Master's Thesis

Vulnerabilities detection in binary code

Nguyen The Hung

Supervisor Mizuhito Ogawa

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

8, 2024

Abstract

For many years, vulnerabilities in software have posed significant security risks, necessitating the development of effective detection methods. Various approaches have been employed for vulnerability detection, including CNNs-based, RNNs-based, autoencoder-based, and transformer-based techniques, utilizing diverse datasets. Recently, graph kernel techniques have emerged as a promising approach for detecting vulnerabilities, particularly for Original Entry Point detection of packed code.

This thesis proposes a novel method for vulnerability detection at both the function-level and line-level using graph kernels in combination with machine learning classifiers. By leveraging the structural properties of control flow graphs (CFGs) and the power of machine learning, this approach aims to improve the accuracy and robustness of vulnerability detection systems. The methodology involves constructing CFGs from binary code, extracting meaningful features from these graphs, and applying graph kernel techniques such as the Weisfeiler-Lehman kernel and Shortest Path kernel to encode the graphs into feature vectors.

Extensive experiments were conducted to evaluate the effectiveness of the proposed method. The results indicate that combining Weisfeiler-Lehman Optimal Assignment (WLOA) kernel with Shortest Path (SP) kernel significantly improves the detection performance.

This research contributes to advancing the field of software vulnerability detection by offering a new perspective on utilizing graph kernels for enhanced security analysis. The findings highlight the superiority of graph kernel-based methods and suggest directions for future work, including further refinement of kernel techniques and exploration of additional machine learning models to further enhance detection accuracy.

Keywords: Vulnerability Detection, Graph Kernels, Machine Learning, Control Flow Graphs, Weisfeiler-Lehman Kernel, Shortest Path Kernel, Binary Code Analysis, Function-level Detection, Line-level Detection, Software Security.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The detection of vulnerabilities in binary code is a critical aspect of software security, which ensures the protection of systems from malicious attacks. Two of the most common methods used for this purpose are RNN-based models using Recurrent Neural Networks (RNNs) and transformer-based models using transformer architectures. Binary code and assembly instructions can be treated as sequences, and RNNs can process these sequences to learn patterns associated with vulnerabilities. The self-attention mechanism in transformers allows the model to weigh the importance of each token in the sequence, enabling it to understand the context and relationships between instructions. Both of these models have demonstrated high accuracy in detecting vulnerabilities at the function level; however, they struggle with precision at the line level, which is crucial for detailed vulnerability identification and remediation.

In 2023, Mr. PHAM Thanh Hung using graph kernel for Original Entry Point detection of packed code in his thesis. Therefore, the emergence of graph kernel techniques presents a promising solution to this challenge. Graph kernel approaches take advantage of the structural information inherent in code representations, enabling more granular analysis and detection. These techniques have the potential to revolutionize vulnerability detection by improving precision and efficiency at the line level. By capturing intricate relationships and dependencies within the code, graph kernel methods can facilitate more accurate and efficient identification of vulnerabilities.

The motivation for this thesis is driven by the need to explore and optimize these emerging graph kernel techniques to address the existing limitations of RNN-based and transformer-based models. By focusing on the

development and application of graph kernel approaches, this research aims to contribute to the advancement of vulnerability detection methodologies, making them more robust and effective in safeguarding software systems.

1.2 Problem Statement

In the context of subsection 1.1, the aim is to address the limitations of current vulnerability detection methods in binary code. Existing models like BVDetector and LineVul exhibit certain strengths in function-level vulnerability detection but also significant weaknesses in line-level vulnerability detection.

This thesis starts by evaluating the feasibility of the LineVul model, which employs a transformer architecture to detect vulnerabilities at the function level. Although it achieves high accuracy at this level, its efficacy decreases significantly when pinpointing vulnerabilities at the line level.

To address these limitations, the potential of graph kernel approaches in vulnerability detection is explored. By leveraging the structural information of code through graph representations, graph kernel techniques may offer a more precise and efficient method for identifying vulnerabilities.

The performance of these graph kernel approaches is compared with the LineVul model to determine its efficacy. Through this comparison, the aim is to identify the most effective methodology for comprehensive and accurate vulnerability detection in binary code.

1.3 Contributions

The main contributions of this thesis are as follows.

1. **Binary Dataset with Line-Level Labeling:** A binary dataset with line-level labeling has been curated, providing a granular view of vulnerabilities within the code. This dataset serves as a valuable resource for training and evaluating models aimed at pinpointing vulnerabilities with high precision.

2. **Application of graph kernel techniques for function-level vulnerability detection:** A novel technique utilizing graph kernel approaches is introduced to detect vulnerabilities at the function level. Using the structural information inherent in the representation of code in graphs, this technique enhances the accuracy and effectiveness of function-level vulnerability detection.

3. **Enhancing Line-Level Vulnerability Detection Using Control Flow Graph (CFG) Features:** A method that integrates features derived from Control Flow Graphs (CFGs) is proposed to improve the performance of vulnerability detection at the line level. This approach combines the CFG-based features with existing detection methods to achieve more precise and reliable identification of vulnerabilities within individual lines of code.

1.4 Thesis outline

The remaining of this thesis is organized as follows:

Chapter 2: Reviews existing literature on various machine learning approaches for vulnerability detection, including CNNs, RNNs, autoencoders, and transformer models. It also discusses commonly used datasets and the significance of CWE and CVE reports in the context of vulnerability detection.

Chapter 3: Provides the foundational knowledge necessary for understanding the research, including the concepts of control flow graphs, graph kernels, and various machine learning models used for classification tasks.

Chapter 4: Describes the datasets used, from obtaining binary code from C/C++ source files to the detection of vulnerabilities at both function and line levels. This includes the processes of CFG construction, labeling, and classification.

Chapter 5: Details the methodological approach taken in this research.

Chapter 6: Discusses the evaluation metrics applied, the experimental setup, and presents the results of the experiments. It also includes a discussion of the findings for both function-level and line-level vulnerability detection.

Chapter 7: Summarizes the key findings of the research, discusses their implications, and outlines potential directions for future work.

Chapter 2

Related Works

2.1 CNN-based for vulnerability detection

Convolutional Neural Networks (CNNs) are a class of deep learning models known for their effectiveness in image and spatial data processing. CNNs are characterized by their ability to automatically and adaptively learn spatial hierarchies of features through backpropagation using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers.CNNs can be effectively used for vulnerability detection in binary code due to their ability to capture and learn hierarchical patterns within the code. Binary code can be represented in formats that maintain spatial relationships, such as images or matrices, making it suitable for CNN processing.

Lee et al. (2017)[1] introduced a novel approach that merges an innovative encoding technique for assembly language instructions, named Instruction2vec[2], with a deep learning model referred to as "Text-CNN" to detect vulnerabilities in binary code. This encoding technique converts each instruction into a fixed-length vector. The instruction components, including opcodes and operand fragments, are encoded using a specialized word2vec model[3]. Their method achieved a 96.1% accuracy in identifying CWE-121 (Stack Overflow) vulnerabilities in the Juliet Test Suite, surpassing the 94.2% accuracy achieved with a conventional word2vec model.

2.2 RNN-based for vulnerability detection

Recurrent Neural Networks (RNNs) are a class of deep learning models designed to recognize patterns in sequences of data, such as time series, text, or code. Unlike feedforward neural networks, RNNs have connections that form directed cycles, allowing information to persist. This feature makes

them well-suited for tasks where the order of the data is essential. RNNs are inherently designed to handle sequential data and maintain the order of information, which is crucial for analyzing binary code. Vulnerabilities often depend on the sequence of instructions, and RNNs can effectively capture these dependencies. By maintaining a memory of previous inputs, RNNs can understand the broader context within which a particular instruction or function operates. This contextual awareness is vital for accurately identifying vulnerabilities that are influenced by preceding code segments.

BVDetector[4] functions using program slices that have been pre-extracted. It uses word2vec encoding per token. The authors experiment with various neural networks to categorize the encoded slices, discovering that a BGRU[5] yields the best performance. They evaluated their technique on program slices obtained from a portion of the Juliet test suite targeting vulnerabilities related to memory corruption and numerical errors, reporting an accuracy of 96.7%.

The concept of program slices[6][7] is adapted to the assembly language by Li et al.[8]. In addition, they propose a combined model of source code and assembly termed 'hybrid slices'. This hybrid slice technique is tested on an aggregated subset of Juliet, achieving 96.9% accuracy, surpassing BVDetector's 88.9%.

2.3 Autoencoder-based for vulnerability detection

Although VulDeePecker primarily uses Bi-directional Long Short-Term Memory (Bi-LSTM) networks, it also incorporates autoencoder techniques for feature extraction and dimensionality reduction. The system identifies vulnerability patterns by analyzing code snippets and extracting features using deep learning models, including autoencoders for preprocessing and denoising.

Le et al. introduce a Maximal Divergence Sequential Auto-Encoder[9] and utilize a constant encoding for opcodes along with a histogram-based encoding for operands to describe assembly language instructions, achieving 85% accuracy.

2.4 Transfomer-based for vulnerability detection

In recent years, Transformer-based models have gained significant popularity over RNNs for various Natural Language Processing (NLP) tasks, including vulnerability detection in source code. Transformers use a self-attention mechanism that allows the model to weigh the importance of different words in a sentence (or tokens in code) when forming an output. This enables the model to capture long-range dependencies and relationships between tokens effectively.Due to their architecture, Transformers can process input data in parallel, leading to faster training times and scalability to larger datasets. Transformers can consider the entire sequence of input tokens at once, which provides a comprehensive understanding of the context.

LineVul[10] is a Transformer-based approach specifically designed to predict vulnerabilities at the line level in C/C++ code. This approach addresses limitations found in existing methods, such as IVDetect, by leveraging advanced deep learning techniques, including BERT architecture, pre-trained CodeBERT[11] models, and attention mechanisms for precise vulnerability localization.

LineVul's methodology involves two key phases: function-level prediction and line-level prediction. At the function level, Byte Pair Encoding (BPE) is used for subword tokenization, and a stack of 12 Transformer encoder blocks with a multi-head self-attention mechanism is employed. The pre-trained CodeBERT model is fine-tuned to generate vector representations of the code, capturing long-term dependencies and interactions within the codebase.

For line-level prediction, LineVul utilizes self-attention scores from the Transformer model to rank lines according to their likelihood of being vulnerable. Attention scores are summarized to prioritize inspection lines, allowing a more precise identification of vulnerabilities within individual lines of code.

The empirical evaluation of LineVul was conducted on a large-scale dataset consisting of over 188,000 C/C++ functions, with line-level ground truths for vulnerabilities. The results demonstrated that LineVul significantly improves upon existing methods, achieving an F1 measure of 0.91 for function-level predictions and a Top-10 accuracy of 0.65 for line-level predictions.

2.5 Common Weakness Enumeration(CWE) and Common Vulnerabilities and Exposures(CVE) report

2.5.1 Common Weakness Enumeration

Common Weakness Enumeration (CWE) is a community-developed list of common hardware and software weaknesses. These weaknesses are identified and categorized to help developers, security practitioners, and organizations understand and mitigate vulnerabilities in software and hardware systems. The primary goal of CWE is to create a standardized taxonomy of weaknesses to improve the development, acquisition, and operation of secure software and hardware.

CWE includes detailed descriptions of each weakness, including examples, potential consequences, and mitigation strategies. The list is regularly updated to reflect new findings and evolving security practices. The CWE list is maintained by the MITRE Corporation and is widely used in various security tools and frameworks for vulnerability detection and analysis.

Standardized Weaknesses: CWE provides a standardized way to identify and describe software and hardware weaknesses, making it easier for organizations to communicate and address security issues.

Comprehensive Coverage: The CWE list covers a wide range of weaknesses, including coding errors, design flaws, and architectural issues.

Mitigation Strategies: Each CWE entry includes potential mitigation strategies to help developers and security practitioners prevent and address the identified weaknesses.

2.5.2 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a list of publicly disclosed information security vulnerabilities and exposures. Each CVE entry includes an identifier, a brief description of the vulnerability or exposure, and references to related vulnerability reports and alerts. The CVE list is maintained by the MITRE Corporation and is widely used by security professionals to track and address vulnerabilities in software and hardware systems.

CVE identifiers are used by various security tools and databases to provide consistent and standardized information about vulnerabilities. This standardization helps organizations quickly assess the impact of vulnerabilities, prioritize remediation efforts, and improve overall security posture.

Unique Identifiers: Each CVE entry is assigned a unique identifier, making it easy to reference and track specific vulnerabilities.

Publicly Disclosed Vulnerabilities: CVE entries are based on publicly disclosed information, ensuring that the list reflects real-world security issues.

References and Advisories: Each CVE entry includes references to related vulnerability reports, advisories, and patches, providing valuable context and remediation information.

2.5.3 Integration of CWE and CVE in Vulnerability Detection

The integration of CWE and CVE is crucial for effective vulnerability detection and management. While CWE focuses on identifying and categorizing weaknesses, CVE provides detailed information about specific vulnerabilities and exposures that exploit these weaknesses. Security tools and frameworks often use CWE and CVE together to provide comprehensive vulnerability assessments.

Examples of integration:

Static and Dynamic Analysis Tools: Security tools use CWE entries to identify potential weaknesses in code during static and dynamic analysis. When a weakness is detected, the tool may reference related CVE entries to provide information about known vulnerabilities that exploit the weakness. Figure 2.1 is an example of CWE-391: Unchecked Error Condition and Figure 2.2 is description of CWE-391.

Vulnerability Databases: Databases such as the National Vulnerability Database (NVD) use CVE identifiers to catalog and provide detailed information about vulnerabilities. These databases often include CWE mappings to help users understand the underlying weaknesses associated with each vulnerability. Figure 2.3 is description of CWE-391 in CVE-2023-32871.

Security Best Practices: Organizations use CWE and CVE to inform their security best practices, development guidelines, and incident response strategies. By understanding common weaknesses and vulnerabilities, organizations can implement more effective security measures and improve their overall security posture.

```c
#include "std_testcase.h"

#include <errno.h>

#include <math.h>

#ifndef OMITBAD

void CWE391_Unchecked_Error_Condition__sqrt_01_bad()
{
    {
        double doubleNumber;
        doubleNumber = (double)sqrt((double)-1);
        /* FLAW: Do not check to see if sqrt() failed */
        printDoubleLine(doubleNumber);
    }
}

#endif /* OMITBAD */
```

Figure 2.1: An example of CWE-391: Unchecked Error Condition

CWE-391: Unchecked Error Condition

Weakness ID: 391
Vulnerability Mapping: PROHIBITED
Abstraction: Base

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

[PLANNED FOR DEPRECATION. SEE MAINTENANCE NOTES AND CONSIDER CWE-252, CWE-248, OR CWE-1069.] Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.

Common Consequences

Scope	Impact	Likelihood
Integrity Other	**Technical Impact:** *Varies by Context; Unexpected State; Alter Execution Logic*	

Potential Mitigations

Phase: Requirements

The choice between a language which has named or unnamed exceptions needs to be done. While unnamed exceptions exacerbate the chance of not properly dealing with an exception, named exceptions suffer from the up call version of the weak base class problem.

Figure 2.2: Description of CWE-391

Figure 2.3: Description of CWE-391 in CVE-2023-32871

Chapter 3

Background Knowledge

3.1 Control Flow Graph of binary code

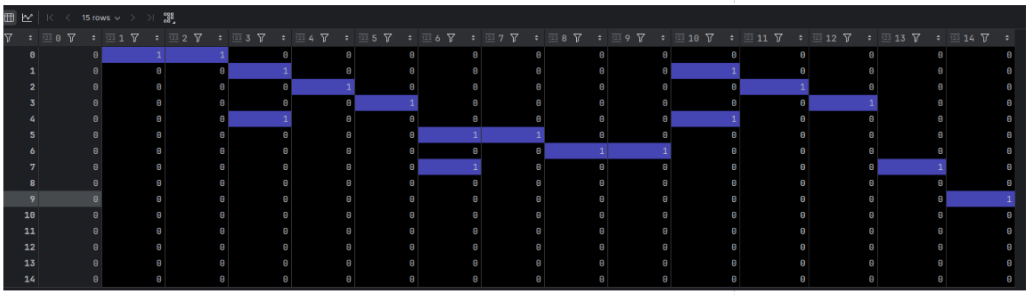
A Control Flow Graph (CFG) is a representation used in computer science to depict all possible execution paths through a program. In the context of binary code, a CFG is a directed graph where nodes represent basic blocks—sequences of consecutive instructions with a single entry point and a single exit point—and edges represent control flow paths between these blocks. Each node in the CFG is labeled with the sequence of instructions it represents. The primary purpose of a CFG is to illustrate the flow of control within a program, making it a crucial tool for various program analysis tasks, such as optimization, debugging, and vulnerability detection.

Formally, a CFG can be defined as a directed graph $G = (V, E)$, where:

- V is a set of vertices (or nodes), each representing a basic block in the binary code. Each node $v \in V$ is labeled with a sequence of instructions.
- E is a set of directed edges, where each edge $(b_i, b_j) \in E$ indicates that there is a possible control flow transfer from basic block b_i to basic block b_j.

3.2 Graph kernel

Accurately measuring graph similarity is crucial for numerous applications across different fields. Recently, graph kernels have become a promising solution to this challenge. GraKeL is a library offering implementations of several established graph kernels, integrating them into a unified framework.



```
{0: 'endbr64\npush\ndec\nmov\npush\ndec\nsub\ndec\nmov\ndec\nmov\nxor\ndec\nmov\nmov\ntest\nje',
1: 'dec\nlea\nmov\nmov\ndec\nmov\ncall',
2: 'mov\nmov\ncall',
3:'mov\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec
\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec
\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\nmov\nmov\ndec\nmov\ndec\nadd
\nmov\ndec\nmov\ndec\nmov\ncall', 4: 'dec\nmov\nmov\ndec\nmov\nmov\ndec\nlea\nmov\nmov\ndec\nmov\ncall',
5: 'dec\ncmp\nje',
6: 'nop\ndec\nmov\ndec\nsub\nje',
7: 'dec\nmov\ndec\nmov\ncall',
8: 'dec\nmov\nleave\nret',
9: 'call',
10: ' ',
11: ' ',
12: ' ',
13: ' ',
14: ' '}
```

Figure 3.1: Adjacency matrix and node labels of graph 1

Written in Python and inspired by scikit-learn's philosophy, GraKeL simplifies the creation of comprehensive machine learning pipelines for tasks like graph classification and clustering.

A graph kernel is a symmetric, positive semidefinite function on the set of graphs $\mathcal{G}$. Once we define such a function $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ on the set $\mathcal{G}$, it is known that there exists a map $\phi : \mathcal{G} \to \mathcal{H}$ into a Hilbert space[16] $\mathcal{H}$, such that:

$$k(G_i, G_j) = \langle \phi(G_i), \phi(G_j) \rangle_{\mathcal{H}}$$

for all $G_i, G_j \in \mathcal{G}$ where $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is the inner product in $\mathcal{H}$. Roughly speaking, a graph kernel is a function that measures the similarity of two graphs.[1]

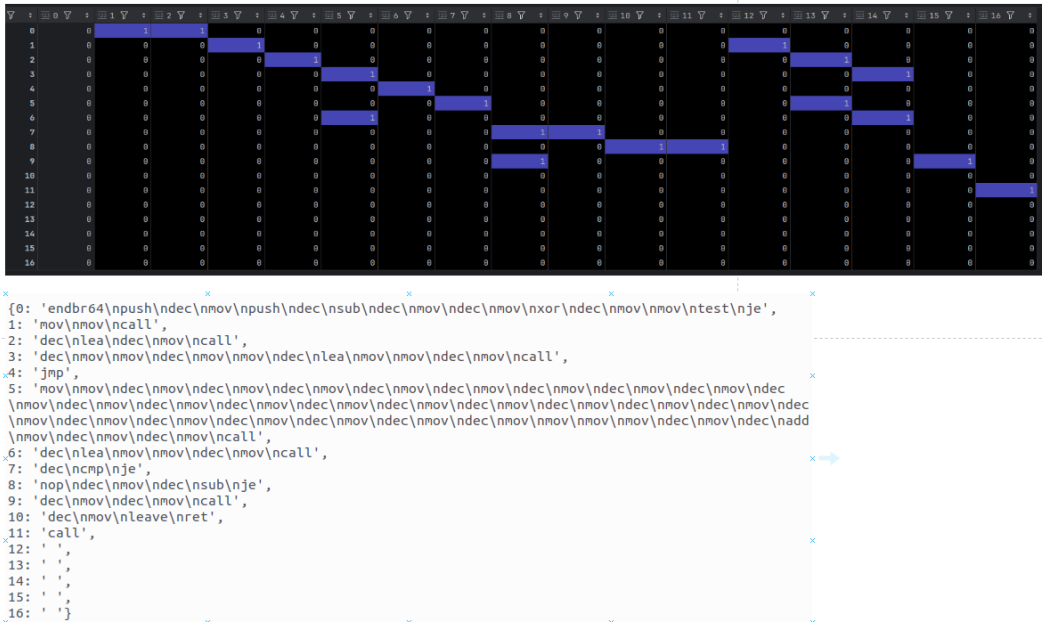
Example:

Figure 3.1 and 3.2 show adjacency matrixes and node labels of graph 1 and graph 2.

3.2.1 Shortest Path Kernels

Shortest Path Kernels compute the similarity between two graphs by comparing the shortest paths within each graph. The kernel considers the length

[1]https://ysig.github.io/GraKeL/0.1a8/documentation/introduction.html



```
{0: 'endbr64\npush\ndec\nmov\npush\ndec\nsub\ndec\nmov\ndec\nmov\nxor\ndec\nmov\nmov\ntest\nje',
1: 'mov\nmov\ncall',
2: 'dec\nlea\ndec\nmov\ncall',
3: 'dec\nmov\nmov\ndec\nmov\nmov\ndec\nlea\nmov\nmov\ndec\nmov\ncall',
4: 'jmp',
5: 'mov\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec
\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec
\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\ndec\nmov\nmov\nmov\nmov\ndec\nmov\ndec\nadd
\nmov\ndec\nmov\ndec\nmov\ncall',
6: 'dec\nlea\nmov\nmov\ndec\nmov\ncall',
7: 'dec\ncmp\nje',
8: 'nop\ndec\nmov\ndec\nsub\nje',
9: 'dec\nmov\ndec\nmov\ncall',
10: 'dec\nmov\nleave\nret',
11: 'call',
12: ' ',
13: ' ',
14: ' ',
15: ' ',
16: ' '}
```

Figure 3.2: Adjacency matrix and node labels of graph 2

	0	1
0	1.000000	0.763090
1	0.763090	1.000000

Figure 3.3: The kernel matrixes using the Shortest Path Kernels of graph 1 and graph 2

and labels of the shortest paths, capturing the global structure of the graphs. This approach is effective in tasks where the relationships between nodes are crucial for understanding the graph's properties. **Example:** Figure 3.1 shows the kernel matrixes using the Shortest Path Kernels of graph 1 and graph 2.

3.2.2 Weisfeiler Lehman Framework

The Weisfeiler-Lehman (WL) Framework is a method for graph isomorphism testing that iteratively refines the labels of nodes in a graph based on the labels of their neighbors. Initially, each node is assigned a label based on its attributes. At each iteration, the label of a node is updated to reflect the multiset of labels of its neighboring nodes. This process continues for

	0	1
0	1.000000	0.947335
1	0.947335	1.000000

Figure 3.4: The kernel matrixes using the Weisfeiler Lehman Framework of graph 1 and graph 2

a predefined number of iterations, resulting in a refined label for each node that captures its local neighborhood structure.

The WL Framework is widely used for enhancing the discriminative power of graph kernels by creating more informative node representations that consider the graph's topology.

Example: Figure 3.1 shows the kernel matrixes using the Weisfeiler Lehman Framework of graph 1 and graph 2.

3.2.3 Weisfeiler-Lehman Optimal Assignment

Valid Assignment Kernels

Valid Assignment Kernels are a class of graph kernels that compute the similarity between two graphs by finding an optimal assignment of nodes between the graphs. The goal is to match nodes in a way that maximizes the similarity between their labels while respecting the structural properties of the graphs. This involves solving an optimization problem to find the best alignment of nodes that produces the highest overall similarity score.

Weisfeiler-Lehman Optimal Assignment he Weisfeiler-Lehman Optimal Assignment (WLOA) Kernel combines the Weisfeiler-Lehman Framework with the concept of valid assignment kernels. In the WLOA Kernel, the node labels are first refined using the Weisfeiler-Lehman Framework. Once the node labels are updated, the kernel computes the optimal assignment between the nodes of two graphs by solving an assignment problem. This process ensures that the similarity measure captures both the local neighborhood structures (through WL refinement) and the best possible node-to-node correspondence (through optimal assignment).

The WLOA Kernel provides a robust measure of graph similarity by integrating the iterative label refinement of the WL Framework with the flexibility of optimal node assignments, making it effective for various graph classification and similarity tasks.

Example: Figure 3.1 shows the kernel matrixes using the WLOA Kernel of graph 1 and graph 2.

	0	1
0	1.000000	0.887151
1	0.887151	1.000000

Figure 3.5: The kernel matrixes using using the WLOA Kernel of graph 1 and graph 2

3.3 Machine learning for classification

3.3.1 Random forest

A Random Forest is an ensemble learning method for classification, regression, and other tasks, which operates by constructing a multitude of decision trees during training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. This method was introduced by Breiman (2001)[17] and is known for its robustness and accuracy in handling various types of data.

The concepts of Random Forests algorithm:

1. **Ensemble Learning**: Random Forests belong to ensemble methods that combine multiple models to improve overall performance and robustness compared to individual models.

2. **Bootstrap Aggregating (Bagging)**: This technique involves creating multiple subsets of the original dataset by sampling with replacement (bootstrap sampling) and training a model on each subset. Bagging helps reduce variance and prevent overfitting.

3. **Random Feature Selection**: At each split in the decision tree, a random subset of features is considered, ensuring diversity among the trees and reducing correlation between them, which improves generalization.

4. **Majority Voting/Averaging**: The final prediction is made by aggregating the predictions from all individual trees in the forest, either by majority voting (for classification) or averaging (for regression).

3.3.2 Support Vector Machine

A Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression tasks. The SVM algorithm aims to find

the hyperplane that best separates the data into different classes with the maximum margin. Introduced by Cortes and Vapnik (1995)[18], SVMs are effective in high-dimensional spaces and are known for their versatility in handling various types of data.

The concepts of SVM:

1. **Hyperplane**: A hyperplane is a flat affine subspace of the feature space that separates different classes. In a two-dimensional space, it is a line, while in higher dimensions, it becomes a plane or a hyperplane.

2. **Support Vectors**: Support vectors are the data points closest to the hyperplane, which are critical in defining the position and orientation of the hyperplane. These points directly affect the margin, and thus the classification boundary.

3. **Margin**: The margin is the distance between the hyperplane and the nearest data points of each class. SVM aims to maximize this margin, resulting in a more robust classifier.

4. **Kernel Trick**: The kernel trick allows SVM to handle non-linearly separable data by mapping the input features into a higher-dimensional space where a linear hyperplane can be used to separate the classes. Common kernels include the polynomial kernel, radial basis function (RBF) kernel, and sigmoid kernel.

5. **Regularization Parameter (C)**: The parameter C controls the trade-off between achieving a large margin and minimizing classification error. A smaller C value encourages a larger margin with more misclassifications, while a larger C value prioritizes minimizing misclassification errors.

3.3.3 XGBoost

XGBoost (eXtreme Gradient Boosting) is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost is known for its speed and performance, making it a popular choice for structured or tabular data. It was introduced by Tianqi Chen and Carlos Guestrin in their 2016 paper "XGBoost: A Scalable Tree Boosting System" (Chen & Guestrin, 2016)[19].

The concepts of XGBoost:

1. **Gradient Boosting**: XGBoost uses the gradient boosting framework, where new models are added to correct the residual errors made by existing models. It iteratively fits new models to the negative gradient of the loss function.

2. **Regularization**: Regularization in XGBoost helps prevent overfitting by penalizing the complexity of the model. It includes both γ (penalty for the number of leaves) and λ(penalty for the sum of squared leaf weights).

3. **Tree Pruning**: XGBoost employs a technique called "maximum depth pruning" in which it grows trees up to a specified maximum depth. This helps control the complexity of the model and prevents overfitting.

4. **Handling Missing Values**: XGBoost can automatically handle missing values by learning the best imputation strategy during the training process.

5. **Parallel and Distributed Computing**: XGBoost is designed to be highly efficient and scalable. It supports parallel processing and can be distributed across multiple machines to handle large-scale datasets.

6. **Weighted Quantile Sketch**: XGBoost uses a weighted quantile sketch algorithm to handle weighted data and approximate the split points for continuous features efficiently.

7. **Cross Validation**: XGBoost supports built-in cross-validation, which allows for the evaluation of the model's performance and helps in tuning hyperparameters to avoid overfitting.

3.3.4 Multilayer perceptron

A Multilayer Perceptron (MLP) is a type of artificial neural network that consists of multiple layers of nodes, including an input layer, one or more hidden layers, and an output layer. Each node in one layer connects with a certain weight to every node in the next layer. MLPs use a supervised learning technique called backpropagation for training. They are used for various tasks such as classification, regression, and pattern recognition, due to their ability to model complex non-linear relationships. (Rosenblatt, 1961)[20]

The concepts of MLP:

1. **Feedforward Architecture:** In an MLP, the data flows in one direction, from the input layer to the output layer, through the hidden layers. There are no cycles or loops in the network.

2. **Activation Functions:** Activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Common activation functions include the sigmoid function, tanh, and ReLU. ReLU is widely used due to its simplicity and efficiency.

3. **Backpropagation:** Backpropagation is the learning algorithm used to train MLPs. It involves calculating the gradient of the loss function with respect to each weight by the chain rule, then updating the weights to minimize the loss function. This process is repeated iteratively until the network converges.

4. **Loss Function:** The loss function measures the difference between the predicted output and the actual target values. Common loss functions include mean squared error (MSE) for regression tasks and cross-entropy loss for classification tasks.

5. **Weight Initialization:** Proper weight initialization is crucial for the convergence of the network. Common initialization methods include random initialization, Xavier initialization, and He initialization.

6. **Learning Rate:** The learning rate determines the size of the steps taken during the optimization process. It is a critical hyperparameter that affects the convergence speed and stability of the training process.

7. **Regularization:** Regularization techniques, such as L1 and L2 regularization or dropout, are used to prevent overfitting by penalizing large weights or randomly dropping units during training.

8. **Optimization Algorithms:** Various optimization algorithms, such as Stochastic Gradient Descent (SGD), Adam, and RMSprop, are used to update the weights based on the computed gradients. These algorithms differ in how they adjust the learning rate and handle the gradients.

3.3.5 Convolutional neural network

A Convolutional Neural Network (CNN) is a class of deep neural networks that is most commonly applied to analyze visual images. CNNs are characterized by their use of convolutional layers that apply a convolution operation to the input, passing the result to the next layer. This architecture allows CNNs to effectively capture spatial hierarchies in data. CNNs are widely used for image classification, object detection, and various other tasks that involve spatial data[21].

The typical structure of a CNN includes:

1. **Convolutional Layers:** Convolutional layers use filters (kernels) to perform convolution operations on the input data, extracting local features. Each filter detects specific patterns, such as edges or textures, in the input image.

2. **Activation Functions:** Activation functions introduce non-linearity into the network, enabling it to learn complex patterns. The ReLU (Rectified Linear Unit) activation function is commonly used in CNNs due to its simplicity and effectiveness.

3. **Pooling Layers:** Pooling layers perform down-sampling operations, such as max pooling or average pooling, to reduce the spatial dimensions of the feature maps. This helps in making the network invariant to small translations and reduces computational load.

4. **Fully Connected Layers:** Fully connected layers are used at the end of the network to combine the extracted features and perform the final classification. Each neuron in these layers is connected to all neurons in the previous layer, similar to traditional neural networks.

5. **Dropout:** Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of the input units to zero during training. This forces the network to learn more robust features.

6. **Batch Normalization:** Batch normalization normalizes the input of each layer to have a mean of zero and a standard deviation of one. This helps in accelerating training and improving the stability of the network.

7. **Strides and Padding:** Strides determine the step size with which the filter moves across the input. Padding is used to add extra pixels around the input image, allowing the filter to fully cover the edges of the input.

8. **Backpropagation and Optimization:** The network is trained using backpropagation, where the gradients of the loss function with respect to the network parameters are computed and used to update the weights through an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam.

Chapter 4

Translation of C/C++ vulnerability dataset to x86 binary code

4.1 Datasets for vulnerability detection

Various datasets have been developed and utilized for vulnerability detection in source code. These data sets differ in terms of programming languages, types of vulnerabilities, and scale of data.

The Juliet C/C++ 1.3 Test Suite is a collection of test cases in the C/C++ language. It contains examples organized under 118 different Common Weakness Enumeration(CWE). It includes 64,099 test cases. The suite is often used as a benchmark for evaluating static analysis tools and vulnerability detection models. In particular, Yamaguchi et al.[12] utilized the Juliet Test Suite to assess their vulnerability discovery approach in source code through graph mining, while Russell et al. (2018)[13] used it in the development of a deep learning model to detect vulnerabilities in C / C++ code. Figures 4.1 and Figure 4.2 show 2 test cases of the Juliet C/C++ 1.3 Test Suite.

The Big-Vul dataset is a large-scale collection of real-world vulnerabilities from various open-source projects, with commit-level information that allows researchers to track how vulnerabilities are introduced and fixed. This dataset primarily includes C/C++ code and is suitable for training machine learning models for vulnerability detection and studying the evolution of vulnerabilities. Fan et al. (2020)[14] introduced Big-Vul to support the development of deep learning-based vulnerability detection models.

The Romeo dataset[15] is a binary vulnerability detection benchmark

```c
#ifndef OMITBAD

void CWE253_Incorrect_Check_of_Function_Return_Value__char_fprintf_01_bad()
{
    /* FLAW: fprintf() might fail, in which case the return value will be negative, but
     * we are checking to see if the return value is 0 */
    if (fprintf(stdout, "%s\n", "string") == 0)
    {
        printLine("fprintf failed!");
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

static void good1()
{
    /* FIX: check for the correct return value */
    if (fprintf(stdout, "%s\n", "string") < 0)
    {
        printLine("fprintf failed!");
    }
}
```

Figure 4.1: CWE-253: Incorrect check of function return value

```c
#ifndef OMITBAD

void CWE252_Unchecked_Return_Value__char_snprintf_01_bad()
{
    {
        /* By initializing dataBuffer, we ensure this will not be the
         * CWE 690 (Unchecked Return Value To NULL Pointer) flaw for fgets() and other variants */
        char dataBuffer[100] = "";
        char * data = dataBuffer;
        /* FLAW: Do not check the return value */
        SNPRINTF(data,100-strlen(SRC)-1, "%s\n", SRC);
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

static void good1()
{
    {
        /* By initializing dataBuffer, we ensure this will not be the
         * CWE 690 (Unchecked Return Value To NULL Pointer) flaw for fgets() and other variants */
        char dataBuffer[100] = "";
        char * data = dataBuffer;
        /* FIX: check the return value */
        if (SNPRINTF(data,100-strlen(SRC)-1, "%s\n", SRC) < 0)
        {
            printLine("snprintf failed!");
        }
    }
}
```

Figure 4.2: CWE-252: Unchecked return value

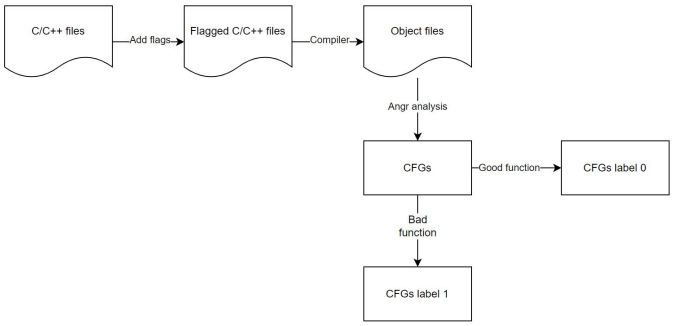


Figure 4.3: The process of CFGs construction and labeling

dataset based on the Juliet Test Suite (version 1.3), with approximately 41,812 test cases according to 91 Common Weakness Enumeration (CWE) categories. The dataset provides a simple text representation of disassembled binaries suitable for various sequence classifiers, incorporating context for cross-function vulnerabilities and preserving semantics to identify API calls while preventing label leakage.

4.2 Obtain binary code from C/C++

4.2.1 Adding Flags in C/C++ Files

The Juliet Test Suite is an extensive collection of synthetic test cases designed to evaluate the security of software programs. Each test case in the suite includes a pair of functions: one that is vulnerable (bad function) and one that has been corrected to remove the vulnerability (good function). The vulnerable function demonstrates a specific software weakness, while the corrected function shows the appropriate fix. The process of CFGs construction and labeling is illustrated in the Figure 4.3.

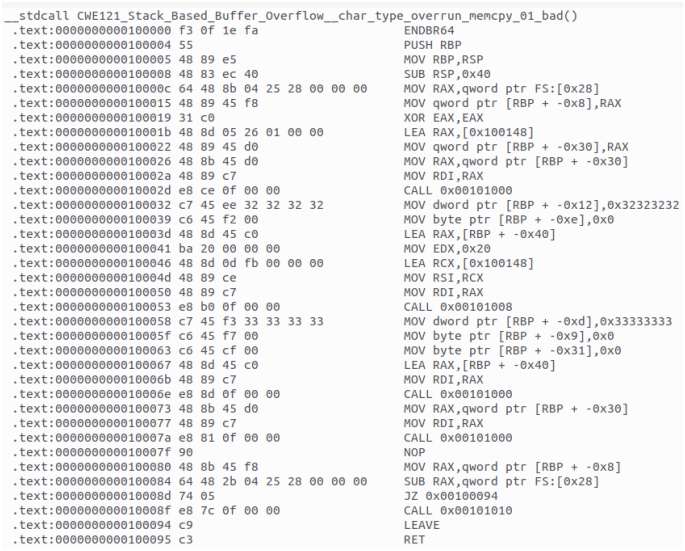
To accurately locate the instructions responsible for the introduction of vulnerabilities, flags are inserted into the C/C++ source files. These flags act as markers, identifying the precise locations in the code where vulnerabilities are present. By doing this, the analysis tools can more easily pinpoint the critical sections of the code that need to be examined. The flagged sections are then used to guide the construction of the CFGs, ensuring that the resulting graphs accurately reflect the presence of vulnerabilities. Figures 4.4 and Figure 4.5 illustrate the C code before and after adding the flags.

```c
typedef struct _charVoid
{
    char charFirst[16];
    void * voidSecond;
    void * voidThird;
} charVoid;

#ifndef OMITBAD

void CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01_bad()
{
    {
        charVoid structCharVoid;
        structCharVoid.voidSecond = (void *)SRC_STR;
        /* Print the initial block pointed to by structCharVoid.voidSecond */
        printLine((char *)structCharVoid.voidSecond);
        /* FLAW: Use the sizeof(structCharVoid) which will overwrite the pointer voidSecond */
        memcpy(structCharVoid.charFirst, SRC_STR, sizeof(structCharVoid));
        structCharVoid.charFirst[(sizeof(structCharVoid.charFirst)/sizeof(char))-1] = '\0'; /* null terminate the string */
        printLine((char *)structCharVoid.charFirst);
        printLine((char *)structCharVoid.voidSecond);
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

static void good1()
{
    {
        charVoid structCharVoid;
        structCharVoid.voidSecond = (void *)SRC_STR;
        /* Print the initial block pointed to by structCharVoid.voidSecond */
        printLine((char *)structCharVoid.voidSecond);
        /* FIX: Use sizeof(structCharVoid.charFirst) to avoid overwriting the pointer voidSecond */
        memcpy(structCharVoid.charFirst, SRC_STR, sizeof(structCharVoid.charFirst));
        structCharVoid.charFirst[(sizeof(structCharVoid.charFirst)/sizeof(char))-1] = '\0'; /* null terminate the string */
        printLine((char *)structCharVoid.charFirst);
        printLine((char *)structCharVoid.voidSecond);
    }
}
```

Figure 4.4: The original C file

```c
typedef struct _charVoid
{
    char charFirst[16];
    void * voidSecond;
    void * voidThird;
} charVoid;

#ifndef OMITBAD

void CWE122_Heap_Based_Buffer_Overflow__char_type_overrun_memcpy_01_bad()
{
    {
        charVoid * structCharVoid = (charVoid *)malloc(sizeof(charVoid));
        if (structCharVoid == NULL) {exit(-1);}
        structCharVoid->voidSecond = (void *)SRC_STR;
        /* Print the initial block pointed to by structCharVoid->voidSecond */
        printLine((char *)structCharVoid->voidSecond);
        /* FLAW: Use the sizeof(*structCharVoid) which will overwrite the pointer y */
        char start_flaw0[] = "2222";
        memcpy(structCharVoid->charFirst, SRC_STR, sizeof(*structCharVoid));
        char stop_flaw1[] = "3333";
        structCharVoid->charFirst[(sizeof(structCharVoid->charFirst)/sizeof(char))-1] = '\0'; /* null terminate the string */
        printLine((char *)structCharVoid->charFirst);
        printLine((char *)structCharVoid->voidSecond);
        free(structCharVoid);
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

static void good1()
{
    {
        charVoid * structCharVoid = (charVoid *)malloc(sizeof(charVoid));
        if (structCharVoid == NULL) {exit(-1);}
        structCharVoid->voidSecond = (void *)SRC_STR;
        /* Print the initial block pointed to by structCharVoid->voidSecond */
        printLine((char *)structCharVoid->voidSecond);
        /* FIX: Use the sizeof(structCharVoid->charFirst) to avoid overwriting the pointer y */
        char start_fix0[] = "4444";
        memcpy(structCharVoid->charFirst, SRC_STR, sizeof(structCharVoid->charFirst));
        char stop_fix1[] = "5555";
        structCharVoid->charFirst[(sizeof(structCharVoid->charFirst)/sizeof(char))-1] = '\0'; /* null terminate the string */
        printLine((char *)structCharVoid->charFirst);
        printLine((char *)structCharVoid->voidSecond);
        free(structCharVoid);
    }
}
```

Figure 4.5: The C file after adding flags



```
__stdcall CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01_bad()
.text:0000000000100000 f3 0f 1e fa                  ENDBR64
.text:0000000000100004 55                           PUSH RBP
.text:0000000000100005 48 89 e5                     MOV RBP,RSP
.text:0000000000100008 48 83 ec 40                  SUB RSP,0x40
.text:000000000010000c 64 48 8b 04 25 28 00 00 00   MOV RAX,qword ptr FS:[0x28]
.text:0000000000100015 48 89 45 f8                  MOV qword ptr [RBP + -0x8],RAX
.text:0000000000100019 31 c0                        XOR EAX,EAX
.text:000000000010001b 48 8d 05 26 01 00 00         LEA RAX,[0x100148]
.text:0000000000100022 48 89 45 d0                  MOV qword ptr [RBP + -0x30],RAX
.text:0000000000100026 48 8b 45 d0                  MOV RAX,qword ptr [RBP + -0x30]
.text:000000000010002a 48 89 c7                     MOV RDI,RAX
.text:000000000010002d e8 ce 0f 00 00               CALL 0x00101000
.text:0000000000100032 c7 45 ee 32 32 32 32         MOV dword ptr [RBP + -0x12],0x32323232
.text:0000000000100039 c6 45 f2 00                  MOV byte ptr [RBP + -0xe],0x0
.text:000000000010003d 48 8d 45 c0                  LEA RAX,[RBP + -0x40]
.text:0000000000100041 ba 20 00 00 00               MOV EDX,0x20
.text:0000000000100046 48 8d 0d fb 00 00 00         LEA RCX,[0x100148]
.text:000000000010004d 48 89 ce                     MOV RSI,RCX
.text:0000000000100050 48 89 c7                     MOV RDI,RAX
.text:0000000000100053 e8 b0 0f 00 00               CALL 0x00101008
.text:0000000000100058 c7 45 f3 33 33 33 33         MOV dword ptr [RBP + -0xd],0x33333333
.text:000000000010005f c6 45 f7 00                  MOV byte ptr [RBP + -0x9],0x0
.text:0000000000100063 c6 45 cf 00                  MOV byte ptr [RBP + -0x31],0x0
.text:0000000000100067 48 8d 45 c0                  LEA RAX,[RBP + -0x40]
.text:000000000010006b 48 89 c7                     MOV RDI,RAX
.text:000000000010006e e8 8d 0f 00 00               CALL 0x00101000
.text:0000000000100073 48 8b 45 d0                  MOV RAX,qword ptr [RBP + -0x30]
.text:0000000000100077 48 89 c7                     MOV RDI,RAX
.text:000000000010007a e8 81 0f 00 00               CALL 0x00101000
.text:000000000010007f 90                           NOP
.text:0000000000100080 48 8b 45 f8                  MOV RAX,qword ptr [RBP + -0x8]
.text:0000000000100084 64 48 2b 04 25 28 00 00 00   SUB RAX,qword ptr FS:[0x28]
.text:000000000010008d 74 05                        JZ 0x00100094
.text:000000000010008f e8 7c 0f 00 00               CALL 0x00101010
.text:0000000000100094 c9                           LEAVE
.text:0000000000100095 c3                           RET
```

Figure 4.6: The assembly code of bad function

4.2.2 Compiling C/C++ Files to object Files

Once the flags have been added, the modified C/C++ files are compiled into object files (.o files) using the GNU Compiler Collection (gcc). This step converts the high-level source code into machine code, which is a lower-level representation that can be directly executed by a computer. The compilation process involves several stages, including preprocessing, parsing, optimization, and code generation. The resulting object files contain the machine code instructions corresponding to the original C/C++ source code, structured in a way that is suitable for further analysis. Figure 4.6 and Figure 4.7 are assembly codes of bad function and good function after compiling process.

Compiler Optimization Compiler optimization refers to the process of improving the performance and efficiency of the code generated by a compiler. The goal is to produce machine code that runs faster, uses fewer resources, or both. Compiler optimization techniques can be applied at various stages of the compilation process, including during the parsing, intermediate code generation, and final code generation phases.

There are many types of compiler optimization, and some of them can automatically correct vulnerabilities during the compiling process.

Examples:

Buffer Overflows and Stack Canaries: Some compilers include options to add stack canaries, which can detect buffer overflow attempts. For example, GCC has the -fstack-protector option, which inserts guard variables

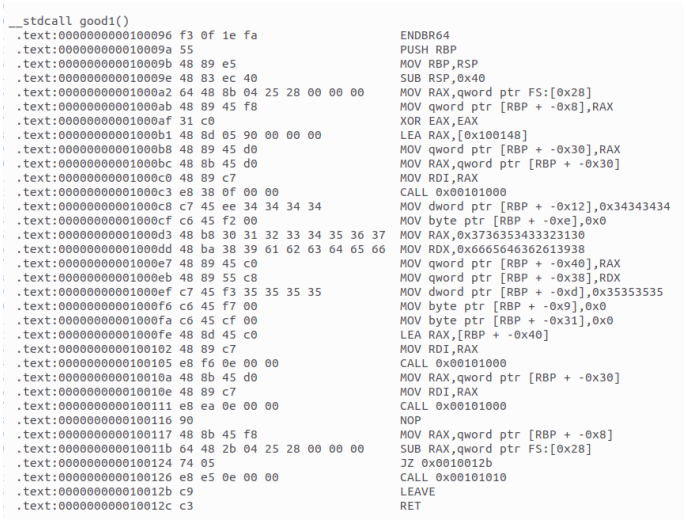


Figure 4.7: The assembly code of good function

to detect stack smashing in Figure 4.8.

```
void vulnerable_function(char *input) {
    char buffer[10];
    strcpy(buffer, input); // Potential buffer overflow
}
```

With stack protection enabled:

```
gcc -fstack-protector -O2 -o program program.c
```

In assembly code without stack protection (a):

1. **Function Setup and Buffer Allocation:**
   - `push %rbp`: Saves the old base pointer on the stack. This is standard for preserving the caller's frame context.
   - `mov %rsp, %rbp`: Sets up the new frame pointer for the current function.
   - `sub $48, %rsp`: Allocates 48 bytes on the stack for local variables, including the buffer. This space is reserved directly from the stack, reducing the stack pointer.
2. **Function Call to `strcpy`:**
   - `mov %rdi, -40(%rbp)`

```
        .file   "program.c"
        .text
        .globl  vulnerable_function
        .type   vulnerable_function, @function
vulnerable_function:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $48, %rsp
        movq    %rdi, -40(%rbp)
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movq    -40(%rbp), %rdx
        leaq    -18(%rbp), %rax
        movq    %rdx, %rsi
        movq    %rax, %rdi
        call    strcpy@PLT
        nop
        movq    -8(%rbp), %rax
        subq    %fs:40, %rax
        je      .L2
        call    __stack_chk_fail@PLT
.L2:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   vulnerable_function, .-vulnerable_function
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

a) Assembly code without stack protection

```
        .file   "program.c"
        .text
        .p2align 4
        .globl  vulnerable_function
        .type   vulnerable_function, @function
vulnerable_function:
.LFB35:
        .cfi_startproc
        endbr64
        subq    $40, %rsp
        .cfi_def_cfa_offset 48
        movq    %rdi, %rsi
        movl    $10, %edx
        movq    %fs:40, %rax
        movq    %rax, 24(%rsp)
        xorl    %eax, %eax
        leaq    14(%rsp), %rdi
        call    __strcpy_chk@PLT
        movq    24(%rsp), %rax
        subq    %fs:40, %rax
        jne     .L5
        addq    $40, %rsp
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        ret
.L5:
        .cfi_restore_state
        call    __stack_chk_fail@PLT
        .cfi_endproc
.LFE35:
        .size   vulnerable_function, .-vulnerable_function
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

b) Assembly code with stack protection

Figure 4.8: The example of Buffer Overflows and Stack Canaries, (a) Assembly code without stack protection, (b) Assembly code with stack protection

- `mov %fs:40, %rax`
- `mov %rax, -8(%rbp)`
- `leaq -18(%rbp), %rax`
- `mov %rdx, %rsi`
- `mov %rax, %rdi`
- `call strcpy@PLT`: Calls the `strcpy` function, which copies the string from `input` (source) to `buffer` (destination). This is where the vulnerability to buffer overflow exists, as `strcpy` does not check the length of the input against the buffer size.

3. **Standard Function Exit:**

   - `leave`: Restores the previous stack frame (equivalent to `mov %rbp, %rsp` followed by `pop %rbp`).
   - `ret`: Returns control to the calling function by popping the return address from the stack.

In assembly code with stack protection (b):

1. **Function Setup and Buffer Allocation:**

   - `push %rbp`: Saves the old base pointer on the stack. This is standard for preserving the caller's frame context.
   - `mov %rsp, %rbp`: Sets up the new frame pointer for the current function.
   - `sub $48, %rsp`: Allocates 48 bytes on the stack for local variables, including the buffer. This space is reserved directly from the stack, reducing the stack pointer.

2. **Function Call to `strcpy`:**

   - `mov %rdi, -40(%rbp)`
   - `mov %fs:40, %rax`
   - `mov %rax, -8(%rbp)`
   - `leaq -18(%rbp), %rax`
   - `mov %rdx, %rsi`
   - `mov %rax, %rdi`

- `call strcpy@PLT`: Calls the `strcpy` function, which copies the string from `input` (source) to `buffer` (destination). This is where the vulnerability to buffer overflow exists, as `strcpy` does not check the length of the input against the buffer size.

3. **Standard Function Exit:**

   - `leave`: Restores the previous stack frame (equivalent to `mov %rbp, %rsp` followed by `pop %rbp`).
   - `ret`: Returns control to the calling function by popping the return address from the stack.

4. **Assembly Code with Stack Protection (Panel b)**

5. **Function Setup and Canary Implementation:**

   - `endbr64`: An instruction used in newer versions of GCC for indirect branch tracking, part of Intel's Control-Flow Enforcement Technology.
   - `sub $40, %rsp`: Allocates 40 bytes on the stack for the buffer and possibly the canary. The exact allocation might slightly differ, showing a different focus in how space is managed due to the canary.
   - `mov %fs:40, %eax`
   - `xor %eax, %eax`: These lines are involved in setting and later checking the canary value. `%fs:40` typically holds the canary value, and `xor` is used for validation.

6. **Function Call with Protection:**

   - `leaq 14(%rsp), %rdi`
   - `call strcpy@PLT`: Similar direct call to `strcpy`, but now under the protection of the canary mechanism.

7. **Check Canary and Error Handling:**

   - `xor %eax, %eax`
   - `test %eax, %eax`
   - `jne .L5`: These instructions check if the canary value has been altered (specifically, `xor` would set flags based on whether the canary is unchanged; `jne` jumps if not equal, indicating a change).

- `.L5:`
- `call __stack_chk_fail@PLT`: If the canary check fails, this call to `__stack_chk_fail` is triggered, which handles the error by typically terminating the program or invoking other security measures.

8. **Enhanced Exit Sequence:**
    - `add $40, %rsp`: Corrects the stack pointer, undoing the earlier subtraction.
    - `ret`: Returns control to the caller, similar to the unprotected version but only after safely verifying the integrity of the stack.

Bounds Checking: Higher-level optimizations can sometimes include bounds checking for arrays. Compilers like GCC and Clang offer options to enable these checks, although they might not be enabled by default for performance reasons. The example in Figure 4.9

```
void vulnerable_function(int index) {
    int array[10];
    array[index] = 0; // Potential out-of-bounds access
}
```

With stack protection enabled:

```
gcc -fstack-protector-all -O2 -o program program.c
```

In assembly code without stack protection (a):

1. **Function Setup and Buffer Allocation:**
    - `push %rbp`: Saves the old base pointer on the stack. This is standard for preserving the caller's frame context.
    - `mov %rsp, %rbp`: Sets up the new frame pointer for the current function.
    - `sub $48, %rsp`: Allocates 48 bytes on the stack for local variables, including the buffer. This space is reserved directly from the stack, reducing the stack pointer.
2. **Function Call to `strcpy`:**
    - `mov %rdi, -40(%rbp)`
    - `mov %fs:40, %rax`

```
        .file   "program_2.c"
        .text
        .globl  vulnerable_function
        .type   vulnerable_function, @function
vulnerable_function:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $48, %rsp
        movq    %rdi, -40(%rbp)
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movq    -40(%rbp), %rdx
        leaq    -18(%rbp), %rax
        movq    %rdx, %rsi
        movq    %rax, %rdi
        call    strcpy@PLT
        nop
        movq    -8(%rbp), %rax
        subq    %fs:40, %rax
        je      .L2
        call    __stack_chk_fail@PLT
.L2:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   vulnerable_function, .-vulnerable_function
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

a) Assembly code without stack protection

```
        .file   "program_2.c"
        .text
        .p2align 4
        .globl  vulnerable_function
        .type   vulnerable_function, @function
vulnerable_function:
.LFB35:
        .cfi_startproc
        endbr64
        subq    $40, %rsp
        .cfi_def_cfa_offset 48
        movq    %rdi, %rsi
        movl    $10, %edx
        movq    %fs:40, %rax
        movq    %rax, 24(%rsp)
        xorl    %eax, %eax
        leaq    14(%rsp), %rdi
        call    __strcpy_chk@PLT
        movq    24(%rsp), %rax
        subq    %fs:40, %rax
        jne     .L5
        addq    $40, %rsp
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        ret
.L5:
        .cfi_restore_state
        call    __stack_chk_fail@PLT
        .cfi_endproc
.LFE35:
        .size   vulnerable_function, .-vulnerable_function
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

b) Assembly code with stack protection

Figure 4.9: The example of Bounds Checking, (a) Assembly code without stack protection, (b) Assembly code with stack protection

- `mov %rax, -8(%rbp)`
- `leaq -18(%rbp), %rax`
- `mov %rdx, %rsi`
- `mov %rax, %rdi`
- `call strcpy@PLT`: Calls the `strcpy` function, which copies the string from `input` (source) to `buffer` (destination). This is where the vulnerability to buffer overflow exists, as `strcpy` does not check the length of the input against the buffer size.

3. **Standard Function Exit:**

   - `leave`: Restores the previous stack frame (equivalent to `mov %rbp, %rsp` followed by `pop %rbp`).
   - `ret`: Returns control to the calling function by popping the return address from the stack.

In assembly code with stack protection (b):

1. **Function Setup and Canary Implementation:**

   - `endbr64`: An instruction used in newer versions of GCC for indirect branch tracking, part of Intel's Control-Flow Enforcement Technology.
   - `sub $40, %rsp`: Allocates 40 bytes on the stack for the buffer and possibly the canary. The exact allocation might slightly differ, showing a different focus in how space is managed due to the canary.
   - `mov %fs:40, %eax`
   - `xor %eax, %eax`: These lines are involved in setting and later checking the canary value. `%fs:40` typically holds the canary value, and `xor` is used for validation.

2. **Function Call with Protection:**

   - `leaq 14(%rsp), %rdi`
   - `call strcpy@PLT`: Similar direct call to `strcpy`, but now under the protection of the canary mechanism.

3. **Check Canary and Error Handling:**

   - `xor %eax, %eax`

- `test %eax, %eax`
- `jne .L5`: These instructions check if the canary value has been altered (specifically, `xor` would set flags based on whether the canary is unchanged; `jne` jumps if not equal, indicating a change).
- `.L5:`
- `call __stack_chk_fail@PLT`: If the canary check fails, this call to `__stack_chk_fail` is triggered, which handles the error by typically terminating the program or invoking other security measures.

4. **Enhanced Exit Sequence:**
   - `add $40, %rsp`: Corrects the stack pointer, undoing the earlier subtraction.
   - `ret`: Returns control to the caller, similar to the unprotected version but only after safely verifying the integrity of the stack.

In this task, our goal is to preserve the vulnerabilities in the binary code to accurately detect and analyze them. Therefore, we do not use optimization during the compilation process. Compiler optimizations, such as constant folding or dead code elimination, can inadvertently correct or remove vulnerabilities, which would defeat the purpose of our analysis.

To ensure that the vulnerabilities remain intact, we compile the code with no optimization using the -O0 flag. This flag instructs the compiler to disable all optimization passes, preserving the original code structure and any existing vulnerabilities.

4.2.3 Removing flags in CFGs

First, build blocks of cfg Figure 4.10a then remove flags instruction in label of block Figure 4.10b. In this section, we describe the process of removing flag instructions from the labels of blocks in the Control Flow Graphs (CFGs). This involves two main steps: first building the blocks of the CFG and then removing the flag instructions.

1. **Building Blocks of CFG:**

   To build the blocks of the CFG, we start by analyzing the binary code to identify basic blocks and their connections. A basic block is a sequence of instructions with a single entry point and a single exit point. The CFG is constructed by identifying these blocks and the control flow paths between them.

Figure 4.10: The blocks of bad function CFG,(a) The blocks of bad function CFG with flags, (b) The blocks of bad function CFG without flags

Figure 4.10a illustrates an example of a CFG with flag instructions included in the basic blocks. These flags were initially added to mark the presence of vulnerabilities in the source code.

2. **Removing Flag Instructions:**

   Once the CFG is built, the next step is to remove the flag instructions from the labels of the blocks. The flag instructions were used to localize vulnerabilities during the analysis phase but are not needed for the final model training and evaluation. Removing these flags ensures that the labels reflect only the functional instructions of the code.

   Figure 4.10b shows the CFG after the flag instructions have been removed. The blocks now contain only the original assembly instructions without any additional markers. This cleaned-up CFG is used for further analysis and classification of vulnerabilities.

   The process of removing the flag only deletes the marked instructions in the labels of the nodes. As a result, it does not alter the executable code, and the structure of the CFG remains unchanged.

After completing the dataset construction process, our dataset contains a total of 35,687 test cases spanning 81 different CWE categories.

Chapter 5

Methodology

5.1 Vulnerability detection in function level

5.1.1 CFGs construction and labeling

The process of constructing and labeling CFG for vulnerability detection at the function level involves a series of steps that ensure accurate identification and analysis of vulnerable and non-vulnerable functions. This section elaborates on the steps taken to achieve this.The process of CFGs construction and labeling is illustrated in the Figure 4.3

1. **Building CFGs with Angr**: Angr is a versatile binary analysis framework that supports the construction and analysis of CFG from compiled binaries. In this methodology, Angr is used to analyze the compiled .o files and generate CFGs for each function. The steps involved in this process are as follows:

    - **Loading the Object Files:** Angr loads the .o files, parsing the machine code instructions and organizing them into a structured representation that it can work with.
    - **Constructing Basic Blocks:** Angr identifies basic blocks within each function. A basic block is a sequence of consecutive instructions with a single entry point and a single exit point. The instructions within a basic block are executed sequentially without any branching, making it a fundamental unit of the CFG.
    - **Labeling the Basic Blocks:** In this step, each basic block is labeled using only the opcodes of the instructions it contains. This means that the labels represent the sequence of operation codes, which are the portion of the instruction that specifies the operation

to be performed. By focusing on opcodes, we capture the essential actions of each block without extraneous details.

- **Building the CFGs:** Once the basic blocks are identified, Angr constructs the CFG by connecting these blocks based on the control flow paths. Edges in the CFG represent possible transitions between basic blocks due to branch, jump, or call instructions. This results in a directed graph that represents all possible execution paths within each function.
- **Labeling the Functions:** After the CFGs are constructed, each function is labeled based on its vulnerability status. Functions identified as bad (containing vulnerabilities) are labeled with a 1, indicating the presence of vulnerabilities. Functions identified as good (corrected to remove vulnerabilities) are labeled with a 0, indicating the absence of vulnerabilities. These labels are crucial for training machine learning models, as they provide the ground truth needed for supervised learning.

5.1.2 CFGs classification

The classification of CFGs involves using advanced graph kernel methods to distinguish between vulnerable and non-vulnerable functions based on their CFG representations. In this methodology, two specific graph kernels—Weisfeiler-Lehman optimal assignment kernel and shortest path kernel—are combined to enhance the classification performance.

1. **Weisfeiler-Lehman Optimal Assignment Kernel**: The WL kernel enhances the discriminative power by capturing local neighborhood structures and their evolution over multiple iterations.

    - **Node Label Refinement:** The WL algorithm iteratively updates node labels by aggregating the labels of neighboring nodes, creating more informative labels that capture the local graph structure.
    - **Optimal Assignment:** The optimal assignment part of the kernel focuses on matching nodes from two graphs in a way that maximizes the similarity between the refined node labels, resulting in a more accurate graph similarity measure.

2. **Shortest Path Kernel**: The shortest path kernel computes the similarity between graphs by comparing the shortest paths within them.

This kernel is effective at capturing the global structure of graphs by considering the distances between all pairs of nodes.

- **Path Enumeration:** The kernel enumerates all shortest paths in the graph, effectively summarizing the graph's structure in terms of the distances between nodes.
- **Path Comparison:** The similarity between graphs is then calculated by comparing the lengths and labels of the shortest paths, providing a comprehensive measure of graph similarity.

3. **Combining the Weisfeiler-Lehman optimal assignment kernel and the shortest path kernel**: These combination leverages the strengths of both methods, leading to a more robust and discriminative kernel for CFG classification.

    - **Local and Global Structure:** The WL kernel excels at capturing local neighborhood structures and their evolution, while the shortest path kernel captures the global structure of the graph. By combining them, the resulting kernel can effectively analyze both local and global patterns within CFGs.
    - **Enhanced Discriminative Power:** Each kernel captures different aspects of graph similarity. The WL kernel focuses on node label refinement and optimal matching, whereas the shortest path kernel emphasizes distances between nodes. Combining these perspectives results in a richer representation of the graph's structural properties.
    - **Improved Classification Accuracy:** Empirical studies have shown that combining multiple graph kernels can lead to improved classification performance. The complementary nature of the WL optimal assignment and shortest path kernels provides a more comprehensive similarity measure, enhancing the ability to distinguish between vulnerable and non-vulnerable CFGs.

The combined kernel is constructed by computing the similarity scores from both the WL optimal assignment kernel and the shortest path kernel and then integrating these scores. The integration can be achieved through techniques such as kernel addition or concatenation, where the final similarity score is a weighted combination of the individual scores from each kernel.

By combining these two powerful graph kernels, the classification process benefits from a more holistic analysis of CFGs, leading to more accurate and reliable detection of vulnerabilities at the function level.

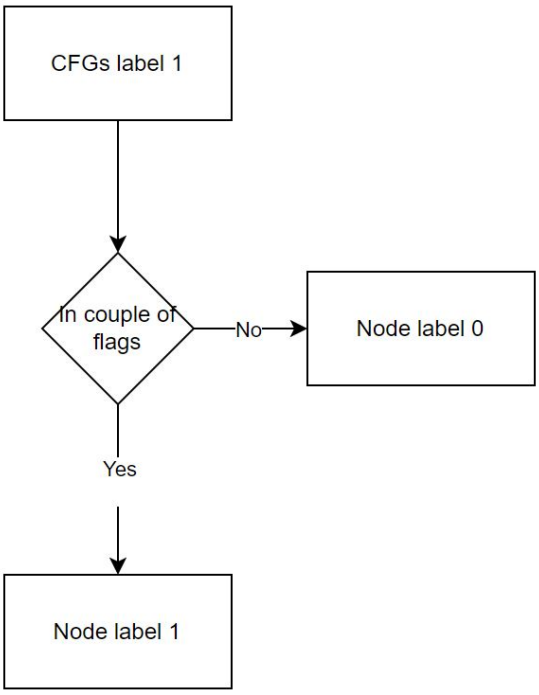


Figure 5.1: The process of node labeling

5.2 Vulnerability detection in line level

5.2.1 Nodes features extraction and labeling

For vulnerability detection at the line level, it is essential to extract and label features from the nodes of the CFGs. This process involves identifying vulnerable nodes based on flagged instructions and extracting relevant features from each node to facilitate accurate classification. This process is illustrated by Figure 5.1

1. **Node Labeling**: In the vulnerable CFGs, nodes are labeled based on the presence of flagged instructions, which were previously identified and marked in the C/C++ source files before being compiled into object files. The labeling process is as follows:

    - Nodes built from instructions flagged as vulnerable in the C/C++ file are labeled as vulnerable nodes (label 1).
    - Remaining nodes, which do not contain flagged instructions, are labeled as non-vulnerable nodes (label 0).

2. **Nodes Features Extraction**: For each node in the CFG, various features are extracted to represent the characteristics of the instructions and their context within the graph. These features are crucial for training machine learning models to detect vulnerabilities. The features extracted for each node are as follows:

    - **Length**: The number of assembly instructions in the node. This feature helps in understanding the complexity and size of the basic block.
    - **Instruction Type**: The ASCII value of the first character of the first instruction type. This feature provides an initial categorization of the instruction type.
    - **Registers Used**: The count of registers (e.g., eax, ebx) used in the instructions. This feature indicates the level of register utilization.
    - **Constants**: The count of constants (numeric values) used in the instructions. This feature highlights the presence of hard-coded values in the code.
    - **Is Jump**: A boolean feature indicating whether the node contains any jump instructions (e.g., jmp, je, jne, jg, jl). This feature is essential for identifying control flow changes.
    - **Memory Address**: A boolean feature indicating whether the node references any memory addresses. This feature helps in identifying instructions that interact with memory.
    - **Is Function Call**: A boolean feature indicating whether the node contains a function call. This feature is crucial for detecting potential vulnerability points where external functions are called.
    - **In-Degree**: The in-degree of the node in the CFG, representing the number of edges entering the node. This feature provides information about the node's connectivity and significance in the control flow.
    - **Out-Degree**: The out-degree of the node in the CFG, representing the number of edges exiting the node. This feature also provides information about the node's connectivity and control flow impact.

By extracting and labeling these features, the nodes in the CFGs are transformed into a structured representation that can be used to train machine

learning models for line-level vulnerability detection. The detailed and comprehensive feature set ensures that the models can accurately distinguish between vulnerable and non-vulnerable nodes, improving the overall effectiveness of the vulnerability detection process.

5.2.2 Nodes classification

The classification of nodes within CFGs is a critical step in detecting vulnerabilities at the line level. In this approach,we employ several classifiers for this task, including Random Forest, XGBoost, CNN, and MLP.

Chapter 6

Experiments

6.1 Vulnerability detection in function-level

6.1.1 Dataset

The dataset for vulnerability detection in function level after was construct to CFGs contain 71,374 CFGs which present for 71,374 functions. We experimentally evaluate our models by dividing the dataset as shown in Table 6.2.

6.1.2 Evaluation matrix

In evaluating function-level vulnerability predictions, we compare our graph kernel model with the LINEVUL model, which serves as a baseline. Similar to LINEVUL, we assess our model using three binary classification measures: Precision, Recall, and F1-score. These metrics provide a comprehensive evaluation of the model's performance in identifying vulnerabilities.

Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positives. It measures the accuracy of the positive predictions made by the model.

$$\text{Precision} = \frac{TP}{TP + FP}$$

	Label	Number of functions
1	0	35,687
2	1	35,687

Table 6.1: Label and number of functions of labels in dataset

	Label	Number of functions
1	train	70%
2	valid	10%
3	test	20%

Table 6.2: The data splits of functions

where:

- *TP* (True Positives) is the number of correctly predicted vulnerable functions.
- *FP* (False Positives) is the number of non-vulnerable functions incorrectly predicted as vulnerable.

Recall: Recall, also known as Sensitivity or True Positive Rate, is the ratio of correctly predicted positive observations to the actual positives. It measures the model's ability to identify all relevant instances of vulnerabilities.

$$\text{Recall} = \frac{TP}{TP + FN}$$

where:

- *TP* (True Positives) is the number of correctly predicted vulnerable functions.
- *FN* (False Negatives) is the number of vulnerable functions incorrectly predicted as non-vulnerable.

F1-score:
The F1-score is the harmonic mean of Precision and Recall.

6.1.3 Setup

To evaluate the effectiveness of our approach for function-level vulnerability detection, we have utilized three graph kernels: Weisfeiler-Lehman Optimal Assignment (WLOA), Shortest Path, and a combination of WLOA and Shortest Path. These kernels are compared with the baseline model, LINEVUL.

Implementation:

- We use the `weisfeiler_lehman`, `shortest_path`, and `weisfeiler_lehman_optimal_assignment` classes from the GraKel library to implement these graph kernels.
- For classification, we use the `svc` (Support Vector Classifier) from the `sklearn.svm` module.

6.1.4 Result and discussion

1. **Result**

Models	F1	Precision	Recall
WLOA	0.79	0.79	0.78
Shortest path	0.80	0.79	0.79
WLOA + SP	0.81	0.81	0.81
LINEVUL	0.99	0.98	0.99

Table 6.3: The results for vulnerability detection in function level

Table 6.3 shows the results on test data. Among the three graph kernels, the combination of Weisfeiler-Lehman Optimal Assignment (WLOA) and Shortest Path (SP) achieves the highest results. However, these results are still lower compared to the LINEVUL model.

2. **Discussion**

   (a) **Reasons for LineVul's effectiveness at the function level**
   The LineVul model incorporates several techniques such as tokenization and utilizes the BERT architecture which includes self-attention layers. These layers excel in capturing long-term dependencies within lengthy sequences through dot-product operations. Rather than relying on project-specific training data, we make use of the CodeBERT pre-trained language model to produce vector representations of source code. CodeBERT underwent training on a substantial dataset comprising 20GB of code, employing a Robustly Optimized BERT pre-training methodology. As a result, LineVul can comprehend a greater breadth of lexical and logical semantics from the given code input, thereby producing a more insightful vector representation.

   (b) **The drawback of graph kernel**

	Label	Number of nodes
1	0	843,900
2	1	35,687

Table 6.4: Label and number of nodes of labels in dataset

	Label	Number of nodes
1	train	70%
2	valid	10%
3	test	20%

Table 6.5: The data splits of nodes

Graph kernels primarily focus on the structure of graph, often ignoring node features. They integrate node labels or attributes, but they do not naturally handle complex node features, such as high-dimensional vectors. In this case, they only processed strings of node labels.

Graph Kernels often rely on pre-defined and hand-crafted features based on graph topology (such as walks, cycles, and subgraphs). While effective at capturing local and basic structural information, these features may not adequately represent high-order interactions that involve complex relationships and dependencies between nodes over larger graph regions. In the context of code representation, they fail to capture high-order interactions among instructions.

6.2 Vulnerability detection in line-level

6.2.1 Dataset

The dataset for vulnerability detection in line level after extraction of CFG nodes contains 879,587 nodes that represent 879,587 sequences of instructions. We experimentally evaluate our models by dividing the dataset as shown in Table 6.5.

6.2.2 Evaluation matrix

To evaluate LineVul approach for line-level vulnerability localization, they use the top-15 Accuracy measure. This metric assesses the percentage of vulnerable functions where at least one actual vulnerable line appears in the

top-15 ranking of predicted lines. The intuition behind this metric is that security analysts are likely to prioritize lines that rank highly in vulnerability predictions, similar to how users interact with recommendations in other systems. By focusing on the top-15 lines, this metric provides a practical measure of the approach's effectiveness in guiding security analysts towards the most critical code sections. Top-15 accuracy measures the ability of the model to correctly identify at least one vulnerable line within the top-15 predicted lines for each vulnerable function. The steps to calculate top-15 accuracy are as follows:

Function-Level Prediction: The model is trained for function-level vulnerability detection and used to predict which functions are vulnerable.

Line-Level Attention Scores: For each correctly predicted vulnerable function, the model calculates an attention score for each line within the function. Attention scores indicate the likelihood that each line is vulnerable.

Top-15 Line Ranking: The lines within each correctly predicted vulnerable function are ranked based on their attention scores. The top-15 lines with the highest attention scores are selected.

Correct Line-Level Prediction: If at least one actual vulnerable line appears in the top-15 ranked lines, it is considered a correct line-level prediction. The model does not consider how many vulnerable lines are present in the top-15, only whether at least one is correctly identified.

Top-15 Accuracy Calculation: Top-15 Accuracy is calculated as the ratio of the total number of correct line-level predictions to the total number of correctly predicted vulnerable functions.

$$\text{Top-15 Accuracy} = \frac{\text{Total Correct Line-Level Predictions}}{\text{Total Correct Vulnerable Functions}}$$

The average count of instructions within a vulnerability node is 15. Consequently, to evaluate four classification models—random forest, Xgboost, CNN, and MLP—we employ these models to identify which node in a CFG represents a vulnerability node.

6.2.3 Setup

We have utilized four classification models: random forest, Xgboost, CNN, MLP compare with baseline - LineVul for line-level vulnerability detection.

Implementation:

- We use the RandomForestClassifier from the sklearn.ensemble module to implement the Random Forest classifier.

- The XGBoost classifier is implemented using the XGBClassifier from the xgboost library.
- CNN and MLP models, we use the PyTorch library. The torch.nn module is used to build the neural network layers, and the torch.optim module is used for optimization.

6.2.4 Result and discussion

1. **Result**

Models	**Top-15 accuracy**
Random forest	0.93
XGboost	0.91
CNN	0.48
MLP	0.43
LineVul	0.76

Table 6.6: The results for vulnerability detection in line level

Table 6.6 shows the results on test data. The Random Forest model achieves the highest top-15 accuracy of 0.93, significantly outperforming the baseline LineVul model, which has an accuracy of 0.76. XGBoost also performs exceptionally well, with a top-15 accuracy of 0.91. In contrast, the CNN and MLP models achieve lower accuracies of 0.48 and 0.43, respectively.

2. **Discussion**

   (a) **The characteristic of data**

   Complexity and Dimensionality: Our data is technically complex as it encompasses numerous fine details about the operation of assembly code and the relationships between code blocks.

   Non-linearity: The relationships between features (such as jump instructions and control flows) are not linear and require complex machine learning models to accurately understand and predict.

   Imbalance: Our data is imbalanced, for example, the number of samples in class 0 is more than number of of samples in class 1, the number of jump instructions is less than other types of instructions.

Diversity in Features: Our data comprises a variety of different features, from quantities to types of data, necessitating the use of complex machine learning techniques for effective processing.

Our data is a typical example where ensemble classifiers can be very effective, especially in reducing the impact of noise and detecting complex patterns through the combination of various models.

(b) **Reasons for effectiveness of RF and XGBoost for our data**

Both RF and XGBoost are types of ensemble classifiers.

Random Forest is an example of bagging (Bootstrap Aggregating). In this method, multiple decision trees are independently trained on different subsets of the data, which are randomly sampled with replacement from the original data set.

XGBoost is an example of boosting. This is an ensemble method in which models are built sequentially to improve upon previous models.

(c) **Reasons for limited results of CNN and MLP**

CNNs and MLPs are not ensemble classifiers. They both construct a single model that computes the output based on a specific neural network architecture. Although there may be multiple layers in each network, each layer is not an independent model and cannot function independently; instead, it must work in close conjunction with the other layers.

Chapter 7

Conclusion and Future Work

Conclusion

In this thesis, we explored various methods for vulnerability detection in binary code at both the function and line levels. Our approach involved constructing and analyzing Control Flow Graphs (CFGs) from binary code, extracting meaningful features from these graphs, and employing multiple machine learning models to detect vulnerabilities. The models compared include graph kernel, Random Forest, XGBoost, Convolutional Neural Networks, Multi-Layer Perceptrons , and the LineVul model.

Our experimental results indicate that ensemble methods such as Random Forest and XGBoost outperform deep learning models like CNN, MLP and LineVul in terms of top-15 accuracy for line-level vulnerability detection. These ensemble methods leverage the diverse and high-dimensional features extracted from CFG nodes more effectively, providing robust and accurate predictions. Despite the lower performance of graph kernel methods compared to the LineVul model at the function-level, they offer valuable insights and demonstrate the potential for further improvement.

Future Work

To enhance the effectiveness of our vulnerability detection approach, several areas for future work are identified:

- Expanding the dataset with a larger number of test cases can significantly improve the training and evaluation of machine learning models.
- Integrating graph kernel methods with deep learning approaches. For example, KerGNNs: Interpretable Graph Neural Networks with Graph Kernels[22].

Bibliography

[1] Lee, Young Jun and Choi, Sang-Hoon and Kim, Chulwoo and Lim, Seung-Ho and Park, Ki-Woong. Learning binary code with deep learning to detect software weakness, *KSII the 9th international conference on internet (ICONI) 2017 symposium*, 2017

[2] Lee, Yongjun and Kwon, Hyun and Choi, Sang-Hoon and Lim, Seung-Ho and Baek, Sung Hoon and Park, Ki-Woong. BVDetector: A program slice-based binary code vulnerability intelligent detection system, *Information and Software Technology*, vol.123, pp.106289 (2020)

[3] Mikolov, Tomas and Chen, Kai and Corrado, Greg and Dean, Jeffrey. Efficient estimation of word representations in vector space, *arXiv preprint arXiv:1301.3781 Sciences*, 2013

[4] Tian, Junfeng and Xing, Wenjing and Li, Zhen. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN, *Applied Sciences*, vol.09, no.19, pp.4086 (2019)

[5] Cho, Kyunghyun and Van Merriënboer, Bart and Gulcehre, Caglar and Bahdanau, Dzmitry and Bougares, Fethi and Schwenk, Holger and Bengio, Yoshua. Learning phrase representations using RNN encoder-decoder for statistical machine translation, *arXiv preprint arXiv:1406.1078*, (2014)

[6] Li, Zhen and Zou, Deqing and Xu, Shouhuai and Jin, Hai and Zhu, Yawei and Chen, Zhaoxuan. Sysevr: A framework for using deep learning to detect software vulnerabilities, *IEEE Transactions on Dependable and Secure Computing*, vol.19, no.4, pp.2244–2258 (2021)

[7] Li, Zhen and Zou, Deqing and Xu, Shouhuai and Ou, Xinyu and Jin, Hai and Wang, Sujuan and Deng, Zhijun and Zhong, Yuyi. Vuldeepecker: A deep learning-based system for vulnerability detection, *arXiv preprint arXiv:1801.01681*, (2018)

[8] Li, Xingzheng and Feng, Bingwen and Li, Guofeng and Li, Tong and He, Mingjin. A vulnerability detection system based on fusion of assembly code and source code, *Security and Communication Networks*, vol.2021, no.1, pp.9997641 (2021)

[9] Le, Tue and Nguyen, Tuan Vu and Le, Trung and Phung, Dinh and Montague, Paul and De Vel, Olivier and Qu, Lizhen. Maximal divergence sequential auto-encoder for binary software vulnerability detection, *International Conference on Learning Representations 2019*, (2019)

[10] Fu, Michael and Tantithamthavorn, Chakkrit. Linevul: A transformer-based line-level vulnerability prediction, *Proceedings of the 19th International Conference on Mining Software Repositories*, pp.608–620 (2022)

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

[12] Suneja, Sahil and Zheng, Yunhui and Zhuang, Yufan and Laredo, Jim and Morari, Alessandro. Learning to map source code to software vulnerability using code-as-a-graph *arXiv preprint arXiv:2006.08614*, (2020)

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

[14] Fan, Jiahao and Li, Yi and Wang, Shaohua and Nguyen, Tien N. AC/C++ code vulnerability dataset with code changes and CVE summaries, *Proceedings of the 17th International Conference on Mining Software Repositories*, pp.508–512 (2020)

[15] Brust, Clemens-Alexander and Sonnekalb, Tim and Gruner, Bernd. ROMEO: A binary vulnerability detection dataset for exploring Juliet through the lens of assembly language, *Computers & Security*, vol.128, pp.103165 (2023)

[16] Young, Nicholas. *An introduction to Hilbert space*, Cambridge university press, (1988)

[17] Breiman, Leo. Random forests, *Machine learning*, vol.45, pp.5–32 (2001)

[18] Cortes, Corinna and Vapnik, Vladimir. Support-vector networks, *Machine learning*, vol.20, pp.273–297 (1995)

[19] Chen, Tianqi and Guestrin, Carlos. Xgboost: A scalable tree boosting system, *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp.785–794 (2016)

[20] Rosenblatt, Frank. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*, Cornell Aeronautical Lab Inc Buffalo NY, (1961)

[21] LeCun, Yann and Bottou, Léon and Bengio, Yoshua and Haffner, Patrick. Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, vol.86, no.11, pp.2278–2324 (1998)

[22] LFeng, Aosong and You, Chenyu and Wang, Shiqiang and Tassiulas, Leandros. Kergnns: Interpretable graph neural networks with graph kernels, *Proceedings of the AAAI conference on artificial intelligence*, vol.36, no.06, pp.6614–6622 (2022)