University of Twente.

Faculty of Electrical Engineering, Mathematics
Computer Science Technology

Master Thesis

for

Master of Science (M.Sc.)

in

CyberSecurity(CybSec)

Automated Vulnerability Detection in Java Source Code using J-CPG and Graph Neural Network

---

Samarjeet Singh Patil, s2078449

Department of EEMCS(CYBSEC)

February 2021

Graduation Committee

Dr.Ing. E. Tews	**Prof.Dr. M. Huisman**	**Dr.Ir. D.C. Mocanu (Decebal)**
Associate Professor	Full Professor	Associate Professor
University of Twente	University of Twente	University of Twente
e.tews@utwente.nl	m.huisman@utwente.nl	d.c.mocanu@utwente.nl

Contents

List of Figures iv

List of Tables viii

Acknowledgement xi

Abstract xii

1 Introduction 1

1.1 Problem 1

1.2 Research Goal and Questions 4

1.3 Outline of the thesis 5

2 Background 6

2.1 Graphical Representation 6

2.1.1 Abstract Syntax Tree(AST) 7

2.1.2 Control Flow Graph(CFG) 9

2.1.3 Program Dependence Graph(PDG) 10

2.1.4 Code Property Graph(CPG) 12

2.2 Neural Networks 13

2.2.1 Introduction 13

2.2.2 Multi Layer Perceptron(MLP) 17

2.2.3 Word Representation 17

2.3 Graph Neural Networks 21

2.3.1 Introduction 21

2.3.2 Functionality 21

2.3.3 Training 23

2.4 Transfer Learning 28

2.4.1 Transfer Learning in Graph Neural Networks 29

3 Related Work 32

3.1 Static Analysis Tool(SATs) for vulnerability detection 32

3.2 Vulnerability detection approach based on traditional machine learning approach 33

3.3 Vulnerability detection approach using Graph based machine learning approach 38

4 Methodology & Implementation 41

4.1 Data Selection 42

4.2 Data Pre-Processing 42

4.2.1 Graphical Code Representation 43

4.2.2 Graph Attribute Embedding 49

4.3 Representation Learning 50

4.3.1 Pre-Training Graph Neural Network(GNN) 52

4.4 Classification 53

5 Evaluation 55

5.1 Dataset 55

5.2 Pre-Processing 55

5.2.1 JCPG Tool 56

5.2.2 Attribute Embedding: Word2Vec Model 57

5.3 Evaluation of the Model 59

5.3.1 Experiments 62

5.3.2 Comparison with Devign Model 81

5.3.3 Comparison with Static Tools 85

5.4 Evaluation Summary 86

6 Discussion 87

7 Limitation and Future Work **89**

7.1 Dataset 89

7.2 Approach 89

7.2.1 JCPG 89

7.2.2 Word2Vec Model 90

7.2.3 GNN and Classifier 90

8 Conclusion **91**

References **93**

A APPENDIX **98**

APPENDIX **98**

A.1 Basic Java Constructs 98

List of Figures

1 Example Code snippet 7
2 AST for example code shown in Figure 1 8
3 CFG for example code shown in Figure 1 9
4 CDG for example code shown in Figure 1 10
5 DDG for example code shown in Figure 1 10
6 PDG for example code shown in Figure 1 11
7 Property Graph 12
8 Code Property Graph for example code shown in Figure 1 14
9 Neural Network 15
10 Single Neuron 15
11 ReLU function 16
12 Sigmoid function 16
13 Multi Layer Perceptron 18
14 One-Hot vector representation 18
15 CBOW architecture 19
16 Skip-Gram architecture 20
17 Network Graph as example 22
18 Computational Graph for Node A 23
19 Multi-head attention 26
20 Injectivity in Computational Graph 27
21 Contextual Prediction from [32] 30
22 Attribute Masking from [32] 30
23 An illustrative example of the attributed graph generation procedure from [33] 31
24 Decomposition of conditional property for a node 32
25 Robust code analysis architecture 34

26 Overview of the approach for automatic feature learning for vulnerability prediction based on LSTM 36

27 Architecture overview of the proposed approach from [38] 37

28 Overview of the framework from [34] 39

29 Example Code Snippet from [74] 40

30 Joint Graph structure for example shown in figure 29 from [74] 40

31 Overview of the implementation architecture 41

32 Example source code snippet 44

33 Example source code snippet for ICFG 44

34 AST generated by JCPG tool for code shown in Figure 32 44

35 CFG generated by JCPG tool for code shown in Figure 32 45

36 ICFG generated by JCPG tool for code snippet shown in Figure 33 45

37 CDG for the example code snippet shown in Figure 32 45

38 DEF-USE analyses of example code 32 46

39 DDG of example code snippet shown in Figure 32 46

40 Code Property Graph for method addNumber from code snippet shown in Figure 33 47

41 Code Property Graph for code snippet shown in Figure 32 48

42 Code statement and Word2Vec dictionary generated for code snippet shown in Figure 32 50

43 Projection of code token for example code shown in Figure 32 in multi-dimensional space 50

44 Code Example of Integer OverFlow 52

45 Attribute Masking strategy on the CPG output for example code shown in Figure 44 52

46 Context Prediction strategy on the CPG output 53

47 Clusters of embedding for tokens in CWE-256 58

48 Binary Classification Confusion Matrix 60

49 Multi-Class Classification Confusion Matrix 62

50 Source code of Const.java 98

51 Source code of ClassTutorial.java 99

52 Source code for StaticBlock.java 99

53 Code Property Graph for 50 100

54 Code Property Graph for 51 101

55 Code Property Graph for 52 102

56 Source code of IfElse.java 103

57 Source code of TradFor.java 103

58 Source code of ForEach.java 104

59 Source code of While.java 104

60 Code Property Graph for 56 105

61 Code Property Graph for 57 106

62 Code Property Graph for 58 107

63 Code Property Graph for 59 108

64 Source code of While.java 109

65 Source code of Label.java 109

66 Code Property Graph for 64 110

67 Code Property Graph for 65 111

68 Source code for Switch.java 112

69 Source code of Synch.java 112

70 Code Property Graph for source code shown in Figure 68 114

71 Code Property Graph for source code shown in Figure 69 116

72 Source code of TryCheck.java 117

73 Source code of TryWithRes.java 117

74 Code Property Graph for 72 118

75 Code Property Graph for 73 119

76 Source code of TryMultiRes.java 120

77 Source code of TryFinally.java 120

78 Code Property Graph for 76 122

79 Code Property Graph for 77 123

80 Source code of Throw.java 124

81 Source code of Throws.Java 124

82 Code Property Graph for 80 125

83 Code Property Graph for 81 126

84 MultiThrowable.java from TomCat project 127

85 File-Level Code Property Graph for source code shown in Figure 84 130

86 CPG for method add from Figure 84 131

87 CPG for method getThrowables from Figure 84 132

88 CPG for method getThrowable from Figure 84 133

89 CPG for method size from Figure 84 134

90 JSON format CPG output for method add from Figure 84 135

91 GML format CPG output for method add from Figure 84 135

92 DOT format output for method add from Figure 84 136

List of Tables

1 Components used for the implementation of our approach and the library/frameworks used for these components . . . 54

2 OWASP Top 10 vulnerabilities of Java . . . 55

3 The filtered list of CWE entries with class distribution used for evaluation experiments . . . 56

4 List of source code files used for tool evaluation for Java construct . . . 57

5 GIT projects used for evaluation of JCPG tool . . . 57

6 The hyper-parameters of the GNN model . . . 59

7 Dataset for Multi-Class Classification . . . 61

8 Confusion Matrix for Cross-Site Scripting . . . 62

9 Confusion Matrix for SQL Injection . . . 63

10 Confusion Matrix for LDAP Injection . . . 63

11 Confusion Matrix for HTTP Response Splitting . . . 63

12 Confusion Matrix for XPath Injection . . . 64

13 Confusion Matrix for Plain-Text Storage of Credentials . . . 64

14 Confusion Matrix for Hard-Coded Credentials . . . 64

15 Confusion Matrix for Sensitive Data Exposure . . . 65

16 Confusion Matrix for Relative Path Traversal . . . 65

17 Confusion Matrix for Absolute Path Traversal . . . 65

18 Confusion Matrix for MultiSet-1 . . . 66

19 Confusion Matrix for MultiSet-2 . . . 66

20 Confusion Matrix for MultiSet-3 . . . 66

21 Confusion Matrix for MultiSet-4 . . . 67

22 Confusion Matrix for Cross-Site Scripting . . . 67

23 Confusion Matrix for SQL Injection . . . 68

24 Confusion Matrix for LDAP Injection . . . 68

25 Confusion Matrix for HTTP Response Splitting 68
26 Confusion Matrix for XPath Injection 69
27 Confusion Matrix for Plain-Text Storage of Credentials 69
28 Confusion Matrix for Hard-Coded Credentials 69
29 Confusion Matrix for Sensitive Data Exposure 70
30 Confusion Matrix for Relative Path Traversal 70
31 Confusion Matrix for Absolute Path Traversal 70
32 Confusion Matrix for MultiSet-1 71
33 Confusion Matrix for MultiSet-2 71
34 Confusion Matrix for MultiSet-3 71
35 Confusion Matrix for MultiSet-4 72
36 Confusion Matrix for Cross-Site Scripting 72
37 Confusion Matrix for SQL Injection 72
38 Confusion Matrix for LDAP Injection 73
39 Confusion Matrix for HTTP Response Splitting 73
40 Confusion Matrix for XPath Injection 73
41 Confusion Matrix for Plain-Text Storage of Credentials 74
42 Confusion Matrix for Hard-Coded Credentials 74
43 Confusion Matrix for Sensitive Data Exposure 74
44 Confusion Matrix for Relative Path Traversal 75
45 Confusion Matrix for Absolute Path Traversal 75
46 Confusion Matrix for MultiSet-1 75
47 Confusion Matrix for MultiSet-2 76
48 Confusion Matrix for MultiSet-3 76
49 Confusion Matrix for MultiSet-4 76
50 Summary of Evaluation results for Binary Classification task for the non-pretrained model and the pre-trained model 78

51 Summary of Evaluation results for Multi-Class classification task for the non-pretrained model and two pretrained model 80

52 Confusion Matrix for Cross-Site Scripting Vulnerability 81

53 Confusion Matrix for SQL Injection Vulnerability 82

54 Confusion Matrix for LDAP Injection Vulnerability 82

55 Confusion Matrix for XPath injection Vulnerability 82

56 Confusion Matrix for Plain-Text Storage of Credential Vulnerability 83

57 Confusion Matrix for Hard-Coded Credential Vulnerability 83

58 Confusion Matrix for Missing Encryption of Sensitive Data Vulnerability . 83

59 Comparison of Evaluation results of the pre-trained model and previous research model(Devign) 84

60 Comparison of Evaluation results of the pre-trained model and Static Analysis Tools(SATs) 85

61 Components used in the implementation of our approach 86

Acknowledgement

I want to extend my gratitude to the faculty of Electrical Engineering, Mathematics, and Computer Science(EEMCS) of the University of Twente to be a part of this institution as a student. With this thesis assignment, I conclude my master's program in computer science(CyberSecurity). I want to thank the computer science department for encouraging me to work on this assignment. I thank Dr.Ing. E. Tews for chairing the graduation committee and providing guidance throughout the thesis. I would like to thank Prof.Dr. M. Huisman and Dr.Ir. D.C. Mocanu for supervising me and providing me guidance throughout the course. I would like to thank Mr. David Vaartjes, the Co-Founder/Director of Agile Security, Securify, for introducing me to the topic of the assignment. Supervise me to complete the thesis with your valuable inputs and be an external member of the graduation committee.

Samarjeet Singh Patil

March 2021

Abstract

In this digital era, detecting a software vulnerability is a crucial yet daunting task to protect the systems from adversarial cybersecurity attacks. Although there has been researching in this direction, vulnerability detection remains open, evidenced by the numerous vulnerabilities reported daily. There are several tools available to mitigate the consequences of software vulnerabilities and improve system security. The traditional tools such as the static analysis tools can detect only generic errors using a list of pre-defined rules and vulnerability patterns or contradict expected software behavior. Hence, these tools cannot easily extend it to more specific vulnerability patterns without thoroughly studying the vulnerability and its causes. Additionally, a new set of modern tools inspired by machine learning models in text/speech processing, image processing, and computer vision are also available. However, these tools consider the source codes as flat sequences which do not alleviate the long-term dependency problem. The vulnerability within a source code must be identified at a finer granularity to localize the vulnerability and facilitate the fix. To alleviate these limitations, inspired by the recent development of Graph Neural Networks and their practical application in various fields, we explore Graph Neural Networks' applicability in learning the properties of source code from a security standpoint. We propose an automatic and intelligent vulnerability detection method that uses a tool operating at the source code level to provide an intermediate graphical representation of the source code and graph neural network-based model for vulnerability prediction at method-level granularity. Working towards this direction, we developed a tool called JCPG that operates at the source code level to capture the data and control flow analyses and generate an intermediate graphical representation of the source codes at the file level and the method level. Our approach uses the JCPG tool to represent source codes as graphs fed to a pre-trained GNN model to perform representation learning and then uses a multilayer perceptron model to perform the classification task. We report our experiments' results and show that our model outperforms the static analyzers and the previously used GNN models for the Juliet Java dataset. Thus, we confirm that using a tool that operates at the source code to generate an intermediate graphical representation combined with a highly expressive GNN model can be used as a vulnerability prediction tool that works even for source code that is not compilable.

1 Introduction

The advancement in technology has transformed the world into a digital society with computer systems at its core connecting various aspects of life, empowering people and businesses worldwide. Software governs these computer systems, which also acts as a medium for human-machine interaction. Although the software is programmed to carry out specific tasks, sometimes it fails to do so because of vulnerabilities in the program. There are numerous definitions of vulnerabilities; after summarizing these definitions, we define a vulnerability as "A fault in the design, development, or the configurational phase of the software that a threat vector can exploit explicitly or implicitly to cross the privilege boundaries within a system causing an error instance." We can refer to these vulnerabilities as weaknesses or backdoor in the system that allows an attacker to compromise one or more of the three essential elements of the security model, i.e., Confidentiality, Integrity, and Availability[9].

Although it is hard to quantify these vulnerabilities' adverse effects, the economic impact of vulnerabilities is catastrophic, which is evident because numerous companies lose millions of dollars because of the exploitation of vulnerabilities by malicious hackers. Hence, the computer systems thus require a significantly high level of security. Unfortunately, due to rapid technological changes, security remains an open problem since research for the ideal security approach or policy cannot keep up with the constant developments.

Human developers develop software, and it is inherently impossible to produce a perfect non-vulnerable code even with the most accurate debugging process. However, our aim should be to produce the best quality system to prevent significant damages due to small failure causing a domino effect. Working towards this direction, our first line of defense to produce secure and reliable software is to perform code reviews by testing and debugging the code, but this is a tedious and challenging job as the coding style and the software's size varies. At the same time, the vulnerabilities are nested and complex. Additionally, it requires reviewers with a certain level of background knowledge to review the codes. Hence, developers and code reviewers are looking for new methods to perform code reviews with less human intervention.

1.1 Problem

Identifying vulnerabilities in a source-code is a crucial yet challenging problem in the field of security. The primitive technique of manual code review requires a code inspector or security expert with a high-level understanding of the code semantics, i.e., sufficient experience and knowledge of the program and programming language, which is vital in the manual code auditing tasks [62]. The second approach is to use the traditional l mechanisms that can be categorized as static [66] [17], dynamic citeCerebro,[19],[39],[63],[67] and hybrid methods of vulnerability detection [54]. Static analysis can be employed in the early stages of the development cycle and have a high coverage but also incurs a high false-positive rate. Dynamic analysis methods find vulnerabilities by running the software program that has to be analyzed. Although it is under a low false-positive rate, its dependency on the test cases incurs low recall. A hybrid analysis approach can be either a static analysis system that leverages dynamic analysis to identify false vulnerabilities or a

dynamic analysis approach that leverages static analysis techniques to guide the test-case selection and analysis process. These rule-based approaches suffer from shortcomings. In addition to these techniques, inspired by the effectiveness of machine-learning techniques from the field of Artificial Intelligence (AI) in practice for multiple application areas, a different class of vulnerability detection techniques that utilize techniques from the field of data science and artificial intelligence are introduced.

We can classify the machine learning-based approaches into different taxonomy based on feature extraction techniques and the underlying vulnerability detection techniques. text-colorblue [27] introduces one such taxonomy that categorizes the approaches into four categories as follows:

1. Vulnerability prediction based on software metrics
2. Anomaly detection approach
3. Vulnerable code pattern recognition
4. Miscellaneous approaches

Vulnerability prediction models is inspired by the field of software quality and reliability assurance and uses data-mining, machine-learning, and statistical analysis techniques to predict vulnerable software artifacts based on well-known software engineering metrics as the feature set such as source-code size, complexity, code-churn, and developer-activity metrics **Anomaly Detection approaches** utilize machine-learning and data-mining techniques to identify software defects. It does it by finding locations in source code that do not conform to usual or expected code patterns for APIs, such as the function-call pair of malloc and free, lock and unlock, or API's share of rules and patterns. **Vulnerable code pattern recognition** utilizes machine-learning and data-mining techniques to analyze and automatically extract features and patterns from the binary machine code, high-level source code of the program, conventional code parser, static data-flow, and control-flow analysis. They are then used to discover software vulnerabilities through pattern-matching techniques. **Miscellaneous approaches** includes some of the notable works that utilize different techniques from the field of AI and data science that do not come under the other mentioned categories or constitute a logical category.

Vulnerability prediction using software metrics-based approaches does not analyze program syntax and semantics. Hence, it lacks better performance and accuracy, whereas anomaly detection, vulnerable code pattern recognition, and various approaches analyze program syntax-semantics to extract features for the vulnerability detection process. Although anomaly detection has the advantage of discovering unknown vulnerabilities, they have a high false-positive and false-negative. Since vulnerability code pattern recognition learns vulnerable patterns from a vulnerable and clean sample, it performs better on accuracy than anomaly detection. It is highly dependent on the quality of the dataset.

To alleviate the limitation of the early vulnerable code pattern recognition models, understanding the vulnerable pattern's underlying semantics as a security expert would cause a semantic gap between security experts and the Artificial Intelligence(AI) based detection system. This semantic gap [40] is the lack of coincidence between the abstracts semantic

meanings of a vulnerability that a practitioner can understand and the obtained semantics that a machine learning algorithm can learn. In most of the previous vulnerability code pattern recognition-based approaches, the source-code is treated as a flat sequence analogous to natural language and processed using Natural Language Processing(NLP) techniques. However, the source code is more structural and logical compared to natural languages. It can represent source-code as a data-structure such as Abstract Syntax Tree, Control Flow Graph, Data Flow Graph. Moreover, specific approaches transform source-code into intermediate graphical representation to capture the code's syntactic and semantic features and then perform machine-learning approaches. However, these approaches that transform source-code to graph structures do not work on the non-compilable source-code and have a coarse detection granularity. These features are a problem since the source-code might not always be fully executable due to issues with built configuration files or other organizational matters. Also, since the neural network's performance depends on the quantity and quality of the training data, the lack of a rich labeled vulnerability dataset limits the performance of neural network-based intelligent ways.

To this end, we propose a vulnerability detection tool based on graph neural networks with a composite intermediate representation of the source code that detects vulnerabilities at the method-level and even operates on non-compilable source codes. The intermediate graphical representation of the source code enables us to encode the semantics and syntactic features of the programming code to capture various vulnerabilities' properties. In this process, we developed a tool to create file-level and method-level code property graphs for Java programs that operate at the source code level and export the generated code property graph in various formats for various applications. The tool also works for source codes that are non-compilable due to missing packages. We then utilize this tool's output to detect vulnerabilities using a modified highly expressive graph neural network. This approach can be applied to every kind of vulnerability as it is general and does not prescribe any prior knowledge of the source code. Moreover, the approach's vulnerability pattern knowledge base can be extended and improved by providing a new vulnerability or a revised dataset for a preceding vulnerability.

1.2 Research Goal and Questions

This research aims to create a tool that performs method-level vulnerability detection in a Java source code even if the source code is non-compilable. This research will infuse an understanding of various technologies and their implementation to create a generalized approach to find vulnerabilities in a java source code.

The initial step in conducting a research thesis comprises framing the primary research objective and the sub-research objectives. This step enables us to provide a well-structured framework to conduct the research study and streamlines the results accordingly. Hence, the following are the research questions in focus for this research.

1. **RQ1:** How to capture the syntactic and semantic properties of a non-compilable Java source code?
   In most previous research, the generation of intermediate graphical representation capturing the Java source code's syntactic and semantic features is by operating on a compiled source code. In most cases, the auditors have the source code that lacks some of the required files and packages that make the source code non-compilable. We need a technique to capture the source code's syntactic and semantic features even from non-compilable source codes.

2. **RQ2:** How to utilize the Graph Neural Networks(GNNs) to alleviate the long-term dependency issue in vulnerability detection?
   In source code, code elements have relationships defining the code's syntactic and semantic features, but there are long-term dependencies in some vulnerable code samples. The previous approaches fail to capture such long-term dependencies, which can be alleviated using the graph embedding methods. Hence, we evaluate how we can use Graph Neural Networks to alleviate these limitations and increase vulnerability detection methods' effectiveness and performance.

   (a) **RQ2a:** In particular, how to utilize the Graph Isomorphism Network (GIN) to alleviate the long-term dependency issue?
   Graph Isomorphism Network (GIN) is the Weisfeiler-Lehman test-based Graph Neural Network. Theoretically, Graph Isomorphism Network(GIN) expressivity is greater than other anisotropic graph neural networks. We evaluate how we can benefit from this expressivity can for the downstream task of vulnerability detection.

3. **RQ3:** How to alleviate the issue of lack of an attested vulnerability dataset?
   The quality and quantity of the dataset determine the effectiveness of an approach. Hence the performance of an approach is limited by the quality of the dataset used. Since there is a lack of an attested vulnerability dataset, it can hinder the approach's effectiveness. We will evaluate how we can utilize the Pre-Training methods to alleviate this problem.

1.3 Outline of the thesis

The following chapter-wise distribution is followed to answer the research questions and sub-research questions listed in section 1.2.

Chapter 1: Introduction
This chapter introduces the research topic to understand the context and knowledge gap addressed by framing the research questions.

Chapter 2: Background
This chapter introduces the various technologies used in implementing the approach to understanding the rationale behind selecting the technologies. This introduction provides readers the required technical background to apprehend the steps and technologies used to answer the research questions and sub-research questions.

Chapter 3: Related Work
This chapter includes selecting the various researches taken as a reference to understand the underlying research gap in the application of machine learning methods in software security. It provides the background information regarding the application of machine learning methods in vulnerability prediction by categorizing the previous researches into two categories. The first category is the work that utilizes machine learning methods considering the source code as a flat sequence like natural text, image, and speech. The second category refers to the works where it transforms source code is into graphical representations, and then these graphical representations are used by the graph-based machine learning approaches.

Chapter 4: Methodology & Implementation
This chapter includes the methodology and its implementation to solve the research questions and sub-research questions introduced in Chapter 1.

Chapter 5: Evaluation
This chapter includes an evaluation of the approach and provides the results of the experiments conducted. First, it provides the evaluation results for each component. Secondly, it provides the evaluation of the approach as a whole, followed by comparing the approach with previous approaches and static analysis tools.

Chapter 6: Discussion
This chapter includes a discussion on the outcomes of the experiments performed in Chapter 6 and highlights the essential aspects.

Chapter 7: Limitation and Future Work
This chapter highlights the limitations of the research work and provides possible solutions for the same. Additionally, it also provides suggestions for extending or enhancing the research work by recommending future research pathways in the domain.

Chapter 8: Conclusion
This chapter concludes the research thesis by ensuring that it answers all the research questions introduced in Section 1.

2 Background

In the following section, we will introduce the technical background required for the implementation of our approach for the reader to understand the rationale behind the methodology, which we discuss in Chapter 4. We will introduce the required technical background based on the three stages of the approach: the Data Creation & Pre-Processing stage, Representation Learning stage, and Classification stage that we describe briefly below.

- **Data Selection & Pre-Processing:** An attested vulnerability dataset is containing Java source code files used as the input to the tool. To perform further analysis on this input, first, the data, i.e., the source code files, are pre-processed to transform the data into an appropriate input data format.

  Pre-Processing is a two-step process that involves the static analysis of the source code to extract the source code's properties and represent them as a graphical data structure, i.e., a Code Property Graph(CPG). Then embedding the graph attributes to capture the semantic meaning of the attributes. The graphical data structure Code Property Graph(CPG) as explained in subsection 2.1.4 is a data structure that combines the abstract syntax trees (AST), control flow graph (CFG), and program dependence graph (PDG) concepts. Each of the concepts is well-explained in subsections 2.1.1, 2.1.2, and 2.1.3 respectively to form the joint data structure capturing the advantages of each format.

  In this step, we use the pre-processed data, i.e., the intermediate graphical representation of the source code, as the input to learn the graph's syntactic and semantical features and output a global vectorized representation of the graph. To achieve this, we used various machine learning models that are explained briefly in the section 2.2.

- **Classification:** In this step, we use a graph-based machine model to classify a given source code input. The machine learning model used is explained in the section 2.2.

Based on the steps described above, first, we will introduce the various graphical representation of the source code that jointly form the final graphical representation, i.e., the code property graph, which is used for the program analysis. Then, we will introduce Neural Networks and the various concepts of neural networks used in the implementation, followed by the introduction of Graph Neural Networks(GNN) and their concepts.

2.1 Graphical Representation

In the Data Pre-processing stage, as the name suggests, before using the dataset for the downstream task, we process the input data to transform it into a form suitable for further processing. The first step in pre-processing is to perform static analysis of the program, which, unlike dynamic analysis, does not require the program's execution and is performed directly on the software program's source code. This analysis is performed to capture/extract the source code's properties necessary for the intermediate representation of the source code into some data-structure that highlights the source code's various elements and their interaction to be used for further analysis.

```c
#include <stdio.h>

int main()
{
   int x, y, temp, gcd;

   printf("Enter two integer: \n");
   scanf("%d %d", &x, &y);

   while (y != 0)
   {
      temp = y;
      y = x % y;
      x = temp;
   }

   gcd = x;
   printf("GCD of given integers is: %d", gcd);

   return 0;
}
```

Figure 1: Example Code snippet

In our case, we will perform static analysis on the source code to capture the source code's semantic and structural features and represent the source code as a graph with the source code elements as the nodes and the edges of the graph defining the interactions and dependencies between these elements. There are various graphical representations for source code, with each representation capturing the source code's specific properties. The three classical graphical representations of the source code are Abstract Syntax Tree(AST), Control Flow Graph(CFG), and Program Dependence Graph(PDG). In our case, we would like to utilize the benefits of the properties captured by all three representations, and to do this; we use Code Property Graph(CPG) as the graphical representation of the source code. A code property graph(CPG) is a joint data structure proposed by Fabian Yamaguchi [68] that combines the abstract syntax tree, the control flow graph, and the program dependence.

In this section, we present the concept of Abstract Syntax Tree(AST), the Control Flow Graph(CFG), and the Program Dependence Graph(PDG), followed by the Code Property graph and its construction using an example source code shown in Figure ??.

2.1.1 Abstract Syntax Tree(AST)

Abstract Syntax Tree(AST) is an intermediate ordered tree that forms the basis for the generation of higher-order graphical code representation and is used in compilers to check code for accuracy and to identify semantically similar codes [10][72]. It is an abstract syntactic structure of the code parser's source code with the tree nodes encoding how the statements and expressions are nested. Unlike the concrete syntax tree, it does not represent the concrete chosen to express its program.

In this tree structure, starting from the node, the program is categorized into code blocks, statements, declarations, expressions, etc. Further categorized into primary tokens forming the leaf nodes of the tree structure. The inner node in an AST graph represents the operators and the leaf node represents the operands as shown in Figure 2 for the source code given in Figure 1. Since ASTs lack the explicit representation of the control flow and the program's data dependencies, they cannot be used for higher-order code analysis.

Figure 2: AST for example code shown in Figure 1

2.1.2 Control Flow Graph(CFG)

Control Flow Graph(CFG) is defined as a graph $G = (N, E)$ where N is a finite set of nodes where each node is a basic that represents the statement and predicates of the source code. E is a finite set of directed edges where an edge $e_{i,j}$ connects two nodes $n_i, n_j \in N$. It describes the traversal of control flow from node n_i to n_j within a program depicting the code statement execution order based on the conditions to be satisfied. Based on the node, every edge in the graph is assigned a label of true, false, or ϵ and does not require ordering, like the abstract syntax tree. A statement node will have one outgoing edge labeled as ϵ. In contrast, a predicate node will have two outgoing edges with labels true and false, representing the predicate's evaluation(true or false).

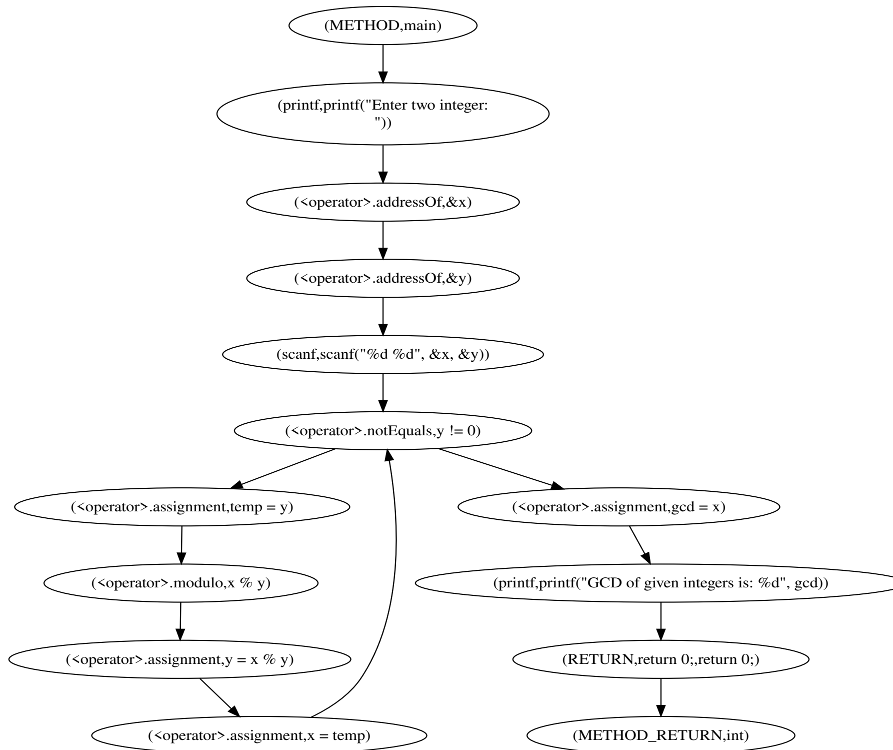


Figure 3: CFG for example code shown in Figure 1

To construct a CFG from the base AST, first, a preliminary CFG is constructed using the structured control statements within the program like 'while', 'if', and 'for'. Additionally, this preliminary CFG is updated using the unstructured control statements within the program like 'goto', 'break', and 'continue'. The Figure 3 shows the control flow graph for the code sample in Figure 1.

From the security perspective, a control flow graph has become a primitive code representation to understand a program in reverse engineering as it exposes the control flow of an application. It is used to ensure secure coding of the applications by guiding fuzz testing tools [56] and by detecting variants of known malicious applications [25]. However, since the Control Flow Graph does not represent the program's data dependence edges, it fails to capture the statements processing the data influenced by an attacker.

2.1.3 Program Dependence Graph(PDG)

Program Dependence Graph(PDG) is the second widely used directed graph for program representation where the program statements constitute the nodes instead of the basic blocks. PDGs explicitly represent two types of dependencies among the statements and predicates of a program and are used as an intermediate representation [22]. The PDG is a joint data structure that combines the Data Dependence Graph(DDG), which constitutes the data dependency edges as shown in Figure 5 representing the dependability/influence of one variable on another and the Control Dependence Graph(CDG) which constitutes of the control dependency edges that represents the influence of a predicate on the values of variables as shown in Figure 4.

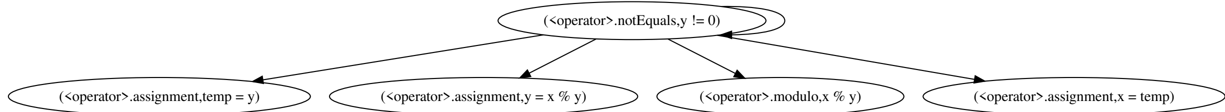


Figure 4: CDG for example code shown in Figure 1

The construction of PDG from a control flow graph requires the following steps: first, we have to determine the DEF-USE pair set, which constitutes the set of variables *defined* and the set of variables *used* by each program statement and followed by calculation of *reaching definitions* for each statement and predicate.

Reaching Definition can be defined as an association between the definition and use of a variable 'V' defined at position 'P' and used at position 'Q' if and only if there exists at least one control flow path from 'P' to 'Q' such that there is no redefinition of the variable 'V' along the path. [?] Figure 6 shows the program dependence graph for the code sample

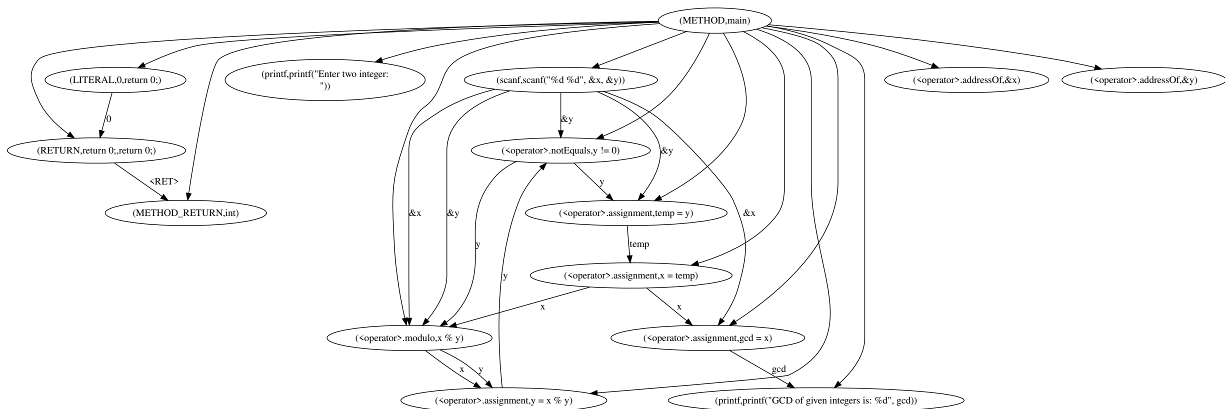


Figure 5: DDG for example code shown in Figure 1

given in Figure 1 where the control dependence edges are labelled as CDG and the data dependence edges are labelled as DDG. Although the graph does not reflect the order of statement execution, it reflects the control and data dependencies between the statements and predicates.

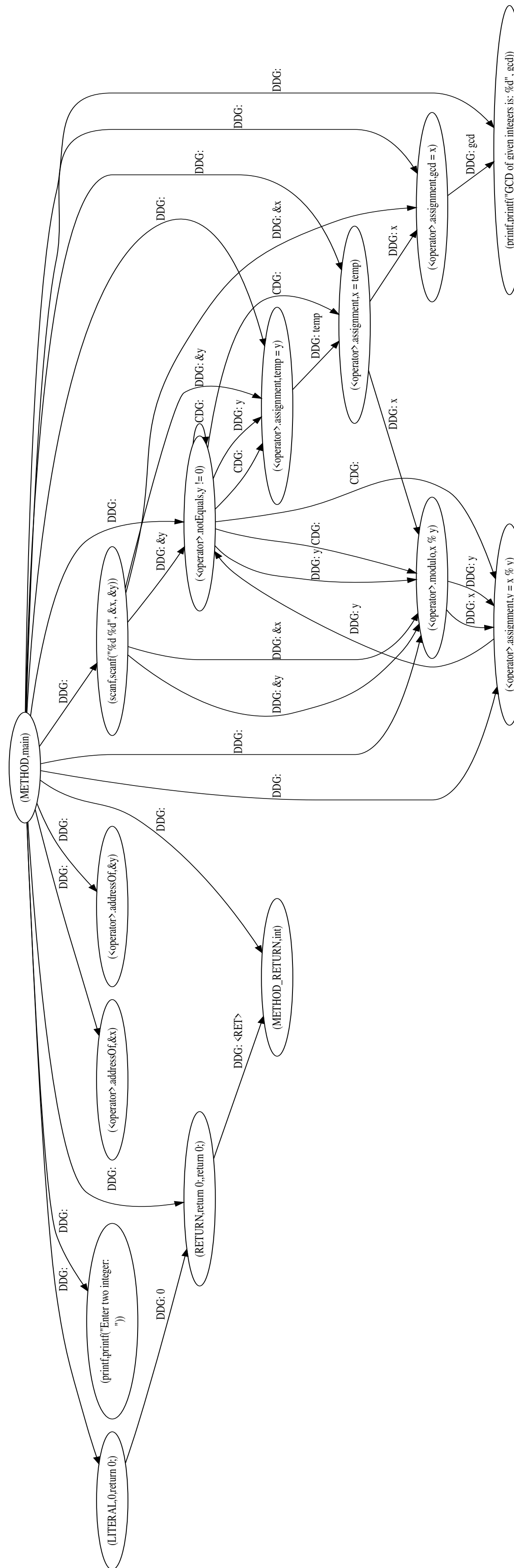


Figure 6: PDG for example code shown in Figure 1

2.1.4 Code Property Graph(CPG)

Although the Abstract Syntax Tree(AST), Control Flow Graph(CFG), and Program Dependence Graph(PDG) capture certain properties of the underlying program, in the majority of the cases, each representation alone is insufficient to characterize all the properties of a source code. To alleviate these limitations, a novel representation called a Code Property Graph(CPG) that combines properties of the three graphs in a joint data structure using the concept of property graphs described below was introduced in [68].
Property Graph: A property graph [48] is used to represent complex domain models with heterogeneous edges where the edges are labeled or typed. The edges along the vertices maintain a set of key-value pairs known as properties that allow non-graphical data representation. A property graph can be formally defined as a labeled, multi directed graph $G = (V, E, \lambda, \mu)$, where V is a set of nodes, E is a set of the edges that are directed and labeled (i.e. $E \subseteq (V \times V)$, λ is an edge labeling function where $(\lambda : E \to \sigma))$, and $\mu : (V \cup E) \times R \to S$ is a function that assigns properties to edges and nodes that are a map from elements and property keys (K) to property values (S).
Abstract Syntax Tree: Abstract Syntax Tree modelled as a property graph is defined

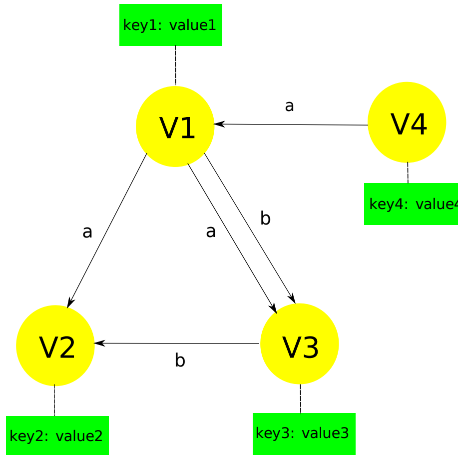


Figure 7: Property Graph

as a directed labelled multi-graph $G_{AST} = (V_{AST}, E_{AST}, \lambda_{AST}, \mu_{AST})$ where V_{AST} is the set of AST nodes and E_{AST} is the set of AST edges connecting the respective nodes. The edges are labeled by a function λ_{AST} that describes the relationship types between the parent and child node and μ_{AST} is a function that assigns properties to the nodes and edges.
Control Flow Graph: Control Flow Graph(CFG) modeled as a property graph is defined as a directed labeled multi-graph $G_{CFG} = (V_{CFG}, E_{CFG}, \lambda_{CFG}, \mu_{AST})$ where V_{CFG} is a set of nodes defining the statements and predicates of the source code and the edges of the graph are assigned the labels for identification using the function λ_{CFG} while the rest of the graph remains the same as the Abstract Syntax Tree.
Program Dependence Graph: A Program Dependency Graph(PDG) modeled as a property graph is defined as a directed labelled multi-graph $G_{PDG} = (V_{CFG}, E_{PDG}, \lambda_{PDG}, \mu_{AST})$ with new set of edges E_{PDG}. The edges of the graph are assigned labels using edge labeling function $\lambda_{PDG} : E_{pdg} \to \sum_{PDG}$ where $\sum_{PDG} = C, D$ with C and D corresponding to control and data dependencies respectively where a property *condition* that indicates the result of the predicate is assigned to control dependency and a property *symbol* that indicated the corresponding symbol is assigned to data dependency.

Construction of Code Property Graph: To construct the Code Property Graph(CPG), first, the Abstract Syntax Tree(ASTs), Control Flow Graph(CFGs), and Program Dependency Graph(PDGs) of the program are modelled as property graphs as shown above and then these property graphs are merged into a graph representation combining all the properties of the individual representations. Formally, a code property graph(CPG) is a property graph constructed from the abstract syntax tree, control flow graph and program dependence graph of the program and is defined as $G_{CPG} = (V, E, \lambda, \mu)$ where V is a set vertices and $V = V_{AST}$, $E = (E_{AST} \cup E_{CFG} \cup E_{PDG})$ is a set of edges, $\lambda = \lambda_{AST} \cup \lambda_{CFG} \cup \lambda_{PDG}$ is a edge labelling function and $\mu = \mu_{AST} \cup \mu_{PDG}$ is a function that assigns property to each node in the graph. Figure 8 shows the code property graph for the code sample shown in Figure 1.

2.2 Neural Networks

In the pre-processing data stage, the second step is 'Attribute Embedding.' The third and fourth stages are the Representation Learning stage and the classification stage, respectively. We present a brief introduction to Neural Networks followed by a brief introduction to the various neural network concept we utilize for the steps described above in the following section.

2.2.1 Introduction

A Neural Network is a computer algorithm analogous to the human brain. Its objectives are to perform the human brain's cognitive functions to capture the underlying relationships in a set of data. A neural network consists of an arbitrary number of layers of neurons where each layer can consist of an arbitrary number of nodes as shown in Figure 9. The first layer is called the input layer, the last layer is called the output layer, and the intermediate layers are called the hidden layers. Every node in the first layer is interconnected with each node in the next layer, and these interconnections between the layers are called weights(W) as shown in Figure 9. At every layer except the final layer, there is an additional weight without an input term called Bias which is independent of the previous layer and is applicable to provide an extra bit of adjustability.

Now to understand the function of these weights, consider Figure 10. A single node N_1^2 (where the subscript value represents the node id and the super-script value represents that layer number) in the hidden layer connects to all nodes in the input layer where each node has its values, and each connection has a specific weight. The intermediate value of the node N_1^2 is represented by equation 1 where x_n represents the value of input nodes, w represents the weights between the nodes, and b is the bias. Then, this intermediate value of the node is used as input to the activation function, and the output of the activation function is the final state value of the node N_1^2.

$$H = (x_1 * w_1 + x_2 * w_2 + x_3 * w_3) + b \tag{1}$$

For the calculating output between two complete layers, the intermediate node values can be represented by equation 2 where $W \in R^{M \times N}$ is the weight matrix where M is the

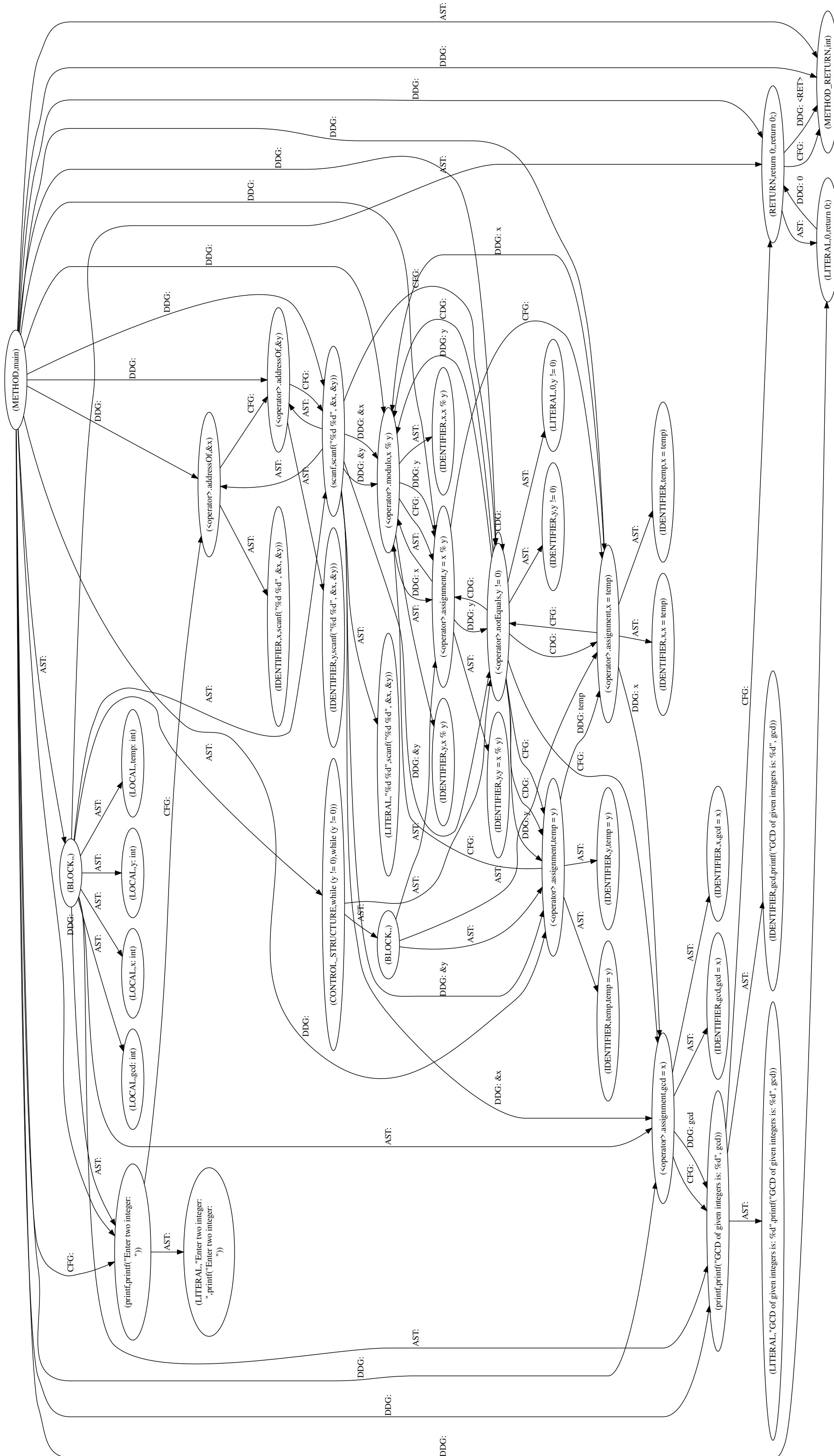


Figure 8: Code Property Graph for example code shown in Figure 1

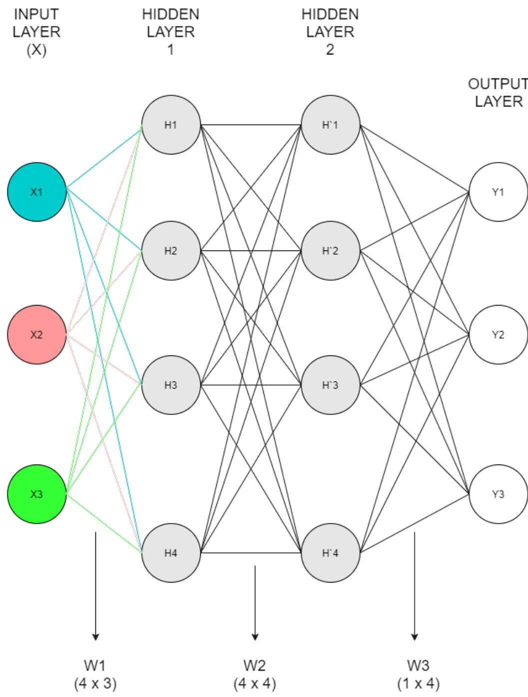


Figure 9: Neural Network

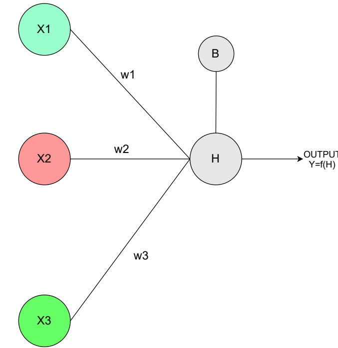


Figure 10: Single Neuron

number of output nodes and N is the number of input nodes, X is the input vector and B is the bias vector.

$$H = (W * X) + B \tag{2}$$

$$Y = f(H) = f(W * H + B) \tag{3}$$

The output vector is defined by equation 3 where $f(.)$ is the activation function. The activation function is a non-linear function used to provide non-linearity in the output because without an activation function the output will have a linear relationship with the input vector in equation 3. There are various activation functions, we will briefly introduce some of the popular activation functions below.

- ReLU: Rectified Linear Unit(ReLU) is also known as a ramp function that provides the positive part of the input argument. When the input x is greater than 0, then it functions as an identity function $y = x$ and becomes 0 otherwise as shown in equation4. Figure 11 shows the plot for a ReLU function.

$$ReLu = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \tag{4}$$

- Sigmoid: Sigmoid is a function bounded between 0 and 1 that corresponds to the output probability. As shown in the equation 5, the function squishes the extremely high and low values as it outputs 0 if the value of the input is a large negative number and outputs 1 if the value of the input is a large positive number. Figure 12 shows the plot for a sigmoid function.

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{5}$$

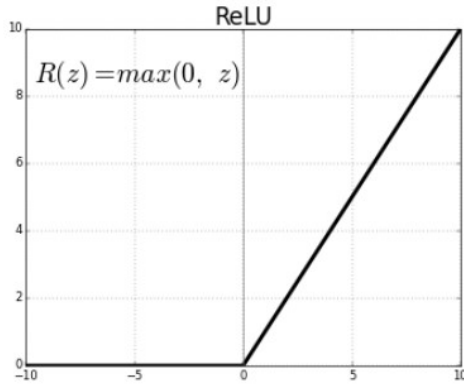


Figure 11: ReLU function

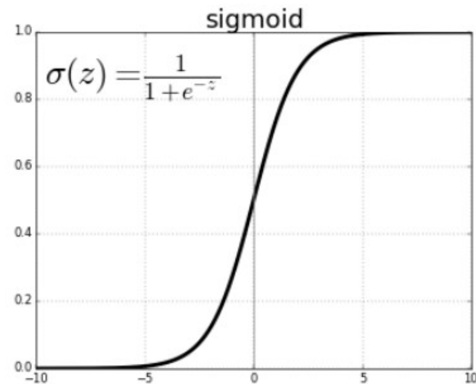


Figure 12: Sigmoid function

- Softmax: The Softmax function is also known as a normalized exponential function widely used for classification problems. Equation 6 defines the softmax function where it takes as input a vector z of k real numbers and outputs the normalized probability distribution consisting of k probabilities proportional to their exponentials.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{6}$$

To train the model to predict correct outputs from the input data, we can change the neural networks' configuration by changing the weights in the network as changing the weights changes the output. The neural network uses a back-propagation algorithm that involves calculating partial derivatives and gradient descent to change the weights. The training process starts with the feed-forward step where first the weights are randomly initialized, then the input data is fed to the network, and the output is calculated. Then the error is calculated by comparing the output of the feed-forward step, which is the predicted value by the network with the actual output value. The error reflects the neural network's performance and is used to calculate the network's performance improvement after updating the weights. There are multiple error calculation methods, and the selection of it depends on the downstream tasks. There are mainly two types of tasks: Regression and Classification.

For regression, the method used is Mean Squared Error defined by the equation 7 that calculates the square of the difference between the actual value and predicted value where Y_{pred} is predicted output and Y is the actual output. The aim is to have a deficient mean squared error.

$$Error(E) = \frac{1}{2n} \times \sum_{i=1}^{n} (Y_{pred} - Y)^2 \tag{7}$$

For classification, the last layer consists of the softmax activation function that gives the probability distribution as output, and then the Cross-Entropy or log loss is used to calculate the error.

$$Error(E) = -(ylog(p) + (1+y)log(1-p)) \tag{8}$$

$$Error(E) = -\sum_{c=1}^{M} y_{o,c} log(p_{o,c}) \tag{9}$$

Equation 8 defines the Cross-Entropy or log loss error if the number of classes is equal to 2 where y is the actual output i.e. the correct class label c, p is the predicted probability output of class c and equation 9 defines the Cross-Entropy or log loss error if the number of classes is greater than 2.

The final step of training a network is the back-propagation that aims at minimizing the error by adjusting the weights as follows: first, the partial derivatives of every wieght with respect to the error is derived at the last layer. If the partial derivative is positive, we decrease the value of the weight since decreasing the weight decreases the loss and if the partial derivative is negative, we increase the weight since increasing the weight decreases the loss. Then we back-propagate to the second last layer, perform the same, and so forth until we reach the input layer and all the weights are updated. When repeated for all the data in the training set, the entire cycle is called one epoch and the neural network takes several such epochs to train.

2.2.2 Multi Layer Perceptron(MLP)

Multilayer Perceptron(MLP)[35] is the simplest form of feedforward artificial neural network, which eliminates the limitations of perceptrons to represent only linear functions. MLP uses a more robust and complex architecture to learn regression and classification models for difficult datasets as shown in Figure 13. It has an input layer that constitutes the feature vector $x_1, x_2, ..., x_n$ and an output layer that fully connects with multiple hidden layers in between them. There are connection weights between the input layer and a hidden layer given as w_{ij}, and the connection weights between the hidden layer and output layer are given as v_{ij}. Training MLP requires three steps: Forward pass, Error or Loss calculation, and Backward pass. In Forward pass, the inputs are pushed forward through MLP, and at every layer, the dot product of the input is taken with the connection weights, add bias that exists between the layers, and fed to the activation function. The hidden layer outputs are given by equation 1 and the output layer output is given by equation 2. MLP also utilizes different activation functions such as sigmoid function, tanh function or rectified linear(ReLU) function.

In loss calculation, the model uses the output of the model, which is called the predicted output, and the actual output to calculate the loss for the ith data point using the cost function based on the downstream task like the squared loss or cross-entry loss. Then it uses the calculated loss in the Backpropagation algorithm. In the Backward pass stage, we use the backpropagation algorithm to minimize the cost function using stochastic gradient descent and update the connection weights. The termination can be when all the training samples are classified correctly. The error between consecutive epochs does not change significantly or applies a limit on the number of epochs.

2.2.3 Word Representation

Natural Language Processing (NLP) are algorithms for computers to learn and understand natural languages to perform various tasks ranging from speech processing to semantic interpretation and discourse processing. In all these NLP tasks, one common denominator

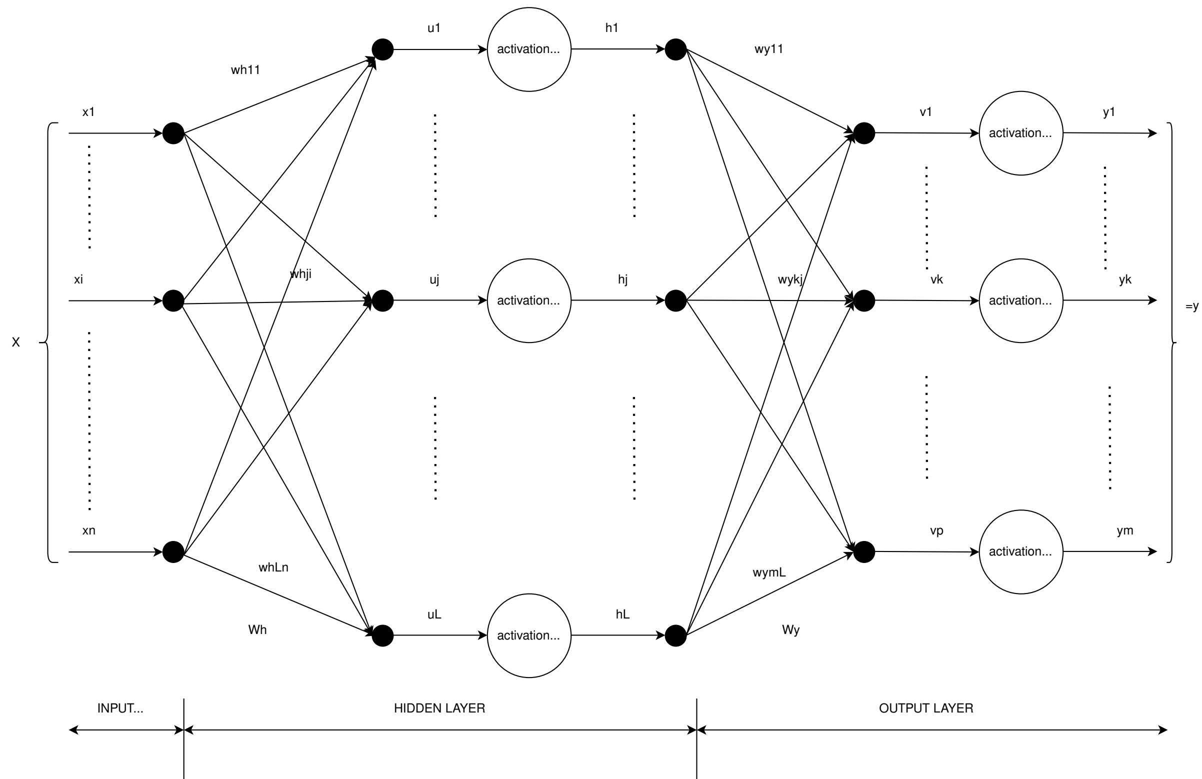


Figure 13: Multi Layer Perceptron

is to learn word representation that provides the notion of some similarity and difference between the words. Word vectors are one such representation with these abilities encoded in them. Thus, we encode each word token into some vector representing a point in some N-dimensional word space sufficient to capture the word's semantic properties such as count, gender, and tense of the natural language. There are three classes of word vector methods. The first trivial method is the one-hot vector in which it represents every word W as a vector $R^{|V|\times 1}$ where V is the size of the vocabulary with 1 at the index of the word within the sorted vocabulary and the rest all 0s as shown in Figure 14. The vocabulary has 'apple' and 'zebra' as the first and last words in the vocabulary as shown in Figure 14. Although

$$w^{apple} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ . \\ . \\ . \\ . \\ 0 \end{bmatrix} \quad w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ . \\ . \\ . \\ . \\ 0 \end{bmatrix} \quad w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ . \\ . \\ . \\ . \\ 0 \end{bmatrix} \quad , \ldots\ldots \quad w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ . \\ . \\ . \\ . \\ 1 \end{bmatrix}$$

Figure 14: One-Hot vector representation

this method provides word representation utterly independent entity, it fails to provide a notion of similarity. The second class of word embedding methods is the Singular value decomposition(SVD) based methods. First, it captures the count of word co-occurrence in the form of a matrix X over the entire dataset. Then a Singular Vector Decomposition

on X is performed to get USV^T decomposition where it uses the rows of U as the word embedding for the words in the vocabulary. These methods differ from each other based on the method of accumulation and formation of the matrix X. The third and the widely used class is the Iteration-based methods. These classes aim to design a model with word vectors as their parameters and then train them to adjust them using a back-propagation algorithm to achieve particular objectives. There are several methods in this class, but the most recent and widely used is the Word2Vec model, briefly introduced below.

Word2Vec

Word2Vec is a probabilistic approach to word embedding. There are two algorithms for word2vec, namely: the continuous bag-of-words (CBOW) shown in Figure 15 and skip-gram shown in Figure 16. The CBOW algorithm's goal is to predict a word vector of the center word from the surrounding context, and the goal of the skip-gram algorithm is to predict the probability distribution of context words from the center word. In the following section, we briefly present the two algorithms.

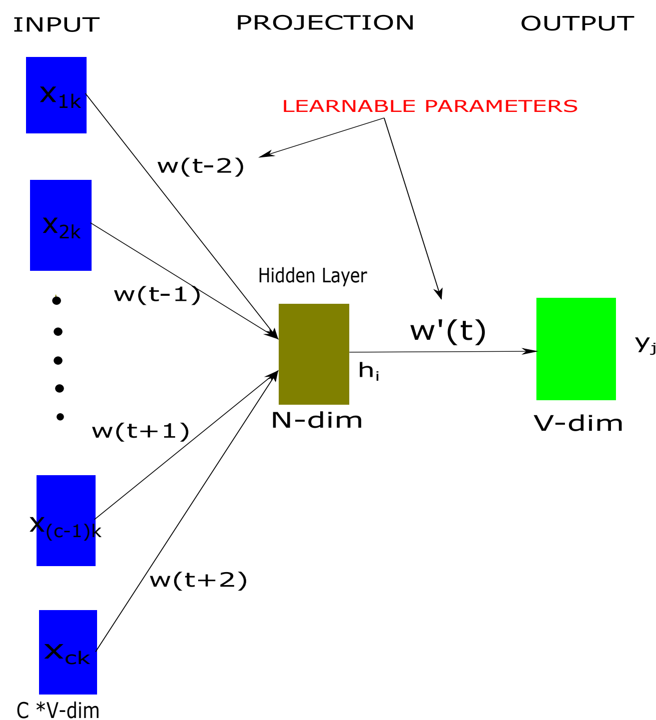


Figure 15: CBOW architecture

Continuous Bag of Words(CBOW)

In this approach, given as input a context like "Ron," "like," "to," "the," "blue," "cheese" the goal is to predict or generate the center word "eat." The approach takes the following steps. The first step of the model is to generate one-hot vectors represented as x^c for the sentences in the input context of size m, and the output of the model y is the one-hot

vector of the known center word. These two are the known parameters of the model. The next step is to create the input word matrix $\mathcal{V} \in R^{n\times|V|}$ and the output word matrix $\mathcal{U} \in R^{|V|\times n}$ where n defines the size of the embedding space. The i-th column of V and j-th row of the U is the n-dimensional embedded vector for the words w_i and w_j and are given as v_i and u_j respectively when w_i and w_j are input and output to the model respectively. Using these matrices we get the embedded word vectors for the context given by $(v_{c-m} = Vx^{(c-m)}, v_{(c-m+1)} = Vx^{(c-m+1)}, \ldots\ldots., v_{c+m} = Vx^{(c+m)} \in R^n)$. Further, we take an average of these vectors as $\hat{v}$, and using this average; we generate a score vector given as $z = U\hat{v}$. Finally, these scores are fed to a softmax activation function to output the scores as probabilities given as $\hat{y} = softmax(z) \in R^{|V|}$. The goal is to have generated probability to match the actual probability. To train this model and learn the parameters matrices $\mathcal{V}$ and $\mathcal{U}$ we use cross-entropy as cost/loss function, and to minimize this cost-function/loss, we use stochastic gradient descent and update the word vectors v_i and u_j.

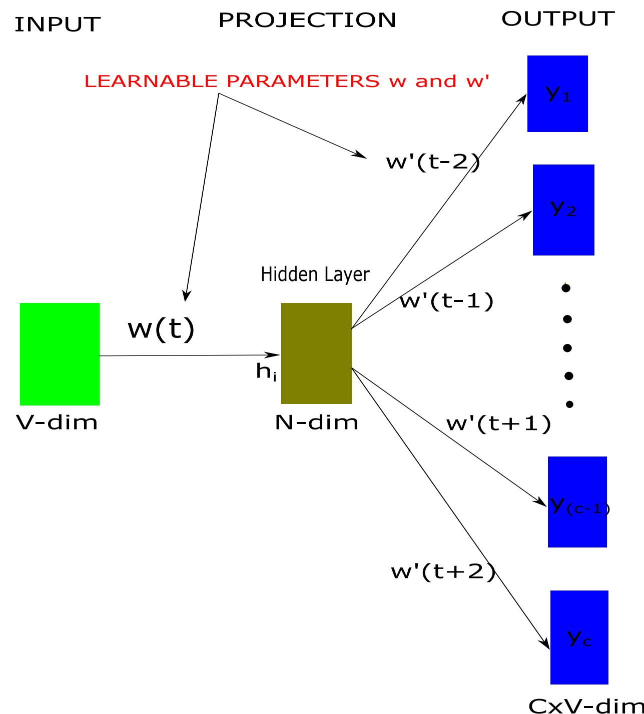


Figure 16: Skip-Gram architecture

Skip-Gram

In this approach, unlike the CBOW, a center word "eat" is given as input, and the goal is to predict or generate the surrounding context words "Ron," "like," "to," "the," "blue," "cheese." The approach takes the following steps. The first step is to generate the one-hot vector of the center word, which is the input to the model given as $x \in R^{|V|}$. Then the next step is to generate the input and output matrix $\mathcal{V}$ and $\mathcal{U}$ as in the CBOW approach, and use it to generate the embedded word vector for the center word given as $v_c = \mathcal{V}_x \in R^n$. Next, we use the embedded word vector to generate a score vector defined by $z = \mathcal{U}v_c$.

The generated score works as an input to the softmax activation function, which gives as output a probabilistic distribution $\hat{y} = softmax(z)$ stating probabilities of observing each context word. The aim is to have the generated probability vector equal to the actual probabilities, which are the one-hot vectors of the actual output. Unlike the CBOW the cost function used in this approach is based on Naive Bayes assumption given by equation 10 where $H(\hat{y}, y_{c+m+j})$ is the cross-entropy between the predicted output vector $\hat{y}$ and one-hot vector input y_{c+m+j}.

$$minimize J = \sum_{j=0, j\neq m}^{2m} H(\hat{y}, y_{c+m+j}) \tag{10}$$

followed by the Stochastic Gradient descent to minimize the cost function and to update the word vectors v_i and u_j.

2.3 Graph Neural Networks

2.3.1 Introduction

The initial deep learning toolbox design operates on images, text, and audio signals represented in a euclidean space as a grid and as sequences. Further, the images as a grid can be fed to convolutional neural networks to perform downstream tasks like image classification. Similarly, the text/audio signals as sequences can be fed to neural network models to perform downstream tasks like song recommendation or text classification. However, data to be processed nowadays is from non-Euclidean space, a data structure representing almost every piece of data. The nodes hold information that describes objects/entities. Edges describe the relationship between those objects. Unlike graphs representing the images and audio/text signals, graphs representing the non-Eucleadian data are often dynamic, arbitrary in size. They have a complex topological structure with multimodal node and edge features with no fixed node ordering or reference point. Hence, inspired by the classical neural networks like Convolutional Neural Network(CNN) [37], Recurrent Neural Networks(RNNs) [30], and AutoEncoders [61], a new class of deep learning model to operate on graphs called Graph Neural Network was developed. The goal was to alleviate the complex problem of feature extraction, scaling the massive amount of data, and allowing flexible implementations to learn on the graph and perform the downstream tasks. The goal of learning on graphs is to map the graph nodes or the entire graphs into an embedding space based on the nodes and edges' structural and individual properties. Furthermore, using this for downstream tasks such as "Node Classification": To predict the class of a given graph node, "Graph Classification": To predict the class of a given graph, and "Link Prediction": Predict the edge existence between nodes of a given graph.

2.3.2 Functionality

To understand the working of the Graph Neural Network consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges, $x_v \in R^F$ for all $v \in V$ are the node features such as for node A, name, age are the node features, $e_{v,u} \in R^d$ for all $(v, u) \in \mathcal{E}$ are the edge feature representing the relationship between two nodes as shown in Figure 17

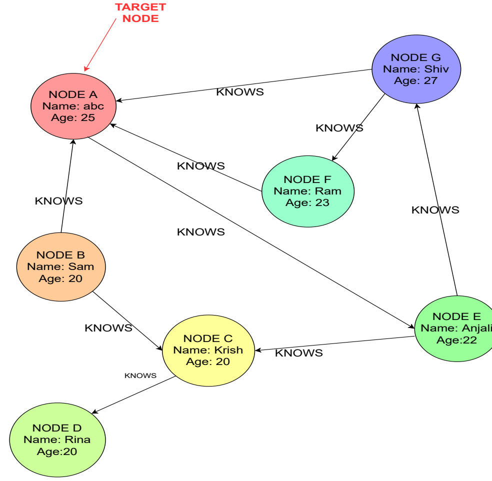


Figure 17: Network Graph as example

which is a network graph. GNN models utilize a form of neural message passing where vector messages are exchanged between nodes and updated using neural network [28]. This message passing framework's key idea is to generate node embedding based on local network neighborhoods working on the intuition that nodes aggregate information from their neighbors using neural networks. In GNN, the node representations $h_u^{(k)}$ for each node $u \in \mathcal{V}$ is updated by repeatedly transforming and aggregating neighboring node $v \in N(u)$ representations. In each message passing iteration, to calculate/update the node representation of the target node as shown in Figure 17, it first unrolls the local network layer where the number of unrolls i.e. layers (L) corresponds to the visibility of the target node i.e. L-hop neighborhood messages are utilized as shown in Figure 18 and each neighbor sends a message as shown in Equation 11.

$$Message : m_{v,u}^{(l)} = MESSAGE(h_v^{(l-1)}, h_w^{(l-1)}) \tag{11}$$

In the next step, these messages across all the neighbors are aggregated using a permutation invariant aggregate function such as sum, mean or max as shown in equation 12.

$$Aggregate : a_v^{(l)} = AGGREGATE(m_{v,u}^{(l)} : w \in N(v)) \tag{12}$$

Finally, the node representation is updated using this neighbor information as shown in equation 13.

$$Update : h_v^{(l)} = UPDATE(h_v^{(l-1)}, a_v^{(l)}) \tag{13}$$

To define it mathematically, the initial embedding for the 0-th layer is given by $h_v^0 = x_v$ which is equal to node feature and the generalized final node representation of the target node A is given by equation 14

$$h_v^k = \sigma(W_k \sum_{u \in N(v)} C_{v,w} h_w^{(k-1)} + B_k h_v^{(k-1)}) \tag{14}$$

where $h_u^{(k-1)}$ and $h_v^{(k-1)}$ are the previous layer embeddings for the target node v and its neighboring node $u \in N(v)$, W_k is the parameter matrix, B_k is the Bias matrix, $C_{u,v}$ is the

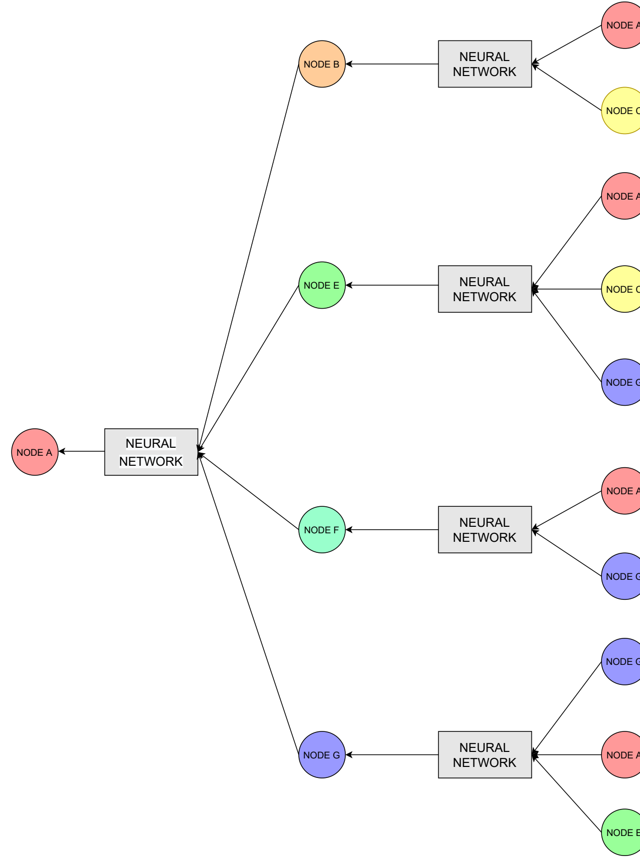


Figure 18: Computational Graph for Node A

normalization coefficient, σ is the activation function. The normalization can be static that means that the normalization coefficient $C_{u,v}$ is always 1, or it can be structure dependent wherein it's normalization is based on the node degree i.e. the normalization coefficient is given by $C_{u,v} = |N(v)|^{-1}$, or it can be data dependent wherein the normalization coefficients are learned in an end-to-end fashion in parallel with node embedding which is also known as attention.

2.3.3 Training

To train the model to generate the embeddings, it learns the trainable parameters W_k and B_k that it uses for the transformation. If W_k is zero, the model learns and generates embedding from the node's features, acting as a multi-layer transformation of the node's features. If B_k is zero, the model learns and generates embedding from the neighboring nodes' features and feeding the final embeddings to a loss function. Then stochastic gradient descent is used to train and adjust the weight parameters to minimize the loss. The training of the model can be in an unsupervised manner or task-specific supervised manner. In an unsupervised manner, the loss function is either based on Random walks, Graph factorization, or Node proximity in the graph, and similar nodes will have similar embedding. In task-specific supervised training, the loss function is selected based on the task. Then the back-propagation algorithm is used to minimize the loss and adjust the matrix accordingly. Additionally, the model can be trained on a subset of computational

graphs for nodes as shown in Figure 18, and then apply the model on new subgraphs since the trainable parameters are shared parameters across all the nodes, i.e., they are the same for all computational graphs. The model uses the final localized node embeddings according to a given downstream task. For node classification, the prediction is based on the computed node embedding as shown in equation 15.

$$\phi(\mathcal{G}, v) = MLP(h_v^{(k)}) \tag{15}$$

For graph classification, it uses an additional function called global readout to aggregate the localized node embeddings to create a global graph representation which it then uses as shown in equation 16.

$$\phi(\mathcal{G}) = MLP(\sum_{v \in \mathcal{V}} h_v^{(k)}) \tag{16}$$

For link prediction, it uses a pair of node embeddings to predict the existence of an edge between two nodes as shown in equation 17.

$$\phi(\mathcal{G}, v, w) = MLP(h_u^{(k)}, h_v^{(k)}) \tag{17}$$

Using the message passing scheme, we can model various transformations apart from the isotropic transformations. They are as follows:

- **Anisotropic:** Unlike the previous isotropic transformation where every node transformation uses the same signal weight matrix, there are cases where the model has to perform an anisotropic transformation. GNN uses a transformation based on MLP to input the target node representation and the difference between the target node and the neighboring node representation. Hence the neural network model still has shared parameters, but the nodes get transformed based on the current source and target node representation. The transformation is defined by the equation 18.

$$MESSAGE(h_v^{(k)}, h_u^{(k)}) = MLP(h_v^{(k)}, h_u^{(k)} - h_v^{(k)}) \tag{18}$$

- **Edge-Relation:** This formulation allows learning on graphs with distinct edge types. This procedure takes various weight matrices depending on the edge type across the source and target node for message function as shown in equation edge relation transformation.

$$MESSAGE(h_v^{(k)}, h_u^{(k)}) = W_{R(u,v)} h_u^{(k)} \tag{19}$$

- **Edge Features:**For graphs with edge features, the neighboring node features are transformed based on their specific edge features using the transformation function γ that takes as input the edge features between the source and target node. The message function is as shown in equation 20.

$$MESSAGE(h_v^{(k)}, e_{u,v}) = \gamma(e_{u,v}) h_u^{(l)} \tag{20}$$

There are several variants of GNNs. They differ from each other based on the implementation of the message passing scheme, i.e., by assuming various forms of message function, aggregate function, update functions, and read-out functions. Some of the relevant GNNs are as follows: **Graph ConvNet(GCN)** [36] also known as Vanilla Graph Convolutional

Network is the simplest form of GNN that calculates target node embedding by incorporating self-embedding of each node from the previous layer and performing an isotropic averaging operation over the embedding of the neighborhood nodes at the previous layer by addition as shown in the update equation 21

$$h_v^k = \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|} + B_k h_v^{k-1}) \tag{21}$$

GraphSAGE [29] is a general inductive framework that explicitly incorporates self-embedding of each node from the previous layer by sampling and aggregation of neighbor embedding by concatenation as shown in the update equation 22

$$h_v^k = \sigma([W_k.AGG(\{h_u^{k-1}, \forall u \in N(v)\}), B_k h_v^{k-1}]) \tag{22}$$

The three variants of aggregation functions are as follows: *Mean aggregator:* It outputs a weighted average of the neighboring nodes embedding at the previous layer as shown in the equation 24.

$$AGG = \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|} \tag{23}$$

LSTM aggreagtor: is an LSTM-based aggregator with a more extensive expressive capability and is not permutation invariant. It uses input reshuffled of neighboring nodes embedding from the previous layer as shown in equation 24.

$$AGG = \gamma(\{Qh_u^{k-1}, \forall u \in N(v)\}) \tag{24}$$

where γ represents symmetric vector function that performs element-wise mean/max. *Pooling aggregator:* In this aggregator, it applies a symmetric vector function on a set of node's neighbors after transforming the neighbor vectors by feeding each neighbor's hidden state through a fully-connected layer depicted by the equation 25 where π is a random permutation operation.

$$AGG = LSTM([h_u^{k-1}, \forall u \in \pi(N(v))]) \tag{25}$$

Graph Attention Network(GAT) [60]: In simple neighborhood aggregation update as shown in equation 21, it aggregates messages across neighborhoods of target node $N(v)$ with the same weighting factor where the weighting factor of the message from node u to node v is given by $a_{vu} = 1/|N(v)|$ and is explicitly defined based on the structural properties of the graph. In Graph Attention Networks(GAT) model, the goal is to implicitly define the weight factor to specify different weights to different neighboring nodes of each target node in the graph. It uses an attention mechanism whose parameters are learned together with the learnable parameters of the neural network in an end-to-end fashion and based on the messages between pair of nodes (u, v). It computes the attention coefficient given that represents the importance of message from u to v by equation 26. It parallelizes the computation across all edges and nodes of the graph to make the computation of attentional coefficients and aggregation efficient.

$$E_{vu} = a(W_k h_u^{k-1}, W_k h_v^{k-1}) \tag{26}$$

Then the attention coefficients are normalized using softmax function to calculate the weighting factor given by the equation 27 and the final node representation is given by equation 28.

$$a_{vu} = \frac{exp(e_{vu})}{\sum_{k \in N(v)} exp(e_{vk})} \tag{27}$$

$$h_v^k = \sigma(\sum_{u \in N(v)} \alpha_{vu} W_k h_u^{k-1}) \tag{28}$$

Multi-Head attention: To further stabilize the attention mechanism, it independently replicates the attention operations R number of times in a given layer where each replica has various parameters called Multi-head attention. Figure 19 shows the multi-head attention. Then the outputs are aggregated either by concatenated or by adding. **Gated Graph**

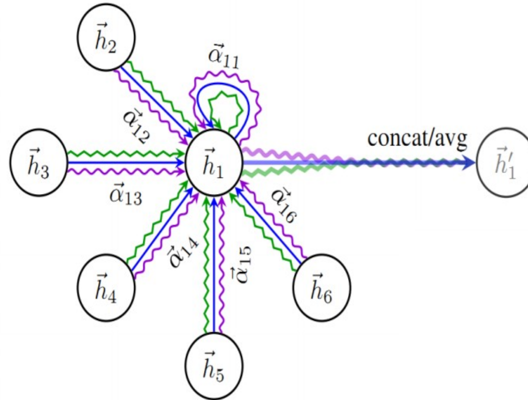


Figure 19: Multi-head attention

ConvNet(GGCN) [14] Gated Graph ConvNet is an anisotropic variant of GCN that utilizes the features of RNN's edge gating mechanism from [41]. The architecture of Convolution Neural Network [50] to explicitly update edge features along with node features while maintaining the edge features at each layer. The update mechanism is as shown in equation 29 where $\odot$ is the Hadamard product and e_{uv} is the edge gate which is defined by equation 30.

$$h_v^k = \sigma(\sum_{u \in N(v)} e_{uv} \odot W^{k-1} h_u^{k-1}) \tag{29}$$

$$e_{uv} = h_u^{k-1} + \sigma(W^{k-1} h_v^{k-1} + W^{k-1} h_u^{k-1}) \tag{30}$$

For Residual Gated Graph ConvNets, it utilizes the residual networks(ResNets) to formulate the node transformation that results in the addition of an identity operator between successive convolutional layers as shown in equation 31.

$$h_v^k = h_v^{k-1} + \sigma(\sum_{u \in N(v)} e_{uv} \odot W^{k-1} h_u^{k-1}) \tag{31}$$

Graph Isomorphism Network [65]: The Graph Neural Network's(GNN) expressivity tightly couples with its power to distinguish the graph structures. The GNN should map isomorphic graphs(topologically identical graphs) to the exact representation. The non-isomorphic graphs to different representations. This expressivity can be related to the Weisfeiler-Lehman test. *Weisfeiler-Lehman test* [64] is a powerful graph isomorphism heuristic based on iterative color refinement. The test has four steps which are as follows: First, to initialize, all the nodes are assigned the same color. Then it iteratively aggregates colors from neighboring nodes and hashes those colors to a unique new coloring. This iterative color refinement procedure iterates for L number of steps. Finally, it checks the

color histograms of the two graphs, and if they are the same, it determines the two graphs are isomorphic. However, the WL-test cannot distinguish all graphs. The iterative color refinement procedure of the WL-test is similar to the neighborhood aggregation of GNNs. Hence, if a GNN is permutation-invariant, it can map isomorphic graphs to the exact representation with the WL-test as an upper-bound for its expressive power. It can be as powerful as the WL-test if the GNN has an injective aggregation procedure where the neighborhood aggregation functions over multi-set. The entire neighborhood aggregation is termed as injective if and only if every step of the neighborhood aggregation is injective as shown in Figure 20. To achieve maximum expressivity, the GNN requires maintaining

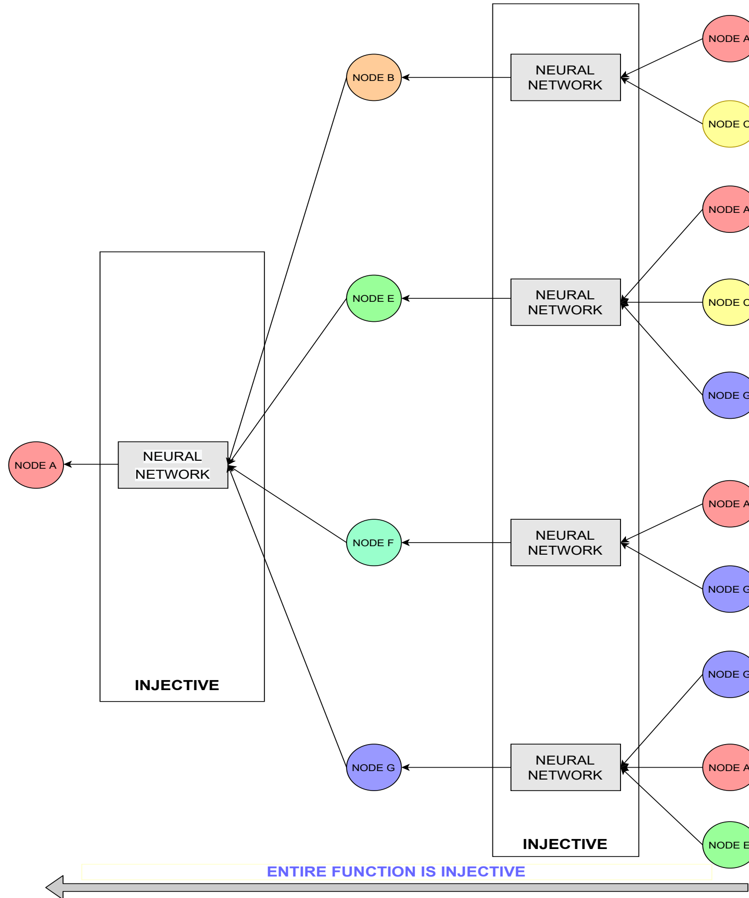


Figure 20: Injectivity in Computational Graph

the neighborhood information and also the information from how many neighboring nodes does the information aggregate. The previous GNNs use mean-pooling and max-pooling as multiset functions that are not injective. The mean pooling fails to distinguish proportionally equivalent multisets, and max-pooling fails to distinguish multisets with the same distinct elements. Hence, sum-pooling is used to have a multiset function that can be expressed as $\sigma(\sum_{x \in S} f(x))$ where σ and f(x) are some non-linear functions that can be modelled using Multi-Layer Perceptron(MLP) and S is a multi-set. Expressive GNNs are developed based on WL-test with injective multiset functions, and one such expressive GNN is Graph Isomorphism Network(GIN). The Graph Isomorphism Network(GIN) [65] is based on the Weisfeiler-Lehman test and uses sum pooling for graph pooling which gives an injective graph pooling. The node update equation is defined as 32 where ϵ^k is a trainable

parameter.

$$h_v^k = MLP^k((1+\epsilon^k)h_v^{k-1} + \sum_{u \in N(v)} h_u^{k-1}) \tag{32}$$

GIN uses an architecture that is analogous to Jumping Knowledge Networks. It gathers information from all iterations of the model to consider all structural information of the input graph. It uses the 'SUM' aggregator as an aggregation function instead of 'MEAN,' or 'MAX' aggregator since the ranking power of 'SUM' aggregator is greater than 'MEAN' and 'MAX' aggregator. 'SUM' aggregator captures the full multiset. In contrast, the 'MEAN' captures only a proportion of elements of a given type, and the 'MAX' aggregator reduces the multiset to a simple set by ignoring the multiplicities. The graph representation is defined by equation 33 where $READOUT$ sums node features from all iterations.

$$h_G = CONCAT(READOUT(h_v^{(k)}|v \in G)|k = 0, 1, ..., K) \tag{33}$$

Finally, it uses graph representations for graph-level predictions.

2.4 Transfer Learning

Most of the deep learning models specialize in a particular domain or task and require a sizeable attested dataset to solve complex problems. Getting such a sizeable attested dataset is tedious, considering the time and effort required to label the dataset manually. Additionally, due to the model's training on a specific task, the model suffers a significant loss in performance when used for a similar new task similar. A technique referred to as the transfer learning/knowledge transfer technique alleviates these limitations leveraging knowledge from a pre-trained model to solve a new problem.

A domain D is a two-element tuple consisting of a feature space X and a marginal probability distribution P(X) over the feature space, where $X = x_1, ..., x_n \in X$ is a sample data point where x_i represents a specific vector. Hence, mathematically, a domain is represented as $D = X, P(X)$. Given a domain $D = X, P(X)$, a task is defined as a two-element tuple consisting of a label space Y and a conditional probability distribution P(Y—X), also referred to as the objective function that typically learns from the training data consisting of pairs of $x_i \in X$ and $y_i \in Y$ [44].

Formally, transfer learning is defined as follows: Given a source domain D_S and a target domain D_T with corresponding tasks T_S and T_T respectively. The transfer learning technique aims to provide the ability to learn the target conditional probability distribution $P(Y_T|X_T)$ in the domain D_T with the information gained from the source domain D_S and source-task T_S where the source domain is not equal to the target domain. The source task is not equal to the target task. Based on the machine learning algorithm used, these transfer learning techniques categorize as Inductive Transfer Learning, Unsupervised Transfer Learning, and Transductive Transfer Learning [44]. Unsupervised and Transductive transfer learning is out of scope since we use the deep learning method's inductive transfer learning technique. In inductive transfer learning, where the domains are the same, but the source and target tasks are different from each other. The technique utilizes the source domain's inductive bias to improve the model's performance on the target task. In the following section, we present the transfer learning techniques for graph neural networks.

2.4.1 Transfer Learning in Graph Neural Networks

One of the limitations of GNNs is that it requires sufficient task-specific labeled data to train the model and have a well-generalized model. However, obtaining a task-specific labeled dataset is challenging. Hence to alleviate this limitation, it uses the transfer learning approach wherein GNN is pre-trained to be generalized with a sparsely labeled dataset for a downstream task. In Wu et.al. [?], the authors performed a systematic large-scale investigation of strategies for pre-training GNNs and introduced an effective pre-training strategy for GNN that first learns domain-specific knowledge about nodes and edges using the node-level information and then graph-level knowledge. This framework suggests that performing node-level pre-training in addition to graph-level pre-training avoids negative transfer and improves performance. These strategies are as follows: **Node-Level Pre-Training** Wu et.al. [?] proposes two self-supervised methods for node-level pre-training, namely Context Prediction and Attribute Masking.

- **Context Prediction:**In the Context Prediction method, the model's aim is similar to word embedding, where the model tries to learn the contextual meaning of a word by using the surrounding words. In this method, the model aims to learn the structural context of a node using its surrounding graph structure and mapping the nodes with similar structural contexts to nearby embeddings. This representation learning process is a three-step process as shown in Figure 21. First, based on the computational graph and the unrolling techniques, the number of unrolls corresponds to the target node's visibility. For every node $v \in V$, it defines the K-hop neighborhood where it contains all the nodes $u \in V$ and edges $e \in E$ that are at most K-hop or K-unrolls away from the node v. The context graph of node v represents the subgraph present between r_1 hop and r_2 hop away from node v where r_1 and r_2 are hyperparameters such that $r1 < K$ so that some are shared between the neighborhood and these are called the context anchor nodes. Then a supplementary GNN is used to obtain the node embedding in the context graph and take the average of the context anchor nodes embeddings to obtain a fixed-length context embedding c_v^G. Finally, it uses negative sampling with log-likelihood function as cost-function to train the GNN and supplementary GNN to predict if a particular context graph and particular neighborhood belong to the same node given as $\sigma(h_v^{(K)T} c_{v'}^{G'}) \approx 1$ where $\sigma(\odot)$ is the sigmoid activation function. It keeps the ratio of negative pair to positive pair as 1.
- **Attribute Masking:** In Attribute Masking, the GNNs are pre-trained by masking the node/edge attributes of the graph and then feeding it to the GNN to predict the masked attributes based on the neighborhood structure as shown in Figure 22. First, it masks the input node/edge attributes randomly by replacing them with particular masked indicators. Then the modified graph is fed to a GNN to obtain node/edge embeddings followed by a linear model on top of the embeddings to predict the masked node/edge attributes [?]. Attribute masking aims to learn the relation and co-relation of various interactions in the given graph and exploit the distribution of graph attributes.

Graph-Level Pre-Training There are two graph-level pre-training strategies. The goal is to pre-train GNNs to generate robust graph embedding that can be transferred across

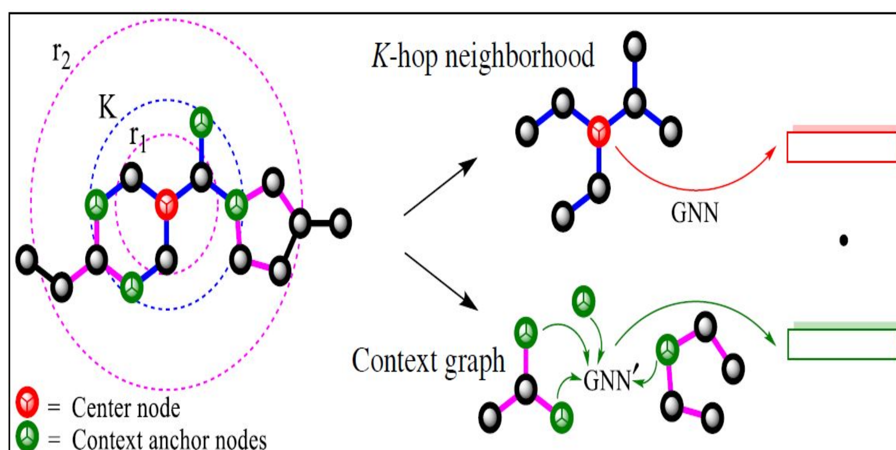


Figure 21: Contextual Prediction from [32]

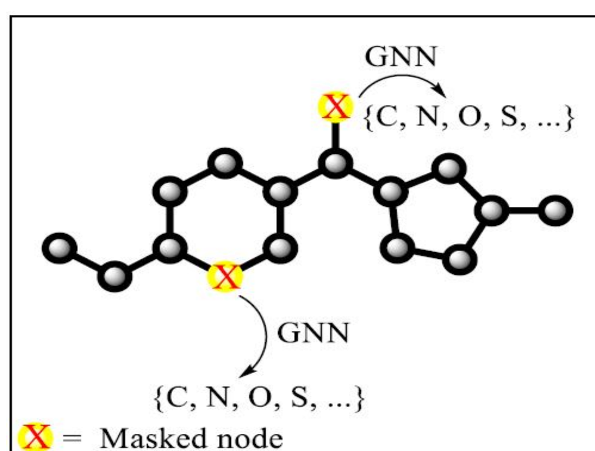


Figure 22: Attribute Masking from [32]

various downstream tasks using the node embeddings obtained by node-level pre-training methods.

- **Supervised Graph-Level Property Prediction:**This strategy is to pre-train a GNN by using a supervised graph-level multi-task prediction to incorporate domain-specific information in the graph-level representation. For example, in the dataset of network graphs, we can pre-train GNNs to predict all the networks' properties identified and validated experimentally so far. Since some supervised pre-training tasks can be unrelated to the downstream task at hand, performing a trivial extensive multi-task graph-level pre-training fails to produce a robust generalized transferable graph level representation. To alleviate this limitation, the supervised pre-training task should highly relatable to the interested downstream task.

- **Structural Similarity Prediction:**This strategy uses a graph-level prediction task of predicting structural similarities between the two graphs to pre-train the GNN.

The local graph embedding created in the graph-level pre-training and the local node embedding created in the node-level pre-training is not meaningful and add to the negative

transfer problem. Hence it is proposed first to utilize node-level pre-training of GNN to perform node level regularization and then performing the graph-level pre-training. It alleviates the limitation of using a pre-training task highly relatable to the interested downstream task.

In Hu et.al. [33], the authors introduced a pre-training framework called the Generative

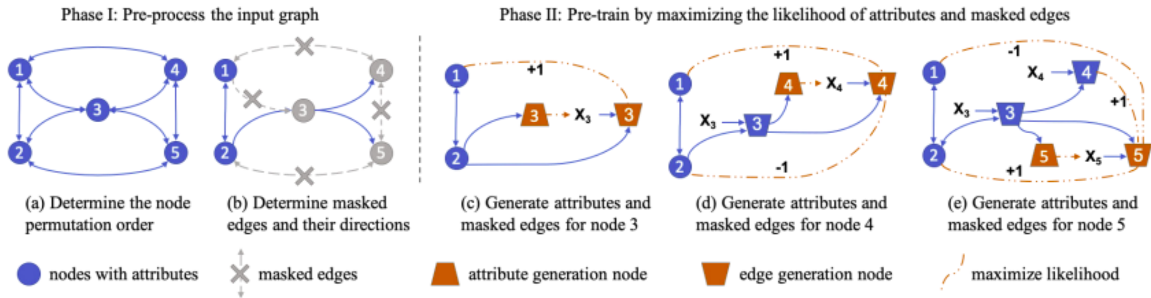


Figure 23: An illustrative example of the attributed graph generation procedure from [33]

Pre-Training of Graph Neural Networks(GPT-GNN). They used the Reddit dataset and the paper citation network extracted from OAG to perform the experiment and evaluate the framework. The GPT-GNN framework introduces a self-supervised graph generation task that allows GNN to capture the graph's structural and semantic properties. The author divides the probability of graph generation into two modules: attribute generation and edge generation, to simultaneously capture the attribute and structure information and learn the relationship between features and structures during the generative process. To model the graph distribution, the author proposes to use a generative model, i.e., to gradually predict features to a new node and which nodes connect to the graph. The framework pre-trains GNN by maximizing the likelihood function of the graph given by $\theta^* = \max_\theta p(G;\theta)$.

$$p(G;\theta) = E_\pi[p_\theta(X^{\pi,E^\pi})] \tag{34}$$

The frameworks suggest that to model p(G;/theta) correctly, the graph distribution can be written as the expected likelihood over all possible permutations given by equation 34. Where $X^\pi \in R^{|V| \times d}$ denotes all edges connected with node i^π denoting the node id of the i-th position in permutation π. Since in graph generation methods, the nodes in the graph come in an order, and edges are generated by connecting each new node to the existing node. Considering that all permutations have equal probability, the framework gives the log-likelihood of generating one node per iteration as given in equation 35.

$$logp_\theta(X, E) = \sum_{i=1}^{|V|} logp_\theta(X_i, E_i | X_{<i} E_{<i}) \tag{35}$$

It divides the conditional property for each node into two terms, feature generation and graph structure generation, as shown in Figure 24 to capture attribute and structure information at the same time. The framework first covers each node's features and some edges providing the remaining parts as the observed edges. In the first module, the framework uses the observed edges to predict the node's characteristics. The second module uses the observed edges and the predicted features to predict the remaining edges in the second module. Figure 23 shows an illustrative example of the attributed graph generation procedure.

$$
\begin{aligned}
& p_\theta(X_i, E_i \mid X_{<i}, E_{<i}) \\
&= \sum_o p_\theta(X_i, E_{i,\neg o} \mid E_{i,o}, X_{<i}, E_{<i}) \cdot p_\theta(E_{i,o} \mid X_{<i}, E_{<i}) \\
&= \mathbb{E}_o \left[p_\theta(X_i, E_{i,\neg o} \mid E_{i,o}, X_{<i}, E_{<i}) \right] \\
&= \mathbb{E}_o \Big[\underbrace{p_\theta(X_i \mid E_{i,o}, X_{<i}, E_{<i})}_{\text{1) generate attributes}} \cdot \underbrace{p_\theta(E_{i,\neg o} \mid E_{i,o}, X_{\leq i}, E_{<i})}_{\text{2) generate edges}} \Big].
\end{aligned}
$$

Figure 24: Decomposition of conditional property for a node

3 Related Work

As vulnerabilities have become a significant threat vector to software applications resulting in high economic losses to the companies, numerous approaches proposing to detect such potential vulnerabilities are available. Traditional approaches like code reviews require expert individuals to inspect the source codes for vulnerability detection manually. Reducing this dependency on security experts and automating the vulnerability detection technique has been an open research question. With the advent of machine learning techniques and their application in image, text/speech processing tasks, researches to implement machine learning techniques in software security are available. The following section will summarize the related research works in vulnerability detection using machine learning methods. We will divide the related research works into two sections. The first section will introduce the approaches that use traditional machine learning approaches used in natural language processing(NLP), image, and text/speech processing. The second section will introduce the approaches that use graph-based machine learning approaches, i.e., Graph Neural Network.

3.1 Static Analysis Tool(SATs) for vulnerability detection

Traditionally, Static Analysis Tools(SATs) that use different analyses and approaches can detect software vulnerabilities. Some of the popular tools are FindBugs, SpotBugs, PMD, and SonarQbe. FindBugs [1] is a static analysis tool that defines vulnerability patterns as a "code idiom that potentially can cause an error." It uses a simple static analysis to verify the code for more than 50 different vulnerability patterns. SpotBug [57] is the successor of the FindBug tool with supports Java 1.9, unlike the FindBug tool that does not support Java 1.9. PMD [4] [5] a static analysis tool that uses the rule-based approach to inspect the code. The tool uses a list of manually defined rules and patterns that are considered common in the applications. The tool extends for new vulnerabilities by adding more rules. Another popular tool is SonarQube [24] which is a quality framework tool that does not use the pre-defined and extendable list of bug patterns and analyses to detect vulnerabilities. The tool outputs the source code's overall quality essence as metrics and coverage test information by providing different views of the source code's various properties.

There have been many studies evaluating static analysis tools such as Charest et.al. [18], Oyetoyan et.al. [43], and Beba et.al. [11]. In [18], to explore the effectiveness of several SATs like CodePro AnalytiX, JLint, FindBugs, and VisualCodeGrepper that detect some of the prevalent software vulnerabilities using a subset of the Juliet Test Suite. The authors

used the metrics such as recall and precision to evaluate the tools and concluded that they performed poorly. In Oyetoyan et.al [43], the authors evaluated six SATs FindBugs, FinsSecurity Bugs, SonarQube, JLint, LAPSE+, and an undisclosed commercial tool on synthetic code using the Juliet Test Suite. The authors concluded that using one SAT tool is not enough to cover the whole range of security weaknesses which was in line with the conclusion by Rutar et.al. [49]. The authors also concluded that the capability of the open-source static tools and the commercial tool is low. Additionally, the authors concluded that developers were interested in using SATs but needed several changes to overcome the underlying obstacles. Similarly, Delaitre et.al [21] presented the evaluation of many undisclosed SATs for languages Java, PHP, and C. using both production software as well as the Juliet Test Suite. The results concluded that the tools struggled with increased complexity like control and data-flow with varying results across different test cases. Unlike the previous research that presented SATs analysis on source code, Beba et.al [11] modified source code of the SATs using proof-of-concept improvement. The authors concluded that detectors that used taint analysis produced fewer false-positive mainly due to the non-reporting of cases where the vulnerable input does not reach an exploitation sink and the detector based on a generalizable and shared implementation than the ones that try their algorithms to track data and vulnerability prediction. They concluded that making the changes to the SATs' implementation improves the SATs' performance. Improvement in the individual detector does not reciprocate for the other detectors, unlike the improvements made to a shared algorithm.

3.2 Vulnerability detection approach based on traditional machine learning approach

In this section, we present the vulnerability detection approaches based on traditional machine learning approaches.

In Yamaguchi et al. [70], the author uses the novel representation of source code called a Code Property Graph(CPG) to model vulnerabilities and assist vulnerability discovery using unsupervised machine learning techniques: dimensionality reduction, anomaly detection, and clustering. The author combines the ideas presented in research work [68], [72], [71], [69], [73]. Figure 25 shows the novel architecture introduced by the author for robust analysis and pattern-based vulnerability discovery consisting of a novel parsing strategy called refinement parsing, code property graph and a graph database. First, the author uses a novel multi-stage technique called refinement parser to alleviate the problem of determining the code structure even if the information is incomplete. Then, the author constructs the code property graph with the interprocedural call edges using the fuzzy parser output to express vulnerabilities as graph traversal in code property graphs and save it on the graph database. Finally, the author uses the graph database as a data source for mining vulnerabilities using machine learning methods.
Further, to embed source code to feature space, the author introduces a three-step method to create bag-of-words embeddings from code property graphs. The first step is called object extraction, where a suitable input space $\mathcal{X}$ is selected that reflects the object of interest, which corresponds to subgraphs of the code property graphs. The second step is called substructure enumeration, where nodes and edges are labeled for each graph, and then the subgraphs in each object are enumerated. The third and last step is to map these

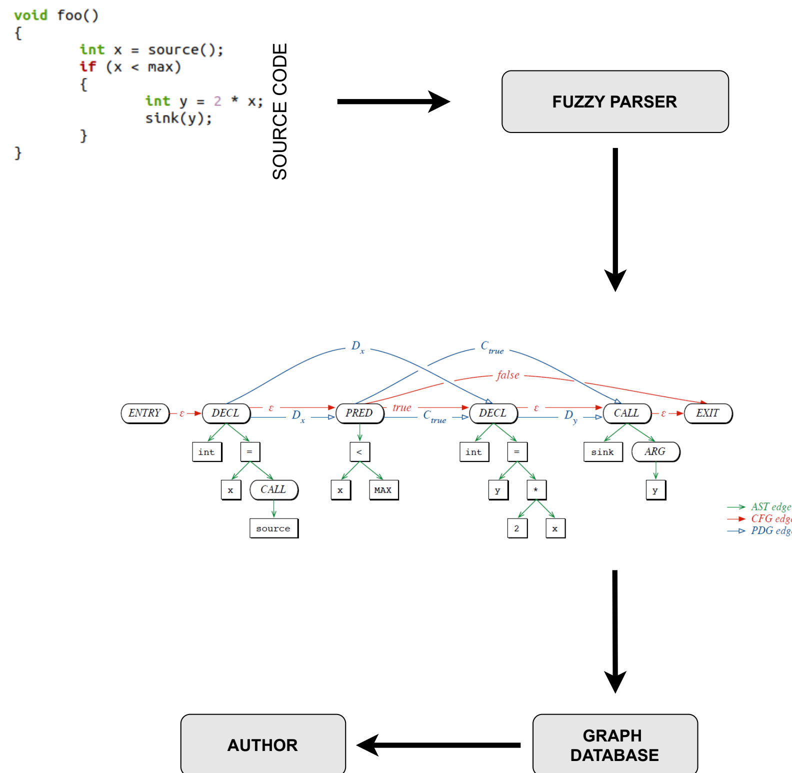


Figure 25: Robust code analysis architecture

objects in the form of subgraphs they contain to a feature space transforming the objects to vectors; this embedding requires choosing an embedding language L and a function s that maps labeled substructures of the objects to the words of the language. Finally, the author uses different unsupervised machine learning methods, as shown below, to perform vulnerability discovery.

- **Dimensionality Reduction:** The author proposes to use a technique based on dimensionality reduction called Latent semantic analysis to identify the topic of discussion in a set of text documents for vulnerability extrapolation. The approach factorizes into four steps: Extraction of syntax trees, Embedding of syntax trees, Identification of structural patterns, and Vulnerability extrapolation.
- **Anomaly Detection:**In anomaly detection, the author uses the previous dimensionality reduction method to identify the functions with similar programming patterns. Then the author analyzes the data flow in these functions, followed by the unsupervised machine learning method of anomaly detection to detect missing checks.

- **Clustering:**Finally, the author proposes an approach to combine the idea of mining vulnerabilities from the graph database by manually creating search patterns and automatically creating search patterns from code using unsupervised machine learning algorithms to introduce a strategy to automate the extraction of search patterns for taint-style vulnerabilities from source code.

The approaches presented by the author in Yamaguchi et al. [70] act as a starting point for the implementation of machine learning methods for vulnerability detection. However, there are some limitations to these approaches as follows: firstly, due to the embedding strategy used for unsupervised machine learning methods, it fails to capture the semantic properties of the code. Secondly, the approach is language-specific and requires a graph database to store the intermediate graph structures, which require complex queries to fetch required data. Finally, it fails to capture vulnerabilities that cross boundaries beyond and does not manifest in just a few functions.

In HK Dam et.al. [20] and N.Saccente et.al. [51], the authors proposed a novel deep learning approach to automatically learn features for predicting vulnerabilities in software code inspired by the advances in Natural Language Processing(NLP) using deep learning [42]. The idea behind the approach was that, like natural languages, source code tokens have a long contextual relationship with code tokens scattered far apart. The machine learning methods based on software metrics and Bag-of-Words fail to capture these long-term dependencies. These limitations can be alleviated using a Long Short Term Memory(LSTM) model for effective learning of long-term dependencies in sequential data by capturing both syntactic and semantic features of the sequence data. Figure 26 depicts an overview of the proposed architecture where with each step, it gathers syntactic and semantic features of the source code to predict vulnerabilities. The authors proposed an architecture that works as follows: it takes as input the source code. It considers the source code as a set of methods with the header containing the declaration of the class variables as method0 as shown in Figure 26. The methods are collectively parsed into variable-sized code token sequences and transformed into fixed-size feature vectors in a multi-dimensional space using the LSTM. The method features gathered from LSTM are aggregated into a single feature vector using pooling operation, which gives a set of syntactic features of the source code. Additionally, for cross-project prediction, semantic features are captured to alleviate the limitation of syntactic features being local to the project. It uses universal bag-of-token states for all the files across the project using the LSTM output to capture the semantic features. A "Codebook" is then created by grouping the tokens into clusters based on the cluster's semantic closeness and centroid called "Codebook." Then the codebook and the bag-of-token states are used to capture the semantic features of the files. Finally, captured syntactic and semantic features are fed to the classifier to learn the representations and output the vulnerability predictions. The approaches presented here learn sparse information about the code features using LSTM as it does not take into account the abstract syntax structural information of the source code. They assume it as a sequence of natural language tokens because of which it fails to capture all the syntactical and semantic features of the source code. Additionally, the tool's effectiveness is determined using an artificial dataset and not with a real-world dataset.

In previous approaches, the methods used for vulnerability detection had certain limitations:

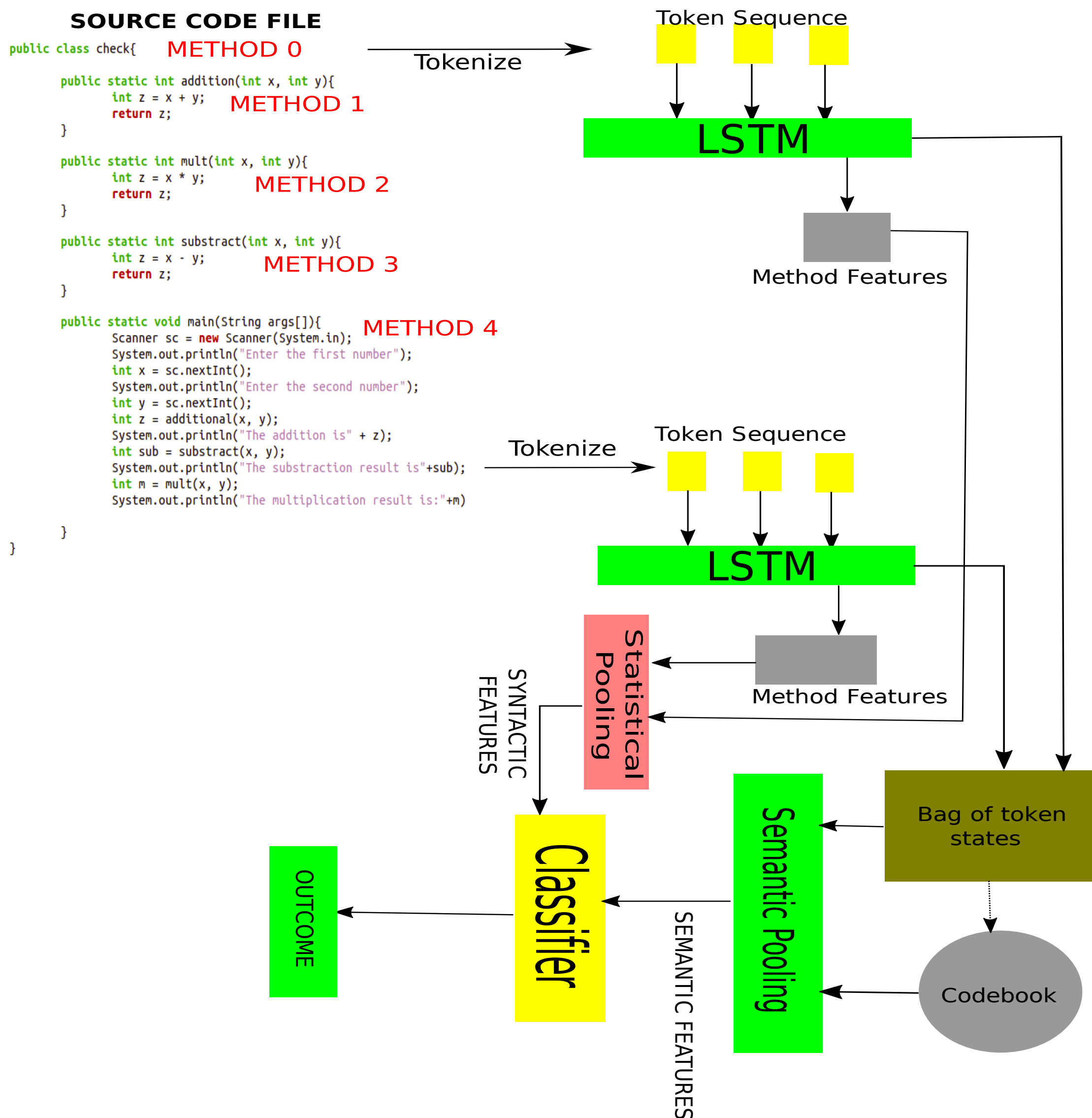


Figure 26: Overview of the approach for automatic feature learning for vulnerability prediction based on LSTM

- The methods failed to capture the correlation between the context of the code elements with a long-term dependency.
- Since programmers can customize identifiers in a programming language, the standard vocabulary fails to incorporate all the possible identifiers. This limitation is called an out-of-vocabulary issue which harms vulnerability detection methods.
- The methods fail to capture vulnerability properties adequately when the vulnerability extends beyond function or file level.
- There is a lack of a reliable dataset for vulnerabilities, which tends to hinder the approach's performance and effectiveness. A machine learning approach's performance depends highly on the dataset used to train the machine learning model.

To alleviate these limitations, X Li et.al [38] proposed a vulnerability detection approach based on minimum intermediate representation learning. It considers two hypotheses which

are as follows:

- The context of a token is given by its preceding and succeeding tokens, and tokens with the same context will have similar semantics.
- Vulnerabilities with the same types have common semantic characteristics and can learn it from the vulnerability context. [38].

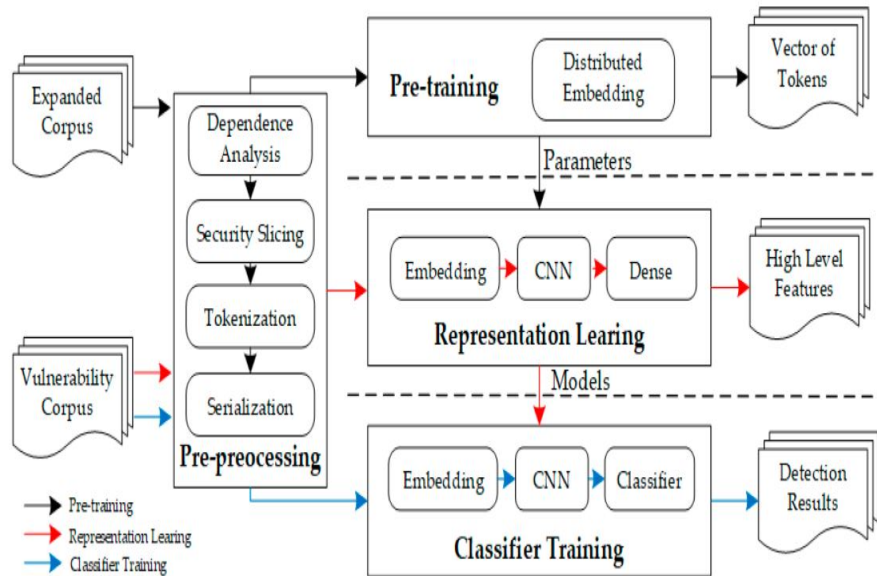


Figure 27: Architecture overview of the proposed approach from [38]

In the first stage, called the pre-processing stage, the author input a source code and transformed it into a minimum intermediate representation that aims to reduce the length of dependencies between the context by eliminating irrelevant codes. To do this, the author utilizes dependency analysis, program slicing, tokenization, and serialization. In the second stage, called Pre-Training, the author alleviates Out-of-Vocabulary(OoV) limitation by training a Continuous Bag-of-Words(CBOW) model on an extended corpus to learn the vector representations and uses them as parameters in the following two stages. In the third stage, called Representation Learning, the author captures the vulnerability dataset's high-level features from the serialized tokens. In the Classifier Training stage, the last step, the author reuses the trained convolutional neural network to obtain the high-level features to output predictions by training classifiers on these high-level features. Although the approach tries to alleviate the limitations of the previous approaches, this approach also has some limitations. First, the approach uses a security slice from the system dependence, which requires a compilable source file. However, in many cases, the available source code is non-compliable due to built configuration issues or other organizational matters. Secondly, it does not alleviate the long-term dependency, which can be improved by transforming the source code into a graph structure and utilizing the structural properties of the graphical representation of the code.

3.3 Vulnerability detection approach using Graph based machine learning approach

In this section, we present the research works that represent the source code as a graphical data structure and then utilizes Graph Neural Networks to develop an approach for vulnerability detection.

In M.Allamanis et.al. [7], the author introduced a graphical representation called "Program Graph" that constitutes Abstract Syntax Tree(AST) to include the syntactic information and data and control flows to include the semantic information. The syntactic and semantic information is included by 10 different edges that contribute proportionally to the runtime complexity as shown below:

- Child/NextToken - edges modeling AST on tokens
- LastRead/LastWrite/ComputedForm: are the edges for variables that model control flow and data flow.
- LastLexicalUse: it chains the uses of the same variable.
- GuardedBy/GuardedByNegation: edges enclosing guard expression that uses this variable.
- FormalArgName: connect method call arguments to its name/type declaration.
- ReturnTo: it links return token to name/type in the method declaration.

To represent the initial node state, the author concatenated embeddings of node label and node type; then, the author used the GGNN model and a task-based "Program Graph" for two downstream tasks VarNaming and VarMisUse. In VarNaming, the author aimed to predict the variable for a given slot out of all the known variables. To achieve this, the author first computed usage representation at the target slot by inserting the candidate variable as a node and connecting it to the target by creating the LastUse, LastWrite, and LastLexicalUse edge for each candidate variable. Then the author used the initial node representation concatenated with an extra bit for the candidate nodes followed by an 8-time step propagating GGNN to get context and usage representation as to the final state of the candidate nodes. Finally, to obtain the correct candidate variable, the author used a linear layer that uses the concatenation of context and usage representation. This research was not for universal vulnerability detection and accounted for vulnerability detection as its complementary task. Nevertheless, it also provided an example of capturing the source code features using a graph neural networks(GNN).
In Giacomo Iadarola et.al. [34], the author proposed a generic bug-finder tool called 'GrapPa'. It uses machine learning models and trains the model to classify an input source code as buggy or a non-buggy code using an artificial dataset containing buggy examples for a specific bug pattern. Figure 28 shows an overview of the tool implementation. The author divided the approach into four steps; the first step was to create the dataset for training the machine learning model by performing mutation on a set of source codes using the Major mutation framework. Then group them by the kind of error raised by testing the mutated code. In the second step, the author generated code property graphs(CPGs) from

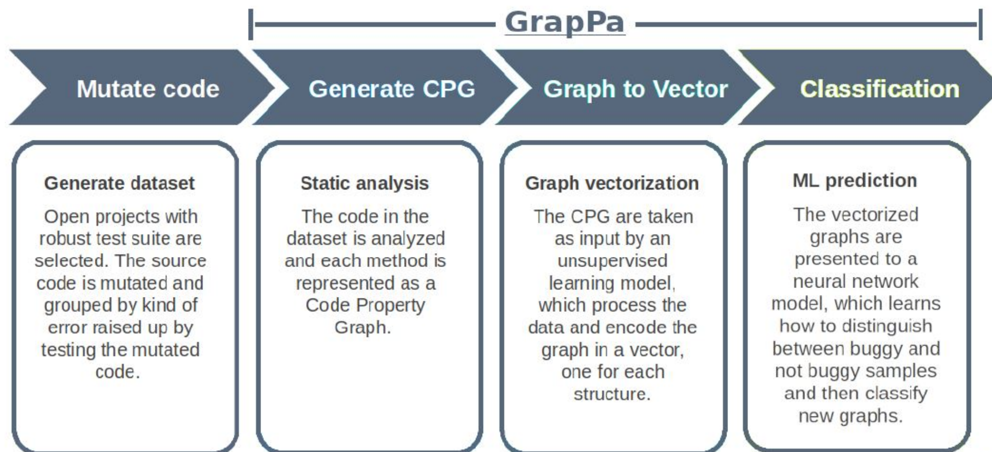


Figure 28: Overview of the framework from [34]

the mutated Java code by performing static analysis using the SOOT framework. The author converted the Java bytecode to Jimple, an intermediate representation for Java bytecode, and performed an analysis to generate CPGs. Next, the author simplified the CPG by removing the intermediate AST nodes between the statements and the tokens. In the third step, the author used the Contextual Graph Markov Model(CGMM) that takes a set of graphs as input to train a model and produces the unsupervised encoder to vectorize the simplified CPG. Finally, the author used MultiLayer Perceptron(MLP) model as the classifier. There were certain limitations to this approach; first, it requires a compilable code to create the CPG as it uses the SOOT framework to perform the static analysis and create the CPG graph. The accuracy of the approach was not precise enough; hence it only produced suggestions rather than finding specific bugs and pinpoint them. Another major limitation was that the model learned how to distinguish between mutated and original code better than distinguishing between a buggy and non-buggy code as the introduction of the bug was synthetic, .
In S.Suneja et.al. [58], the authors proposed a pipeline called AI4VA. It explores the applicability of the Graph Neural Networks in learning the signatures of the vulnerabilities in a source code in terms of the relationship between the nodes and edges in the source code's graphical representation. The authors first encode the sample source code into a Code Property Graph using Joern, an open-source tool, followed by vectorizing the graph while preserving its semantic information. Then the author used a Gated Graph Neural Network to automatically extract templates distinguishing the vulnerable code samples from the non-vulnerable ones using the vectorized graphs. The author used three different datasets to perform experiments evaluating the approach and showcased that the proposed model outperformed the static analyzers, classical machine learning approaches, and CNN and RNN-based deep learning models. The author concluded that representing code as a graph for encoding is more meaningful for vulnerability detection than the existing approach of representing source code as a picture and flat linear sequence for encoding. In Zhou et.al [74], the authors propose an approach similar to Suneja et al., where a GNN based graph-level classification model uses a rich set of code semantic representation. Furthermore, a novel classification method called the Conv module to extract the features useful for vulnerability detection is downstream. The author divided the approach into three layers where the first layer was called the Graph Embedding layer, the second layer

was called the Gated Graph Recurrent layer, and the final layer was called the Conv layer. In the graph embedding layer, the author transformed the source code into a joint data structure similar to Suneja et al. with a slight modification. The joint data structure combined the abstract syntax tree(AST), control flow graph(CFG), data flow graph(DFG), and natural code sequence(NCS) as shown in Figure 30 for the example code snippet shown in Figure 29 and embedded the graph attributes.

```
short add(short b){
        short a = 32767;
        if(b > 0){
                a = a + b;
        }
        return a;
}
```

Figure 29: Example Code Snippet from [74]

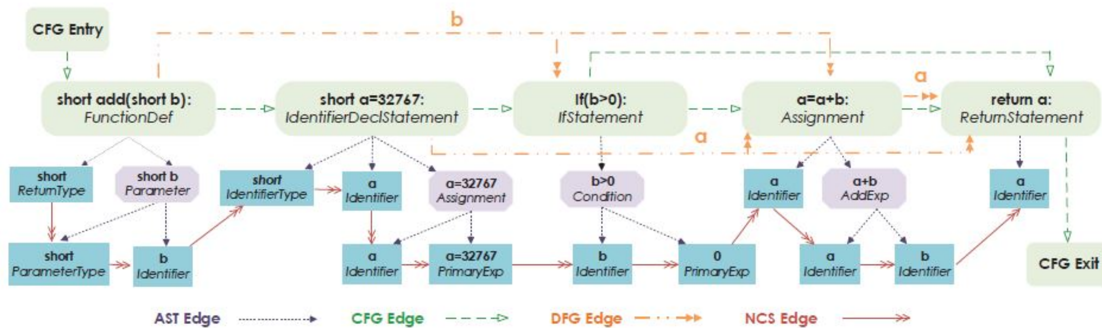


Figure 30: Joint Graph structure for example shown in figure 29 from [74]

In the second layer, called the Gated Graph Recurrent layer, same as the previous research work, the author used a gated graph recurrent neural network to learn the node representation as it allowed a deeper propagation than the GCN and GraphSAGE model. In the final layer, the author presented a novel module called the Conv Layer. It directly applies a 1-D convolution and dense neural network to the node representation generated by the gated recurrent graph layers for the graph-level prediction by selecting the set of nodes and edges relevant to the downstream task. The author used this approach for C/C++ source code and improved the detection of intra-procedural vulnerabilities but failed to cover inter-procedural vulnerabilities. The approach used the Gated Graph Recurrent Neural Network model that learned the model parameters by back-propagation through time(BPTT). Thus, it made it unsuitable for large graphs as it requires running the recurrent function multiple times over all the nodes, which requires storing the intermediate state of all nodes in memory.

4 Methodology & Implementation

In the following section, we present our methodology and its implementation for solving the research questions introduced in Section 1. Our approach aims to automatically detect vulnerabilities in the source code to a finer granularity to assist and provide helpful suggestions to the code auditors regarding the vulnerabilities in the source code using the technologies we presented in Section 2. Figure 31 shows an overview of the implementation architecture where the red lines depict the pre-training workflow, and the black line depicts the classification workflow.

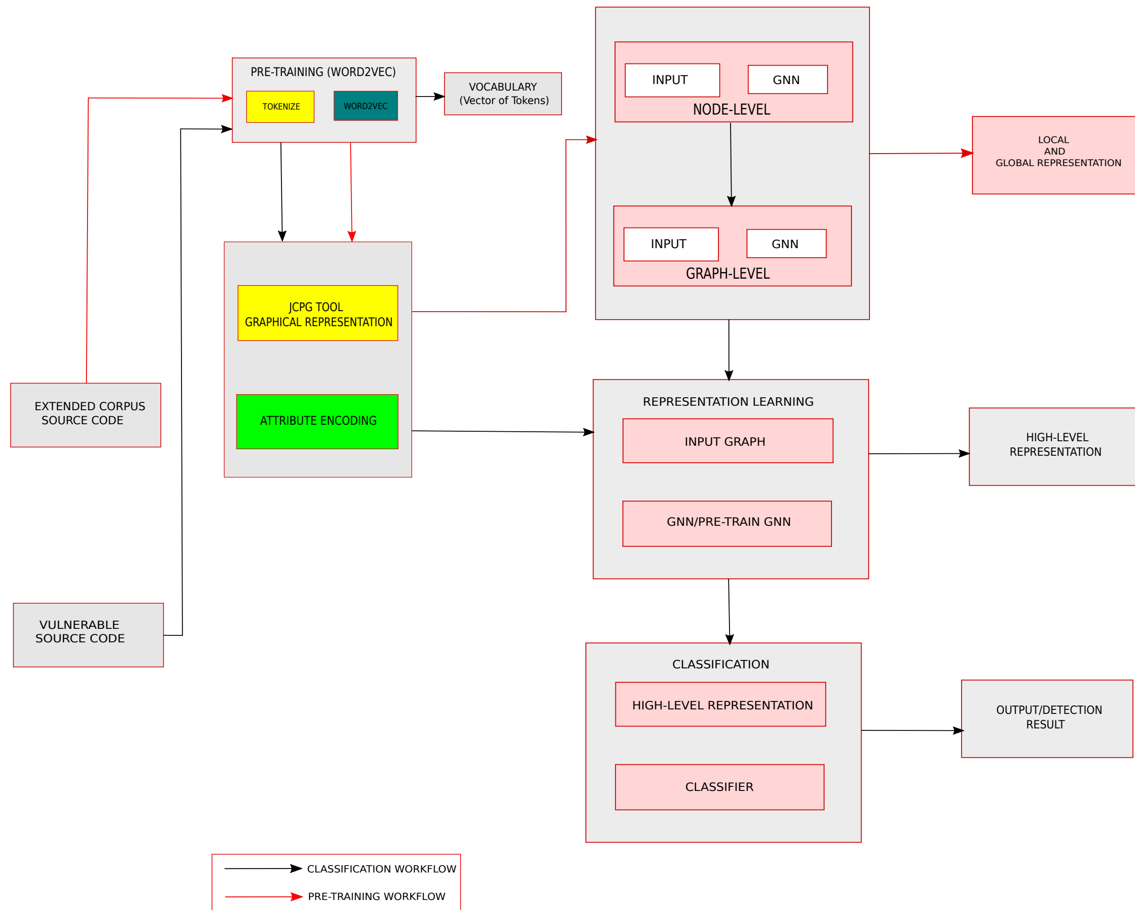


Figure 31: Overview of the implementation architecture

Our methodology summarizes in three steps, namely:

- **Data Selection & Pre-Processing:** The first step of the approach is selecting a dataset and pre-processing it. The pre-processing step factorizes into two steps: **intermediate graphical representation** where we transform the source code into graphical structure capturing the source code's structural and semantic properties. **Atrribute embedding** where we embed the graph attributes to vector representations capturing their semantic meaning.
- **Representation Learning:** In the second step of the approach, a graph neural network learns the semantic and structural features of the source code and outputs

a global vectorized representation of the graph. The GNN learns the vulnerability patterns via this step.

- **Classification:** The third step is to train a graph-based machine learning model to classify a given unseen source input as vulnerable or non-vulnerable.

These steps are explained in Section 4.1, Section 4.2, Section 4.3, and Section 4.4 respectively.

4.1 Data Selection

Dataset selection was a crucial process for training and evaluating a neural network model. An application's effectiveness based on the neural network method highly correlates with the training dataset's quality and quantity. Hence, access to a high-quality dataset was quintessential. For our approach, we needed an attested Java vulnerability dataset. However, since most previous studies were on C programs' vulnerabilities, there was a lack of an attested dataset for benchmarking our proposed approach. The criteria were to select an attested dataset to evaluate our approach and compare it with other approaches.

Since creating a dataset, we required manual labeling of the functions, challenging and tedious. Based on the criteria mentioned above, we chose the SARD's Juliet Test Suite [12] for our implementation. The SARD's Juliet Test Suite was a synthetic dataset created by the National Security Agency's Center for Assured Software(CAS) as a framework to assess static analysis tools. The test suite was a collection of 112 CWE entries with close to 26k test cases. There were multiple test cases for each vulnerability. Each of these test cases had a unique composition of which source it uses and what control and data flow complexity it added. Each test case contained a primary wrong method that either contained the flawed constructs or called another method containing the flaw. Similarly, each test case also contained an excellent primary method that does not contain the non-flawed construct but calls the suitable secondary method containing the actual non-flawed construct. For our implementation, we extracted the test cases for a couple of vulnerabilities and labeled the methods tagged as 'bad' as one and the methods tagged as 'good' as zero.

4.2 Data Pre-Processing

In our approach, the key idea is to enhance the information provided by reading the source code file and using that enhanced information to detect vulnerable patterns in the source code file using Neural Networks. Towards this direction, to enhance the information in the source code, we represent the source codes as graphs instead of a sequence of code lines as it provides information regarding the data and control dependencies between the statements. Finally, the graph attributes are embedded, preserving their semantic properties to create the final dataset fed to the Graph Neural Network model. The following section presents the method used to extract the suitable code properties from the source code file and represent it as a graph. Finally, we present the method used for embedding the graph attributes preserving their semantic meaning.

4.2.1 Graphical Code Representation

In this step, we transformed the source code into an intermediate graphical data structure highlighting all the elements' properties and their interrelations in a source code. There are various graphical representations of source code developed in the field of program analysis. Some of the primitive graphical representations used are Abstract Syntax Trees, Control Flow Graphs, and Program Dependence Graphs. For our analysis, we utilized the code property graph (CPG) 2.1.4 representation which is a joint data structure proposed by Fabian Yamaguchi that merges the concepts of classical program representations, namely Abstract Syntax Tree(AST) 2.1.1, Control Flow Graph(CFG) 2.1.2 and Program Dependence Graph(PDG) 2.1.3.

There are some well-known frameworks available to create a graphical representation of a source code like SOOT [6], Java Objective-sensitive ANAlysis(JOANA) [55] and Joern [53]. These frameworks operate at the byte-code level. Since these frameworks operate on a normalized intermediate code rather than the abstract syntax tree, they simplify the computation by not dealing with full program language and lose high-level abstraction during the translation to intermediate code [59]. Additionally, the Joern framework is not compatible with Java source codes. For our approach, the essential requirement was that the operation should be performed at the source code level and not at the intermediate/machine code level. Since source code available might not always be fully executable due to some issues with building configuration files or organizational matters, the source code itself cannot compile directly. To alleviate this problem, we developed an open-source framework that utilizes the CPG concept for C-language to create code property graphs for Java directly from the source code files with slight representation changes. The details of the tool are as follows.

Java Code Property Graph(JCPG) Tool

To alleviate the lack of tools or frameworks that operate at the source code level to capture the data-flow and control-flow analyses and provide an intermediate graphical representation of the source code, we developed a Java Code Property Graph(JCPG) tool. The tool is based on ANTLR and Java Graph Library and operates at the source code level to extract the graphical representation of the source code. The tool's ability to operate on the source code level instead of the intermediate/machine code level provides the ability to perform control-flow and data-flow analyses directly on the AST level. This ability is beneficial since during translation to intermediate/machine code, the high-level abstractions do not compile, which is particularly vital for processes like code refactoring, bug detection, and coding convention violations detection [59]. Additionally, the tool can export the graphical representation in formats like GraphML, JSON, and DOT to be used further for various analyses.

Working

The tool implementation was based on the ANTLR(ANother Tool for Langauge Recognition) parser generator and Java Graph Library. The tool creates primitive graph represen-

tations and overlays them to create the final graphical representation, i.e., code property graph where the base is the Abstract Syntax Tree. To briefly describe the working of the tool, we used the example code shown in Figure 32. For creating the Abstract Syntax

```
public class gcd{
        public int gcd(int x, int y){
                int tmp = 0;
                while(y != 0){
                        tmp = x % y;
                        x = y;
                        y = tmp;
                }
                return x;
        }
}
```

Figure 32: Example source code snippet

```
public class test{

        //method 1
        public int addNumbers(int a, int b){
                int sum = a + b;
                return sum;
        }

        //method2
        public int multiply(int a, int b){
                int value = addNumbers(a, b);
                int mul = value * 5;
                return mul;
        }

        public static void main(String[] args){
                int num1 = 25;
                int num2 = 15;

                test obj = new test();
                int result = obj.multiply(num1, num2);
                System.out.println("The result is:" + result);
        }
}
```

Figure 33: Example source code snippet for ICFG

Tree(AST), the tool first creates the Parse Tree for the source code, a hierarchical, rooted tree representation of terminals or non-terminals according to some context-free grammar. Additionally, the tool creates a map that includes the nodes' contextual properties in the parse tree. For creating the AST for a source code, the tool uses the ANTLR visitor mechanism instead of the listener mechanism where the walk is performed with an explicit visit call and returns a custom type value. Hence, the tool uses a visitor class that extends the ANTLR parse tree visitor to construct the Abstract Syntax Tree from the parse tree. The tool uses the map of contextual properties and the parse tree to link the graph with the other graphs. Figure 34 shows the Abstract Syntax Tree generated by the tool for the example code shown in Figure 32. The tool used the exact mechanism with a control flow

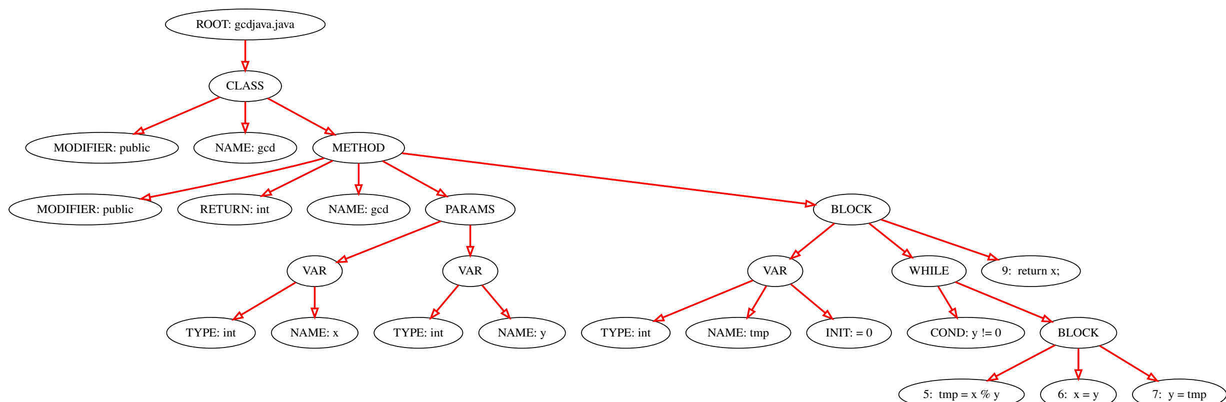


Figure 34: AST generated by JCPG tool for code shown in Figure 32

visitor class that walks through the parse tree to create the control flow graph as shown in Figure 35 and the tool then uses this primitive graphical representation for the generation of the remaining graphical representations. The tool uses the same mechanism as mentioned above to capture the interprocedural calls and create the inter-procedural control flow graph, followed by the creation of JavaClass structure by extracting all class-info from

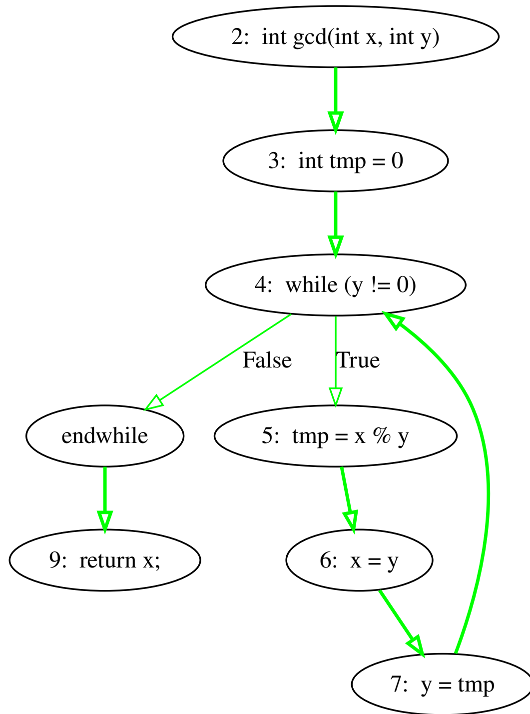


Figure 35: CFG generated by JCPG tool for code shown in Figure 32

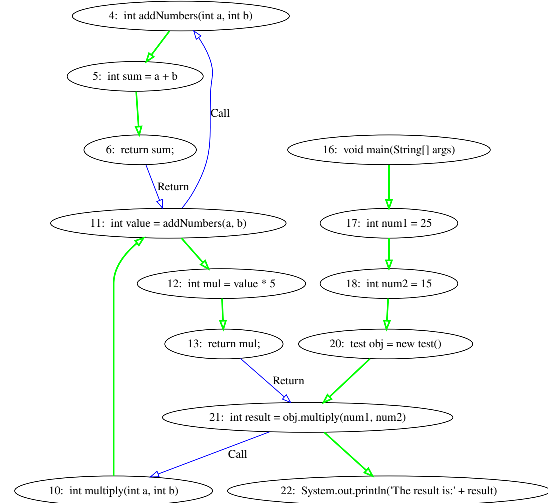


Figure 36: ICFG generated by JCPG tool for code snippet shown in Figure 33

the given source file and the JDK. Next, the tool constructs visitors for each parse tree and contextual property map using the ANTLR visitor pattern for java files and used the contextual property map to create the CFG. Additionally, the tool creates a list of method entries for each java file, creates a new CFG with the method entries and the CFGs, and creates a second contextual map capturing the method calls from the code statements. For instance, consider the example code shown in Figure 33 consisting of three methods. The main method calling the method 'Mul,' which in turn calls method 'Add.' Figure 36 represents the inter-procedural calls with 'CALL' and 'RETURN' edge depicting the procedural call and return from one method to the other in the source code.

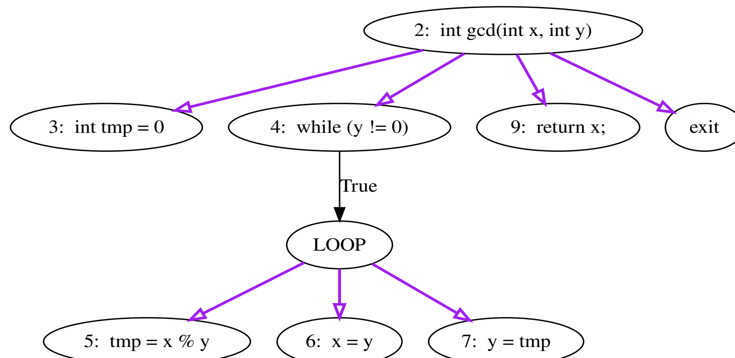


Figure 37: CDG for the example code snippet shown in Figure 32

The JCPG tool generates the Program Dependence Graph(PDG) by first constructing the Control Dependence Graph(CDG) and the Data Dependence Graph(DDG) and then combining them. For constructing the control dependence graph, the tool used the control

flow graph and the concept of Post-Domination [8] where Post-Domination is defined as follows: Each node N in CFG has at least one path from the start to the node N and from node N to stop. The node N post-dominates a node V if every directed path from N to stop contains V. Figure 37 shows the control dependence graph for the example code shown in Figure 32. For building the data dependence graph, the tool used the CFG and the analysis of the Definition-Use pair. Def-Use pair defines the program's point where a value is defined and the point where it uses the defined value. We first performed the Definition-Use analysis at every code statement in the program. We then used the gathered information while traversing the Control Flow Graph to extract data-dependence relations among statements. The DEF-USE pair analyses for the example code 32 is shown in the Figure 38. The data dependence graph for the example code 32 is shown in the Figure 39.

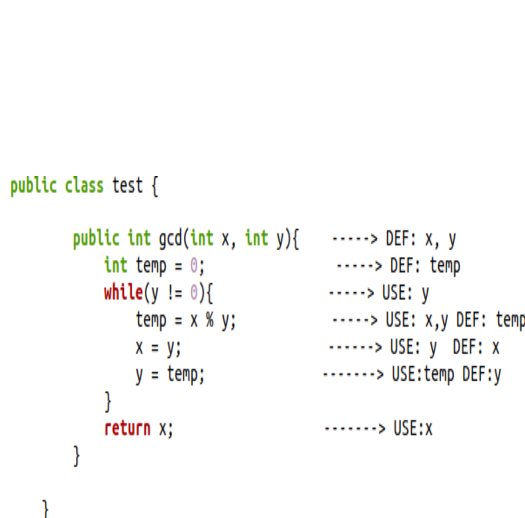


Figure 38: DEF-USE analyses of example code 32

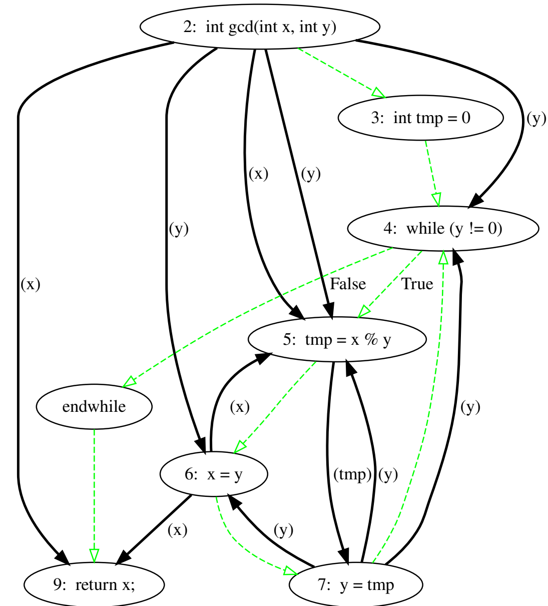


Figure 39: DDG of example code snippet shown in Figure 32

Finally, the tool combines all the primitive graphical representations as shown in Figure 41 for the example code shown in Figure 32 to create a joint data structure with four types of subgraphs(AST, CFG, ICFG, and PDG) that share the same set of vertices $V = V^{ast}$ and set of edges E capturing the properties of all the four sub-graphs as shown below:

- **AST:** edge connecting the AST vertices,
- **FLOWS_TO:** edge connects statement nodes to the successor in the control flow where the edge label value is ϵ,
- **FLOWS_TO_TRUE:** edge defines the connection of statement node to its successor if the condition is true,
- **FLOWS_TO_FALSE:** edge defines the connection of statement node to its successor if the condition is false,
- **Throws:** connects the statement node to its successor in the control flow graph when an exception is thrown.

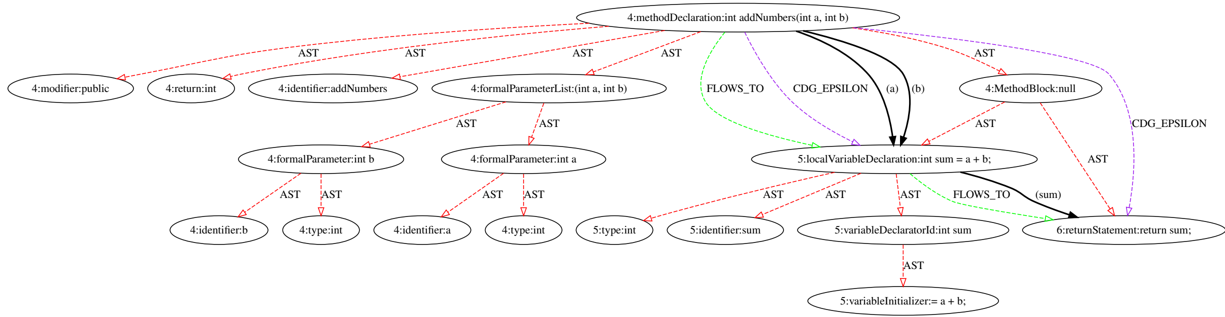


Figure 40: Code Property Graph for method addNumber from code snippet shown in Figure 33

- **CDG_EPSILON:** connects the statement node to its successor node in the control dependence graph if the edge value is ϵ,
- **CDG_TRUE:** edge connects statement nodes to the successor in the control dependence flow where the condition statement is true,
- **CDG_FALSE:** edge connects statement nodes to the successor in the control dependence graph where the edge condition is false,
- **CDG_THROWS** and **CDG_NOT_THROWS:** are edges connecting statement node to its successor if an exception is thrown or not thrown respectively,
- **DDG:** edge defines the data dependence of one statement node to other with the data variable as its edge label,
- **CALL** and **RETURN** edge define the control flow from callee method to calling method and the return of control back to the callee method respectively in ICFG.

Each vertice/node in the graph G has four attributes as follows:

- **CODE:** refers to the code statement represented by node v.
- **TYPE:** refers to the type of the node v, there is a finite number of node types.
- **DEF:** refer to the variable defined in the code statement represented by v if any.
- **USE:** refer to the variable used in the code statement represented by v if any.

The edge $e_{i,j} \in E$ connecting the nodes v_i and v_j has one attribute, namely:

- **TYPE:** refers to the type of edge, there is a finite number of edges.

The tool outputs method-level CPGs of the source code and the file-level CPGs without losing the ability to produce interprocedural control flow for the methods. Figure 40 shows the method-level code property graph for the code snippet shown in Figure 33.

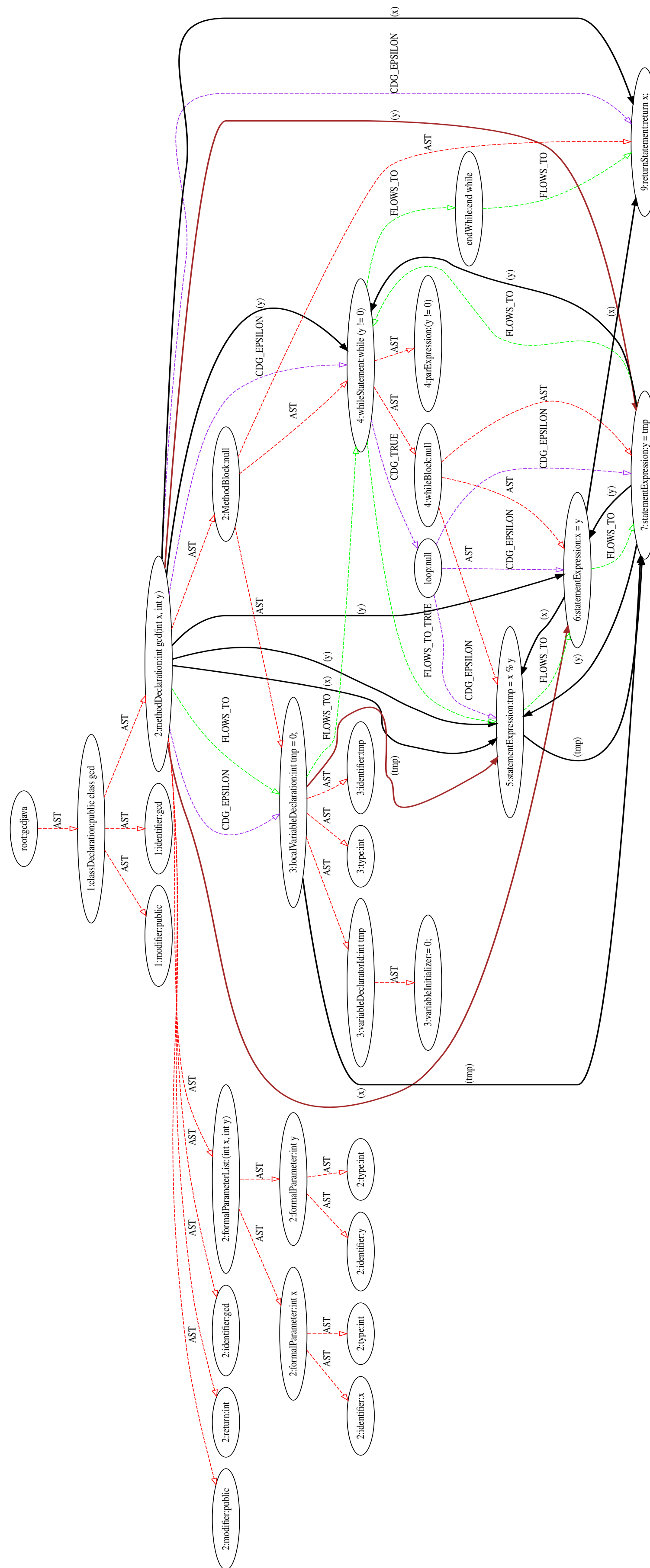


Figure 41: Code Property Graph for code snippet shown in Figure 32

4.2.2 Graph Attribute Embedding

After converting the source code files into code property graphs, the next step was to convert the human-readable intermediate graphical representation into vector representations consumable by Graph Neural Networks. For this, we convert each node and edge to an N-dimensional vector where each node contains various pieces of information depending upon its source. The edge contains information about the relationship between the nodes. To preserve the nodes' semantic attributes, we used the pre-trained Word2Vec model to encode node attributes to the N-dimensional vector. In contrast, we embed the node type attribute and edge type attribute using the label encoder of the sklearn python library [16]. The final node representation is the concatenation of the vectors corresponding to the node attributes' Type', 'DEF,' 'USE,' and the mean of the code attribute's tokens.

Pre-Training Word2Vec Model

Representation Learning is a vital technique that has revolutionized NLP problem-solving approach, and several approaches use it to capturing the source code's semantic context. After generating the code property graph for the source code in the pre-processing stage of the architecture, the next step aimed to learn to capture and retain the semantic meaning of the graph attributes. To achieve that, we used the Word2Vec embedding model. We first pre-trained the Word2Vec model to learn the semantic meaning of the tokens from the source code as follows:

1. **Dataset selection:** extract the dataset to train the model.
2. **Pre-Process Data:** pre-process the extracted dataset.
3. **Training:** train the Word2Vec model using the pre-processed data and visualize the resulting embedding to evaluate the result.

We used the entire source code repository as our extended code corpus to train the Word2Vec model. Since the source codes are similar to texts, we converted the source codes into textual code sentences. To do that, we tokenized the source code file using JavaLang which is a pure Python library providing a lexer and parser targeting Java8 as shown in Figure 42 for the example code shown in Figure 32. Then we took tokens of each file and considered it as one textual code sentence. After pre-processing the dataset and creating a training dataset, we trained the Word2Vec model. For Word2Vec model implementation, we used the Gensim [47] python library. To learn the semantic meaning of the code tokens, the Word2Vec model projects each token vector in a multi-dimensional space as shown in Figure 43 for example code is shown in Figure 32. It places the tokens with similar semantics close to each other to capture the code tokens' similarity and semantics.

```
Starting building of dictionary

Filename is: gcdjava.java

single statement is:
['public', 'class', 'gcd', '{', 'public', 'int', 'gcd', '(', 'int', 'x', ',', 'int', 'y', ')', '{', 'int', 'tmp', '=', '0', ';', 'while', '(', 'y', '!=', '0', ')', '{', 'tmp', '=', 'x', '%',
'y', ';', 'x', '=', 'y', ';', 'y', '=', 'tmp', ';', '}', 'return', 'x', ';', '}', '}']

dict building tokens are:
('y', ';', 'int', 'x', '=', '{', 'tmp', '}', 'public', 'gcd', '(', ')', '0', 'class', ',', 'while', '!=', '%', 'return')

count is:
[['UNK', 0], ('y', 5), (';', 5), ('int', 4), ('x', 4), ('=', 4), ('{', 3), ('tmp', 3), ('}', 3), ('public', 2), ('gcd', 2), ('(', 2), (')', 2), ('0', 2), ('class', 1), (',', 1), ('while', 1),
('!=', 1), ('%', 1), ('return', 1)]

count length is: 20

dictionary is:
{'UNK': 0, 'y': 1, ';': 2, 'int': 3, 'x': 4, '=': 5, '{': 6, 'tmp': 7, '}': 8, 'public': 9, 'gcd': 10, '(': 11, ')': 12, '0': 13, 'class': 14, ',': 15, 'while': 16, '!=': 17, '%': 18, 'return':
19}
```

Figure 42: Code statement and Word2Vec dictionary generated for code snippet shown in Figure 32

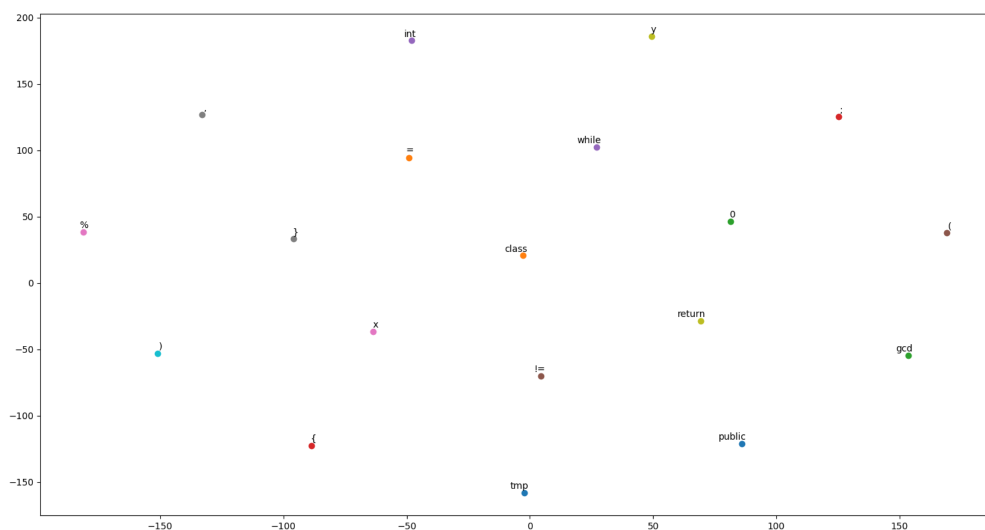


Figure 43: Projection of code token for example code shown in Figure 32 in multi-dimensional space

4.3 Representation Learning

In the representation learning stage, we use a Graph Neural Network(GNN) model to learn the high-level representation of the intermediate graphical representation of the source code to perform the domain-specific downstream tasks. Broadly, the Graph Neural Network models divide into two categories: Spectral-based Graph Neural Network and Spatial-based Graph Neural Networks as described in Section 2 where the models vary based on different neighborhood aggregation techniques. Since, in our case, the contribution of neighboring nodes was not identical to the representation of a node as considered in Spectral-based Graph Neural Network. We required a Spatial-based Convolutional Graph Neural Network(ConvGNN) or a Recurrent Graph Neural Network(RecGNN), which takes each neighboring node's weighted contribution. However, RecGNNs and Spatial-based Convolutional Graph Neural Networks (ConvGNN) share the same underlying idea of message passing. RecGNNs use the same graph recurrent layer in every layer for updating the node representation. In contrast, ConvGNNs use a different graph convolutional layer in every layer for updating the node representations. One of the prominently used

RecGNNs for graph classification tasks is Gated Graph Neural Network which uses gated recurrent unit(GRU) as recurrent function to alleviate the need to constrain parameters to ensure convergence and reducing the recurrence to a fixed count. However, GGNN is not suitable for large graphs. It uses a backpropagation algorithm to learn the training parameters that require running the recurrent function multiple times over all the nodes that need storing all node's intermediate state in memory. Hence, we chose Spatial-GCNN introduced in Section 2 for our graph-level prediction.

There are numerous variants of spatial-based Graph Convolutional Networks as described in Section 2, based on our downstream task of vulnerability prediction by pattern matching, we required a model with high expressivity. Hence, we chose the Graph Isomorphism Network(GIN) [65] introduced in Section 2 which is a state-of-the-art GNN architecture. The node update function of the Graph Isomorphism Network is given by the equation 36

$$h_v^k = MLP^k((1+\epsilon^k)h_v^{k-1} + \sum_{u \in N(v)} h_u^{k-1}) \tag{36}$$

where the goal of the aggregation function and the update function is making the message passing framework equivalent to the Weisfeiler-Lehman test.

As explained in Section 4.2.1, the input graph in our implementation had node features as well as edge features. Hence we modified the GIN architecture to incorporate the edge features. Equation 37 represents the changed node update function of the Graph Isomorphism where the feature dimensionalities of the node and edge should match.

$$h_v^k = MLP^k((1+\epsilon^k)h_v^{k-1} + \sum_{uinN(v)} ReLU(h_u^{k-1} + e_{j,i}^{k-1})) \tag{37}$$

In Graph Isomorphism Network(GIN), the readout function produces the embedding of the entire graph utilizing embeddings of individual nodes that get more refined and global with increment in the number of iterations. Hence, many iterations are vital to achieving good discriminatory power, although features from earlier iterations may sometimes generalize better. GIN uses an architecture that is analogous to Jumping Knowledge Networks. It gathers information from all iterations of the model to consider all structural information of the input graph. It uses the 'SUM' aggregator as an aggregation function instead of 'MEAN,' or 'MAX' aggregator since the ranking power of 'SUM' aggregator is greater than 'MEAN' and 'MAX' aggregator. 'SUM' aggregator captures the full multiset. In contrast, the 'MEAN' captures only a proportion of elements of a given type, and the 'MAX' aggregator reduces the multiset to a simple set by ignoring the multiplicities. The graph representation is defined by equation 38 where *READOUT* sums node features from all iterations.

$$h_G = CONCAT(READOUT(h_v^{(k)}|v \in G)|k = 0, 1, ..., K) \tag{38}$$

Since the node has four features and the edge has one feature, we add padding to match the node and edge feature dimensionalities. We applied batch normalization before applying the ReLU activation function in Equation 37. We applied the dropout function after the ReLU activation function at all GIN layers except at the final layer so that the final node representation can take negative values, which are pivotal for the node-level pre-training strategy described in Section 2. With this node update equation, we obtain the final node representation h_v^k, which gives the local representation of the graph. We pool the final node state representations using the graph pooling mechanism to learn the global representation. As explained in Section 2 we used 'sum' graph pooling mechanism.

4.3.1 Pre-Training Graph Neural Network(GNN)

We utilized the transfer learning technique to alleviate the lack of an attested vulnerability dataset for effective and accurate prediction of vulnerability in a source code file. We pre-trained our Graph Neural Network model using the approach shown in Section 2.4. We pre-train our model at the individual node-level and the entire-graph level to increase the performance of the model. The transfer learning approach introduced two node-level pre-training strategies, namely, Attribute Masking and Context Prediction, and two graph-level pre-training strategies, namely Property Prediction and Similarity Prediction, as described in Section 2.4.1. We implemented two pre-training pipelines where one pipeline was the combination of attribute masking for node-level pre-training and property prediction for graph-level pre-training while the other pipeline is the combination of context prediction for node-level pre-training and property prediction for graph-level pre-training.

```java
public void bad() throws Throwable
{
    byte data;

    /* POTENTIAL FLAW: Use the maximum size of the data type */
    data = Byte.MAX_VALUE;

    /* POTENTIAL FLAW: if data == Byte.MAX_VALUE, this will overflow */
    data++;
    byte result = (byte)(data);

    IO.writeLine("result: " + result);

}
```

Figure 44: Code Example of Integer OverFlow

In the first pipeline, for node-level pre-training, we used the attribute masking strategy to randomly mask the graph attributes, i.e., node/edge attributes and replaced them with masking indicators. By this model learned the regularities of these attributes distributed over graph structure to capture the domain-specific knowledge. To do that, we used the file-level CPG generated by the JCPG tool as the input. Figure 45 shows the attribute

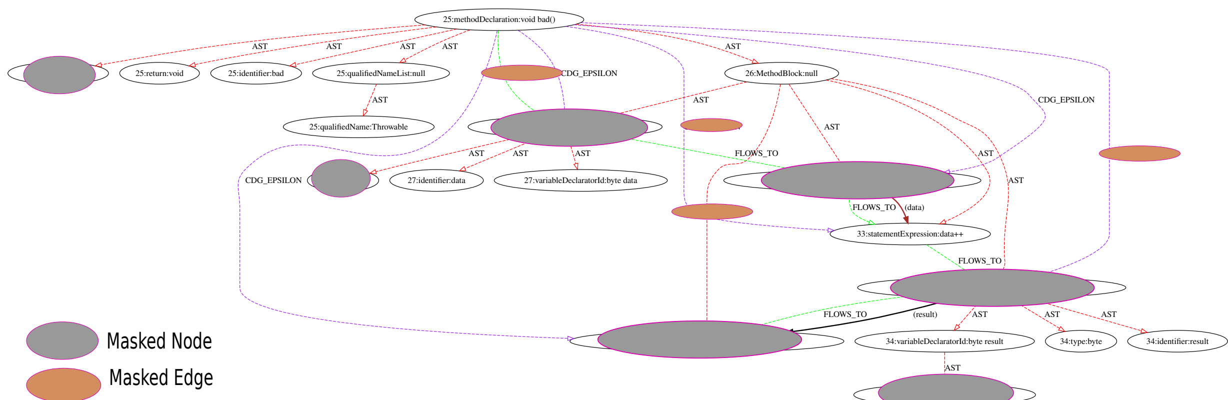


Figure 45: Attribute Masking strategy on the CPG output for example code shown in Figure 44

masking in which the node and edges are randomly masked and replaced with masking indicators. We used the property prediction strategy for graph-level pre-training, where we use a supervised task as a pre-training task. For that, we used a dataset similar to the dataset used for the downstream task. In the second pipeline, for node-level pre-training,

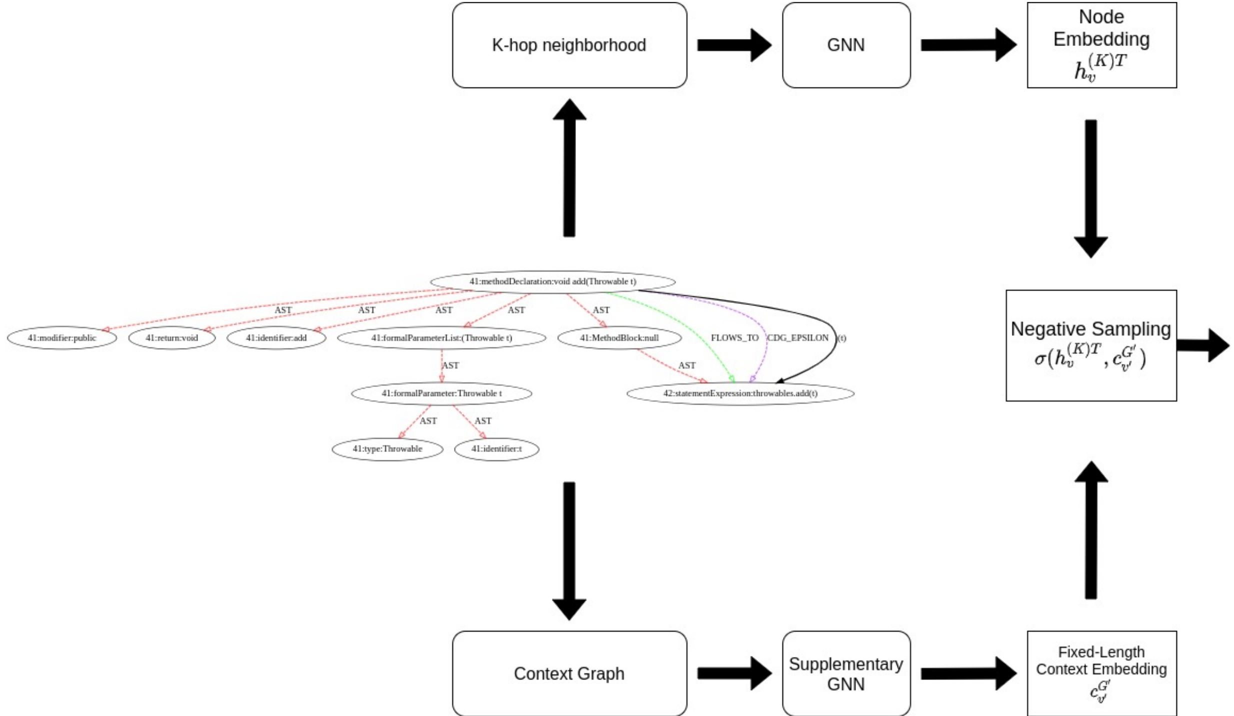


Figure 46: Context Prediction strategy on the CPG output

we used the context prediction strategy as described in Section 2.4.1 where the aim of the model was similar to word embedding, where the model tries to learn the contextual meaning of a word using the surrounding words. In this approach, the model aims at learning the structural context of a node by using its surrounding graph structure and mapping the nodes with similar context to nearby embeddings. The model uses the K-hop neighborhood and context graph for a node v to achieve this. Figure 46 shows the context prediction strategy. We used the property prediction strategy for graph-level pre-training, where we use a supervised task as a pre-training task. For that, we used a dataset similar to the dataset used for the downstream task.

4.4 Classification

The last step of the approach concerns the machine learning model that performs the classification task. As explained in Section 4.2, the Graph Neural Network model performs a representation learning task to output a global vector representation of the input graph that a machine learning model can use to perform the training and the validation tasks of node-level and graph-level classification. In our approach, we used a Multi-Layer perceptron which is explained in Section 2 to classify the input source code as vulnerable or non-vulnerable. We create MLP as a Sequential model, where the input layer is a dense layer with each neuron connected to every other neuron in the next layer [45]. The dimension of the input layer was the same as the embedding dimension. We used the ReLU

activation function, and the output layer dimension was equal to the number of classes for classification.

We used various libraries and frameworks for the implementation of our approach. Table 1 provides an overview of the components and the libraries or frameworks used for the implementation of those components. The implementation of the proposed methodology

Table 1: Components used for the implementation of our approach and the library/frameworks used for these components

Module	**Task**	**Component Used**	**Library/Framework Used**
Data Pre-Processing	Intermediate Graphical Representation	JCPG Tool	ANTLR
			Java Graph Library
	Attribute Masking	Word2Vec	Genism Python library
		Skilearn Encoder	Skilearn Preprocessing API
Representation Learning	GNN Model	Graph Isomorphism Network model	Pytorch Geometric
			Pytorch
	Transfer Learning	SNAP pre-training strategies(Node-level + Graph-Level): Attribute Masking + Property Prediction Context Prediction + Property Prediction	Pytorch Geometric
			Pytorch
Classification	Classification	Multi-Layer Perceptron	Pytorch

as a tool called GraphVul is available online on GitHub. [2] [3]

5 Evaluation

In the following section, we evaluate the various components of the implementation architecture. Firstly, we describe the criteria for the dataset selection, and then we evaluate the selected dataset. Secondly, we evaluate the components used for data pre-processing. Thirdly, we evaluate our Graph Neural Network model and the Graph Neural Network model used in the previous researches [74] [58] on the selected dataset. Finally, we perform the comparison of evaluation results of the two models. Additionally, we compare our model with the well-known Static Analysis Tools(SAT).

5.1 Dataset

As mentioned in Section 4, based on specific criteria, we chose the SARD's Juliet Test Suite [12] to perform the experiments to evaluate the effectiveness of our implementation.

Table 2: OWASP Top 10 vulnerabilities of Java

Rank	Vulnerability	Description
A1	Injection	Injection flaws that occur when untrusted data is sent to an interpreter as part of a command or query.
A2	Broken Authentication	Vulnerabilities due to incorrect implementation of application functions related to authentication and session management.
A3	Sensitive Data Exposure	Vulnerabilities due to weak protection of sensitive data
A4	XML External Entities(XXE)	Attack against application that parses XML input and occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser.
A5	Broken Access Control	Attacks due to improper access control that the adversary can exploit to access unauthorized functionalities and/or data.
A6	Security Misconfiguration	Attacks due to misconfiguration of secure settings for the application, framework, application server, web server, database server, and platform.
A7	Cross-Site Scripting(XSS)	Attacks that occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping.
A8	Insecure Deserialization	Vulnerability in which untrusted or unknown data is used to either inflict a denial of service attack(DoS attack), execute code, bypass authentication, or further abuse the logic behind an application.
A9	Using components with known vulnerabilities	Application and APIs using components such as libraries, frameworks, and other software modules with known vulnerabilities.
A10	Insufficient logging & monitoring	Exploitation of insufficient logging and monitoring.

The SARD's Juliet Test Suite contains various vulnerabilities, and we categorize these vulnerabilities based on OWASP Top 10 Java vulnerabilities shown in Table 2. Since it is time-consuming to evaluate all the vulnerabilities, we narrow the scope by narrowing the list of vulnerabilities to analyze by defining the criterion that the vulnerability must compare results between the different Graph Neural Network models and static models. Table 3 shows the filtered list of vulnerabilities from all the CWE entries such that at least two of the Static Analysis Tools(SATs) cover the vulnerabilities [11]. Additionally, Table 3 shows the class-distribution for the CWE entries.

5.2 Pre-Processing

The next step after the dataset selection step is to pre-process the dataset. The first step of pre-processing is to transform the source code dataset to an intermediate graphical

Table 3: The filtered list of CWE entries with class distribution used for evaluation experiments

CWE		GRAPH COUNT			CLASS DISTRIBUTION	
ID	NAME	FILE LEVEL	METHOD LEVEL		NON-VULNERABLE (∼Approx)	VULNERABLE (∼Approx)
			Non-Vulnerable Graphs (#count)	Vulnerable Graphs (#count)		
A1	**Injection**					
78	OS Command Injection	720	1872	744	70	30
89	SQL Injection	3662	13881	3721	80	20
90	LDAP Injection	720	1872	744	70	30
113	HTTP Response Splitting	2202	8316	2232	80	20
643	XPath Injection	735	2793	744	80	20
A2	**Broken Authentication**					
256	Plaintext Storage of Password	64	252	62	80	20
259	Hard Coded Password	182	489	165	75	25
321	Hard Coded Cryptographic Key	62	156	62	70	30
A3	**Sensitive Data Exposure**					
315	Cleartext Sensitive Info in Cookie	62	241	62	80	20
319	Sensitive Cleartext Transmission	610	2310	620	80	20
A5	**Broken Access Control**					
23	Relative Path Traversal	722	1893	744	70	30
36	Absolute Path Traversal	722	2637	744	80	20
A7	**Cross-Site Scripting**					
80	Basic XSS	1080	2829	1116	70	30
81	XSS Error Message	541	1401	557	70	30
83	XSS Attribute	542	1425	558	70	30

representation dataset. The second step is to embed the graph attributes to capture the semantic properties of the attributes. In the following section, we present the criteria for selecting components for the two pre-processing steps, followed by evaluating the selected components.

5.2.1 JCPG Tool

The research requirement was to have a tool that operates at the source code level and produces an intermediate graphical representation even for non-compilable source codes. Additionally, since the aim of the approach was to find vulnerabilities at the method level, it was required that the tool outputs intermediate graphical representation at the method-level also. These criteria emphasize that most of the available code repositories for code review lack the company-specific packages that make the source-codes non-compilable. As described in Section 4, we used Code Property Graph as the intermediate graphical representation of the source code. The available tools and tools used in previous research do not comply with these requirements since they operate at the machine code level that requires compilation of the source code. To alleviate this problem, we developed a tool called JCPG that operates at the source code level and outputs the Code Property Graph of the source code even if the source code is non-compilable due to missing packages and supporting files.

The JCPG tool is based on ANTLR [46] and Java graph library [26]. It operates at the source code level and produces a code property graph for the Java source code, even in supporting files. The tool uses the ANTLR parser to parse the Java source code, and then it uses the ANTLR visitor pattern to create the graphs. The tool uses the Java graph library to output the graph and exports it in three formats: DOT, JSON, and GML format. To evaluate the tool for the required criteria, we first evaluated the tool for basic

Table 4: List of source code files used for tool evaluation for Java construct

CONSTRUCT	SOURCE CODE	DESCRIPTION
Constructor	Const.java	source code to illustrate the Java constructors with and without parameters
Class	ClassTutorial.java	source code to illustrate constructing and accessing Java class
Static Keyword	StaticBlock.java	source code to illustrate the use of static keyword
If-else control	Ifelse.java	to illustrate if-else control statement
2*For loop	TradFor.java	to illustrate the traditional for loop
	ForEach.java	to illustrate the for-each loop
While loop	While.java	to illustrate the while loop
Do-While loop	DoWhile.java	to illustrate the do-while loop
Label	Label.java	to illustrate the label object
Switch	Switch.java	to illustrate switch case
Synchronized Block	Synch.java	to illustrate the Java synchronized block
Try block	TryCheck.java	to illustrate the Try-Catch block
	TryWithRes.java	to illustrate the Try block with resource
	TryMultiRes.java	to illustrate the Try block with multiple resources
	TryFinally.java	to illustrate the finally block
Throw Keyword	Throw.java	to illustrate the Throw keyword
Throws Keyword	Throws.java	to illustrate the Throws keyword

Java constructs. Followed by this, we evaluated the tool for research requirements. Table 4 shows the source codes used for the evaluation of the tool for the basic Java constructs. The detailed results of the evaluation are shown in Appendix A.1. We observed that the tool generated code property graphs for all Java constructs excepts enumerators. To evaluate the tool for the requirement of code property graph generation in the absence of third-party packages and files, we used various projects from GIT, Table 5 shows some of these projects. We observed that the tool could generate code property graphs even in the absence of required third-party packages and supporting files. Additionally, we evaluated the tool for the method-level CPG generation for the example codes.

Table 5: GIT projects used for evaluation of JCPG tool

Project	Repository Link
Apache Tomcat	https://github.com/apache/tomcat.git
neo4j-geoff	https://github.com/neo4j-contrib/neo4j-geoff.git
sli4j	https://github.com/99soft/sli4j.git
japex-plugin	https://github.com/jenkinsci/japex-plugin.git
DevBukkit	https://github.com/Bukkit/DevBukkit.git

5.2.2 Attribute Embedding: Word2Vec Model

The second step of pre-processing is Attribute Embedding, where we embed the graph attributes. In our approach, the graph had four-node attributes, namely Node Type, Node Label, Defined Variable, Used Variable, and one edge attribute Edge Type. The criteria for embedding the graph attributes was that the embedding captures the semantic meaning of

the attributes. Considering the attributes Node Label, Defined Variable, and Used Variable that constitutes the tokens from the source code, we used the Word2Vec model. For the attributes Node Type and Edge type, we used the scikit learn label encoder.

To evaluate the Word2Vec model for learning the semantic meaning of tokens from the source code, we implemented the Word2Vec model using the Gensim Python library. We then used the extended code corpus comprising of the entire test suite to train the Word2vec model. The experiment's approach was as follows: firstly, we took the source code repository and extracted all the files. Secondly, we extracted the tokens for each of the files to create code sentences where each code sentence corresponds to a single file. Finally, we trained the Word2Vec model on these code sentences. The training parameters for the model were as follows:

- sentences: refers to the dataset to train the model. We used the tokenized source code file converted into sentences where each sentence refers to one source code file as the dataset.
- vector size: defines the dimensionality/embedding size of the word vectors. Output with higher dimensionality leads to better model accuracy but requires a larger training dataset. To retain higher accuracy with increasing the GNN model's complexity, we selected the vector size as 100.
- min_count: refers to the min-count or threshold for the total frequency of a word. Since the dataset is not large and the source code might have tokens with frequency 1, we selected the min_count value to be 1.

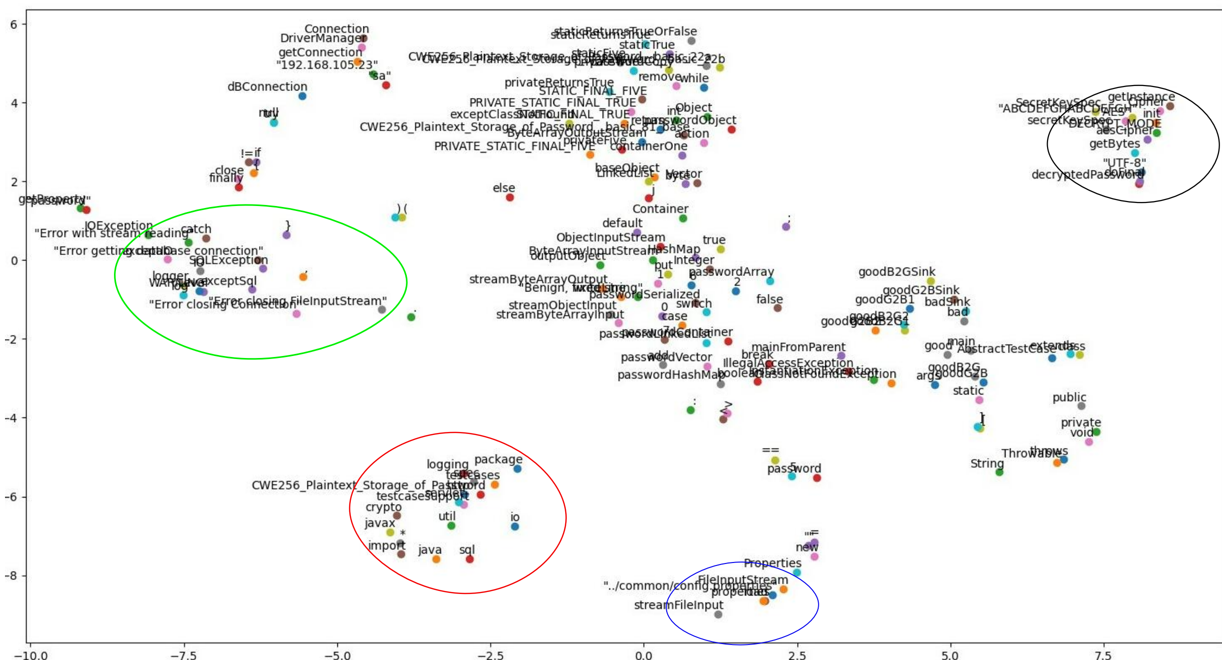


Figure 47: Clusters of embedding for tokens in CWE-256

To visualize the evaluation results, we used a subset of the dataset, i.e., the Broken Authentication vulnerability dataset. Figure 47 visualizes the embeddings for the tokens in

CWE-256 using t-SNE. It can be seen from Figure 47 that it clusters the embeddings for tokens with similar semantic meaning together, highlighting that the Word2Vec model successfully captures the semantic meaning of the tokens.

5.3 Evaluation of the Model

Finally, after pre-processing the data, a Graph Neural Network model performs method-level vulnerability prediction. We used the Graph Isomorphism Network as the GNN model, and to implement the model, we used the Pytorch Geometric library. To evaluate and validate our GNN model, we performed various experiments using the dataset consisting of the CWE entries given in Table 3. As described in Section 4.3.1, we used the pre-training strategy before performing the downstream tasks on the model. In subsection 5.3.1, first, we present the evaluation results for the experiments performed to evaluate our GNN model without pre-training the model for binary as well as multi-class classification tasks with the hyperparameters given in Table 6. These hyperparameters are variables are define the network structure and the variables determining the training strategy. Followed by the evaluation results for the same experiments on the pre-trained GNN models.

Table 6: The hyper-parameters of the GNN model

Hyperparameter	**Value**
Optimizer	Adam
Number of layer	5
Initial Learn Rate	0.0001
Batch Size	32
Readout Function	sum
Dropout Ratio	0.5
Loss	Cross Entropy
Number of epochs	100

Due to the lack of a benchmark dataset and model, in the subsection 5.3.2, we present the comparison of our model with the model used in previous research papers [74] [58]. Since the previous research papers' model and dataset were not available, we implemented the model using the papers' details using the Pytorch-Geometric GNN library. Additionally, since we had an imbalanced dataset to perform a comparison between the two GNN models, we used Stratified K-Fold cross-validation to split the dataset into training, validation, and test dataset instead of a random split which is a version of k-fold cross-validation that splits the dataset randomly while maintaining the same class distribution in each fold.

Finally, in subsection 5.3.3 we present the comparison of our model with the static analysis tools SpotBugs [57], Find Security Bugs [15], and Early Security Vulnerability Detector(ESVD) [52] with improved proof-of-concept to address their limitations [11]. **SpotBugs** utilizes bug patterns as code idiom that is likely to be an error [31] to discover bugs in the Java project. To perform the bug detection, it analysis the Java bytecode using the Byte Code Engineering Library(BCEL) of the Apache Common project [23] instead of the Java programming language. **Find Security Bugs** has a very similar implementation to SpotBugs and uses JVM bytecode instead of Java source code for analysis. It

consists of added new vulnerability detectors based on the functionality provided by SpotBugs and techniques such as taint analysis for detecting new classes of vulnerabilities. The vulnerability detectors use resource files to list their vulnerable sources and sinks, making it possible to easily add or update the detectors without changing their code itself. In addition to Java, Find Security Bugs also supports Apache Groovy, Scala, and Kotlin as they all compile into JVM bytecode. Unlike SpotBugs and Find Security Bugs, **ESVD** analyzes Java code for detecting vulnerabilities using verifiers similar in function to the detectors of SpotBugs and Find Security Bugs. All the verifiers use the same detection algorithm based on tracking vulnerable input from a source to sink.

Evaluation Metrics for Binary Classification

As seen in Table 3, the dataset is an imbalanced dataset which makes the common metric for classification Prediction Accuracy an inappropriate and misleading metric. Hence, we used alternative performance metrics such as Confusion Matrix, Precision, Recall, and F1-score to evaluate the model for binary classification. These metrics are defined below:

- **Confusion Matrix:**is a tabular representation of the counts from predicted and actual values as shown in Figure 48. A binary classification problem has two classes, and the confusion matrix for the same is as shown in Figure 48. True Positive(TP) refers to the number of predictions output where the classifier predicted the positive class as positive. True Negative(TN) refers to the number of predictions output where the classifier predicted the negative class as negative. False Positive (FP) refers to the number of predictions output with classifier predicting the negative class as positive and False Negative(FN) refers to the number of predicted output where the classifier predicted the positive class as negative.

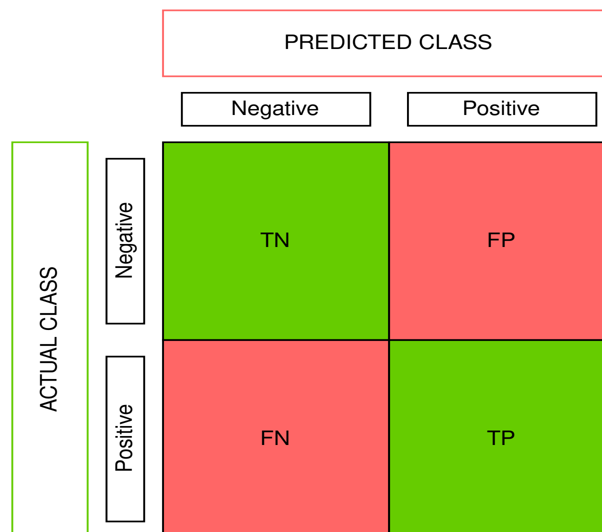


Figure 48: Binary Classification Confusion Matrix

- **Accuracy:**refers to the overall accuracy of the model and is defined by the equation 39.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{39}$$

- **Precision:**is the ratio of accurately predicted positive outcomes to the total predicted positive outcomes given by equation 40.

$$PrecisionScore = \frac{truepositive}{truepositive + falsepositive} \tag{40}$$

- **Recall:**is a ratio of correctly predicted positive outcomes to all the observations of the positive class and is defined by the equation 41. Recall is also known as True Positive Rate(TPR).

$$RecallScore = \frac{truepositive}{truepositive + falsenegative} \tag{41}$$

- **F1 score:**is a weighted average of Precision and Recall and is defined by the equation 42.

$$F1Score = 2 \times \frac{precision \times recall}{precision + recall} \tag{42}$$

Evaluation Metrics for Multi-Class Classification

To evaluate the model for multi-class classification, we use the dataset given in Table 7. The datasets are combinations of different CWE entries, and as seen in Table 3 these datasets are imbalanced. Hence, we use alternative metrics such as the Confusion Matrix, Precision, Recall, and F1 score instead of the accuracy metric to evaluate the classification model. Unlike binary classification confusion matrix with positive and negative class, the confusion matrix for multi-class classification consists of True Positive, True Negative, False Positive and False Negative for each class as shown in Figure 49.

Table 7: Dataset for Multi-Class Classification

Dataset	Vulnerability	CWE Entries
Multiset-1	Hard-Coded Credentials, Missing Encryption of Sensitive Data	CWE-259,321, 315,319
Multiset-2	NULL Pointer Dereference, XPath Injection	CWE-476,643
Multiset-3	LDAP Injection, PlainText Storage of a Password, NULL Pointer Dereferencce, XPath Injection	CWE-90,256, 476,643
Multiset-4	Relative Path Traversal, Absolute Path Traversal, Cross-Site Scripting	CWE-23,36,80, 81,83

Instead of precision, recall, and F1-score for each class, we calculate Micro-Precision, Micro-Recall, and Micro-F1 scores. The metrics are calculated globally by using the total true positives, false negatives, and false positives. We also calculated the Weighted-Precision, Weighted-Recall, and Weighted-F1 score, where it calculates the metrics for each label and takes their weighted mean, accounting for the label imbalance.

Figure 49: Multi-Class Classification Confusion Matrix

5.3.1 Experiments

Non Pre-Training Model

Binary Classification:

- Cross-Site Scripting Table 8 shows the confusion matrix for the non-pre-trained model on the XSS dataset where n=755 refers to the total number of test data points, actual class refers to the True class distribution, and prediction class refers to the Predicted Class Distribution. The model produces True Negative=560, False Positive=02, False Negative=09, and True Positive=214. The accuracy, precision, recall, and f1-score of the dataset model are 98.59%, 99.07%, 95.96%, and 97.49%, respectively.

Table 8: Confusion Matrix for Cross-Site Scripting

n=785		**Prediction**	
		No	**Yes**
Actual	**No**	560	02
	Yes	09	214

- Injection
  - **SQL Injection:**Table 9 shows the confusion matrix for the non-pre-trained model on the SQL injection dataset where n=1761 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=1234, False Positive=15, False Negative=23, and True Positive=489. The accuracy, precision, recall, and f1-score of the dataset model are 97.84%, 97.02%, 95.50%, and 96.25%, respectively.
  - **LDAP Injection:**Table 10 shows the confusion matrix for the non-pre-trained model on the LDAP injection dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution,

Table 9: Confusion Matrix for SQL Injection

n=1761		Prediction	
		No	Yes
Actual	No	1234	15
	Yes	23	489

and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=187, False Positive=01, False Negative=02, and True Positive=72. The accuracy, precision, recall, and f1-score of the dataset model are 98.85%, 98.63%, 97.29%, and 97.95%, respectively.

Table 10: Confusion Matrix for LDAP Injection

n=262		Prediction	
		No	Yes
Actual	No	187	01
	Yes	02	72

- **HTTP Response Splitting:**Table 11 shows the confusion matrix for the non-pre-trained model on the HTTP Response Splitting dataset where n=539 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=427, False Positive=02, False Negative=03, and True Positive=107. The accuracy, precision, recall, and f1-score of the dataset model are 99.07%, 98.16%, 97.27%, and 97.71%, respectively.

Table 11: Confusion Matrix for HTTP Response Splitting

n=539		Prediction	
		No	Yes
Actual	No	427	02
	Yes	03	107

- **XPath Injection**Table 12 shows the confusion matrix for the non-pre-trained model on the XPath injection dataset where n=353 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=276, False Positive=03, False Negative=01, and True Positive=73. The accuracy, precision, recall, and f1-score of the dataset model are 98.86%, 97.33%, 98.64%, and 97.98%, respectively.

- **Broken Authentication**
  - **Plain-Text Storage of Credentials:**Table 13 shows the confusion matrix for the non-pre-trained model on the Plain-Text Storage of Credentials dataset where n=30 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=23, False Positive=01, False Negative=01, and True Positive=05. The accuracy, precision, recall, and

Table 12: Confusion Matrix for XPath Injection

n=353		**Prediction**	
		No	**Yes**
Actual	**No**	276	02
	Yes	01	73

f1-score of the dataset model are 93.33%, 83.33%, 83.33%, and 83.33%, respectively.

Table 13: Confusion Matrix for Plain-Text Storage of Credentials

n=30		**Prediction**	
		No	**Yes**
Actual	**No**	23	01
	Yes	01	05

  - **Hard-Coded Credentials** Table 14 shows the confusion matrix for the non-pre-trained model on the Hard-Coded Credentials dataset where n=88 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=23, False Positive=01, False Negative=01, and True Positive=05. The accuracy, precision, recall, and f1-score of the dataset model are 98.86%, 100%, 96%, and 97.95%, respectively.

Table 14: Confusion Matrix for Hard-Coded Credentials

n=88		**Prediction**	
		No	**Yes**
Actual	**No**	63	00
	Yes	01	24

- Sensitive Data ExposureTable 15 shows the confusion matrix for the non-pre-trained model on the Sensitive Data Exposure dataset where n=323 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=249, False Positive=06, False Negative=01, and True Positive=67. The accuracy, precision, recall, and f1-score of the dataset model are 97.83%, 91.78%, 98.52%, and 95.03%, respectively.
- Broken Access Control:
  - **Relative Path Traversal** Table 16 shows the confusion matrix for the non-pre-trained model on the Relative Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=183, False Positive=05, False Negative=00, and True Positive=74. The accuracy, precision, recall, and f1-score of the dataset model are 98.09%, 93.67%, 100%, and 96.73%, respectively.

Table 15: Confusion Matrix for Sensitive Data Exposure

n=323		**Prediction**	
		No	**Yes**
Actual	**No**	249	06
	Yes	01	67

Table 16: Confusion Matrix for Relative Path Traversal

n=262		**Prediction**	
		No	**Yes**
Actual	**No**	183	05
	Yes	00	74

- **Absolute Path Traversal** Table 17 shows the confusion matrix for the non-pre-trained model on the Absolute Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=187, False Positive=00, False Negative=02, and True Positive=73. The accuracy, precision, recall, and f1-score of the dataset model are 99.23%, 100%, 97.33%, and 98.64%, respectively.

Table 17: Confusion Matrix for Absolute Path Traversal

n=262		**Prediction**	
		No	**Yes**
Actual	**No**	187	00
	Yes	02	73

Multi-Class Classification

- Multiset-1: Table 18 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 99.02%, 99.02%, and 99.02% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 99.06%, 99.02%, and 99.03% respectively.

- Multiset-2: Table 19 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 98.37%, 98.37%, and 98.37% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 98.45%, 98.37%, and 98.39% respectively.

- Multiset-3:Table 20 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is LDAP Injection vulnerability, **Class-3** is PlainText Storage of a Password, **Class-4** is NULL Pointer

Table 18: Confusion Matrix for MultiSet-1

n=410		**Predicted Class**		
		Class-1	**Class-2**	**Class-3**
Actual Class	**Class-1**	313	01	03
	Class-2	00	25	00
	Class-3	00	00	68

Table 19: Confusion Matrix for MultiSet-2

n=493		**Predicted Class**		
		Class-1	**Class-2**	**Class-3**
Actual Class	**Class-1**	382	02	05
	Class-2	01	28	00
	Class-3	00	00	75

Dereference and **Class-5** is XPath Injection vulnerability. The model gave micro precision, micro recall, and micro f1-score of 95.91%, 95.91%, and 95.91% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 96.43%, 95.91%, and 95.77% respectively.

Table 20: Confusion Matrix for MultiSet-3

n=493		**Predicted Class**				
		Class-1	**Class-2**	**Class-3**	**Class-4**	**Class-5**
Actual Class	**Class-1**	600	00	00	00	00
	Class-2	00	30	00	00	00
	Class-3	00	00	70	04	00
	Class-4	02	00	26	46	00
	Class-5	00	00	00	00	06

- Multiset-4: Table 21 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is Relative Path Traversal vulnerability, **Class-3** is Relative Path Traversal vulnerability and **Class-4** is Cross-Site Scripting(XSS) vulnerability. The model gave micro precision, micro recall, and micro f1-score of 95.36%, 95.36%, and 95.36% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 95.93%, 95.36%, and 95.11% respectively.

Table 21: Confusion Matrix for MultiSet-4

n=493		Predicted Class			
		Class-1	Class-2	Class-3	Class-4
Actual Class	Class-1	368	00	02	02
	Class-2	00	12	13	02
	Class-3	00	01	32	04
	Class-4	00	00	01	102

Pre-Trained Model

As described in Section 4.3.1, pre-training is a 2-step process where first we perform node-level pre-training followed by graph-level pre-training. There are two techniques for node-level pre-training, namely, Attribute Masking and Context Prediction. In the following section, we first present evaluation results for the pre-trained model with attribute masking as the node-level pre-training technique followed by the pre-training model with context prediction as to the node-level pre-training technique. For node-level pre-training, we used the file-level graph dataset shown in Table 3. For graph-level pre-training, we used the OS Command Injection dataset shown in Table 3 for the supervised learning method.

Attribute Masking + Property Prediction

Binary Classification:

- Cross-Site ScriptingTable 22 shows the confusion matrix for the non-pre-trained model on the XSS dataset where n=785 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=562, False Positive=00, False Negative=07, and True Positive=216. The accuracy, precision, recall, and f1-score of the dataset model are 99.10%, 100%, 96.86%, and 98.40%, respectively.

Table 22: Confusion Matrix for Cross-Site Scripting

n=785		Prediction	
		No	Yes
Actual	No	562	00
	Yes	07	216

- Injection:
  - **SQL Injection:** Table 23 shows the confusion matrix for the non-pre-trained model on the SQL injection dataset where n=1761 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=1238, False Positive=11, False Negative=17, and True

Positive=495. The accuracy, precision, recall, and f1-score of the dataset model are 98.40%, 97.82%, 96.67%, and 97.24%, respectively.

Table 23: Confusion Matrix for SQL Injection

n=1761		**Prediction**	
		No	**Yes**
Actual	**No**	1238	11
	Yes	17	495

– **LDAP Injection:** Table 24 shows the confusion matrix for the non-pre-trained model on the LDAP injection dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=187, False Positive=01, False Negative=02, and True Positive=72. The accuracy, precision, recall, and f1-score of the dataset model are 98.85%, 98.63%, 97.29%, and 97.95%, respectively.

Table 24: Confusion Matrix for LDAP Injection

n=262		**Prediction**	
		No	**Yes**
Actual	**No**	188	00
	Yes	00	74

– **HTTP Response Splitting** Table 25 shows the confusion matrix for the non-pre-trained model on the HTTP Response Splitting dataset where n=539 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=429, False Positive=00, False Negative=03, and True Positive=107. The accuracy, precision, recall, and f1-score of the dataset model are 99.44%, 100%, 97.27%, and 98.61%, respectively.

Table 25: Confusion Matrix for HTTP Response Splitting

n=539		**Prediction**	
		No	**Yes**
Actual	**No**	429	00
	Yes	03	107

– **XPath Injection:**Table tab:XPathInjectionMask shows the confusion matrix for the non-pre-trained model on the XPath injection dataset where n=353 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=277, False Positive=01, False Negative=00, and True Positive=74. The accuracy, precision, recall, and f1-score of the dataset model are 99.71%, 98.66%, 100%, and 99.32%, respectively.

- Broken Authentication

Table 26: Confusion Matrix for XPath Injection

n=353		**Prediction**	
		No	**Yes**
Actual	**No**	277	01
	Yes	00	74

- **Plain-Text Storage of Credentials:** Table 27 shows the confusion matrix for the non-pre-trained model on the Plain-Text Storage of Credentials dataset where n=30 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=23, False Positive=01, False Negative=01, and True Positive=05. The accuracy, precision, recall, and f1-score of the dataset model are 93.33%, 83.33%, 83.33%, and 83.33%, respectively.

Table 27: Confusion Matrix for Plain-Text Storage of Credentials

n=30		**Prediction**	
		No	**Yes**
Actual	**No**	23	01
	Yes	01	05

- **Hard-Coded Credentials:** Table 28 shows the confusion matrix for the non-pre-trained model on the Hard-Coded Credentials dataset where n=88 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=63, False Positive=00, False Negative=01, and True Positive=24. The accuracy, precision, recall, and f1-score of the dataset model are 98.86%, 100%, 96%, and 97.95%, respectively.

Table 28: Confusion Matrix for Hard-Coded Credentials

n=88		**Prediction**	
		No	**Yes**
Actual	**No**	63	00
	Yes	01	24

- Sensitive Data Exposure Table 29 shows the confusion matrix for the non-pre-trained model on the Sensitive Data Exposure dataset where n=323 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=252, False Positive=03, False Negative=01, and True Positive=67. The accuracy, precision, recall, and f1-score of the dataset model are 98.76%, 95.71%, 98.52%, and 97.09%, respectively.
- Broken Access Control

Table 29: Confusion Matrix for Sensitive Data Exposure

n=323		Prediction	
		No	Yes
Actual	No	252	03
	Yes	01	67

– **Relative Path Traversal** Table 30 shows the confusion matrix for the non-pre-trained model on the Relative Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=188, False Positive=00, False Negative=01, and True Positive=73. The accuracy, precision, recall, and f1-score of the dataset model are 99.61%, 100%, 98.64%, and 99.31%, respectively.

Table 30: Confusion Matrix for Relative Path Traversal

n=262		Prediction	
		No	Yes
Actual	No	188	00
	Yes	01	73

– **Absolute Path Traversal:** Table 31 shows the confusion matrix for the non-pre-trained model on the Absolute Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=187, False Positive=00, False Negative=02, and True Positive=73. The accuracy, precision, recall, and f1-score of the dataset model are 99.23%, 100%, 97.33%, and 98.64%, respectively.

Table 31: Confusion Matrix for Absolute Path Traversal

n=262		Prediction	
		No	Yes
Actual	No	187	00
	Yes	02	73

Multi-Class Classification

- Multiset-1: Table 32 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 99.02%, 99.02%, and 99.02% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 99.06%, 99.02%, and 99.03% respectively.
- Multiset-2:Table 33 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded

Table 32: Confusion Matrix for MultiSet-1

n=410		Predicted Class		
		Class-1	Class-2	Class-3
Actual Class	Class-1	313	01	03
	Class-2	00	25	00
	Class-3	00	00	68

credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 99.39%, 99.39%, and 99.39% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 99.40%, 99.39%, and 99.38% respectively.

Table 33: Confusion Matrix for MultiSet-2

n=493		Predicted Class		
		Class-1	Class-2	Class-3
Actual Class	Class-1	388	00	01
	Class-2	01	27	01
	Class-3	00	00	75

- Multiset-3: Table 34 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is LDAP Injection vulnerability, **Class-3** is PlainText Storage of a Password, **Class-4** is NULL Pointer Dereference and **Class-5** is XPath Injection vulnerability. The model gave micro precision, micro recall, and micro f1-score of 96.17%, 96.17%, and 96.17% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 97.05%, 96.17%, and 96.04% respectively.

Table 34: Confusion Matrix for MultiSet-3

n=493		Predicted Class				
		Class-1	Class-2	Class-3	Class-4	Class-5
Actual Class	Class-1	599	01	00	00	00
	Class-2	00	30	00	00	00
	Class-3	00	00	73	01	00
	Class-4	00	00	28	46	00
	Class-5	00	00	00	00	06

- Multiset-4: Table 35 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is Relative Path Traversal vulnerability, **Class-3** is Relative Path Traversal vulnerability and **Class-4** is Cross-Site Scripting(XSS) vulnerability. The model gave micro precision, micro recall, and micro f1-score of 95.36%, 95.36%, and 95.36% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 95.93%, 95.36%, and 95.11% respectively.

Table 35: Confusion Matrix for MultiSet-4

n=493		Predicted Class			
		Class-1	Class-2	Class-3	Class-4
Actual Class	Class-1	368	00	02	02
	Class-2	00	12	13	02
	Class-3	00	01	32	04
	Class-4	00	00	01	102

Context Prediction + Property Prediction

Binary Classification:

- Cross-Site Scripting Table 36 shows the confusion matrix for the non-pre-trained model on the XSS dataset where n=755 refers to the total number of test data points, actual class refers to the True class distribution, and prediction class refers to the Predicted Class Distribution. The model produces True Negative=562, False Positive=00, False Negative=07, and True Positive=216. The accuracy, precision, recall, and f1-score of the dataset model are 99.10%, 100%, 96.86%, and 98.40%, respectively.

Table 36: Confusion Matrix for Cross-Site Scripting

n=755		Prediction	
		No	Yes
Actual	No	562	00
	Yes	07	216

- Injection
  - **SQL Injection:** Table 37 shows the confusion matrix for the non-pre-trained model on the SQL injection dataset where n=1761 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=1238, False Positive=11, False Negative=17, and True Positive=495. The accuracy, precision, recall, and f1-score of the dataset model are 98.40%, 97.82%, 96.67%, and 97.24%, respectively.

Table 37: Confusion Matrix for SQL Injection

n=1761		Prediction	
		No	Yes
Actual	No	1238	11
	Yes	17	495

  - **LDAP Injection:** Table 38 shows the confusion matrix for the non-pre-trained model on the LDAP injection dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution,

and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=187, False Positive=01, False Negative=02, and True Positive=72. The accuracy, precision, recall, and f1-score of the dataset model are 98.85%, 98.63%, 97.29%, and 97.95%, respectively.

Table 38: Confusion Matrix for LDAP Injection

n=262		**Prediction**	
		No	**Yes**
Actual	**No**	188	00
	Yes	00	74

– **HTTP Response Splitting:** Table 39 shows the confusion matrix for the non-pre-trained model on the HTTP Response Splitting dataset where n=539 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=427, False Positive=02, False Negative=00, and True Positive=110. The accuracy, precision, recall, and f1-score of the dataset model are 99.62%, 98.21%, 100%, and 99.09%, respectively.

Table 39: Confusion Matrix for HTTP Response Splitting

n=539		**Prediction**	
		No	**Yes**
Actual	**No**	427	02
	Yes	00	110

– **XPath Injection:** Table 40 shows the confusion matrix for the non-pre-trained model on the XPath injection dataset where n=353 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=277, False Positive=01, False Negative=00, and True Positive=74. The accuracy, precision, recall, and f1-score of the dataset model are 99.71%, 98.66%, 100%, and 99.32%, respectively.

Table 40: Confusion Matrix for XPath Injection

n=353		**Prediction**	
		No	**Yes**
Actual	**No**	277	01
	Yes	00	74

- Broken Authentication

  – **Plain-Text Storage of Credentials:** Table 41 shows the confusion matrix for the non-pre-trained model on the Plain-Text Storage of Credentials dataset where n=30 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted

Class Distribution. The model produces True Negative=24, False Positive=00, False Negative=01, and True Positive=05. The accuracy, precision, recall, and f1-score of the dataset model are 96.66%, 100%, 83.33%, and 90.90%, respectively.

Table 41: Confusion Matrix for Plain-Text Storage of Credentials

n=30		**Prediction**	
		No	**Yes**
Actual	**No**	24	00
	Yes	01	05

- **Hard-Coded Credentials:** Table 42 shows the confusion matrix for the non-pre-trained model on the Hard-Coded Credentials dataset where n=88 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=63, False Positive=00, False Negative=01, and True Positive=24. The accuracy, precision, recall, and f1-score of the dataset model are 98.86%, 100%, 96%, and 97.95%, respectively.

Table 42: Confusion Matrix for Hard-Coded Credentials

n=88		**Prediction**	
		No	**Yes**
Actual	**No**	63	00
	Yes	01	24

- Sensitive Data Exposure Table 43 shows the confusion matrix for the non-pre-trained model on the Sensitive Data Exposure dataset where n=323 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=252, False Positive=03, False Negative=01, and True Positive=67. The accuracy, precision, recall, and f1-score of the dataset model are 98.76%, 95.71%, 98.52%, and 97.09%, respectively.

Table 43: Confusion Matrix for Sensitive Data Exposure

n=323		**Prediction**	
		No	**Yes**
Actual	**No**	252	03
	Yes	01	67

- Broken Access Control
  - **Relative Path Traversal:** Table 44 shows the confusion matrix for the non-pre-trained model on the Relative Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=183, False Positive=05, False

Negative=00, and True Positive=74. The accuracy, precision, recall, and f1-score of the dataset model are 98.09%, 93.67%, 100%, and 96.73%, respectively.

Table 44: Confusion Matrix for Relative Path Traversal

n=262		Prediction	
		No	Yes
Actual	No	183	05
	Yes	00	74

- **Absolute Path Traversal:** Table 45 shows the confusion matrix for the non-pre-trained model on the Absolute Path Traversal dataset where n=262 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=184, False Positive=03, False Negative=00, and True Positive=75. The accuracy, precision, recall, and f1-score of the dataset model are 98.85%, 96.15%, 100%, and 98.03%, respectively.

Table 45: Confusion Matrix for Absolute Path Traversal

n=262		Prediction	
		No	Yes
Actual	No	184	03
	Yes	00	75

Multi-Class Classification

- Multiset-1: Table 46 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 99.02%, 99.02%, and 99.02% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 99.06%, 99.02%, and 99.03% respectively.

Table 46: Confusion Matrix for MultiSet-1

n=410		Predicted Class		
		Class-1	Class-2	Class-3
Actual Class	Class-1	313	01	03
	Class-2	00	25	00
	Class-3	00	00	68

- Multiset-2: Table 47 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerability class, **Class-2** is Hard-Coded credentials vulnerability, and **Class-3** is Missing Encryption of Sensitive Data vulnerability. The model gave micro precision, micro recall, and micro f1-score of 99.39%,

99.39%, and 99.39% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 99.40%, 99.39%, and 99.38% respectively.

Table 47: Confusion Matrix for MultiSet-2

n=493		Predicted Class		
		Class-1	Class-2	Class-3
Actual Class	Class-1	388	00	01
	Class-2	01	27	01
	Class-3	00	00	75

- Multiset-3: Table 48 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is LDAP Injection vulnerability, **Class-3** is PlainText Storage of a Password, **Class-4** is NULL Pointer Dereference and **Class-5** is XPath Injection vulnerability. The model gave micro precision, micro recall, and micro f1-score of 96.68%, 96.68%, and 96.68% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 96.91%, 96.68%, and 96.65% respectively.

Table 48: Confusion Matrix for MultiSet-3

n=493		Predicted Class				
		Class-1	Class-2	Class-3	Class-4	Class-5
Actual Class	Class-1	600	00	00	00	00
	Class-2	00	30	00	00	00
	Class-3	00	00	68	06	00
	Class-4	00	00	20	54	00
	Class-5	00	00	00	00	06

- Multiset-4: Table 49 shows the confusion matrix for the vulnerability dataset Multiset-1 shown in Table 7 where **Class-1** is Non-Vulnerable class, **Class-2** is Relative Path Traversal vulnerability, **Class-3** is Relative Path Traversal vulnerability and **Class-4** is Cross-Site Scripting(XSS) vulnerability. The model gave micro precision, micro recall, and micro f1-score of 95.54%, 95.54%, and 95.54% respectively. The model gave weighted precision, weighted recall , and weighted f1-score of 96.04%, 95.54%, and 95.27% respectively.

Table 49: Confusion Matrix for MultiSet-4

n=493		Predicted Class			
		Class-1	Class-2	Class-3	Class-4
Actual Class	Class-1	368	00	02	02
	Class-2	00	12	13	02
	Class-3	00	01	32	04
	Class-4	00	00	00	103

Result Summary

Table 50 shows the evaluation results of the non-pre-trained GNN model and the two pre-trained GNN models for the binary classification tasks. We took the average of the evaluation metrics to compare the three models. The non-pre-trained GNN model outputs an average precision, recall, and F1-score of 95.89%, 96.13%, and 95.83%, respectively. The GNN model pre-trained using attribute masking technique for node-level pretraining and supervised learning for graph-level pretraining gave an average precision, recall, and F1-score of 97.21%, 96.19%, and 96.78%. The GNN model pre-trained using the context prediction technique and supervised learning for graph-level pretraining gave an average precision, recall, and F-1 score of 97.90%, 96.86%, and 97.27%.

In the first pre-trained model, we performed node-level pretraining using the attribute masking technique described in Section 4.3.1 to learn the local representation of the graph attributes. The supervised learning using the OS command injection dataset for graph-level representation learning follows the node-level technique. This approach improved the precision, recall performance of the GNN model by 1.32% and 0.06%, respectively.

In the second pre-trained model, we performed node-level pretraining using the context prediction technique described in Section 4.3.1 to learn the local representation of the graph attributes. The supervised learning using the OS command injection dataset for graph-level representation learning follows the node-level pretraining technique. This approach improved the precision, recall performance of the GNN model by 2.01% and 0.73%, respectively.

Table 50: Summary of Evaluation results for Binary Classification task for the non-pretrained model and the pre-trained model

CWE		GNN MODEL								
ID	**NAME**	**Non Pre-Trained**			**Pre-Trained (Attribute Masking+ Supervised Learning)**			**Pre-Trained (Context Prediction+ Supervised**		
A1	**Injection**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
89	SQL Injection	97.02	97.02	96.25	97.82	96.67	97.24	97.82	96.67	97.24
90	LDAP Injection	98.63	97.29	97.95	98.63	97.29	97.95	98.63	97.29	97.95
113	HTTP Response Splitting	98.16	97.27	97.71	100	97.27	98.61	98.21	100	99.09
643	XPath Injection	97.33	98.64	97.98	98.66	100	99.32	98.86	100	99.32
A2	**Broken Authentication**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
256	Plain-Text Storage of Password	83.33	83.33	83.33	83.33	83.33	83.33	100	83.33	90.90
259,321	Hard Coded Password, Hard Coded Cryptographic Key	100	96	97.95	100	96	97.95	100	96	97.95
A3	**Sensitive Data Exposure**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
315,319	Clear-Text Sensitive Info in Cookie, Sensitive Clear-Text Transmission	91.78	98.52	95.03	95.71	98.52	97.09	95.71	98.52	97.09
A5	**Broken Access Control**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
23	Relative Path Traversal	93.67	100	96.73	100	98.64	99.31	93.67	100	96.73
36	Absolute Path Traversal	100	97.33	98.64	100	97.33	98.64	96.15	100	98.03
A7	**Cross-Site Scripting**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
80,81,83	Basic XSS, XSS Error Message, XSS Attribute	99.07	95.96	97.49	100	96.86	98.40	100	96.86	98.40
Average Metric Scores		**Average Precision (%)**	**Average Recall (%)**	**Average F1-Score (%)**	**Average Precision (%)**	**Average Recall (%)**	**Average F1-Score (%)**	**Average Precision (%)**	**Average Recall (%)**	**Average F1-Score (%)**
		95.89	96.13	95.83	97.21	96.19	96.78	97.90	96.86	97.27

Table 51 shows the evaluation results of the GNN model without pre-training and the two pre-trained GNN models for the multi-class classification tasks. Similar to the binary classification task, we took the evaluation metrics' average to compare the three models. The average micro precision, micro recall, and micro f1-score of the GNN model without pre-training were 97.16%, 97.16%, and 97.16%. Additionally, the average weighted precision, weighted recall, and weighted f1-score of the model were 97.16%, 97.16%, and 97.07%. The average micro precision, micro recall, and micro f1-score of the GNN model pre-trained using the attribute masking technique for node-level, and graph-level pre-training using supervised learning were 97.43%, 97.43%, and 97.43%. Additionally, the average weighted precision, weighted recall, and weighted f1-score of the model were 97.86%, 97.48%, and 97.39%. The average micro precision, micro recall, and micro f1-score of the GNN model pre-trained using the context prediction technique for node-level, and graph-level pre-training using supervised learning were 97.65%, 97.65%, and 97.65%. Additionally, the average weighted precision, weighted recall, and weighted f1-score of the model were 97.85%, 97.65%, and 97.58%.

In the first GNN model, we used the attribute masking technique for node-level pre-training and supervised learning on the OS command injection dataset for graph-level pre-training. 0.27% improved the micro-precision and recall score, and 0.7% and 0.29% respectively improved the weighted precision and recall.

In the second GNN model, we used the context prediction technique for node-level pre-training and supervised learning on the OS command injection dataset for graph-level pre-training, micro-precision, and recall 0.49% improved score. 0.69% and 0.49% respectively improved the weighted precision and recall.

Table 51: Summary of Evaluation results for Multi-Class classification task for the non-pretrained model and two pretrained model

CWE		GNN MODEL																	
ID	Name	Non Pre-Trained						Pre-Trained (Attribute Masking+ Property Prediction)						Pre-Trained (Context Prediction + Property Prediction)					
Set 1	**Multiset-1**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**
259,321, 315,319	Hard-Coded Credentials, Missing Encryption of Sensitive Data	99.02	99.02	99.02	99.06	99.02	99.03	99.02	99.02	99.02	99.06	99.02	99.03	99.02	99.02	99.02	99.06	99.02	99.03
Set 2	**Multiset-2**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**
476,643	NULL Pointer Dereference, XPath Injection	98.37	98.37	98.37	98.45	98.37	98.39	99.39	99.39	99.39	99.40	99.39	99.38	99.39	99.39	99.39	99.40	99.39	99.38
Set 3	**Multiset-3**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**
90,256,476, 643	LDAP Injection, PlainText Storage of a Password, NULL Pointer Dereference, XPath Injection	95.91	95.91	95.91	96.43	95.91	95.77	96.17	96.17	96.17	97.05	96.17	96.04	96.68	96.68	96.68	96.91	96.68	96.65
Set 4	**Multiset-4**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**	**Micro Precision (%)**	**Micro Recall (%)**	**Micro F1-Score (%)**	**Weighted Precision (%)**	**Weighted Recall (%)**	**Weighted F1-Score (%)**
23,36,80,81,83	Relative Path Traversal, Absolute Path Traversal, Cross-Site Scripting	95.36	95.36	95.36	95.93	95.36	95.11	95.36	95.36	95.36	95.93	95.36	95.11	95.54	95.54	95.54	96.04	95.54	95.27
Average Scores		**Average Micro Precision (%)**	**Average Micro Recall (%)**	**Average Micro F1-Score (%)**	**Average Weighted Precision (%)**	**Average Weighted Recall (%)**	**Average Weighted F1-Score (%)**	**Average Micro Precision (%)**	**Average Micro Recall (%)**	**Average Micro F1-Score (%)**	**Average Weighted Precision (%)**	**Average Weighted Recall (%)**	**Average Weighted F1-Score (%)**	**Average Micro Precision (%)**	**Average Micro Recall (%)**	**Average Micro F1-Score (%)**	**Average Weighted Precision (%)**	**Average Weighted Recall (%)**	**Average Weighted F1-Score (%)**
		97.16	97.16	97.16	97.16	97.16	97.07	97.43	97.43	97.43	97.86	97.48	97.39	97.65	97.65	97.65	97.85	97.65	97.58

5.3.2 Comparison with Devign Model

To evaluate and validate our model's efficiency in the absence of a benchmark dataset and model, we utilized the model used in the previous research work to compare the efficiency of our model. The model used in the previous research works uses the Gated Graph Sequence Neural Networks for learning the global representation and a module called 'Conv' for classification. Since there were no datasets or model implementation available, we implemented the model based on the research paper's details to perform the comparison. We used the Pytorch-Geometric library for the implementation of the Gated Graph Sequence Neural Network used for the representation learning phase and the Conv module, which was a fully connected layer of 1-D convolution layer with max-pooling defined by the equation 43.

$$\sigma(.) = MAXPOOL(ReLu(CONV(.))) \tag{43}$$

In the Conv module, first traditional 1-D convolutional and dense layer are applied on the concatenation of the final node representation and the initial node representation state $[H_i^{(T)}, x_i]$ and the final node feature $H_i^{(T)}$ respectively. Followed by this, the two outputs undergo pairwise multiplication, and then the average of the resulted vector is taken to make a prediction.

Binary Classification

- Cross-Site Scripting: Table 52 shows the confusion matrix for the devign model on the Cross-Site Scripting dataset where n=785 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=550, False Positive=12, False Negative=20, and True Positive=203. The accuracy, precision, recall, and f1-score of the dataset model are 95.92%, 94.41%, 91.03%, and 92.68%, respectively.

Table 52: Confusion Matrix for Cross-Site Scripting Vulnerability

n=785		**Predicted Class**	
		Class-1	**Class-2**
Actual Class	**Class-1**	550	12
	Class-2	20	203

- Injection
  - **SQL Injection:** Table 53 shows the confusion matrix for the Devign model on the SQL injection dataset where n=1761 refers to the total number of test data points, actual class refers to the True class distribution, and prediction class refers to the Predicted Class Distribution. The model produces True Negative=1230, False Positive=19, False Negative=22, and True Positive=490. The accuracy, precision, recall, and f1-score of the dataset model are 98.85%, 98.63%, 97.29%, and 97.95%, respectively.

Table 53: Confusion Matrix for SQL Injection Vulnerability

n=1761		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	1230	19
	Class-2	22	490

- **LDAP Injection:** Table 54 shows the confusion matrix for the Devign model on the LDAP injection dataset where n=262 refers to the total number of test data points, actual class refers to the True class distribution, and prediction class refers to the Predicted Class Distribution. The model produces True Negative=180, False Positive=08, False Negative=09, and True Positive=65. The accuracy, precision, recall, and f1-score of the dataset model are 93.51%, 89.04%, 87.83%, and 88.43%, respectively.

Table 54: Confusion Matrix for LDAP Injection Vulnerability

n=262		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	180	08
	Class-2	09	65

- **XPath Injection:** Table 55 shows the confusion matrix for the Devign model on the XPath injection dataset where n=352 refers to the total number of test data points, actual class refers to the True class distribution, and prediction class refers to the Predicted Class Distribution. The model produces True Negative=270, False Positive=08, False Negative=03, and True Positive=71. The accuracy, precision, recall, and f1-score of the dataset model are 96.87%, 89.87%, 95.94%, and 92.80%, respectively.

Table 55: Confusion Matrix for XPath injection Vulnerability

n=352		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	270	08
	Class-2	03	71

- Broken Authentication
  - **Plain-Text Storage of Credentials:** Table 56 shows the confusion matrix for the Devign model on the Plain-Text Storage of Credentials dataset where n=30 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=21, False Positive=03, False Negative=01, and True Positive=05. The accuracy, precision, recall, and f1-score of the dataset model are 86.66%, 62.50%, 83.83%, and 71.42%, respectively.

Table 56: Confusion Matrix for Plain-Text Storage of Credential Vulnerability

n=262		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	21	03
	Class-2	01	65

- **Hard-Coded Credentials:**Table 57 shows the confusion matrix for the Devign model on the Hard-Coded Credentials dataset where n=88 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=60, False Positive=03, False Negative=05, and True Positive=20. The accuracy, precision, recall, and f1-score of the dataset model are 90.90%, 86.95%, 80%, and 83.33%, respectively.

Table 57: Confusion Matrix for Hard-Coded Credential Vulnerability

n=88		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	60	03
	Class-2	05	20

- Sensitive Data Exposure Table 58 shows the confusion matrix for the Devign model on the Missing Encryption of Sensitive Data dataset where n=323 refers to the total number of test data points. The actual class refers to the True class distribution, and the prediction class refers to the Predicted Class Distribution. The model produces True Negative=249, False Positive=06, False Negative=04, and True Positive=64. The accuracy, precision, recall, and f1-score of the dataset model are 96.90%, 91.42%, 94.11%, and 92.74%, respectively.

Table 58: Confusion Matrix for Missing Encryption of Sensitive Data Vulnerability

n=323		Predicted Class	
		Class-1	Class-2
Actual Class	Class-1	249	06
	Class-2	04	64

Result Summary

Table 59 shows the comparison of the evaluation results of the pre-trained models and the previous research model [74] [58]. To compare the two pre-trained models and the

Table 59: Comparison of Evaluation results of the pre-trained model and previous research model(Devign)

CWE		GNN MODEL								
ID	NAME	PRE-TRAINED (Attribute Masking+ Supervised Learning)			PRE-TRAINED (Context Prediction+ Supervised Learning)			DEVIGN MODEL		
A1	**Injection**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**
89	SQL Injection	97.82	96.67	97.24	97.82	96.67	97.24	98.63	97.29	97.95
90	LDAP Injection	98.63	97.29	97.95	98.63	97.29	97.95	89.04	87.83	88.43
643	XPath Injection	98.66	100	99.32	98.86	100	99.32	89.87	95.94	92.80
A2	**Broken Authentication**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**
256	Plain-Text Storage of Password	83.33	83.33	83.33	100	83.33	90.90	62.50	83.83	71.42
259,321	Hard Coded Password, Hard Coded Cryptographic Key	100	96	97.95	100	96	97.95	86.95	80	83.83
A3	**Sensitive Data Exposure**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**
315,319	ClearText Sensitive Info in Cookie, Sensitive Cleartext Transmission	95.71	98.52	97.09	95.71	98.52	97.09	91.42	94.11	92.74
A7	**Cross-Site Scripting**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**	**Precision(%)**	**Recall(%)**	**F1-score(%)**
80,81,83	Basic XSS, XSS Error Message, XSS Attribute	100	96.86	98.40	100	96.86	98.40	95.92	91.03	92.68
Average Score		**Average Precision(%)**	**Average Recall(%)**	**Average F1-score(%)**	**Average Precision(%)**	**Average Recall(%)**	**Average F1-score(%)**	**Average Precision(%)**	**Average Recall(%)**	**Average F1-score(%)**
		96.21	95.52	95.89	98.71	95.47	96.97	87.68	90.04	88.55

previous research model, we considered the evaluation metrics' mean. Table 59 shows that the mean precision, recall, and f1-score of the GNN model pre-trained using the attribute masking technique for node-level pre-training and supervised learning for graph-level pre-training were 96.21%, 95.52%, and 95.89%. The mean precision, recall, and f1-score of the GNN model pre-trained using context prediction technique for node-level pre-training and supervised learning for graph-level pre-training were 98.71%, 95.47%, and 96.97%. Whereas the mean precision, recall, and f1-score of the previous research model were 87.88%, 90.04%, and 88.55%.

In summary, our GNN model, when pre-trained with attribute masking strategy for node-level pre-training and supervised learning for graph-level pre-training, outperforms the previous research model and provides 5.48% and 8.33% improvement in recall and precision score respectively. Thus, our model can detect 5.48% more vulnerabilities with 8.33% more precision than the previous research model. Similarly, our GNN model, when pre-trained with context prediction strategy for node-level pre-training and supervised learning for graph-level pre-training, outperforms the previous research model and provides 5.43% and 10.83% improvement in recall and precision score respectively. Thus, our model can detect 5.43% more vulnerabilities with 10.83% more precision than the previous research model.

5.3.3 Comparison with Static Tools

Table 60 shows the comparison of evaluation results of the pre-trained models with static analysis tools ESVD, SpotBugs, and Find Security Bugs with improved proof of concept to address the limitations of the SATs. The static analysis tool that operates at the source code level, ESVD, does not cover all the vulnerabilities. The vulnerabilities covered by the analyzer provide low recall with a high precision score that states that the analyzer has a low detection rate but high precision. For example, from the Table 60 it is seen that ESVD can detect the only 22% of the SQL injection defect in the Juliet Test Suite but with 100% precision. Similarly, SpotBugs that operate at the Java ByteCode level also fails to cover all the vulnerabilities but produces a better recall score than ESVD. Among all the three SATs, FindSecBugs outputs the best evaluation results. Hence, we compared FindSecBugs with our two pre-trained GNN models.

Table 60: Comparison of Evaluation results of the pre-trained model and Static Analysis Tools(SATs)

CWE		GNN MODEL						TOOLS								
ID	**NAME**	**PRE-TRAINED (Attribute Masking+ Supervised Learning)**			**PRE-TRAINED (Context Prediction+ Supervised Learning)**			**ESVD**			**SpotBugs**			**FindSecBugs**		
A1	**Injection**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
89	SQL Injection	97.82	96.67	97.24	97.82	96.67	97.24	100	22	36.06	70	84	76.36	100	89	94.17
90	LDAP Injection	98.63	97.29	97.95	98.63	97.29	97.95	100	19	31.93	0	0	0	100	89	94.17
113	HTTP Response Splitting	100	97.27	98.61	98.21	100	99.09	100	19	31.93	100	47	63.94	100	89	94.17
643	XPath Injection	98.66	100	99.32	98.86	100	99.32	0	0	0	0	0	0	100	89	94.17
A2	**Broken Authentication**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
259	Hard Coded Password	100	95	97.43	100	95	97.43	87	18	32.96	100	14	24	100	43	30.06
321	Hard-Coded Cryptographic Key	100	83.33	90.90	100	83.33	90.90	0	0	0	0	0	0	100	43	30.06
A5	**Broken Access Control**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
23	Relative Path Traversal	100	98.64	99.31	93.67	100	96.73	100	19	31.93	100	47	63.94	88	100	93.61
36	Absolute Path Traversal	100	97.33	98.64	96.15	100	98.03	100	19	31.93	100	47	63.94	88	100	93.61
A7	**Cross-Site Scripting**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
80	Basic XSS	100	92.85	96.29	100	92.85	96.29	100	19	31.93	100	46	63.01	100	89	94.17
81	XSS Error Message	100	96.36	98.14	100	96.36	98.14	100	19	31.93	100	46	63.01	100	89	94.17
83	XSS Attribute	98.24	100	99.11	98.24	100	99.11	100	19	31.93	100	46	63.01	100	89	94.17

We compared the SAT and the GNN models for each vulnerability because they use a separate detector. For injection vulnerabilities, the recall score, i.e., vulnerability detection rate of the GNN models, is approx 8.80% better than the SAT FindSecBugs. Similarly, for Broken Authentication and Cross-Site Scripting vulnerabilities, the GNN model outputs a better vulnerability detection rate than FindSecBugs. For Broken Access Control vulnerability, the SAT FindSecBugs outperforms the GNN model in vulnerability detection rate by approx 2% but with 12% lower precision compared to the GNN models only after adding additional tainted sources. In summary, our GNN model outperforms the Static Analysis Tools even after addressing the SATs' limitations by improving the proof of concept.

5.4 Evaluation Summary

As mentioned in Section 4, our implementation factorizes into four modules: Data selection, Pre-Processing of the dataset, Representation Learning, and Classification and uses various components in these modules based on the task as shown in Table 61.

Table 61: Components used in the implementation of our approach

Module	Task	Component Used
Data Pre-Processing	Intermediate Graphical Representation	JCPG Tool
	Attribute Masking	Word2Vec and Skilearn
Representation Learning and Classification	GNN Model	Graph Isomorphism Network model
	Transfer Learning	SNAP pre-training strategies(Node-level + Graph-Level): Attribute Masking + Property Prediction Context Prediction + Property Prediction

In this section, we present the summary of the evaluation results for each component evaluated in Section 5. In section 5.2.1, we evaluated the JCPG tool that we developed to generate the code property graph from the source code. We showcased that the tool works for all java constructs by performing experiments using various source codes. Table 4 shows some of the source codes. Then we showcased that the tool met the following requirements: generation of file-level and method-level code property graph, generation of CPG for compilable and non-compilable source code files, and generate the output in various formats like DOT, JSON, and GML to be used for further processing. In section 5.2.2, we showcased that the Word2Vec model learns the semantic meaning of the graph attributes and outputs embedding for the attributes where the attribute with similar semantic meaning cluster together. In section 5.3, we evaluated our model in two steps. First, we evaluated the non-pre-trained model and showcased that our model without pre-training outputs an average precision, recall, and f1-score of 95.89%, 96.13%, and 95.83% for binary classification and the mean micro precision, micro recall, and micro f1-score of 97.16%, 97.16%, and 97.16% for multi-class classification. This result states that without pre-training, our model can detect 96.13% of the vulnerabilities with a precision of 95.89% in binary classification tasks and can detect 97.16% of the vulnerabilities with a precision of 97.16% in multi-class classification. Then we evaluated the model with pre-training and showcased that transfer learning improves the model's performance. Hence, the same model can extend for new vulnerabilities without causing a negative transfer effect. In section 5.3.2, we compared our model with the model used in the previous research [58] [74] and showcased that our model outperforms the previous models and provides 5.48% and 8.33% improvement in recall and precision score respectively. Finally, in section 5.3.3, we compared our model with the static analysis tools that were improved using the proof of concepts [11]. It showcased that our model outperformed the static analysis tool even after addressing the static analysis tools' limitations by improving the proof of concepts.

6 Discussion

In this chapter, we present a short discussion about the evaluation results presented in Section 5. As described in Section 4, in our approach, we first take the source code file and input it to the JCPG tool, which outputs CPGs for each method in the source code. We then feed the CPGs to the GNN model that performs method-level vulnerability prediction. The vulnerability prediction tool helps the developers and code reviewers to perform the preliminary checks for software vulnerabilities and focus their attention on the specific source code file with potential vulnerabilities even when the source code is non-compilable. In the following section, we will discuss the results for each component of the implementation architecture.

Dataset: The previous research for vulnerability detection using Graph Neural Networks was on C-language. Hence, multiple vulnerability datasets are available to be used as benchmark datasets for evaluating the GNN model, which is not the case with Java vulnerabilities. Hence due to the lack of an attested dataset for Java, as seen in Section 4, we used an artificial dataset Juliet Test Suite used for the evaluation of static tools to evaluate our model. Compare its performance with previous research models and static tools. The Juliet Test Suite dataset is an artificial vulnerability dataset where the test-cases are curated with vulnerabilities. Hence, it fails to capture all the constructs for each vulnerability category, and the test cases for a single vulnerability category have a similar method signature. Additionally, the dataset has an imbalanced class distribution. Hence to retain the same class distribution for training, validation, and test dataset, we used the Stratified K-Fold cross-validation to split the dataset instead of a random dataset. Additionally, to validate the model's robustness, we used a vulnerability dataset with varied sample sizes from the Juliet Test Suite dataset.

JCPG: The first step of the pre-processing stage was to create the code property graph for the Java source code. The tools available for creating code property graphs for Java source code worked on the ByteCode, requiring compiling the source code. However, we are required to find the source code's vulnerability even when the source code is non-compilable due to missing supporting files. We developed a tool called the JCPG tool based on ANLTR and Java Graph library operating at the source code level to alleviate this. We first evaluated the tool for the introductory Java constructs. The tool can create a code property graph for the source code with all Java's basic constructs except the enumerator. Similarly, to evaluate the tool for non-compilable source codes and generation of method-level code property graphs, we used various projects from GIT repositories Table 5 shows a few of the GIT repositories we use. The tool was able to successfully generate code property graphs for the selected source codes and export the output in three formats, namely DOT, JSON, and GML that provides the ability to use the output for further use.

GNN model: We use a modified Graph Isomorphism Model that includes the edge features to calculate the final global graph representation in the representation learning stage. In our approach, we first pre-train the GNN model using the pre-training strategy introduced by Hu et.al. [?] and then use the pre-trained GNN model for the downstream tasks. As shown in Section 5.3, to evaluate our implementation strategy, we first evaluated our implementation strategy without pre-training the model. We follow it by evaluating our implementation strategy with the pre-trained models for binary and multi-class classifica-

tion. We conclude that the transfer learning approach could be beneficial for vulnerability prediction, and pre-training the model at node-level to learn the local representation and graph-level to learn the global representation slightly improves the model's performance. We conclude that the tool could automatically learn the vulnerability patterns from the dataset by learning the global representation of the vulnerability patterns from the code property graph, unlike the static tools where the vulnerabilities are detected based on the rules and patterns defined manually, which triggers the bug. Our results show that our model can learn vulnerability patterns without manual feature generation for the vulnerability and identify if the source can better identify code contains a vulnerable method. Since the transfer learning strategy does not cause a negative transfer and the tool can automatically learn vulnerability patterns, we can extend the tool for new vulnerability patterns without affecting the tool's performance. Our results show that for binary classification tasks, our pre-trained model, on average, detects 96.86% of the vulnerabilities with 97.90% precision. In contrast, for the downstream task of multi-class classification, the average detects 97% of the vulnerabilities with 97.85% precision. These high recall and precision values for the model can correspond to the artificial vulnerability dataset without any noise. The model's performance might decrease for natural code samples with a wide variety of vulnerability constructs for a single vulnerability. We also validated our model's efficiency compared to the previously used model and concluded that our model outperformed the previous model. Our model was able to detect 5.48% more vulnerabilities with 8.33% more precision than the previous model. Finally, we also compared our model with the static analysis tools with improved proof of concept to address their limitations. We conclude from test results shown in Table 60 even the state of the static analysis tool FindSecBugs fails to outperform our model to detect vulnerabilities even after improving the proof of concept.

In conclusion, we can use our JCPG tool to produce an intermediate graphical representation combined with the highly expressive GIN Graph Neural Network as a vulnerability detection tool at the source code level. Finally, the only language-dependent component in the entire implementation architecture is the JCPG tool used to generate code property. The model can extend to programming languages that can parse using the ANTLR parser generator.

7 Limitation and Future Work

In this chapter, we present each component's limitations in the current approach and some ideas on to improve it.

7.1 Dataset

The dataset plays a crucial part, right from the dataset's training, to evaluate the model's efficiency and correctness. A biased or flawed dataset used for training the model can lead to misclassification in the validation and test phase. Previous research for vulnerability prediction was for vulnerabilities in the C/C++ programming language; hence, there was a lack of attested vulnerability dataset for Java programming language. Due to this, and since the manual classification and labeling of a dataset is a tedious and challenging task, we trained and evaluated the model using the Juliet Test Suite dataset used to evaluate static tools. The Juliet Test Suite is a synthetic vulnerability dataset with 118 separate Common Weakness Enumeration(CWE). The dataset does not cover all the vulnerabilities, and also it does not cover all the variations within the covered vulnerability categories. Hence the performance of the model for a real-world dataset with noise might decline to certain levels as shown in [58]. A dataset with a good mix of real-world and synthetic code could be a practical approach to train and validate the model to fortify the model's validity. Hence, future improvements would be to improve the datasets' quality by taking real-world and synthetic codes to train and evaluate the machine learning model. However, manually classifying and labeling the dataset is a highly tedious and expensive task. Hence, we can develop an improved approach to reduce labeling's manual task and curate a dataset.

7.2 Approach

The next part of implementation architecture is pre-processing the dataset, which requires a tool to transform a source code into an intermediate graphical representation and a tool to embed the graph attributes.

7.2.1 JCPG

The tools available to generate code property graphs for Java source code operate at the byte-code level, requiring compiling the source codes. In most cases, during code reviews, the source codes are non-compilable. The requirement was to have a tool that operates at the source code level and generates code property graphs even for non-compilable source codes. We developed a tool based on ANTLR and Java graph library that operates on the source code level to achieve this. The tool produced CPGs for file-level and method level for all basic java constructs except enumeration and interface. We would like to include these constructs for the tool. Additionally, we would like to add slicing to output program slices instead of a method-level code property graph to increase vulnerability prediction's granularity.

7.2.2 Word2Vec Model

To embed the graph attributes, we used the Word2Vec model that trains on a set of text data and produces embeddings for the words in the dataset, capturing each word's semantic meaning. Hence, the Word2Vec model fails to produce embedding for a word out of its vocabulary. The limitations can be alleviated using the FastText model, an extension of the Word2Vec model and treats each word as a composition of character N-grams [13]. The N-gram features provide the Out-of-Vocabulary issue solution providing the FastText model ability to produce embedding for unseen words.

7.2.3 GNN and Classifier

The last part of the implementation architecture is the Graph Neural Network model and the classifier. We used the modified Graph Isomorphism Network to learn the global representation of the intermediate graphical representation, which the tool uses to train the classifier and classify if the function is vulnerable or not. We can improve the model to more fine granularity vulnerability localization. Presently, our approach divides source code files into methods and finds vulnerabilities at the method level. However, reducing the vulnerability to few code statements rather than methods, using slicing techniques would increase the model's granularity. The current pipeline/tool acts as a static analyzer that can extend to new vulnerabilities. As future work, we would like to extend the pipeline by creating a Siamese Network that could identify a vulnerability with a sparse dataset or train the model for custom potential vulnerabilities labeled by the user with a sparse dataset.

8 Conclusion

In this work, we present a tool that automatically extracts the signature of the software vulnerabilities in terms of node connectivity diagram over a graph-level representation of the java source code. We developed a pipeline that utilizes a tool that generated intermediate graph representation of the source code, preserving its structural and semantic information and a graph neural network model to classify a Java function as vulnerable or non-vulnerable.

To implement our approach, we developed a tool called JavaVul that consists of three components. The first component of JavaVul was a tool that operates at the source code level to generate an intermediate graphical representation of the source code even if it is not compilable. The intermediate graphical representation enhances the information captured at the source code level by providing an intermediate graphical representation of the source code that combines the primitive graphical representation properties. It further uses it to automatically learn vulnerability features and find vulnerable patterns in a source code at the method level. We chose the code property graph, a data structure that combines the primitive data structures Abstract Syntax Tree, Control Flow Graph, and Program Dependence Graph as the intermediate graphical representation. Since the available tools did not meet the requirements, we developed a tool based on ANTLR and Java graph library called JCPG. It operates at the source code level to analyze the data-flow and control-flow in a Java source code and generate the code property graph. The tool uses the ANTLR parser to parse the source code and ANTLR visitor pattern and Java Graph Library to construct the graph by walking the parse tree and producing the code property graph in three formats DOT, JSON, and GML for further use. The second component of the JavaVul tool was a tool to embed the graph attributes capturing the attributes' semantic meaning. For this, we chose the Word2Vec model trained on an extended code corpus. The third component of the JavaVul tool is the GNN model. It uses the GNN model for representation learning and a classifier to classify if the method is vulnerable. To increment the Graph Neural Networks model's classification power, we employ the transfer learning approach introduced by Hu et.al. [] that uses the node-level and graph-level pre-training approach. To evaluate the approach, we use the SARD's Juliet dataset. Additionally, to alleviate the lack of benchmark dataset and model, we used the Gated Graph Neural model with a novel classifier model Conv used in the previous researches [74] [58] to compare the evaluation results of our model. Additionally, we performed a comparison of our model with state-of-the-art Static Analysis Tools.

The experiments we performed to evaluate the approach show that our model without pre-training can detect 96.13% of the vulnerabilities with a precision of 95.89%, and transfer learning improves the model's performance. Hence, the model can extend to new vulnerabilities without negative transfer. Additionally, after pre-training, our model's recall and precision scores for the binary classification task were 96.86% and 97.90% respectively, the micro recall and precision scores of our model for the multi-class classification task were 97.65% and 97.65% respectively. The weighted precision and recall of the model for the multi-class classification were 97.65% and 97.85%. Our model outperforms the model based on Gated Graph Neural Network used in the previous research works. Our model detects 5.43% more vulnerabilities with 10.83% more precision. Our model also outperforms the state-of-the-art static analysis tool even after improving the proof of concept to

address the static analysis tool's limitations. Additionally, unlike the static analyzer that requires manual rule definition, our model automatically learns the vulnerability features and differentiates the vulnerable methods from the healthy counterpart.

These results show that Graph Neural Networks, combined with the JCPG, can be trained to predict vulnerable patterns at the method level and be used as a bug pattern finding tool instead of static analysis tools. We implemented the approach in a tool called GraphVul. It already contains the node-level and graph-level pre-trained model along with models for different Java Vulnerabilities. The tool code and the pre-processed data are published online for further research and analysis. We implemented the approach in a tool called GraphVul. It already contains the node-level and graph-level pre-trained model along with models for different Java Vulnerabilities. The tool code and the pre-processed data are published online for further research and analysis. [2] [3]

References

[1] Findbugs-find bugs in java programs.

[2] Java code property graph tool(jcpg tool). https://github.com/Samarjeet93/Java-Code-Property-Graph-Generator--JCPG-Tool.git.

[3] Jvul:java vulnerability detection tool. https://github.com/Samarjeet93/Java-Vulneraility-Tool-JVUL-Tool-.git.

[4] Pmd - an extensible cross-language static code analyzer.

[5] Pmd: A code analyzer for java programmers.

[6] Soot:java optimization framework. https://github.com/soot-oss/soot.

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[8] Frances E. Allen. Control flow analysis. 5(7):1–19, July 1970.

[9] Fady Bashay. *What is the CIA triangle and why is it important for cybersecurity management?* Logsign Blog, 2018.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[11] Sindre Beba and Magnus Melseth Karlsen. Implementation analysis of open-source static analysis tools for detecting security vulnerabilities. 2019.

[12] P. Black. Juliet 1.3 test suite: Changes from 1.2. https://doi.org/10.6028/NIST.TN.1995, 2018.

[13] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016.

[14] Xavier Bresson and Thomas Laurent. Residual gated graph convnets, 2018.

[15] Find Security Bugs. Find security bugs - the spotbugs plugin for security audits of java web applications. https://find-sec-bugs.github.io/, 2019.

[16] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[17] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. FSE 2016, page 678–689, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Nick Rodgers Charest, Thomas and Yan Wu. Comparison of static analysis tools for java using the juliet test suite. In *Proceedings of the 11th Interna- tional Conference on Cyber Warfare and Security, ICCWS*, 2016.

[19] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. *CoRR*, abs/1708.02368, 2017.

[21] Black Paul E Okun Vadim Ribeiro Athos Cohen Terry S Delaitre Aurelien, Stivalet Bertrand. Sate v report: ten years of static analysis tool expositions. In *National Institute of Standards and Technology*, 2018.

[22] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization, July 1987.

[23] The Apache Software Foundation. Apache commons bcel. `https://commons.apache.org/proper/commons-bcel/`, 2019.

[24] Javier García-Muñoz, Marisol García-Valls, and Julio Escribano-Barreno. Improved metrics handling in sonarqube for software quality monitoring. In Sigeru Omatu, Ali Selamat, Grzegorz Bocewicz, Pawel Sitek, Izabela Ewa Nielsen, Julián Alberto García-García, and Javier Bajo, editors, *Distributed Computing and Artificial Intelligence, 13th International Conference, DCAI 2016, Sevilla, Spain, 1-3 June, 2016*, volume 474 of *Advances in Intelligent Systems and Computing*, pages 463–470. Springer, 2016.

[25] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, page 45–54, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Seyed Mohammad Ghaffarian. Java graph library. `https://github.com/ghaffarian/graphs.git`, 2018.

[27] Seyed Mohammad Ghaffarian and H. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50:56:1–56:36, 2017.

[28] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[29] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[31] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[32] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. Pre-training graph neural networks. *CoRR*, abs/1905.12265, 2019.

[33] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. Gpt-gnn: Generative pre-training of graph neural networks, 2020.

[34] Giacomo Iadarola. Graph-based classification for detecting instances of bug patterns, October 2018.

[35] S. M. Jadhav, S. Nalbalwar, A. Ghatol, and B. Ambedkar. Performance evaluation of multilayer perceptron neural network based cardiac arrhythmia classifier. 2012.

[36] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[37] Yann LeCun and Yoshua Bengio. *Convolutional Networks for Images, Speech, and Time Series*, page 255–258. MIT Press, Cambridge, MA, USA, 1998.

[38] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5), 2020.

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. ESEC/FSE 2017, page 627–637, New York, NY, USA, 2017. Association for Computing Machinery.

[40] G. Lin, S. Wen, Q. L. Han, J. Zhang, and Y. Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[41] D. Marcheggiani and I. Titov. Encoding sentences with graph convolutional networks for semantic. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1506–1515, 2017.

[42] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A survey of the usages of deep learning in natural language processing. *CoRR*, abs/1807.10854, 2018.

[43] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Cruzes. Myths and facts about static application security testing tools: An action research at telenor digital. In *XP*, 2018.

[44] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[45] G. Panchal, A. Ganatra, Y. Kosta, and D. Panchal. Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers. *International Journal of Computer Theory and Engineering*, pages 332–337, 2011.

[46] Terence Parr. Antlr(another tool for language recognition). `www.antlr.org`, 1992.

[47] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[48] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern, 2010.

[49] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

[50] A Szlam S. Sukhbaatar and R. Fergus. Learning multiagent communication with backpropagation. *NIPS'16: Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 2252–2260, 2016.

[51] Nicholas Saccente, Josh Dehlinger, L. Deng, Suranjan Chakraborty, and Yin Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 114–121, 2019.

[52] Luciano Sampaio and Alessandro Garcia. Early security vulnerability detector - esvd. `https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/`, 2019.

[53] ShiftLeft Security. Joern-the bug hunters workbench. `https://github.com/ShiftLeftSecurity/joern.git`, 2020.

[54] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3), June 2012.

[55] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, November 2014.

[56] Sherri Sparks, S. Embleton, Ryan Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486, 2007.

[57] SpotBugs. Spotbugs:find bugs in java programs. `https://spotbugs.github.io/`, 2018.

[58] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph, 2020.

[59] Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809 – 1827, 2013. Special section on Language Descriptions Tools and Applications (LDTA'08 '09) Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).

[60] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[61] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. 11:3371–3408, December 2010.

[62] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391, 2018.

[63] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[64] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein.

[65] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.

[66] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472, 2017.

[67] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering*, 45(11):1125–1149, 2019.

[68] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[69] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, 2015.

[70] Fabian Yamaguchi. Pattern-based methods for vulnerability discovery. *it - Information Technology*, 59(2):101–106, 2017.

[71] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, page 13, USA, 2011. USENIX Association.

[72] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 359–368, New York, NY, USA, 2012. Association for Computing Machinery.

[73] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*, CCS '13, page 499–510, New York, NY, USA, 2013. Association for Computing Machinery.

[74] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.

A APPENDIX

A.1 Basic Java Constructs

In the following section, we present the JCPG tool evaluation result for the Basic Java constructs. Table 4 shows the Java source codes and their description used for the following experiments.

```
public class Company {

    private String domainName = null;

    // constructor with no parameter
    public Company(){
        this.domainName = "default";
    }

    // constructor with single parameter
    public Company(String domainName){
        this.domainName = domainName;
    }

    public void getName(){
        System.out.println(this.domainName);
    }

    public static void main(String[] args) {
        // calling the constructor with no parameter
        Company defaultObj = new Company();

        // calling the constructor with single parameter
        Company programizObj = new Company("programiz.com");

        defaultObj.getName();
        programizObj.getName();
    }
}
```

Figure 50: Source code of Const.java

```java
class Outer_class {
   int num;

   // inner class
   private class Inner_class {
      public void print() {
         System.out.println("This is an inner class");
      }
   }

   // Accessing the inner class from the method within
   void display_Inner() {
      Inner_class inner = new Inner_class();
      inner.print();
   }
}

public class My_class {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_class outer = new Outer_class();

      // Accessing the display_Inner() method.
      outer.display_Inner();
   }
}
```

Figure 51: Source code of ClassTutorial.java

```java
class JavaExample2{
   static int num;
   static String mystr;
   //First Static block
   static{
      System.out.println("Static Block 1");
      num = 68;
      mystr = "Block1";
  }
  //Second static block
  static{
      System.out.println("Static Block 2");
      num = 98;
      mystr = "Block2";
  }
  public static void main(String args[])
  {
      System.out.println("Value of num: "+num);
      System.out.println("Value of mystr: "+mystr);
   }
}
```

Figure 52: Source code for StaticBlock.java

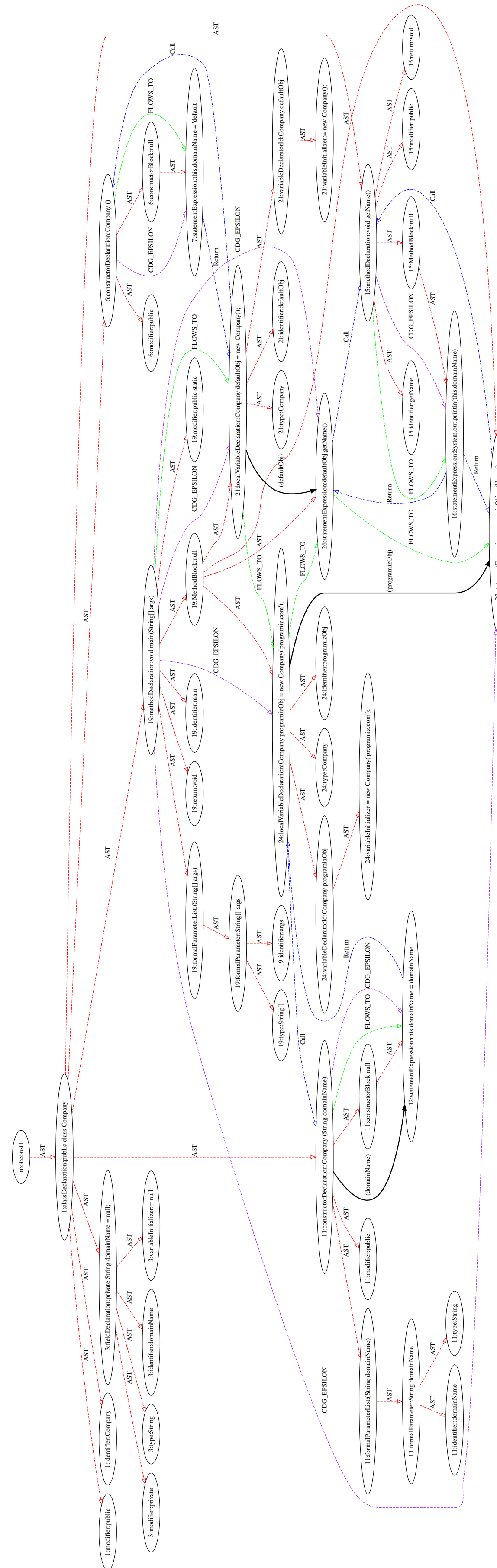


Figure 53: Code Property Graph for 50

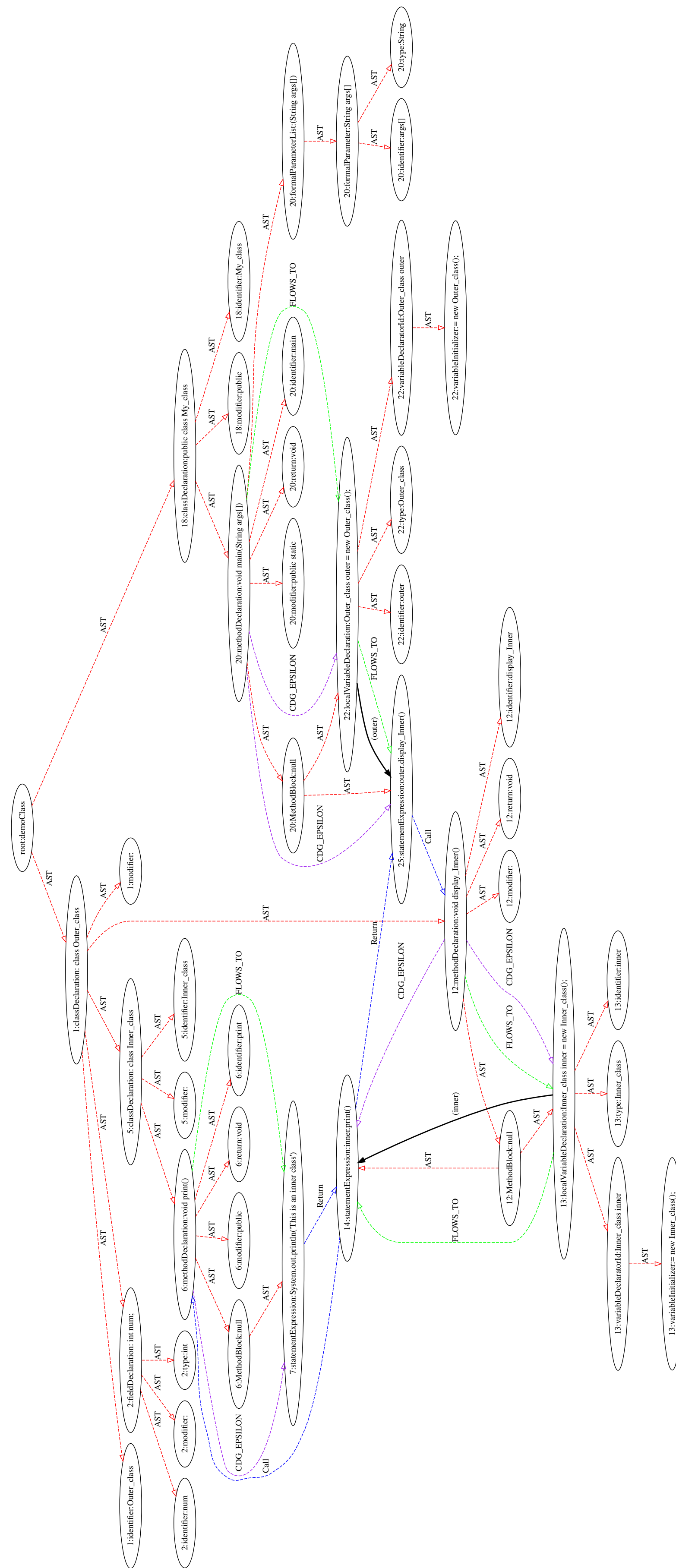


Figure 54: Code Property Graph for 51

Figure 55: Code Property Graph for 52

```java
class Main {
  public static void main(String[] args) {

    int number = 0;

    // checks if number is greater than 0
    if (number > 0) {
      System.out.println("The number is positive.");
    }

    // checks if number is less than 0
    else if (number < 0) {
      System.out.println("The number is negative.");
    }

    // if both condition is false
    else {
      System.out.println("The number is 0.");
    }
  }
}
```

Figure 56: Source code of IfElse.java

```java
public class PyramidExample2 {
        public static void main(String[] args) {
                int term=6;
                for(int i=1;i<=term;i++){
                for(int j=term;j>=i;j--){
                        System.out.print("* ");
                }
                System.out.println();//new line
                }
        }
}
```

Figure 57: Source code of TradFor.java

```java
// Java program to illustrate
// for-each loop
class For_Each
{
    public static void main(String[] arg)
    {
        {
            int[] marks = { 125, 132, 95, 116, 110 };

            int highest_marks = maximum(marks);
            System.out.println("The highest score is " + highest_marks);
        }
    }
    public static int maximum(int[] numbers)
    {
        int maxSoFar = numbers[0];

        // for each loop
        for (int num : numbers)
        {
            if (num > maxSoFar)
            {
                maxSoFar = num;
            }
        }
    return maxSoFar;
    }
}
```

Figure 58: Source code of ForEach.java

```java
// Program to display numbers from 1 to 5

class Main {
  public static void main(String[] args) {

    // declare variables
    int i = 1, n = 5;

    // while loop from 1 to 5
    while(i <= n) {
      System.out.println(i);
      i++;
    }
  }
}
```

Figure 59: Source code of While.java

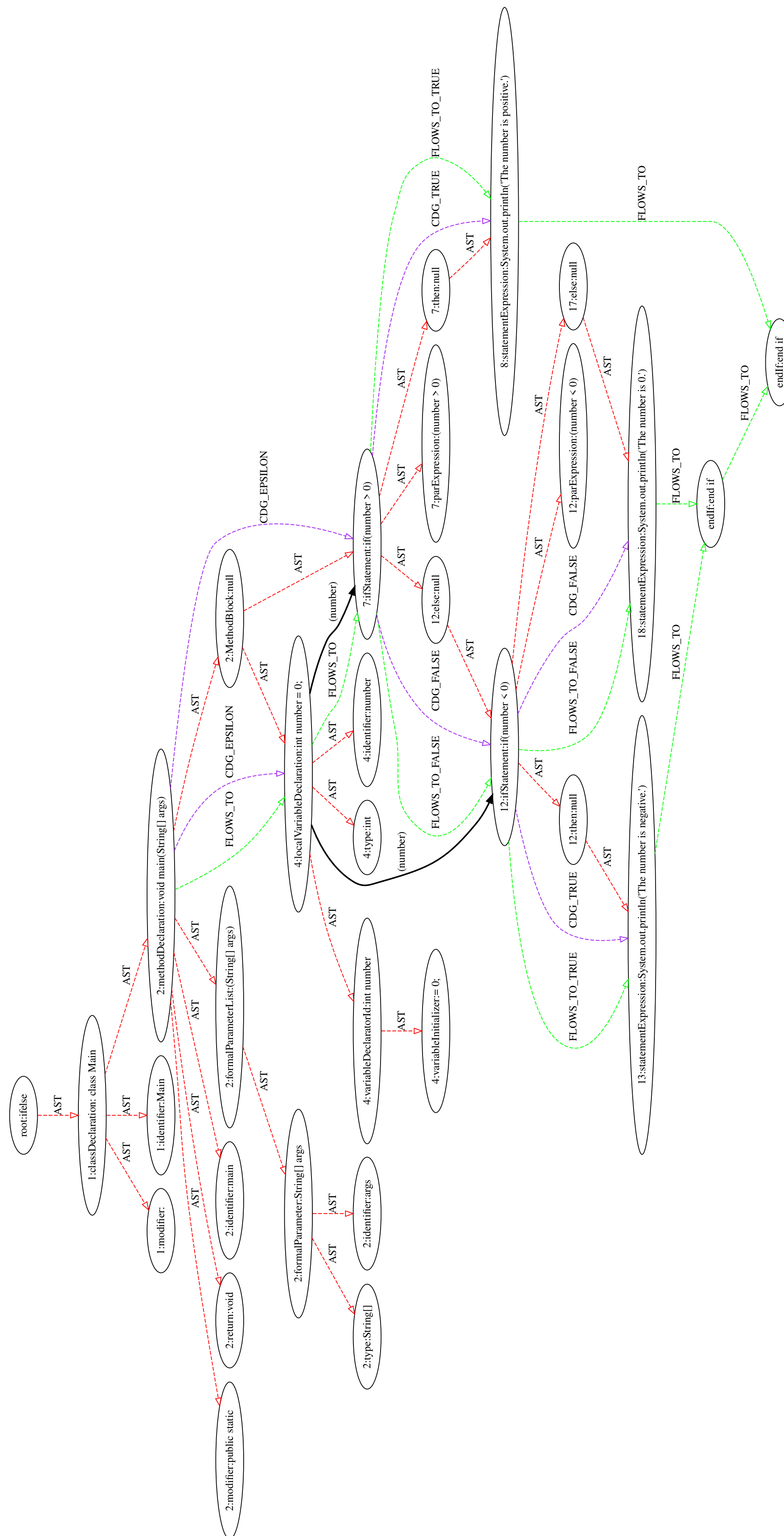


Figure 60: Code Property Graph for 56

Figure 61: Code Property Graph for 57

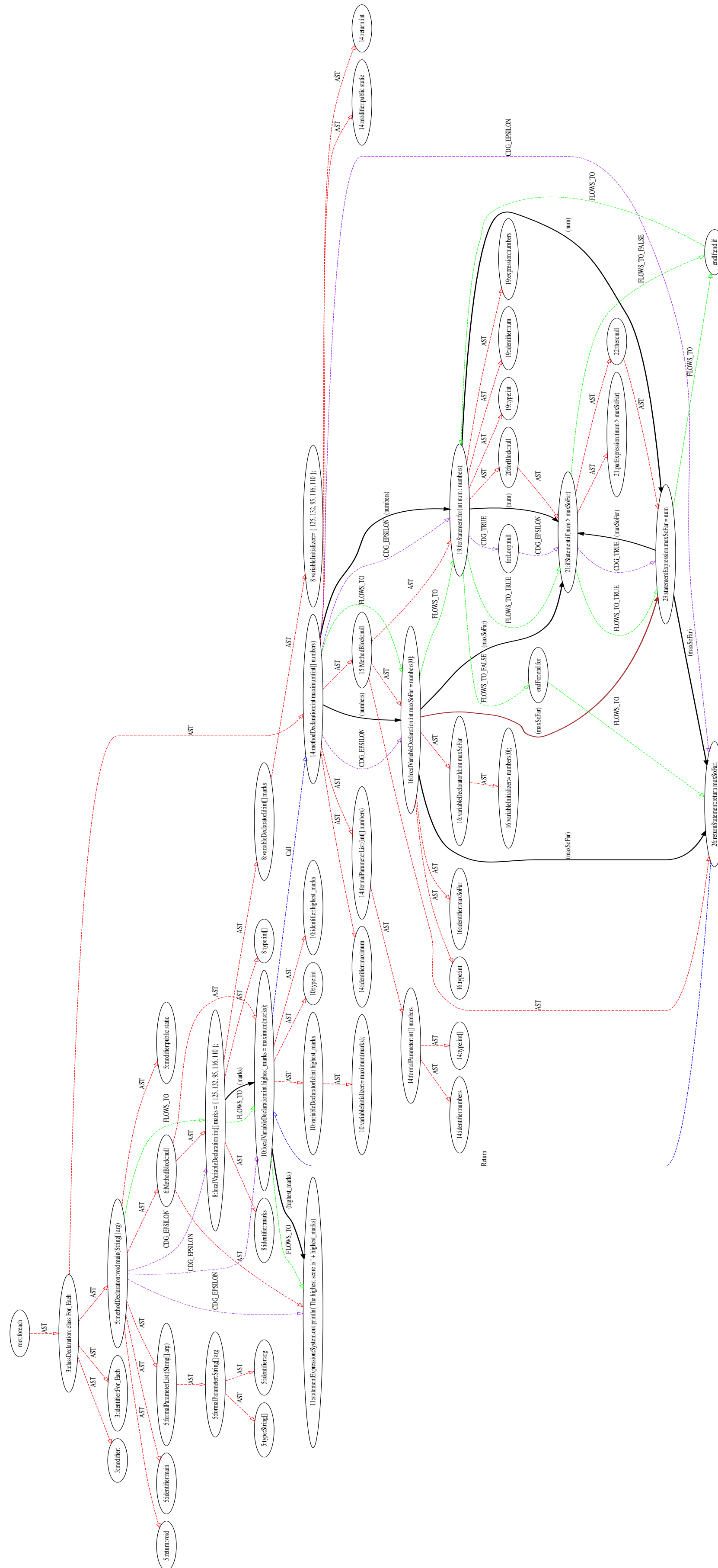


Figure 62: Code Property Graph for 58

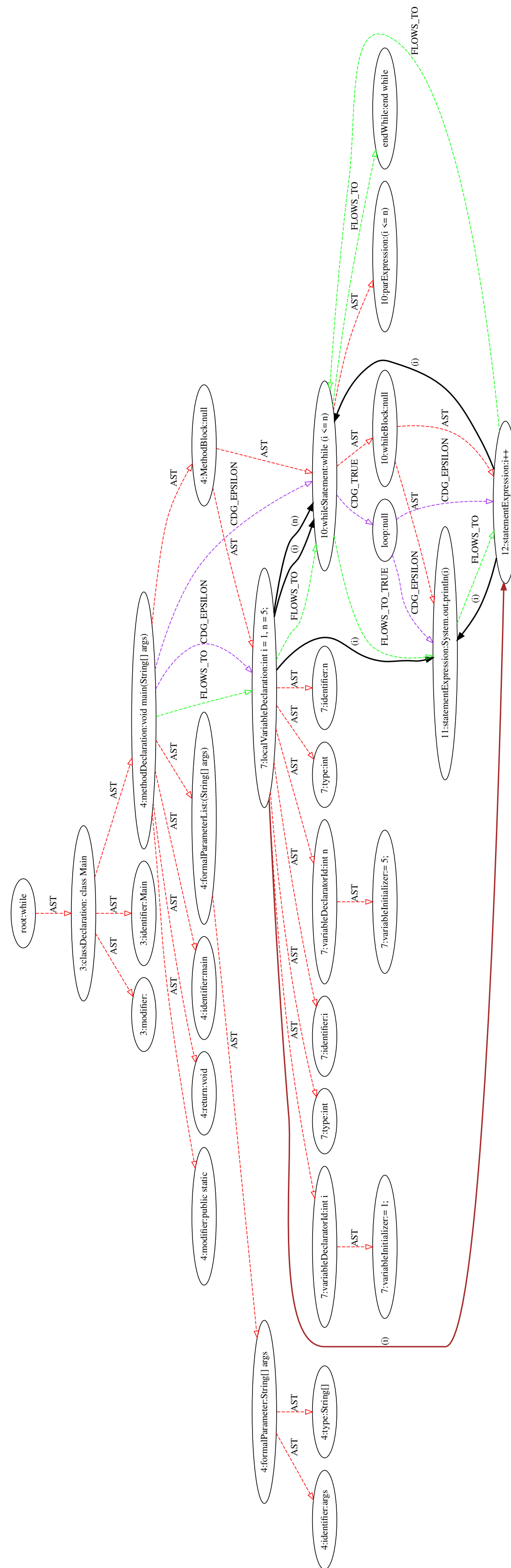


Figure 63: Code Property Graph for 59

```java
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Figure 64: Source code of While.java

```java
public class Tester {
   public static void main(String args[]) {

      first:
         for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++){
               if(i == 1){
                  continue first;
               }
               System.out.print(" [i = " + i + ", j = " + j + "] ");
            }
         }
         System.out.println();

      second:
         for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++){
               if(i == 1){
                  break second;
               }
               System.out.print(" [i = " + i + ", j = " + j + "] ");
            }
         }
   }
}
```

Figure 65: Source code of Label.java

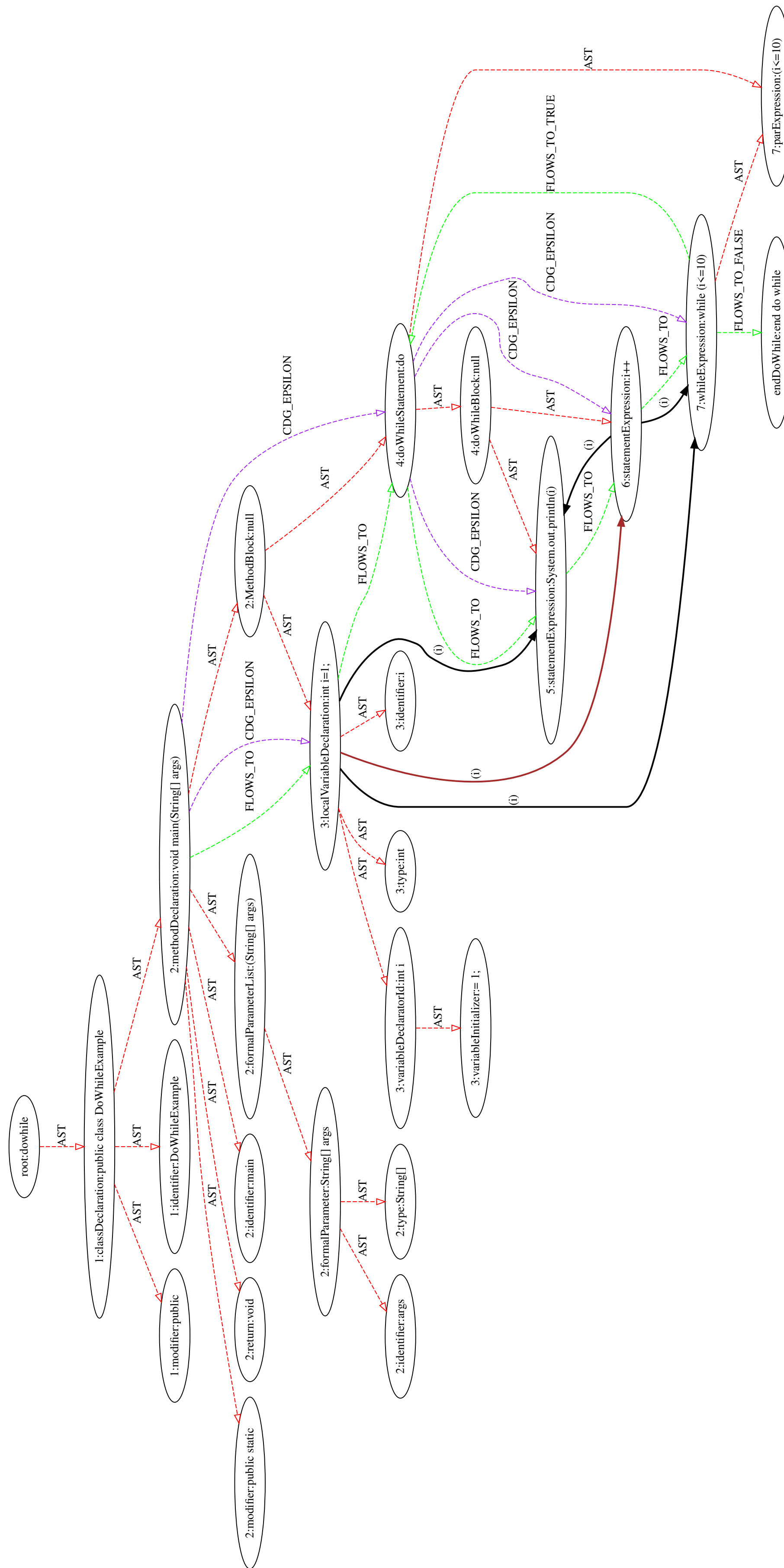


Figure 66: Code Property Graph for 64

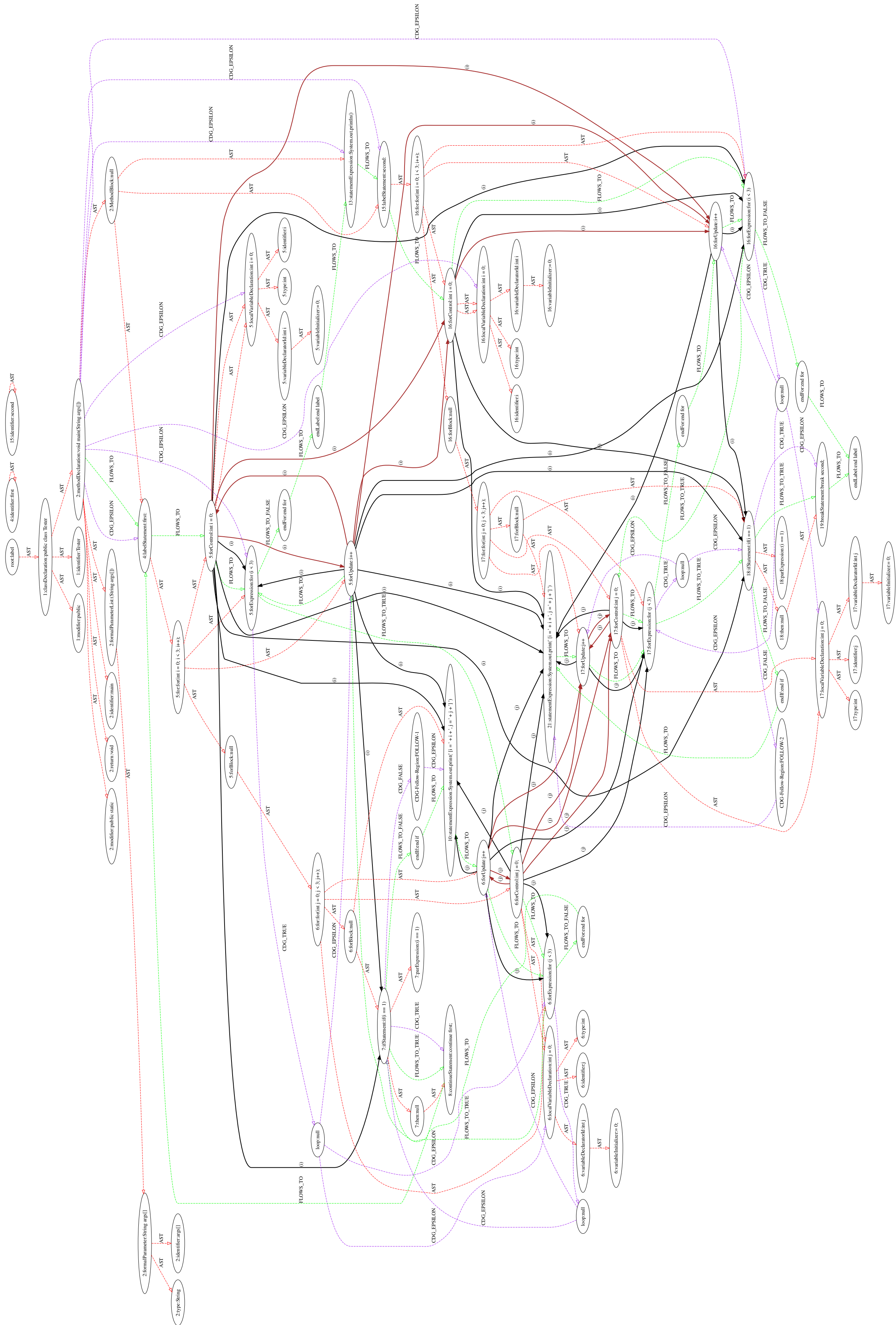


Figure 67: Code Property Graph for 65

```java
// Java program to demonstrate switch case
// with primitive(int) data type
public class Test {
    public static void main(String[] args)
    {
        int day = 5;
        String dayString;

        // switch statement with int data type
        switch (day) {
        case 1:
            dayString = "Monday";
            break;
        case 2:
            dayString = "Tuesday";
            break;
        case 3:
            dayString = "Wednesday";
            break;
        case 4:
            dayString = "Thursday";
            break;
        case 5:
            dayString = "Friday";
            break;
        case 6:
            dayString = "Saturday";
            break;
        case 7:
            dayString = "Sunday";
            break;
        default:
            dayString = "Invalid day";
            break;
        }
        System.out.println(dayString);
    }
}
```

Figure 68: Source code for Switch.java

```java
class Table{

        void printTable(int n){
          synchronized(this){//synchronized block
            for(int i=1;i<=5;i++){
             System.out.println(n*i);
             try{
              Thread.sleep(400);
             }catch(Exception e){System.out.println(e);}
            }
          }
        }//end of the method
}

class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
        this.t=t;
        }
        public void run(){
        t.printTable(5);
        }

}
class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
        this.t=t;
        }
        public void run(){
        t.printTable(100);
        }
}

public class TestSynchronizedBlock1{
        public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
        }
}
```

Figure 69: Source code of Synch.java

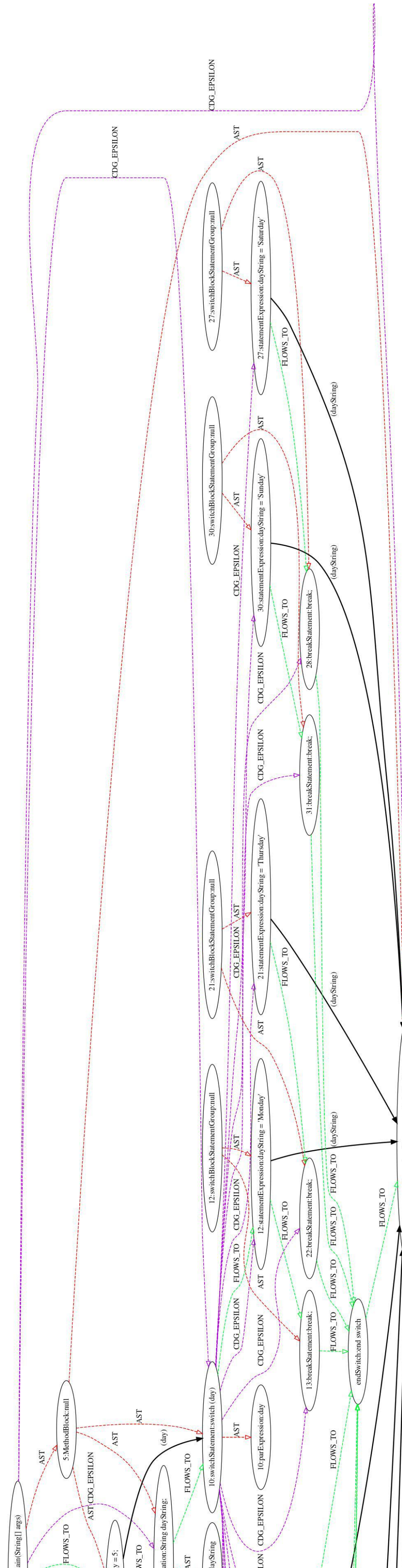
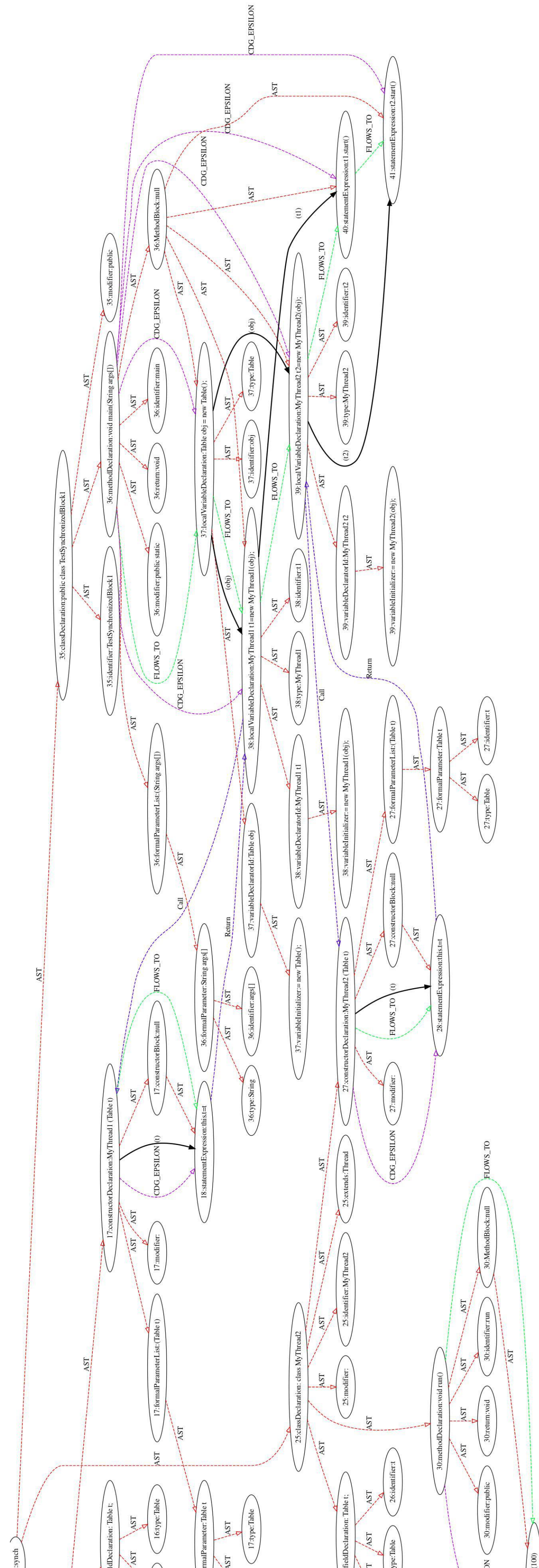

CDG_EPSILON
AST
FLOWS_TO
(dayString)
(day)
27:switchBlockStatementGroup:null
27:statementExpression:dayString = 'Saturday'
30:switchBlockStatementGroup:null
30:statementExpression:dayString = 'Sunday'
28:breakStatement:break;
31:breakStatement:break;
21:switchBlockStatementGroup:null
21:statementExpression:dayString = 'Thursday'
12:switchBlockStatementGroup:null
12:statementExpression:dayString = 'Monday'
22:breakStatement:break;
13:breakStatement:break;
endSwitch:end switch
36:statementExpression:System.out.println(dayString)
5:MethodBlock:null
10:switchStatement:switch (day)
10:parExpression:day

Figure 70: Code Property Graph for source code shown in Figure 68



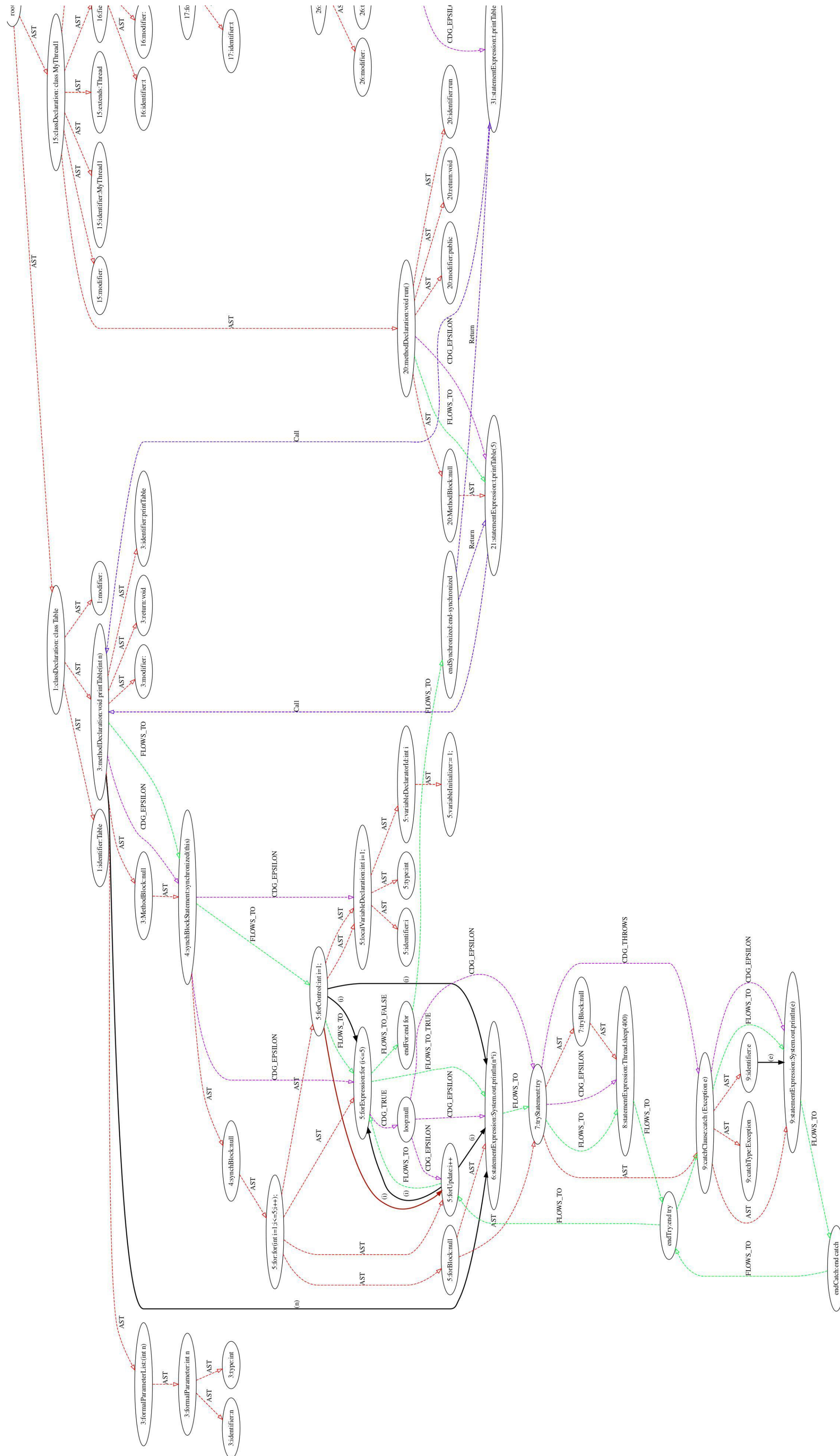


Figure 71: Code Property Graph for source code shown in Figure 69

```java
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Arrays;
public class Test101{
                public static void main(String[] args) {
                        try {
                        if (i > 0) {
                                System.out.println("Positive");
                        } else {
                                if (i == 0)
                                        throw new Exception("ZERO! [ A Pointless Exception! ]");
                                System.out.println("Negative");
                        }
                } catch (Exception ex) {
                        System.err.println(ex);
                }

        }
}
```

Figure 72: Source code of TryCheck.java

```java
import java.io.FileOutputStream;
public class TryWithResources {
        public static void main(String args[]){
                // Using try-with-resources
        try(FileOutputStream fileOutputStream =newFileOutputStream("/java7-new-features/src/abc.txt"))-
{
        String msg = "Welcome to javaTpoint!";
        byte byteArray[] = msg.getBytes(); //converting string into byte array
        fileOutputStream.write(byteArray);
        System.out.println("Message written to file successfuly!");
        }catch(Exception exception){
               System.out.println(exception);
        }
        }
}
```

Figure 73: Source code of TryWithRes.java

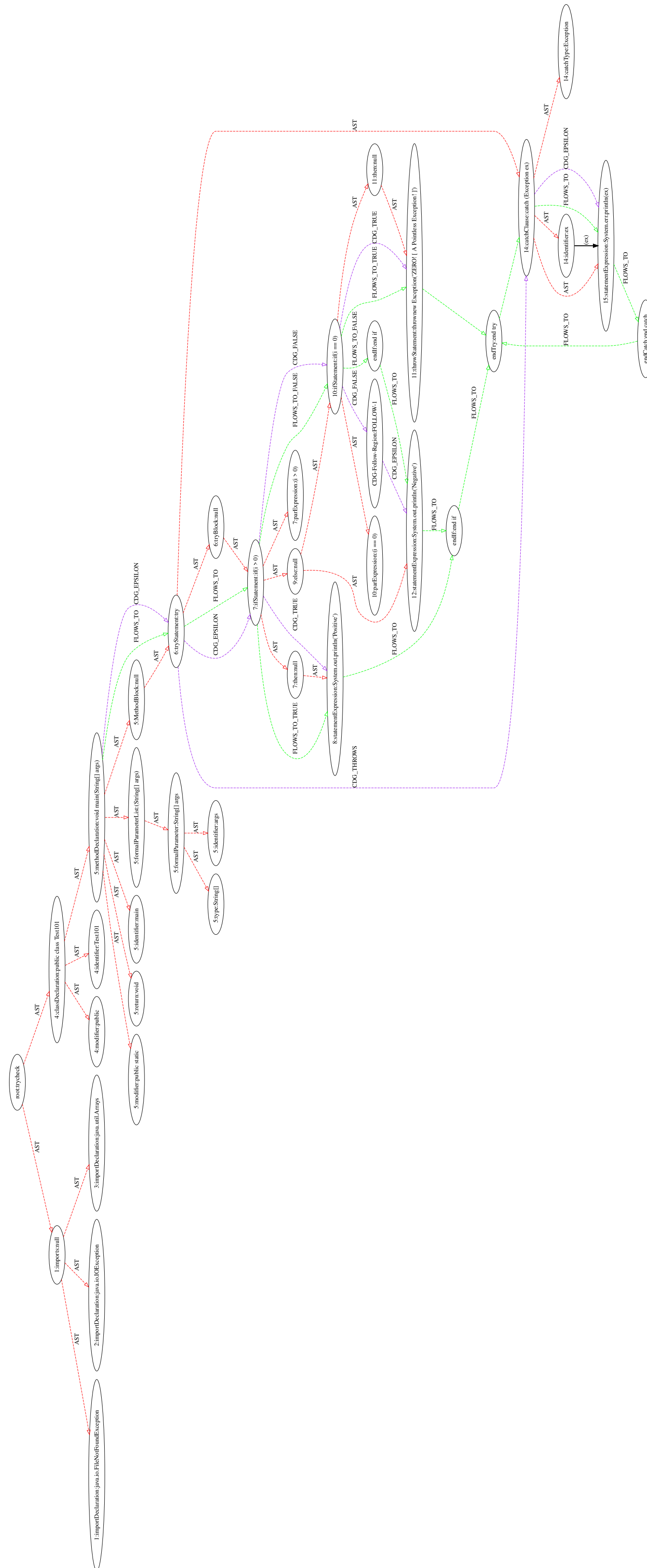


Figure 74: Code Property Graph for 72

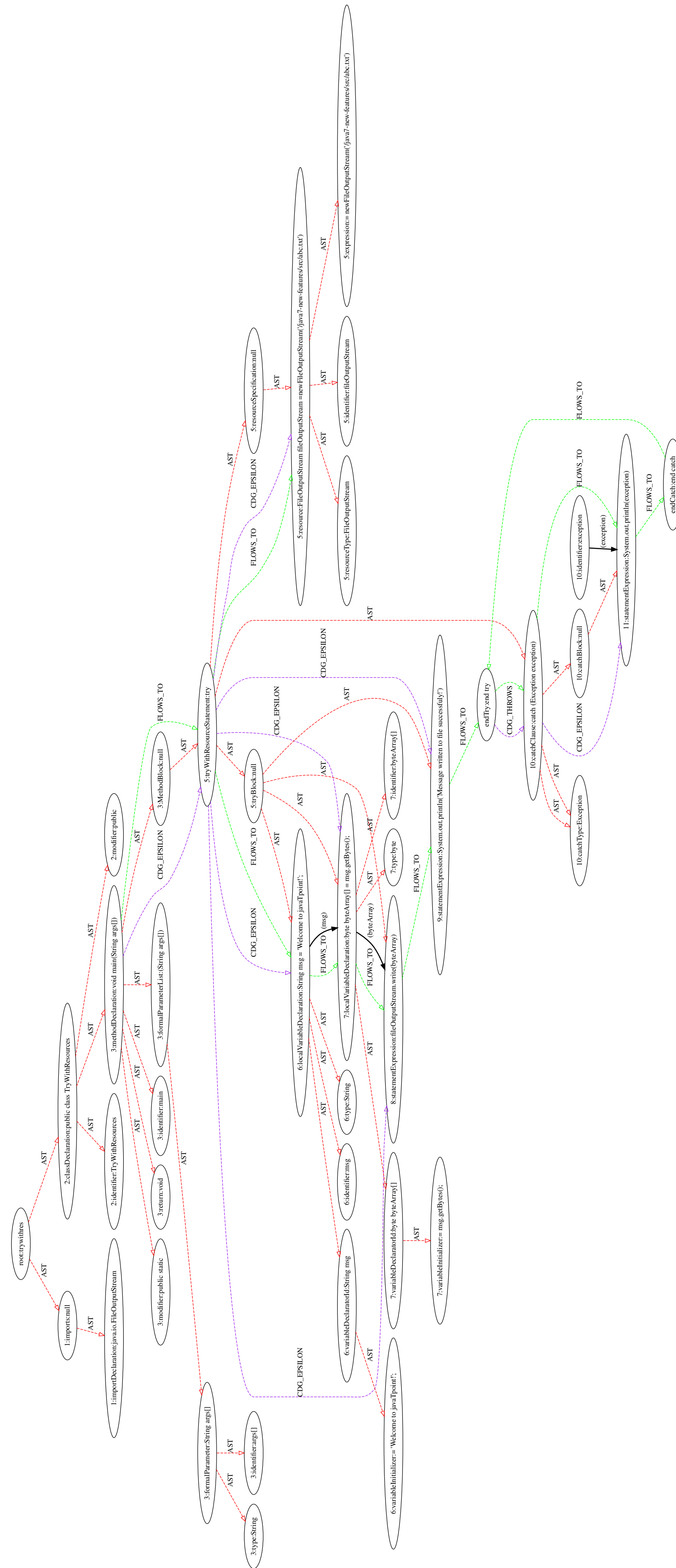


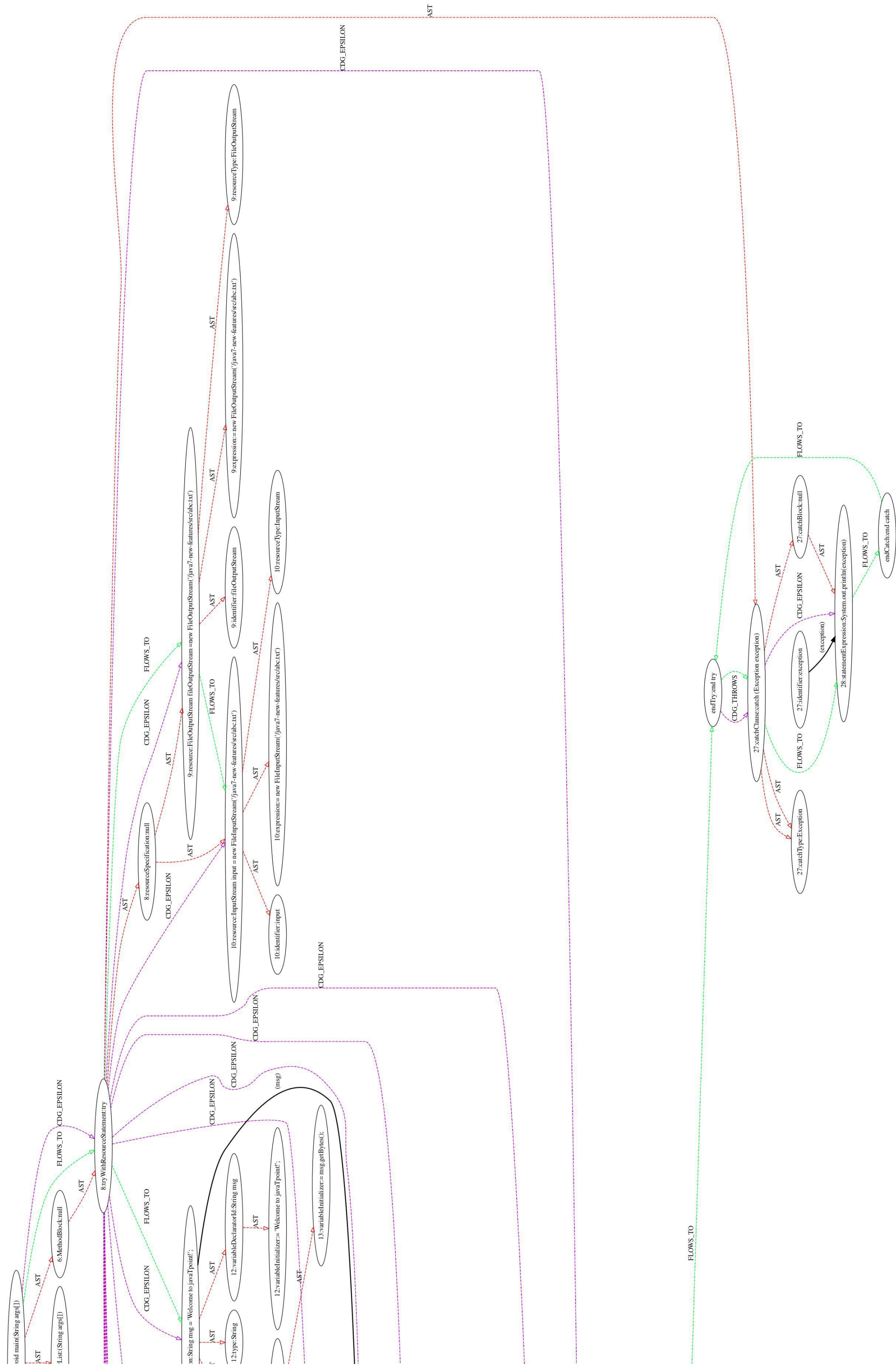
Figure 75: Code Property Graph for 73

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
public class TryWithResources {
public static void main(String args[]){
        // Using try-with-resources
try(    // Using multiple resources
        FileOutputStream fileOutputStream =new FileOutputStream("/java7-new-features/src/abc.txt");
        InputStream input = new FileInputStream("/java7-new-features/src/abc.txt")){
        // -----------------------------Code to write data into
file--------------------------------------------//
        String msg = "Welcome to javaTpoint!";
        byte byteArray[] = msg.getBytes();  // Converting string into byte array
        fileOutputStream.write(byteArray);  // Writing  data into file
        System.out.println("------------Data written into file--------------");
        System.out.println(msg);
        // -----------------------------Code to read data from
file---------------------------------------------//
        // Creating input stream instance
        DataInputStream inst = new DataInputStream(input);
        int data = input.available();
        // Returns an estimate of the number of bytes that can be read from this input stream.
        byte[] byteArray2 = new byte[data]; //
        inst.read(byteArray2);
        String str = new String(byteArray2); // passing byte array into String constructor
        System.out.println("------------Data read from file--------------");
        System.out.println(str); // display file data
}catch(Exception exception){
       System.out.println(exception);
}
}
}
```

Figure 76: Source code of TryMultiRes.java

```
import java.io.FileOutputStream;
public class TryWithResources {
public static void main(String args[]){
        try(    FileOutputStream fileOutputStream=
          new FileOutputStream("/home/irfan/scala-workspace/java7-new-features/src/abc.txt")){
            // -----------------------------Code to write data into
file--------------------------------------------//
                String msg = "Welcome to javaTpoint!";
                byte byteArray[] = msg.getBytes();  // Converting string into byte array
                fileOutputStream.write(byteArray);  // Writing  data into file
                System.out.println("Data written successfully!");
        }catch(Exception exception){
               System.out.println(exception);
        }
        finally{
               System.out.println("Finally executes after closing of declared resources.");
        }
        }
}
```

Figure 77: Source code of TryFinally.java



8:tryWithResourceStatement:try
8:resourceSpecification:null
9:resource:FileOutputStream fileOutputStream = new FileOutputStream('/java7-new-features/src/abc.txt')
9:expression:= new FileOutputStream('/java7-new-features/src/abc.txt')
9:resourceType:FileOutputStream
9:identifier:fileOutputStream
10:resource:InputStream input = new FileInputStream('/java7-new-features/src/abc.txt')
10:expression:= new FileInputStream('/java7-new-features/src/abc.txt')
10:resourceType:InputStream
10:identifier:input
6:MethodBlock:null
12:variableDeclaratorId:String msg
12:variableInitializer:= 'Welcome to javaTpoint!';
13:variableInitializer:= msg.getBytes();
12:type:String
endTry:end try
27:catchClause:catch (Exception exception)
27:catchBlock:null
27:identifier:exception
27:catchType:Exception
28:statementExpression:System.out.println(exception)
endCatch:end catch
AST
FLOWS_TO
CDG_EPSILON
CDG_THROWS
(exception)
(msg)

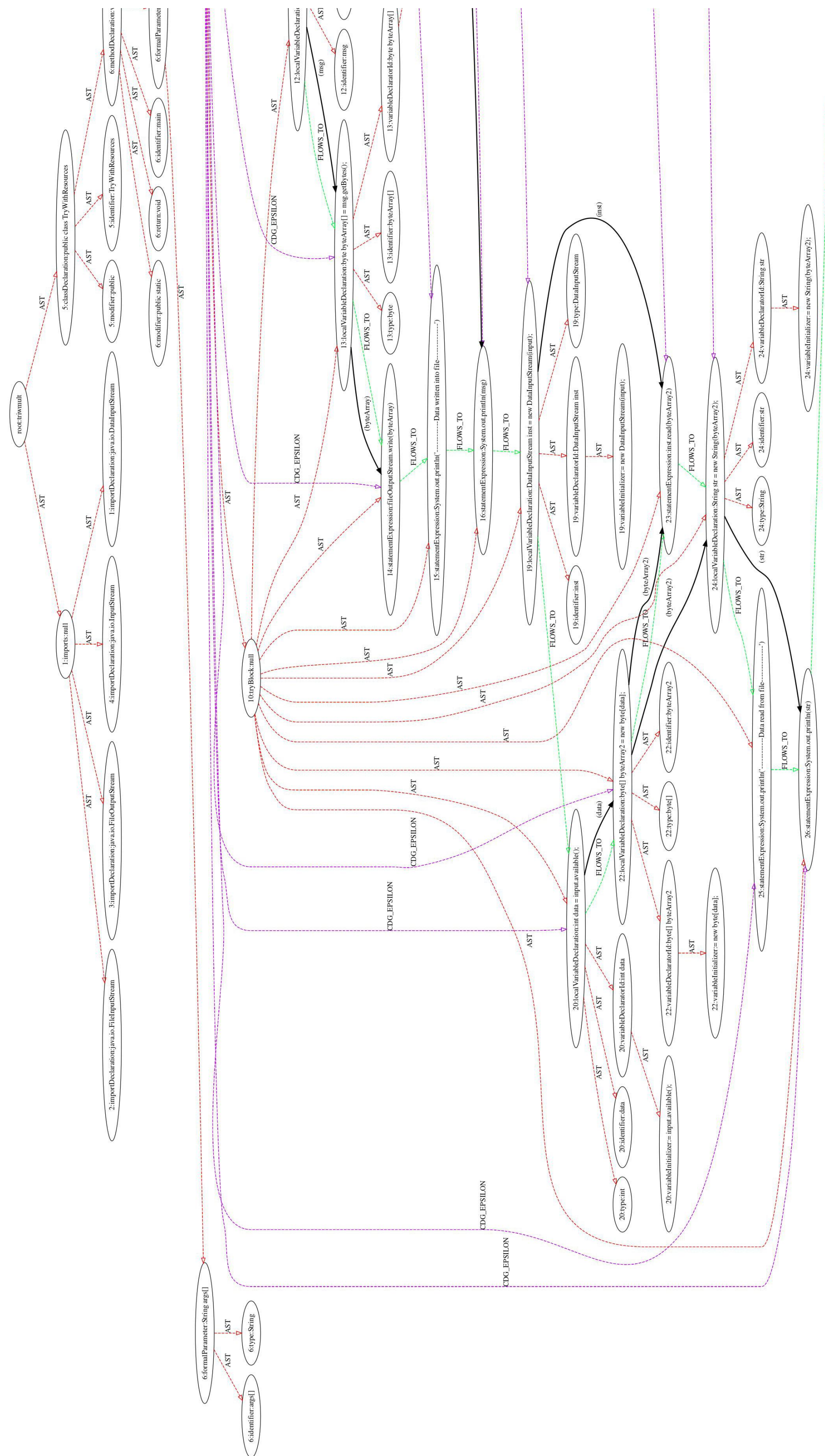


Figure 78: Code Property Graph for 76

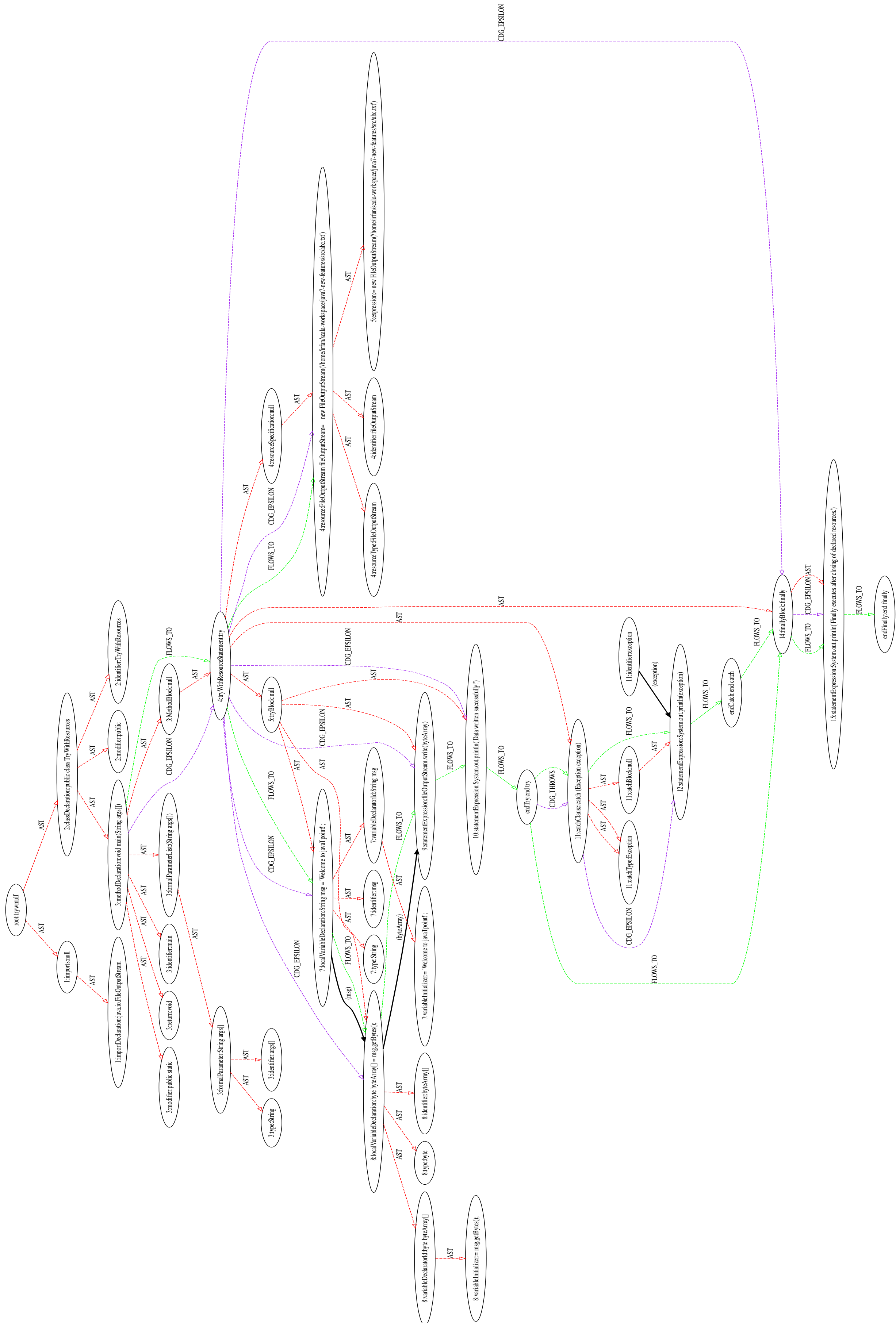


Figure 79: Code Property Graph for 77

```java
// Java program that demonstrates the use of throw
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Figure 80: Source code of Throw.java

```java
// Java program to demonstrate working of throws
class ThrowsExecp
{
        static void fun() throws IllegalAccessException, IOException
        {
                System.out.println("Inside fun(). ");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                try
                {
                        fun();
                }
                catch(IllegalAccessException e)
                {
                        System.out.println("caught in main.");
                }
        }
}
```

Figure 81: Source code of Throws.Java

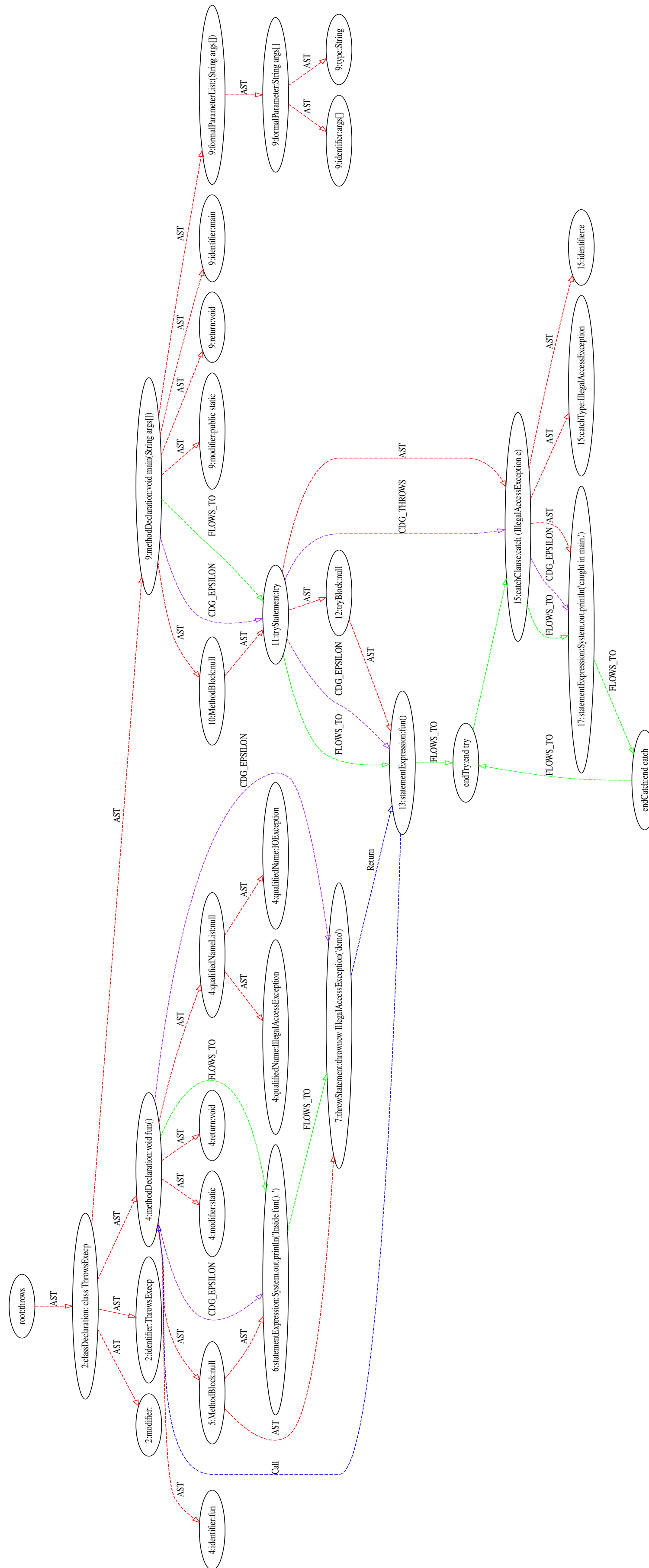


Figure 82: Code Property Graph for 80

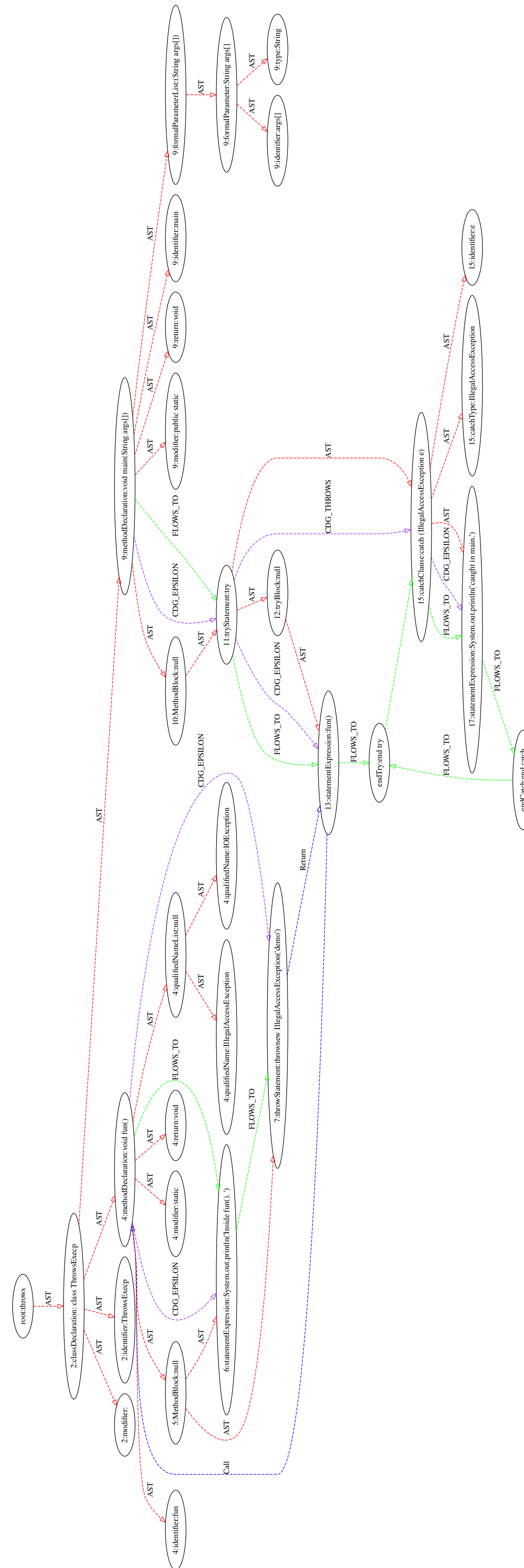
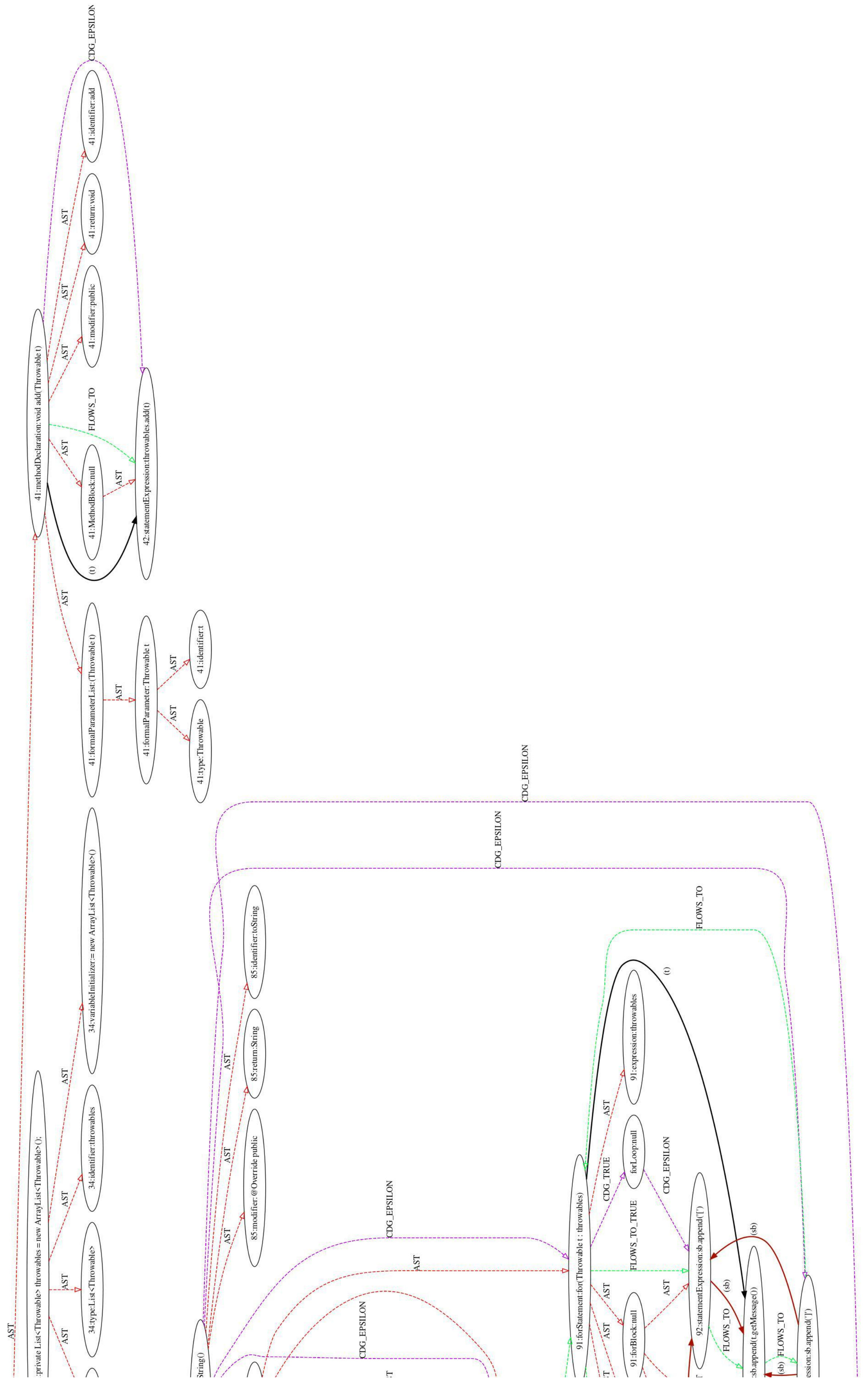
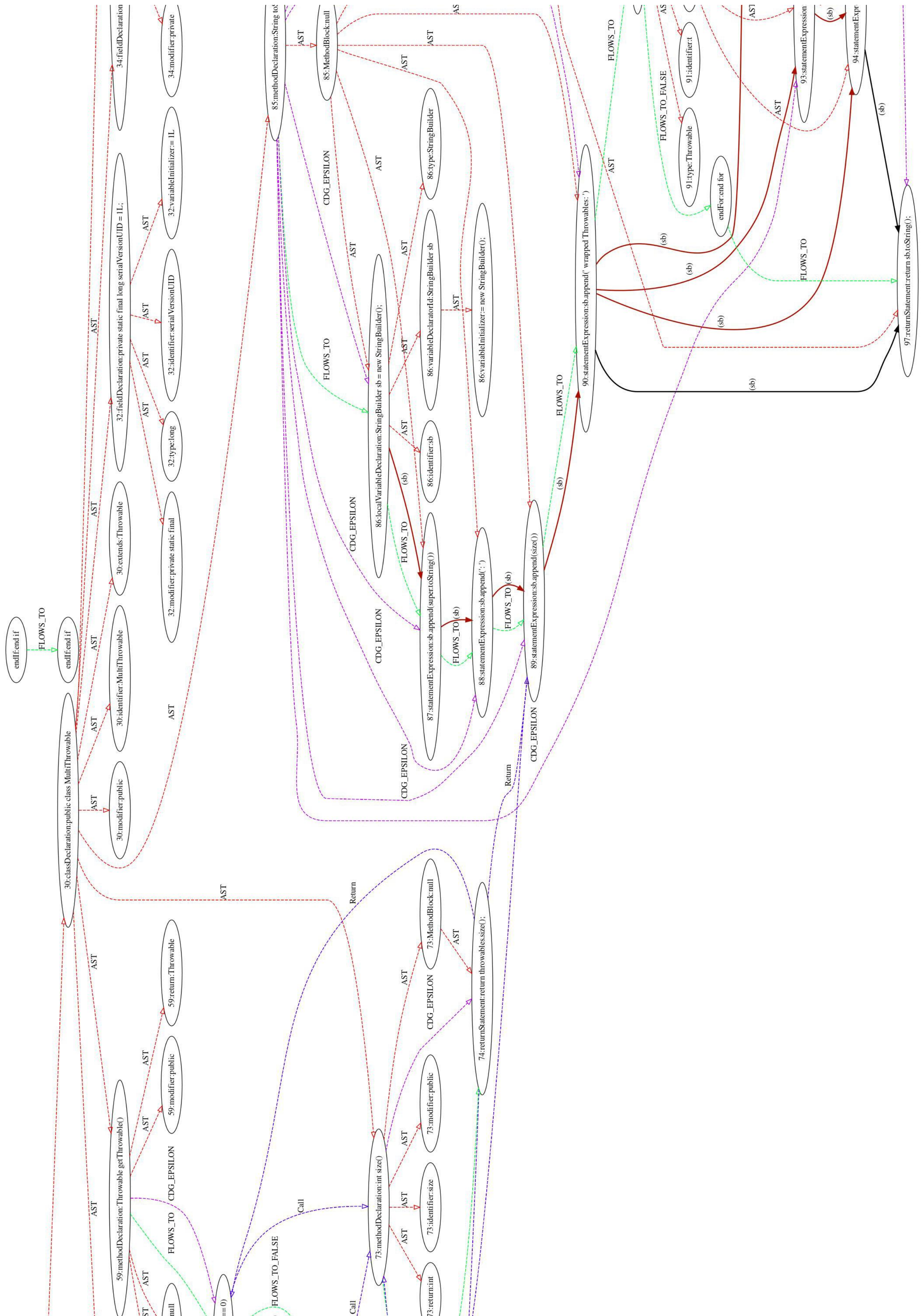


Figure 83: Code Property Graph for 81

```java
package org.apache.tomcat.util;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MultiThrowable extends Throwable {

    private static final long serialVersionUID = 1L;

    private List<Throwable> throwables = new ArrayList<Throwable>();

    public void add(Throwable t) {
        throwables.add(t);
    }

    public List<Throwable> getThrowables() {
        return Collections.unmodifiableList(throwables);
    }

    public Throwable getThrowable() {
        if (size() == 0) {
            return null;
        } else if (size() == 1) {
            return throwables.get(0);
        } else {
            return this;
        }
    }

    public int size() {
        return throwables.size();
    }


    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(super.toString());
        sb.append(": ");
        sb.append(size());
        sb.append(" wrapped Throwables: ");
        for (Throwable t : throwables) {
            sb.append("[");
            sb.append(t.getMessage());
            sb.append("]");
        }

        return sb.toString();
    }
}
```

Figure 84: MultiThrowable.java from TomCat project




endIf:end if
endIf:end if
FLOWS_TO
30:classDeclaration:public class MultiThrowable
30:modifier:public
30:identifier:MultiThrowable
30:extends:Throwable
32:fieldDeclaration:private static final long serialVersionUID = 1L;
32:modifier:private static final
32:type:long
32:identifier:serialVersionUID
32:variableInitializer:= 1L
34:fieldDeclaration
34:modifier:private
59:methodDeclaration:Throwable getThrowable()
59:modifier:public
59:return:Throwable
73:methodDeclaration:int size()
73:return:int
73:identifier:size
73:modifier:public
73:MethodBlock:null
74:returnStatement:return throwables.size();
85:MethodBlock:null
86:localVariableDeclaration:StringBuilder sb = new StringBuilder();
86:type:StringBuilder
86:variableDeclaratorId:StringBuilder sb
86:identifier:sb
86:variableInitializer:= new StringBuilder();
87:statementExpression:sb.append(super.toString())
88:statementExpression:sb.append(': ')
89:statementExpression:sb.append(size())
90:statementExpression:sb.append(' wrapped Throwables:')
91:type:Throwable
91:identifier:t
endFor:end for
97:returnStatement:return sb.toString();
AST
FLOWS_TO
FLOWS_TO_FALSE
CDG_EPSILON
Call
Return
(sb)

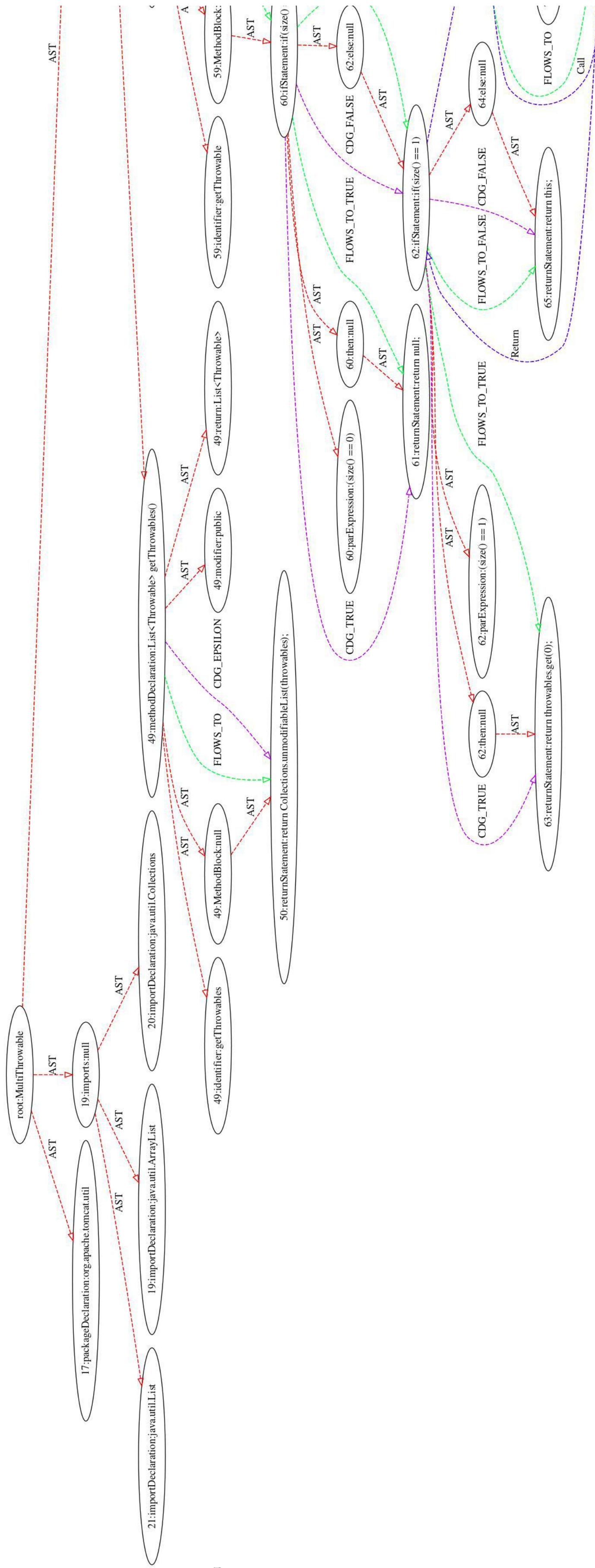


Figure 85: File-Level Code Property Graph for source code shown in Figure 84

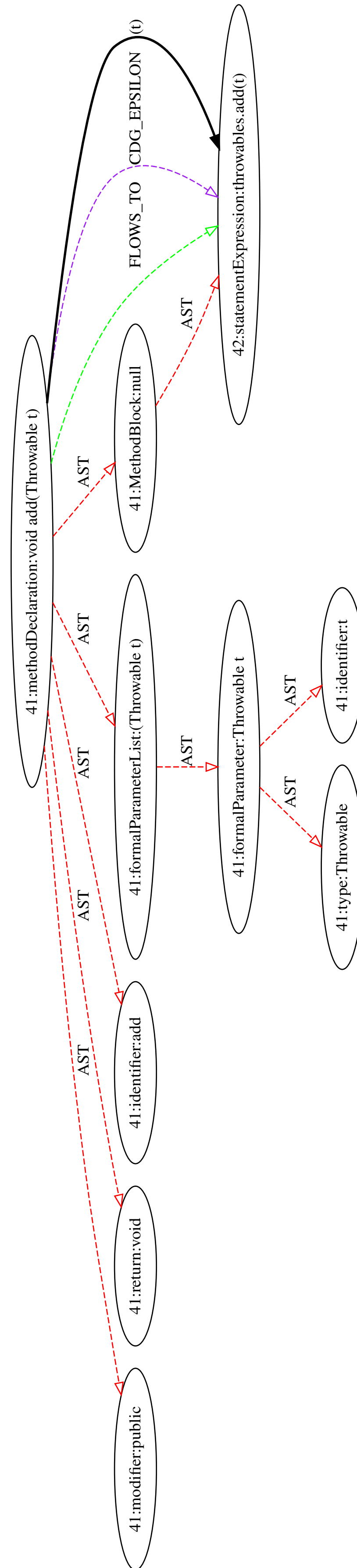


Figure 86: CPG for method add from Figure 84

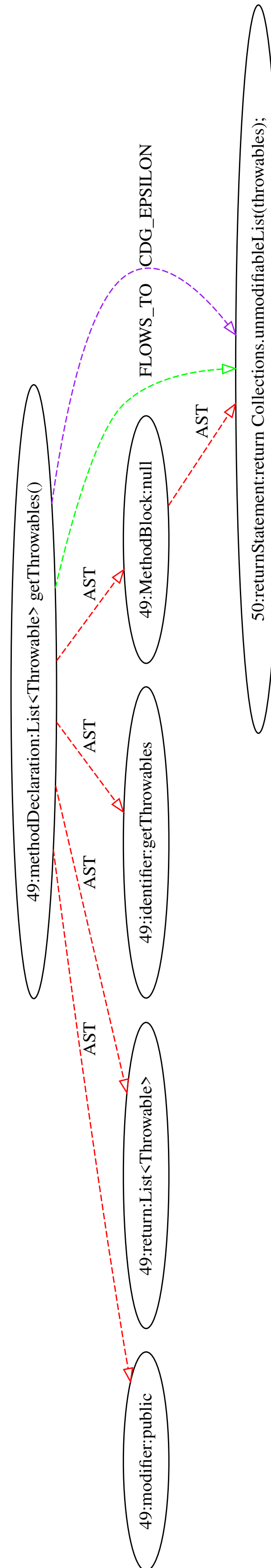


Figure 87: CPG for method getThrowables from Figure 84

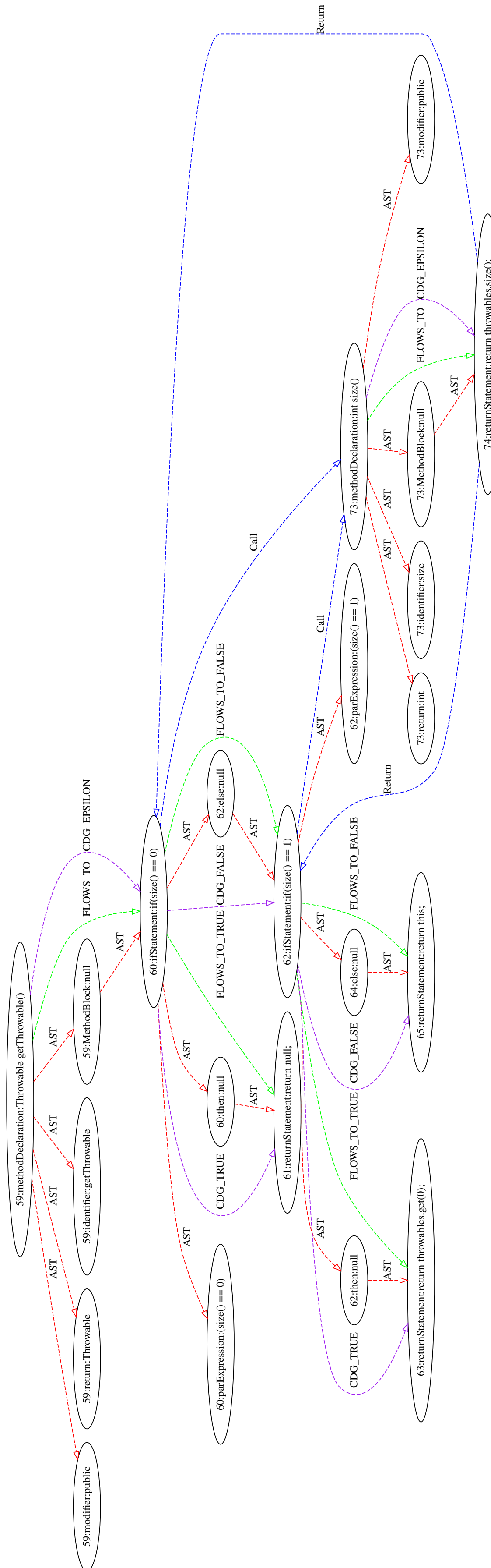


Figure 88: CPG for method getThrowable from Figure 84

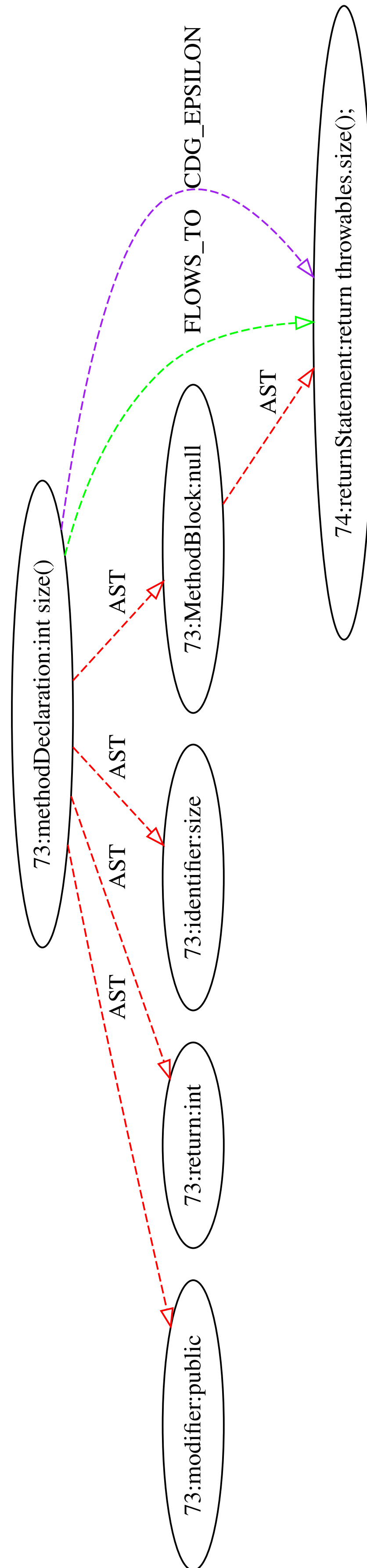


Figure 89: CPG for method size from Figure 84

```
{
 "directed": true,
 "multigraph": true,
 "label": "methodCPG ofMultiThrowable.java",
 "type": "CodeProperty Graph (CPG)",
 "file": "MultiThrowable.java",

 "nodes": [
  {
   "id": 0,
   "line": 12,
   "label": "void add(Throwable t)",
   "type": "methodDeclaration",
   "defs": ["t"],
   "uses": []

  {
   "id": 1,
   "line": 12,
   "label": "public"

  {
   "id": 2,
   "line": 12,
   "label": "void"

  {
   "id": 3,
   "line": 12,
   "label": "add"

  {
   "id": 4,
   "line": 12,
   "label": "(Throwable t)"

  {
   "id": 5,
   "line": 12,
   "label": "Throwable t"

  {
   "id": 6,
   "line": 12,
   "label": "Throwable"

  {
   "id": 7,
   "line": 12,
   "label": "t"

  {
   "id": 8,
   "line": 12,
   "label": "null"

  {
   "id": 9,
   "line": 13,
   "label": "throwables.add(t)",
   "type": "statementExpression",
   "defs": ["$THIS.serialVersionUID", "$THIS.throwables"],
```

```
   "uses": ["t", "$THIS.throwables"]
  }
 ],

 "edges": [
  {
   "id": 0,
   "source": 0,
   "target": 1,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 1,
   "source": 0,
   "target": 2,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 2,
   "source": 0,
   "target": 3,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 3,
   "source": 0,
   "target": 4,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 4,
   "source": 4,
   "target": 5,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 5,
   "source": 5,
   "target": 6,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 6,
   "source": 5,
   "target": 7,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 7,
   "source": 0,
   "target": 8,
   "type": "AST",
   "label": "AST"
  },
  {
```

```
  },
  {
   "id": 8,
   "source": 0,
   "target": 9,
   "type": "CFG",
   "label": "FLOWS_TO"
  },
  {
   "id": 9,
   "source": 0,
   "target": 9,
   "type": "CDG",
   "label": "CDG_EPSILON"
  },
  {
   "id": 10,
   "source": 8,
   "target": 9,
   "type": "AST",
   "label": "AST"
  },
  {
   "id": 11,
   "source": 0,
   "target": 9,
   "type": "Data",
   "label": "t"
  },
 ]
}
```


Figure 90: JSON format CPG output for method add from Figure 84


```
graph [
  directed 1
  multigraph 1
  label "methodCPG ofMultiThrowable.java"
  type "CodeProperty Graph (CPG)"
  file "MultiThrowable.java"

  node [
    id 0
    line 12
    label "void add(Throwable t)"
    type "methodDeclaration"
    defs [var "t"]
    uses []
  ]
  node [
    id 1
    line 12
    label "public"
    type "modifier"
  ]
  node [
    id 2
    line 12
    label "void"
    type "return"
  ]
  node [
    id 3
    line 12
    label "add"
    type "identifier"
  ]
  node [
    id 4
    line 12
    label "(Throwable t)"
    type "formalParameterList"
  ]
  node [
    id 5
    line 12
    label "Throwable t"
    type "formalParameter"
  ]
  node [
    id 6
    line 12
    label "Throwable"
    type "type"
  ]
  node [
    id 7
    line 12
    label "t"
    type "identifier"
  ]
  node [
    id 8
    line 12
    label "null"
    type "MethodBlock"
```

```
  ]
  node [
    id 9
    line 13
    label "throwables.add(t)"
    type "statementExpression"
    defs [var "$THIS.serialVersionUID" var "$THIS.throwables"]
    uses [var "t" var "$THIS.throwables"]
  ]

  edge [
    id 0
    source 0
    target 1
    type "AST"
    label "AST"
  ]
  edge [
    id 1
    source 0
    target 2
    type "AST"
    label "AST"
  ]
  edge [
    id 2
    source 0
    target 3
    type "AST"
    label "AST"
  ]
  edge [
    id 3
    source 0
    target 4
    type "AST"
    label "AST"
  ]
  edge [
    id 4
    source 4
    target 5
    type "AST"
    label "AST"
  ]
  edge [
    id 5
    source 5
    target 6
    type "AST"
    label "AST"
  ]
  edge [
    id 6
    source 5
    target 7
    type "AST"
    label "AST"
  ]
```

```
  ]
  edge [
    id 7
    source 0
    target 8
    type "AST"
    label "AST"
  ]
  edge [
    id 8
    source 0
    target 9
    type "FLOWS_TO"
    label "FLOWS_TO"
  ]
  edge [
    id 9
    source 0
    target 9
    type "CDG_EPSILON"
    label "CDG_EPSILON"
  ]
  edge [
    id 10
    source 8
    target 9
    type "AST"
    label "AST"
  ]
  edge [
    id 11
    source 0
    target 9
    type "DDG_U"
    label "t"
  ]
]
```


Figure 91: GML format CPG output for method add from Figure 84

```
digraph MultiThrowable_CPG {
 // graph-vertices
 v1 [label="41:methodDeclaration:void add(Throwable t)";]
 v2 [label="41:modifier:public";]
 v3 [label="41:return:void";]
 v4 [label="41:identifier:add";]
 v5 [label="41:formalParameterList:(Throwable t)";]
 v6 [label="41:formalParameter:Throwable t";]
 v7 [label="41:type:Throwable";]
 v8 [label="41:identifier:t";]
 v9 [label="41:MethodBlock:null";]
 v10 [label="42:statementExpression:throwables.add(t)";]
 // graph-edges
 v1 -> v2 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v3 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v4 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v5 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v5 -> v6 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v6 -> v7 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v6 -> v8 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v9 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v10 [arrowhead=empty, color=green, style=dashed, label="FLOWS_TO"];
 v1 -> v10 [arrowhead=empty, color=purple, style=dashed, label="CDG_EPSILON"];
 v9 -> v10 [arrowhead=empty, color=red, style=dashed, label="AST"];
 v1 -> v10 [style=bold, label=" (t)"];
 // end-of-graph
}
```

Figure 92: DOT format output for method add from Figure 84