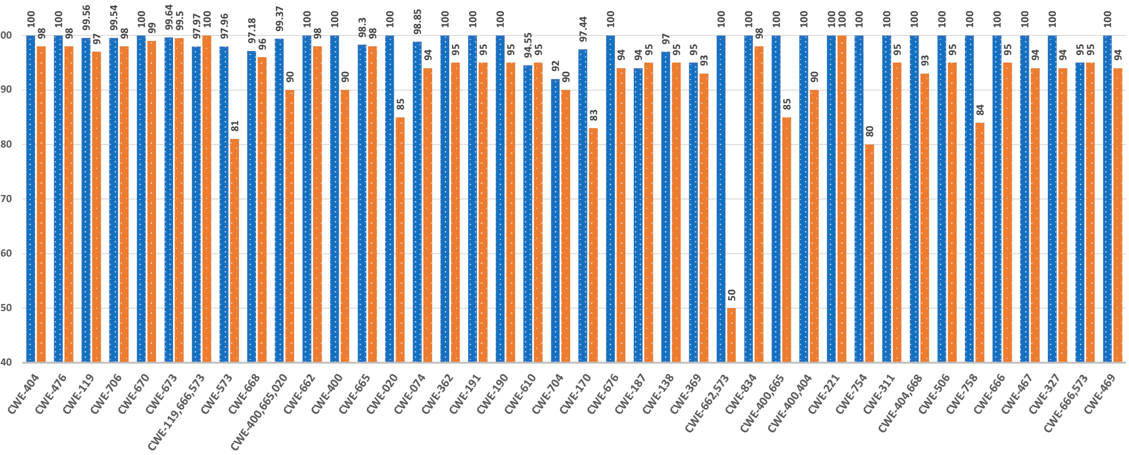


Figure 4: CWE class-by-class F1 score comparison on our proposed Model (RoBERTa-PFGCN), vs. μVulDeepecker on MVD dataset provided by μVulDeePecker including 40 CWE classes. The blue bar corresponds to RoBERTa-PFGCN, while the orange bar represents μVulDeepecker.

can clearly see our model consistently generates higher F1 scores for all the 40 CWE classes.

Table 4 demonstrates our model's performance with two other datasets. These results also illustrate that baseline models like BiLSTM and TextCNN significantly underperform compared to the pre-trained programming language (PL) models like CodeBERT and RoBERTa, as well as our proposed model Multitask RoBERTa-PFGCN, with the FFMpeg+Qemu dataset. Compared to non-PL-based models, our model shows an improvement of 3.92% and 2.60% over BiLSTM and TextCNN and 2.24%, 1.21%, and 4.72% compared to RoBERTa, CodeBERT, and Devign, respectively.

For the D2A dataset, we compared our work with VELVET [19], and see an improvement in classification Accuracy by 1.9%, Recall by 11.67%, and F1 score by 2.41%.

RQ2: Can our classifier learn vulnerability in a biased setting?

An important limitation of real-world source code data is its imbalanced nature. Since real-world projects have very few vulnerable but very many non-vulnerable programs, vulnerability models suffer from data imbalance [10]. The imbalance scenario renders the model more biased towards the non-vulnerable class rendering a higher accuracy with lower precision and recall scores. Of the datasets we have discussed thus far, ReVEAL is highly imbalanced; hence have used it to test against biased settings. A sound vulnerability system should not have a poor F1, Precision, and Recall score despite a potentially high Accuracy, as this would prove bias towards the majority class. We investigated the usefulness of the Focal Loss [23] function and how they prevent biases in the model. For this experiment, we fixed the weight hyperparameters $\alpha = 0.1$ and $\gamma = 2$ for ReVEAL dataset with our RoBERTa-PFGCN model and SVG as input. Since Focal Loss uses a cross-entropy loss function underneath its implementation, we set the learning rate to $1e-5$ with a batch size of 256.

Discussion. We used the ReVEAL dataset for this experiment, which has a 1 to 9 ratio of vulnerable to non-vulnerable code. Table 5 shows that the works proposed by Russell et al. [36], VulDeepecker [33] SySeVR [8] and Devign [7] achieve high accuracy, but their Precision, Recall, and F1 scores drop significantly.To overcome this issue, Focal Loss is used to add more weight to the loss if the model incorrectly predicts the minority class. Thus, we set the value of weight α by taking an inverse ratio of vulnerable to non-vulnerable example codes during training and we set the value of γ to 2 by experimental analysis. However, γ is used as an exponent in Equation 7, so when we run RoBERTa-GCN w/ WL, γ is set to 0 in order to ignore its effect. Observing the last two rows in Table 5, we find that, initially, we tested our model using Weighted Loss (WL) only, and later we tested with Focal Loss, which is a combination of weighted loss α with the hyperparameter γ.

In both cases, we observe that our model has surpassed previous models in terms of Precision, Recall, and F1 score, indicating lower false positive and false negative rates with WL and with FL. However, these numbers improved slightly when compared with performance between weighted loss and Focal Loss, with the exception of the precision metric. From the results, it is observed that with Focal Loss, we achieved an improvement of 11.18% in Precision, 1.06% in Recall, and 0.61% on our F1 score compared to the weighted loss. Nevertheless, using either weighted loss or Focal Loss, the accuracy of our model drops by almost 1% compared to [36], indicating that these previous models are highly biased with an assumption of non-vulnerability. Compared to the next best model [7], our model with Focal Loss shows an improvement of 22.91% in Precision, 23.53% in Recall, and 18.04% on F1 score. Thus, Focal Loss improves precision, recall and F1 on imbalanced data while not causing a negating impact on a balanced dataset only by adjusting the parameters α and γ

RQ3: Is our classifier generalized enough to detect vulnerabilities in N-day and zero-day program sam-

TABLE 5: Vulnerability Classification Using Focal Loss. This table shows the effectiveness of our model using the Focal Loss function. Since only ReVEAL is imbalanced, this dataset is used for result comparison. Results are taken from [10].

Model	Input	Acc.	Prec.	Recall	F1
Russell et al. [36]	Token	**90.98**	24.63	10.91	15.24
VulDeePecker [33]	Slice + Token	89.05	17.68	13.87	15.7
SySeVR [8]		84.22	24.46	40.11	30.25
Devign [7]	CPG	88.41	34.61	26.67	29.87
RoBERTa-PFGCN w/ Weighted Loss	Semantic Vulnerability Graph	88.07	46.34	49.14	47.31
RoBERTa-PFGCN w/ Focal Loss	Semantic Vulnerability Graph	89.88	**57.52**	**50.20**	**47.91**

TABLE 6: N-day and zero-day comparison of our work with previous works.

Model	N-day	Correctly Predicted	zero-day	Correctly Predicted
Devign	273	202	4	1
VELVET		208		1
RoBERTa-PFGCN		**255**		**4**

ples?

We evaluated our classifier's performance based on its ability to accurately predict vulnerability with 273 N-day real-world sample programs. These sample programs are never used during training. We also used 4 zero-day examples in order to evaluate our classifier on predicting zero-day vulnerabilities as well. The classifier predicts the vulnerability classif the vulnerability exists in the code and predicts non-vulnerable when the vulnerability does not exist. Out of these 273 N-day and 4, zero-day code samples, some vulnerability classes exist that are not part of our VulF dataset from table 1. For example, the VulF dataset has no code samples for CWE-787 for training. But a few samples from our VulF dataset have code samples for CWE-787. So when we evaluated our classifier on 273, N-day and 4, zero-day code samples,our model could not predict the vulnerability classification of that particular case. Table 6 provides a more in-depth analysis of our work with the recent models like μVulDeepecker [34], Devign [7] and VELVET [19]. We can see that our model was able to detect most N-day and zero-day vulnerabilities compared to previous works.

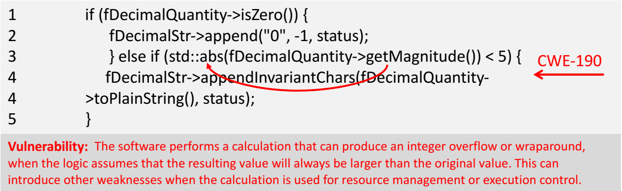
Discussion. In this experiment, we observed that our model achieves an accuracy of 93.00% when trained with our VulF dataset and tested against N-day sample programs of 273 examples. Out of the three datasets that we have used for different experiments, the VulF dataset shows the best performance when we trained our model with SVG. Out of the 273, N-day code samples our model can successfully predict 255 as vulnerable, achieving an accuracy of 93%. Moreover, for zero-day analysis, we tested the same model for 4, zero-day examples, and out of those, our model was able to predict all examples correctly. Furthermore, we provide four case studies, 3, N-day and 1, zero-day program samples, with a critical analysis and the reasoning behind our classifier's outcome.

Case Study 1. We collected this N-day sample program from International Components for Unicode (ICU) repository. In this example, the program in Figure 5 attempts to get the value of $fDecimalQuantity$ without checking the possible integer limit. As a result, a potential buffer overflow could potentially crash the program when the abs function is called. The red edge in Figure 5 shows the Data Processing Edge, which hints to the classifier that a vulnerability may exist. Thus, the classifier emphasized the information provided by this edge, detected the code as vulnerable, and classified the vulnerability as CWE-190.

Case Study 2. We collected this N-day sample program from the FFMpeg repository from GitHub. Here the method $avformat\_new\_stream$ is called without checking the possible value of the output. The output can potentially trigger a null pointer dereference error causing the application to crash. When this code goes through our classifier, the classifier observes the PF edge, observed in red in Figure 6 (Data Processing edge), and classifies this code as vulnerable (CWE-476).

Case Study 3. This zero-day vulnerable example is also part of the TensorFlow C repository. Figure 7 shows a fraction of a large function used during evaluation. In line 4, the code tries to create a mutex lock by invoking $mutex\_lock\_session$. However, till the end of the function, there was no call to unlock the resource $it.first->mu$. As a result, this resource is locked indefinitely, causing a resource management issue. Since a Resource Management edge exists for this situation, our classifier detects the vulnerability and classifies it as CWE-404, which eventually creates a deadlock situation.

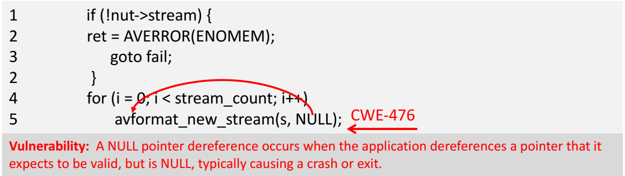
Case Study 4. We collected this N-day, sample program from Linux repository. From GitHub commit messages, we learned that this vulnerability occurs when the function attempts to write data expanded into a page but fails to set up the *inode*, which serves as a unique identifier for information on a specific filesystem. Consequently, a null pointer dereferencing error could occur during writeback if the inode is not created. Figure 8 shows that the statement *writepage* attempts to write a page using the parameters *page and udf_wbc*. Nevertheless, a checking statement (line 5) already exists before assigning a value in line 7. Our classifier identifies the code snippet as non-vulnerable as it is aware that the



```
if (fDecimalQuantity->isZero()) {
    fDecimalStr->append("0", -1, status);
    } else if (std::abs(fDecimalQuantity->getMagnitude()) < 5) {
   fDecimalStr->appendInvariantChars(fDecimalQuantity-
>toPlainString(), status);
}
```

Vulnerability: The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.

Figure 5: An example code for CWE-190, which our classifier predicted accurately. The red edge shows a Poacher Flow edge that captures the Data Processing of the code. Hence, our classifier was able to detect the vulnerability with a description.



```
        if (!nut->stream) {
        ret = AVERROR(ENOMEM);
            goto fail;
         }
        for (i = 0; i < stream_count; i++)
             avformat_new_stream(s, NULL);
```

CWE-476

Vulnerability: A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

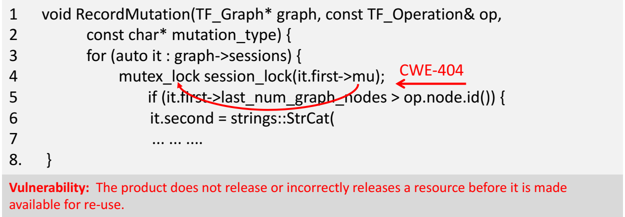
Figure 6: A sample code for CWE-476, which our classifier accurately predicted. The red edge shows a Poacher Flow edge that captures the Access Control of the code. Thus, our classifier was able to detect the vulnerability with a description.

parameters are not null. However, a different API call (*filemap_fdatawrite*) generates this vulnerability (CWE-476) as a result of an inappropriate request for the overall task. Since no logical Poacher edge could be identified for this vulnerability and the other edge do not contribute to detecting vulnerabilities from the implementation of API $writepage$, our model incorrectly classifies this code as non-vulnerable.

5.4. Time and Memory Analysis

In this section, we analyze model complexity in terms of memory consumption and processing time. We assessed our classifier using AST and SVG. We generated an AST using a tool called TreeSitter [75]. Table 7 demonstrates that AST generation incurs significant time and memory overhead. It takes approximately 18 minutes to construct the AST of 141285 functions but only 2 minutes and 32 seconds to generate our SVG for the same VulF dataset. We observe a similar case for the ReVEAL and the D2A datasets. In the ReVEAL dataset, the AST input preprocessing time is 13 times higher than the creation time of our SVG. In the D2A dataset, the AST pre-processing time is 16 times higher. Finally, for the MVD dataset, the AST pre-processing time is 11 times higher.

Furthermore, we discovered high memory overhead issues during the creation of the ASTs. For their generation, a process must traverse the tree recursively in order to build it. As a result, the internal stack expands exponentially as the size of the program increases. In comparison, our SVG doesn't rely on a recursive program for its generation. We perform our analysis in Python and Pytorch. Python's standard stack size is 1,000. However, the stack size had to be increased to 3,000 to build ASTs for all the functions presented in the different datasets we used. A larger stack size causes at least three times more memory consumption when the function grows too large.

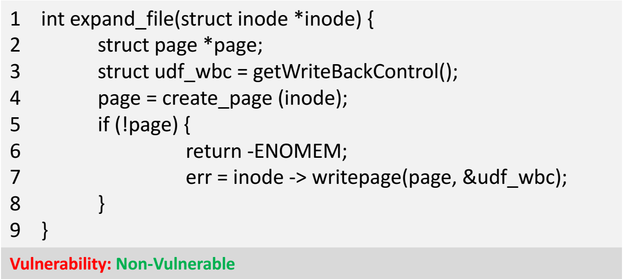


```
void RecordMutation(TF_Graph* graph, const TF_Operation& op,
        const char* mutation_type) {
        for (auto it : graph->sessions) {
              mutex_lock session_lock(it.first->mu);
                   if (it.first->last_num_graph_nodes > op.node.id()) {
                   it.second = strings::StrCat(
                    ... ... ....
 }
```

CWE-404

Vulnerability: The product does not release or incorrectly releases a resource before it is made available for re-use.

Figure 7: A sample code for CWE-404, which our classifier accurately predicted. The red edge shows a Poacher Flow edge that captures the Resource Management of the code. Thus, our classifier was able to detect the vulnerability with a description.



```
int expand_file(struct inode *inode) {
          struct page *page;
          struct udf_wbc = getWriteBackControl();
          page = create_page (inode);
          if (!page) {
                         return -ENOMEM;
                         err = inode -> writepage(page, &udf_wbc);
          }
}
```

Vulnerability: Non-Vulnerable

Figure 8: An example code for CWE-476 that our classifier could not predict accurately. No poacher edges exist for this code. Hence our model predicted it as Non-Vulnerable

5.5. Ablation Studies

We evaluated our model's performance with and without the influence of Poacher Flow (PF) edges. As observed in Table 8, we trained and evaluated the performance of our classifier on four datasets to ensure that the observed performance improvement was not a coincidental occurrence. The experimental hyperparameters were similar across the experiments we performed with the exception of the hyperparameters tailored for **RQ2**. Initially, we trained and evaluated two models, RoBERTa-PFGCN and RoBERTa-GCN, with and without PF edges, respectively. Afterwards, we trained new versions of the RoBERTa-GCN model by adding each component of our PF edges, such as the *Data Processing (DP) Edge*, *Access Control (AC) Edge*, and *Resource Management (RM Edge* to understand the contribution that each PF component had on the model's performance. As seen in Table 8, our proposed RoBERTa-PFGCN vulnerability detection model achieves better performance when trained with PF edges resulting in an 8.74% improvement in accuracy and 8.62% in F1 score when trained and tested using the VulF dataset.

TABLE 7: Comparing the execution times of Generating SVG and AST Graph

Training Data	Graph	Execution Time
VulF	SVG AST	2m 32s 18m
MVD	SVG AST	28m 53s 3m 18
FFMpeg+Qemu	SVG AST	38s 7m 20s
D2A	SVG AST	11s 2m 52s
ReVEAL	SVG AST	20s 4m 15s

TABLE 8: In-depth ablation study for each component of our proposed PF Edges

Dataset	Model	Acc.	Prec.	Recall	F1
VulF	GCN	87.50	87.36	87.94	87.23
	GCN w/ AC	91.23	91.75	91.56	91.23
	GCN w/ DP	90.40	89.65	90.95	90.50
	GCN w/ RM	89.54	90.28	90.61	90.83
	RoBERTa-PFGCN	**96.24**	**96.18**	**95.15**	**95.85**
MVD	GCN	86.10	85.75	85.24	86.50
	GCN w/ AC	92.16	91.90	91.10	91.70
	GCN w/ DP	91.70	90.15	91.57	90.30
	GCN w/ RM	91.46	92.40	91.43	91.33
	RoBERTa-PFGCN	**98.23**	**98.28**	**98.23**	**98.01**
FFMpeg + Qemu	GCN	56.34	57.47	57.28	56.57
	GCN w/ AC	60.94	60.12	60.35	60.48
	GCN w/ DP	61.50	60.51	61.23	60.80
	GCN w/ RM	61.90	61.08	61.40	61.74
	RoBERTa-PFGCN	**63.29**	**63.08**	**62.97**	**62.99**
D2A	GCN	57.95	58.80	57.69	58.46
	GCN w/ AC	61.29	62.50	61.58	62.20
	GCN w/ DP	61.65	60.10	61.95	60.28
	GCN w/ RM	60.30	59.28	59.10	59.25
	RoBERTa-PFGCN	**61.20**	**61.97**	**62.07**	**61.21**

6. Conclusion and Future Work

This paper employed a unique set of edges, including novel Poacher Flow edges to generate richer vulnerability detection and description features. With Poacher Flow edges, our classifier can detect vulnerability that may arise due to the dynamic behavior of a program during execution and assignments. We propose a set of algorithms to generate PF edges from source code.We used a classification model for detecting and classifying source code vulnerabilities based on Multitask RoBERTa-GCN with a Focal Loss and their corresponding classification. We utilized Focal Loss to rectify our model's bias toward the majority class, decreasing our model's false positives and false negatives and resulting in state-of-the-art source code vulnerability detection for real-world projects.We also provided an in-depth ablation study for evaluating the performance impact that each component of our PF edges has on our model. In addition, we performed a time and memory analysis for the generation of ASTs and our SVG. Finally, we introduced the VulF dataset which provides software developers with vulnerability detection and CWE description, helping them resolve source code vulnerability issues. Our future work will focus on reasoning and counterfactual explanations for code vulnerability localization and corrections.

Acknowledgements

This research project and the preparation of this publication were funded in part by the National Science Foundation under Grant No. 2230086.

References

[1] IBM. Compromised employee accounts led to most expensive data breaches over past year. *https://newsroom.ibm.com/2020-07-29-IBM-Report-Compromised-Employee-Accounts-Led-to-Most-Expensive-Data-Breaches-Over-Past-Year*, 2020.

[2] Common vulnerabilities and exposures. https://www.cve.org/.

[3] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

[4] Keke Gai, Meikang Qiu, Bhavani Thuraisingham, and Lixin Tao. Proactive attribute-based secure data schema for mobile cloud in financial industry. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1332–1337. IEEE, 2015.

[5] Kutub Thakur, Meikang Qiu, Keke Gai, and Md Liakat Ali. An investigation on cyber security threats and security models. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 307–311. IEEE, 2015.

[6] Log4j, https://nvd.nist.gov/vuln/detail/CVE-2021-44228.

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

[8] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[9] Fujin Hou, Kun Zhou, Longbin Li, Yuan Tian, Jie Li, and Jian Li. A vulnerability detection algorithm based on transformer model. In *International Conference on Artificial Intelligence and Security*, pages 43–55. Springer, 2022.

[10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[11] Justin M Johnson and Taghi M Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6(1):1–54, 2019.

[12] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[13] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Sara Beery, et al. Wilds: A benchmark of in-the-wild distribution shifts 2021. *arXiv preprint arXiv:2012.07421*, 2020.

[14] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[15] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[16] Bolun Wu, Futai Zou, et al. Code vulnerability detection based on deep sequence and graph models: A survey. *Security and Communication Networks*, 2022, 2022.

[17] Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems*, 28(7):1329–1343, 2019.

[18] Na Li, Haoyu Zhang, Zhihui Hu, Guang Kou, and Huadong Dai. Automated software vulnerability detection via pre-trained context encoder and self attention. In *International Conference on Digital Forensics and Cyber Crime*, pages 248–264. Springer, 2022.

[19] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Velvet: a novel ensemble learning approach to automatically locate vulnerable statements. *arXiv preprint arXiv:2112.10893*, 2021.

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

[21] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[23] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[24] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[25] Flawfinder. https://dwheeler.com/flawfinder/.

[26] Rats. https://security.web.cern.ch.

[27] Cppcheck. https://cppcheck.sourceforge.io/.

[28] Coverity. https://scan.coverity.com/.

[29] Infer. https://fbinfer.com/.

[30] Fabian Yamaguchi. Pattern-based vulnerability discovery. 2015.

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

[32] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2018.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[34] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2019.

[35] Common weakness enumeration. https://cwe.mitre.org/.

[36] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[37] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[38] Abdullah Al-Boghdady, Mohammad El-Ramly, and Khaled Wassif. idetect for vulnerability detection in internet of things operating systems using machine learning. *Scientific Reports*, 12(1):1–12, 2022.

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

[40] Zhen Li, Jing Tang, Deqing Zou, Qian Chen, Shouhuai Xu, Chao Zhang, Yichen Li, and Hai Jin. Towards making deep learning-based vulnerability detectors robust. *arXiv preprint arXiv:2108.00669*, 2021.

[41] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. *CoRR*, abs/2205.12424, 2022.

[42] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Ro{bert}a: A robustly optimized {bert} pretraining approach, 2020.

[43] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection: Performance, model's security and platforms. *arXiv preprint arXiv:2204.03214*, 2022.

[44] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[45] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Deep Learning for Code Workshop*, 2022.

[46] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. *arXiv preprint arXiv:2203.05181*, 2022.

[47] Van Nguyen, Trung Le, Olivier De Vel, Paul Montague, John Grundy, and Dinh Phung. Information-theoretic source code vulnerability highlighting. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[48] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. 03 2022.

[49] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. Vulchecker: Graph-based vulnerability localization in source code.

[50] Miltiadis Allamanis. Graph neural networks in program analysis. In *Graph Neural Networks: Foundations, Frontiers, and Applications*, pages 483–497. Springer, 2022.

[51] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[52] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021.

[53] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR, 2021.

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

[55] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[56] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[57] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[58] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, 2022.

[59] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78, 2004.

[60] Joern. https://github.com/ShiftLeftSecurity/.

[61] Lianzhe Huang, Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Text level graph neural network for text classification. *arXiv preprint arXiv:1910.02356*, 2019.

[62] Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every document owns its structure: Inductive text classification via graph neural networks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 334–339, 2020.

[63] OWASP. Owasp top ten. Accessed: 2023-03-13.

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

[65] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

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

[67] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: a dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120. IEEE, 2021.

[68] Shibani Santurkar, Dimitris Tsipras, and Aleksander Madry. Breeds: Benchmarks for subpopulation shift. *arXiv preprint arXiv:2008.04859*, 2020.

[69] National vulnerability database. https://nvd.nist.gov/.

[70] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

[71] Sharma A Zhou Y. Automated identification of security issues from commit messages and bug reports. *In Proceedings of the 2017 11th joint meeting on foundations of software engineering (pp. 914-919)*, 2017.

[72] Nist, https://www.nist.gov/.

[73] Sard, https://samate.nist.gov/SRD/.

[74] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[75] Tree-sitter. https://tree-sitter.github.io/tree-sitter.