Representation Learning for Vulnerability Detection on Assembly Code

Ashita Diwan

School of Computer Science

McGill University, Montreal

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science in Computer Science

Abstract

Software vulnerability detection is one of the most challenging tasks faced by reverse engineers. Recently, vulnerability detection has received a lot of attention due to a drastic increase in the volume and complexity of software. Reverse engineering is a time-consuming and labor-intensive process for detecting malware and software vulnerabilities. However, with the advent of deep learning and machine learning, it has become possible for researchers to automate the process of identifying potential security breaches in software by developing more intelligent technologies. This has indeed opened a new paradigm for researchers in the area of software security, and it has helped to alleviate the cumbersome process of reverse engineering. In this research, we propose *VDGraph2Vec*, an automated deep learning method to generate representations of assembly code for the task of vulnerability detection. Previous approaches failed to attend to topological characteristics of assembly code while discovering the weakness in the software. VDGraph2Vec embeds the control flow and semantic information of assembly code efficiently using the expressive capabilities of message passing neural networks and the *RoBERTa*

model. Our model is able to learn the important features that help distinguish between vulnerable and non-vulnerable software. We carry out our experimental analysis for performance benchmark on three of the most common weaknesses and demonstrate that our model can identify vulnerabilities with high accuracy and outperforms the current state-of-the-art binary vulnerability detection models.

Abrégé

La détection des vulnérabilités logicielles est l'une des tâches les plus difficiles auxquelles sont confrontés les ingénieurs inverses. Récemment, la détection de vulnérabilités a reçu beaucoup d'attention en raison de l'augmentation drastique du volume et de la complexité des logiciels. L'ingénierie inverse est un processus long et laborieux pour détecter les logiciels malveillants et les vulnérabilités logicielles. Cependant, avec l'avènement de l'apprentissage profond et de l'apprentissage automatique, il est devenu possible pour les chercheurs d'automatiser le processus d'identification des failles de sécurité potentielles dans les logiciels en développant des technologies plus intelligentes. Cela a effectivement ouvert un nouveau paradigme pour les chercheurs dans le domaine de la sécurité logicielle, et cela a permis d'alléger le processus lourd de l'ingénierie inverse. Dans cette recherche, nous proposons *VDGraph2Vec*, une méthode d'apprentissage profond automatisé pour générer des représentations de codes assembleurs pour la tâche de détection de vulnérabilité. Les approches précédentes ne tenaient pas compte des caractéristiques topologiques des codes assembleurs lors de la découverte des faiblesses du logi-

ciel. VDGraph2Vec intègre efficacement le flux de contrôle et les informations sémantiques des codes assembleurs en utilisant les capacités expressives des réseaux de neurones à passage de messages et le modèle *RoBERTa*. Notre modèle est capable d'apprendre les caractéristiques importantes qui permettent de distinguer les logiciels vulnérables des logiciels non vulnérables. Nous effectuons notre analyse expérimentale pour une comparaison de performance sur trois des faiblesses les plus courantes et nous montrons que notre modèle peut identifier les vulnérabilités avec une grande précision et qu'il surpasse les modèles binaires de détection des vulnérabilités actuellement à la pointe de la technologie.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, *Prof. Benjamin Fung*, for his invaluable contributions and generous suggestions during the research. I appreciate his kindness, patience, and the amount of time he spent to provide his insightful feedback. I feel privileged and honoured to have worked under him, and to be part of the *Data Mining and Security (DMaS)* lab.

I am immensely grateful to *Miles Li* and *Prof. Steven Ding* for guiding me at each step with their abundant pool of knowledge and expertise in the topic. I would also like to thank *Guillaume Breyton* for helping me with the French abstract, and all my colleagues in the DMaS lab for the fruitful discussions in our weekly meetings.

Last but not least, I would like to specially thank my family and friends for their unconditional love and support. They kept me motivated throughout my research, especially during the pandemic.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's digital era, massive volumes of open source software code are readily available on the Internet. They are susceptible to malicious use by hackers; hence it has become easy to exploit the vulnerabilities present in these code, posing serious security threats to systems and users. Software vulnerabilities are defects or weaknesses in system design, implementation, or operation management that, if exploited, can lead to various attacks or can even cause the systems to crash [37]. The ramifications of these attacks and crashes can be outrageous and catastrophic. Each year, large numbers of software vulnerabilities are being detected in production software, either released publicly through the *Common Vulnerabilities and Exposures* (*CVE*) database[1] or internally discovered in proprietary code [32]. Thus, the detection of software vulnerabilities garners significant interest from the software security community. Traditionally, software vulnerabilities were detected by

[1]https://cve.mitre.org/

reverse engineering, which is the process of analyzing the design of software from its binary executables [65]. Reverse engineering is a complex and time-consuming process that requires expert knowledge and extensive experience [20]. Security experts often apply this technique to understand the software, especially when the source code is not available. However, this process is manually intensive, making it unfeasible, especially for mitigation of zero-day vulnerabilities. Therefore, we require automated tools to expedite the process for reverse engineers. Recent achievements in machine learning in computer vision, speech recognition, and natural language processing have encouraged researchers in cybersecurity to gauge its effectiveness for vulnerability detection.

There have been several advances in the recent literature on vulnerability detection using machine learning. The majority of the recent work in this direction focuses on the use of classical machine learning, requiring extraction of handcrafted features from the code [12]. Identifying the important features is time-consuming and requires immense human efforts as well. Deep learning has shown its prowess in automatically learning these features from the plain code. Thus, there have been several attempts to detect vulnerabilities using deep learning [6, 46, 47, 74]. Furthermore, researchers tend to detect vulnerabilities mostly at the source code level [12, 32, 61]. Despite the increasing amount of insightful work, vulnerability detection remains a challenging and arduous task, and we need more efficient automated approaches to tackle it, particularly when the source code is unavailable.

There have been a few research studies on vulnerability detection at the assembly code level [15, 42]. Though these studies offer promising results, they are tailored to apprehend only to the semantics of the binaries by capturing the relationships between the different tokens in an assembly instruction as an embedding. These studies do not incorporate the important information available in the topological structure of the assembly code. Thus, we propose to identify the vulnerabilities in software at the assembly code level through deep learning by capturing its meanings as well as structure.

In this thesis, we perform all our experimental analysis at the binary level. Using a disassembler, we can easily disassemble binary executable files to their corresponding assembly code required for our task. We explore a novel representation learning approach that leverages graph neural networks [62]. We focus explicitly on *Message Passing Neural Networks* (*MPNN*) [27], which have achieved state-of-the-art performance in various tasks. Our research highlights that by employing them, we are able to generate improved representation of our code as for each node, they aggregate the messages from all its neighbors. We also ensure that semantically similar instructions have embeddings close to each other by using the capability of a pre-trained Transformer model, *RoBERTa* [49]. This is the first work that utilizes the RoBERTa model for assembly instruction representation. To encapsulate, the workflow of our model is organized as follows: i) Disassembling the software and creating the control flow graphs of the assembly codes, ii) Generating the initial

basic block embeddings using RoBERTa, iii) Using MPNN to generate representations of the entire assembly code, and iv) Detecting the vulnerabilities using those embeddings. We also compare the performance of our model with the state-of-the-art for vulnerability detection. This research also seeks to address some of the additional questions raised in those previous studies and we show our proposed solution is superior to the state-of-the-art solutions. The primary focus of the study is a representation learning problem where we are trying to generate effective representations of assembly code. Additionally, in this research we work on the task of binary vulnerability detection; it is possible to further extend this study by proving the efficiency of these vector representations for other downstream tasks such as binary clone detection [18, 21, 22, 59].

Specifically, our contributions are:

- We propose a novel approach for assembly code representation. It is the first work that employs a hybrid structural and semantic representation learning model at the assembly code for vulnerability detection.

- Our model, *VDGraph2Vec*, is able to generate a latent representation for the entire assembly code, rather than for just an assembly function. It is easier to utilize it for both function-level and code-level analysis. Previous approaches mostly cater to representation at the function-level [19]. Also, this approach is especially useful when source code is unavailable.

- Extensive experiments on publicly available datasets illustrate the efficacy of the semantic and structural components of our proposed model. We capture the semantics by using a language model to learn the instruction embeddings. Further, we demonstrate that using a control flow graph with a message passing neural network helps in attaining enhanced learned representations. By combining these two aspects, our model significantly outperforms current state-of-the-art vulnerability detection methods at the assembly code level.

The rest of the thesis is structured as follows: Chapter 2 discusses the relevant work in the literature and how our model differs from the current models. Chapter 3 provides a formal formulation for the research problem. In Chapter 4 we describe the relevant concepts and systematically present our representation learning model. In Chapter 5 we elaborate on our experimental results and analysis. Finally, Chapter 6 summarizes our thesis and suggests future directions of research.

Chapter 2

Related Work

In this chapter we explore a review of recent works on vulnerability detection, source code, and assembly code representations. Due to the severity of the problem, researchers are developing automated methods for detecting vulnerability detection. Previous work on vulnerability detection is mostly at the source code level. In our work, we investigate vulnerability detection as a representation learning problem at the assembly code level. We primarily focus on the research works that employ machine learning and deep learning, and we explain how our proposed methodology differs from the existing literature.

2.1 Vulnerability Detection

With the advances in machine learning, it has become pivotal to assess its capability in the field of cybersecurity. *Harer et al.* [32] elucidated on two approaches to

detect vulnerabilities in C/C++ code. The first uses features obtained from the intermediate representation, while the second operates directly on source code. The authors used Clang and LLVM [38] tools to extract the control flow graphs to obtain features of the operations and variables. They also implemented a custom C/C++ lexer to get the representations of the tokens, and then converted the lexed tokens into their vector representations using *Bag-of-Words* and *Word2Vec* [52] representations. They further used a *TextCNN* [35] for learning more enhanced features along with an extremely randomized trees classifier [26]. *Russell et al.* [61] proposed a vulnerability detection tool based on deep feature representation learning. They created a custom C/C++ lexer to capture the relevant meanings of the 156 critical tokens as useful features. For generating the embeddings of these tokens, the authors used Word2Vec [52] and then employed convolution and recurrent feature extractors. Using the neural features as inputs, they finally applied the random forest classifier [8] to classify vulnerabilities. Additionally, the authors built an extensive C/C++ source code dataset, called *Draper*, which is mined from Debian and GitHub repositories. They labeled the dataset using three static analysis tools, and combined these findings with the SATE IV Juliet Test Suite to create a large dataset. *Chernis and Verma* [12] demonstrated the effectiveness of extracting text features from functions in C source code and analyzing them with a machine learning classifier. Their experimentation shows that simple features (character count, entropy, and arrow count) achieve a better accuracy than complex features (char-

acter n-grams, word n-grams, and suffix trees). Several researchers also presented comprehensive surveys outlining automated ways to detect software vulnerability [45, 48, 80].

Li et al. [47] developed *VulDeePecker* that relies on the generation of code gadgets, which are a group of semantically related program statements. These code gadgets are transformed into symbolic representations that are used for detecting vulnerabilities using *Bidirectional Long Short-Term Memory* [28]. It was found that deep learning provides higher accuracy compared to pattern-based and code-similarity-based vulnerability detection systems. They also introduced the *SySeVR* [46] framework, which focuses on obtaining program representations that can accommodate syntactic and semantic information pertinent to vulnerabilities by leveraging the abstract syntax trees and program dependency graphs. They conducted empirical studies to show the potential of a *Bidirectional Gated Recurrent Unit* (*BGRU*) [51] for vulnerability detection. A major contribution from the authors is a vulnerability detection dataset [1], collected from two data sources, the *National Vulnerability Database* (*NVD*)[2] and the *Software Assurance Reference Dataset* (*SARD*)[3].

At the assembly code level, *Zheng et al.* [81] proposed a study to evaluate the performance of *recurrent neural networks* (*RNNs*) [60] for binary vulnerability detection. They performed their experimental analysis on four types of vulnerabilities

[1]https://github.com/SySeVR/SySeVR
[2]https://nvd.nist.gov/
[3]https://samate.nist.gov/SRD/index.php

using RNNs such as simple RNN (SRNN) [60], bidirectional SRNN (BSRNN) [63], long short-term memory (LSTM) [33], bidirectional LSTM (BLSTM) [28], gated recurrent unit (GRU) [13] and bidirectional GRU (BGRU) [51]. Their results highlighted that BSRNN is a better RNN model for vulnerability detection as compared to the other models. *Dahl et al.* [15] conducted research that demonstrates the feasibility of RNNs for stack based buffer overflow vulnerability detection. They generated their own dataset by defining safe and vulnerable functions, built around the C system calls. They hypothesized that assembly code can be treated as natural language, and to assess this hypothesis, they applied RNNs on the vulnerability data to capture the differences between the functions based on their context. They concluded that not very deep RNNs were able to satisfactorily detect the stack-based vulnerabilities.

2.2 Source Code Representation

Recent research demonstrates the success of message passing neural networks for source code representation. *Zhou et al.* [83] explored the efficacy of using *Graph Neural networks* for detecting software vulnerabilities by developing a model called *Devign*. Their model encodes the raw source code of a function into a joint graph structure consolidating the syntax via *abstract syntax trees* (*AST*) and semantics via dependency and control flow graphs. The authors implemented a gated graph neural network [44] model for generating the embeddings of each node. This is fur-

ther utilized by the Conv module for graph-level classification. In another work, *Wang et al.* [73] constructed graph representation of programs called *flow-augmented abstract syntax tree (FA-AST)* for detecting code clones. They generated FA-AST by augmenting the abstract syntax trees with explicit control and data flow edges. The authors applied two different types of graph neural networks, gated graph neural networks [44] and graph matching networks [43], on FA-AST to measure the similarity of code pairs. A prominent work by *Allamanis et al.* [3] introduced strategies to learn program structures using graph-based deep learning. They demonstrated the scalability of gated graph neural networks on two tasks, *VARNAMING*, which predicts the name of the variable depending on its usage, and *VARMISUSE*, which seeks to select the correct variable that should be used at a given program location. In this work, programs are represented as graphs by capturing the syntax and semantic relationships between the tokens using different edge types from AST. Our proposed work is significantly different from these approaches because we work at the assembly code level. We aim to improve the performance of graph neural networks models by experimenting with different pre-trained models such as RoBERTa [49] for the initial node representations of the basic blocks.

2.3 Assembly Code Representation

Since an assembly code shares some commonalities with natural text, researchers often employ natural language processing models on programs. *Lee et al.* [42] intro-

duced *Instruction2Vec*, a framework for modeling assembly code. It is an improved version of the Word2Vec [52] model that considers the syntax of the assembly code as well. It uses Word2Vec to generate a lookup table, through which each instruction is represented as a fixed dimension vector containing an opcode and two operands. Furthermore, their model deliberates on the potential of TextCNN [35] for detecting software vulnerabilities. Another research work that deliberates on using the word2vec model for embedding the assembly instructions is by *Redmond et al.* [58]. This work explored techniques adopted from natural language processing to jointly learn multilingual word embeddings [50], and adapted it for cross-architectural binary code analysis. *Ding et al.* [19] proposed an assembly code representation method based on the PV-DM model [39] incorporating the rich semantic information between the tokens. However, all these approaches are only catering to the semantics of the code and not to the actual flow of code execution.

Researchers are now utilizing graph embedding networks to learn representations of assembly functions. *Genius* was one of the first scalable graph based bug search model for firmware images, implemented by *Feng et al.* [23]. The model converted the CFGs of the binary functions to high-level numeric feature vectors and performed the searching by using state-of-the-art hashing techniques with the learned feature vector, rather than performing pair-wise matching. Further, *Xu et al.* [77] developed a neural-based graph embedding model for cross-platform binary code similarity detection, called *Gemini*. Gemini utilized the control-flow

graph and represented it with attributes attached to each node, and used structure2Vec [66] as a graph embedding network to convert the graph into an embedding for similarity detection. They did it by combining the graph embedding network into a Siamese network [10], which captures the objective that similar graph embeddings should be close to each other. *Yan et al.* [78] presented a model that uses *deep graph convolutional neural network (DGCNN)* to embed the structural information inherent in a CFG for malware classification. They also first convert their CFG into attributed CFG (ACFG), where each vertex is represented by certain block-level attributes. Finally, DGCNN is applied on the graph data which transforms it for classification.

Baldoni et al. [5] proposed an unsupervised feature learning approach to automatically extract features from a CFG of an assembly function. Their graph embedding network is the union of two main elements, the vertex feature extraction and structure2Vec network. The first element is responsible for associating a feature vector with each vertex, and the second element combined the feature vectors through a deep neural architecture to generate the final embedding of the graph. For the feature extractor, they compared manual feature engineering with unsupervised feature learning ideas adopted from natural language processing. The authors investigated two instruction embedding aggregation techniques for generating vertex embeddings. They utilized the Word2Vec [52] model (i2v) for obtaining instruction embeddings and combined them using, attention *(i2v_attention)* and

RNN *(i2v_RNN)*. Further, they used a structure2vec deep neural network module for updating the vertex vectors according to the graph topology and vertex features. The final graph embedding is generated by aggregating the vertex vectors obtained after several rounds. The authors conducted an experimental study to evaluate their model on the tasks of binary similarity and compiler provenance. We use the model described in this paper as one of our baselines, and we compare its performance with our model on the task of vulnerability detection.

As compared to these research methods, we analyze the assembly code for the entire file, instead of restraining to only a function. Moreover, our research caters to the task of vulnerability detection. It is also the first work to experiment with RoBERTa for initial node embeddings, which is further used by a message passing neural network.

Chapter 3

Problem Description

In this chapter we provide a formal definition to our problem along with the used notations. The input to our model is a binary file. Using a disassembler, we can retrieve the *Control Flow Graph* (*CFG*) of a function. To construct the CFG of the entire program, we create edges between the basic blocks of a function that call the other function's blocks. For the input to our Message Passing Neural Network (MPNN), we need to represent our graph as $G = (X, E)$, where X is the set of the initial representations of the basic blocks in the CFG, and E is the set of edges between the basic blocks. Each basic block v is represented by a feature vector x_v. The sequence of instructions in a basic block, I_v are mapped to their corresponding feature vector x_v by an embedding function f_E.

$$f_E(I_v) = x_v$$

After obtaining the initial block embeddings (x_v) and edge connections between the blocks, we apply a graph neural network f_g that transforms the block embeddings ($x_v^{'}$). Further, we apply a global pooling (readout) layer to generate the embedding for the entire control flow graph, θ_g, which is used for our downstream task of binary vulnerability detection to yield a label, $\hat{y} \in \{0, 1\}$.

$$f_g(G) = \hat{y}$$

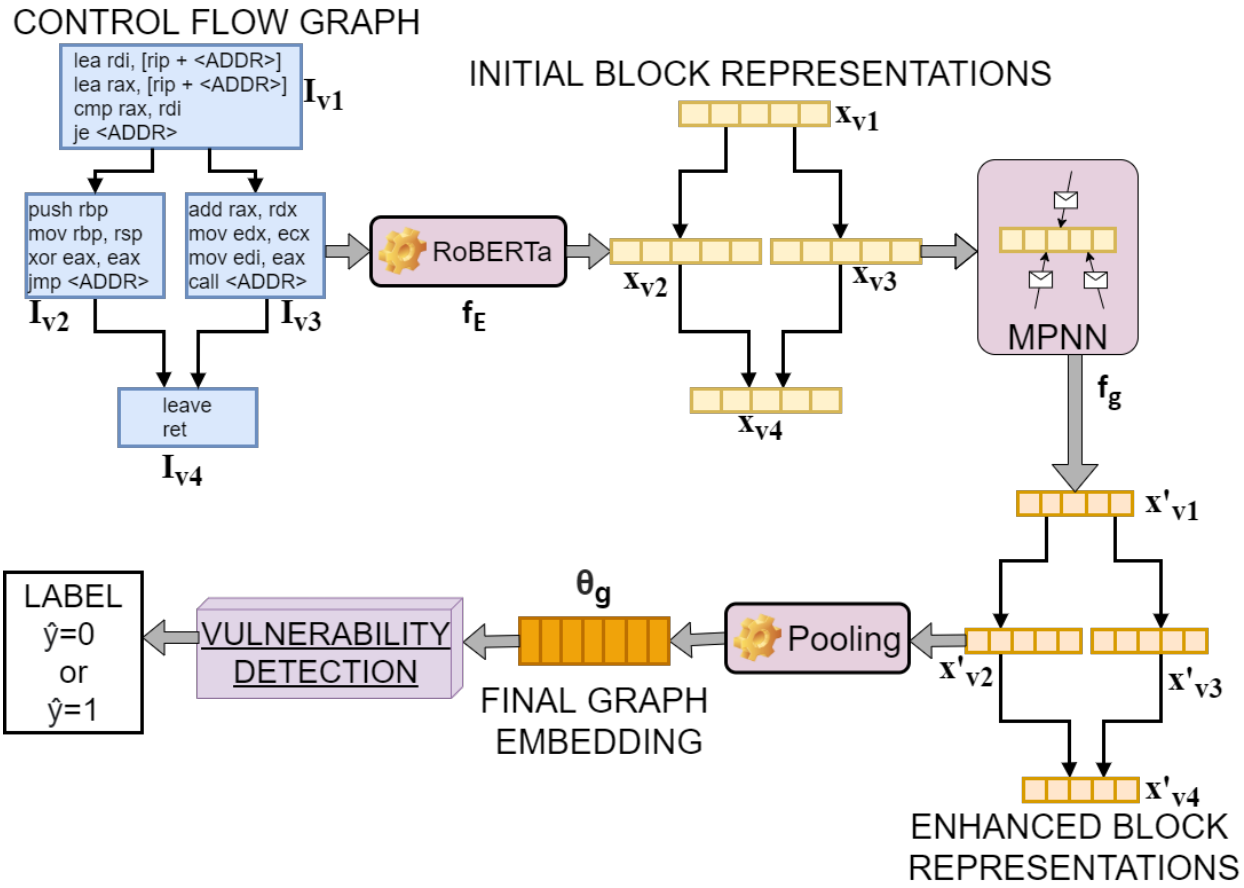


Figure 3.1: Workflow of our VDGraph2Vec model.

The workflow of our VDGraph2Vec model is illustrated in Figure 3.2. Finally, we define our vulnerability detection research problem as follows,

Definition 1 (Vulnerability Detection)**.** Consider a collection of binary files B along with their labels Y signifying whether the binary files contain a certain type of vulnerability or not. Let b be an unknown binary such that $b \notin B$. The vulnerability detection problem is to build a classification model M based on B and Y such that M can be used to determine whether the binary, b, is vulnerable ($\hat{y} = 1$) or non-vulnerable ($\hat{y} = 0$). ■

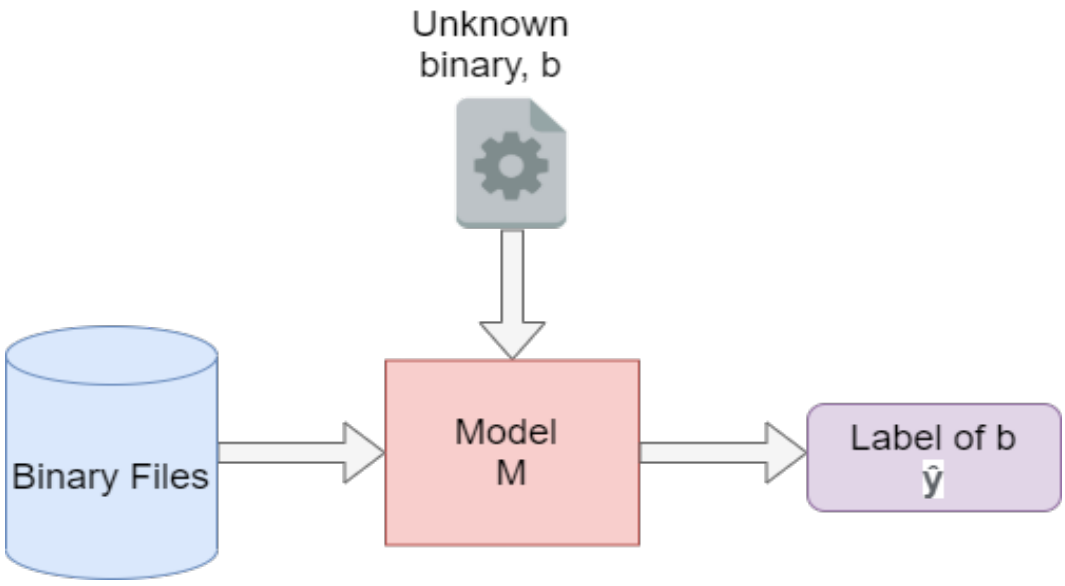


Figure 3.2: Vulnerability detection as a binary classification problem.

Chapter 4

Graph-based Assembly Code Representation Learning for Vulnerability Detection

Before diving into the experimental section, we first explicate more on graph representation learning and how it can be leveraged for efficient assembly code representation and detection of software vulnerabilities. In this chapter we discuss the preliminaries requisite for understanding the model.

4.1 Preliminaries

Given a dataset in binary format, first the files are disassembled into their equivalent assembly code. A disassembler translates the binary machine code into as-

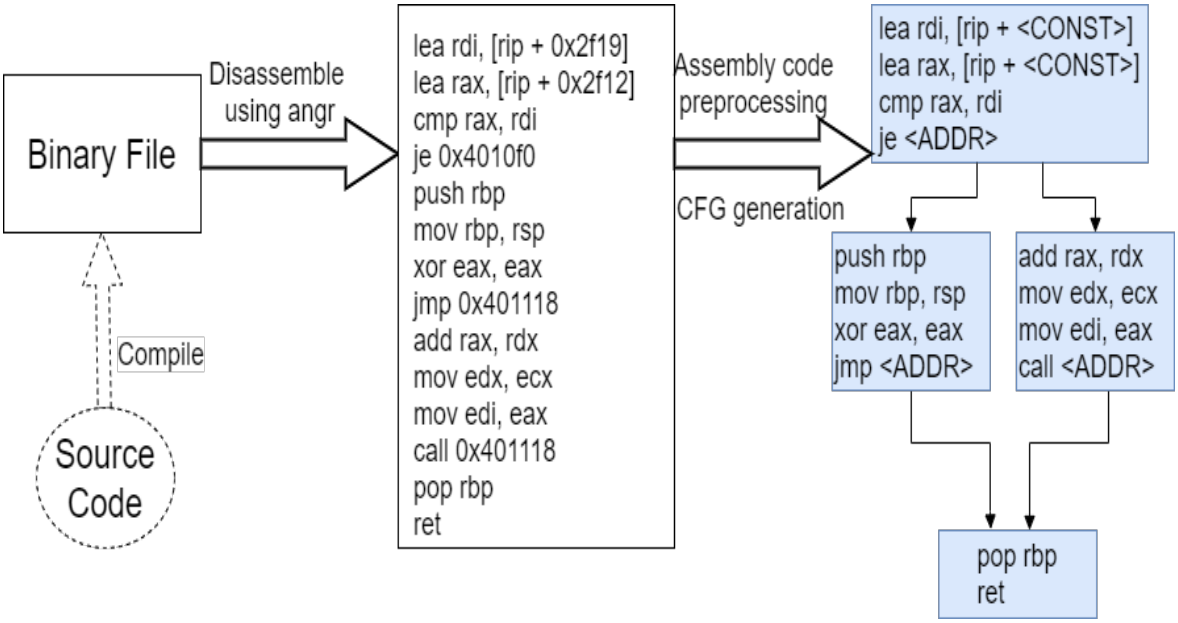


Figure 4.1: Extraction of CFG and its preprocessing using angr.

sembly code. There are a variety of disassemblers including *IDA Pro*[1] and *Ghidra*[2] that can be used for obtaining the CFG. We use *angr*[3] in our research because it is open source, and hence it is easier to replicate results. Therefore, we start by extracting the Control Flow Graph (CFG) of the program. A CFG [14] is a graphical representation of the different execution paths of a program. Each basic block (a group of sequential statements) of the control flow graph is represented by a node. The edges of the graph connect basic blocks that can flow into each other during execution. The process of extracting CFG using angr is depicted in Figure 4.1. At the high level, this representation is especially beneficial for vulnerability detection because it has the ability to uncover risky and unsafe program execution

[1]https://www.hex-rays.com/products/ida/
[2]https://ghidra-sre.org/
[3]https://angr.io/

topologies. Further, we use a Message Passing Neural Network (MPNN) [27] to obtain the representation of the assembly code. Conceptually, it works better because for each node, it accumulates the messages (hidden layer representations) from all of its neighbors and aggregates them to get the final node embeddings. In order to pass the CFG as an input to the MPNN, we need to represent the sequence of instructions in the basic block, I_v, as an embedding. Pre-trained language models [57] have achieved impressive results for various tasks in both natural language processing and in source code representation [24]. To capture the meaning of the assembly instructions, we utilize pre-trained language models (f_E) to extract the initial block embeddings, x_v. In the machine learning community, we generally believe that the semantics of a token (e.g., word, sentence, or instruction, function) is captured by its relationships with other tokens. Embedding captures the relationships, hence we often mention that the embedding captures the semantics, but the software engineering community may have a different interpretation on the term "semantic". Assembly code follows a grammar for writing the instructions and relationship between the operation and operands is important to be captured by the embeddings. We do not want to lose the semantic information available in the assembly instructions. In this work, we use semantics to imply that our embeddings are taking into account the crucial relationships among different tokens in an assembly instruction.

4.2 Graph Neural Networks

Neural networks have shown remarkable progress in computer vision [72], natural language processing [79], and speech recognition [53]. However, we may find other complex data which does not fall under the domain of images, text or speech. If we consider the case of social media networks, the data is highly irregular but it can be easily represented as graphs with nodes as the users and the edges as connections. Thus, this complexity in data structures led to several advancements in machine learning with the introduction of graph neural networks. Owing to the immense expressive power of graphs, these graph representations are extremely useful for non-euclidean data, and hence graph neural networks have attained state-of-the-art results for many tasks [82].

Before probing into graph neural networks, we first explain the representation of graph. A graph $G = (X, E)$ where E is the set of edges and X is the set of nodes, which can either be directed or undirected based on the dependencies between different nodes. Furthermore, a graph can be homogeneous or heterogeneous. Nodes and edges in homogeneous graphs have same types, while nodes and edges have different types in heterogeneous graphs. A graph structure can be a useful representation for many domains — social media networks, molecules, recommender systems, knowledge graphs, etc. These graphs can contain large numbers of edges and nodes, hence requiring them to be represented as latent feature vectors without losing the crucial information inherent in their hierarchical graphical structure.

Therefore, in order to achieve this, we need to adopt machine learning techniques for encoding the nodes, edges, and graphs. Hence, there can be three kinds of loss functions and learning tasks associated with graphs: *node-level*, *edge-level*, and *graph-level*. Node-level tasks focus on nodes, which include node classification, node regression, node clustering, etc. Edge-level tasks include edge classification and link prediction. Finally, graph-level tasks include graph classification, graph regression, and graph matching. GNNs are neural networks that can be directly applied to graphs, and provide an easy way to do node-level, edge-level, and graph-level prediction tasks.

4.2.1 Node embeddings

The rationale behind the concept of node embeddings is that we want to map nodes to a d-dimensional embedding space such that similar nodes in the graph are embedded close to each other. Therefore, if z_u and z_v are the feature vectors associated with the neighboring nodes u and v in the d-dimensional space, we want to define the encoding function such that $similarity(u, v) \approx z_u^T z_v$ (similarity of the embedding). Nodes that are connected by an edge between them are considered as neighboring nodes in the graph. This concept is illustrated in Figure 4.2. We can think of it as an encoder-decoder architecture, where the encoder maps the nodes to embeddings and the decoder maps from embeddings to the similarity score.

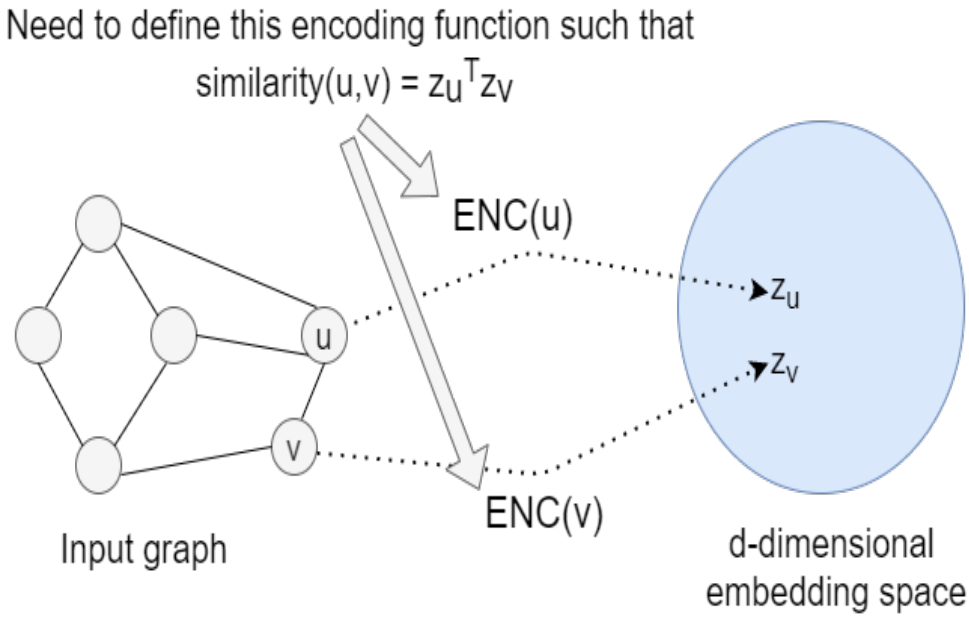


Figure 4.2: The concept of defining the encoding function for node embeddings such that neighboring nodes in a graph have their feature vectors close to one another in the d-dimensional space as well.

Shallow Embeddings

DeepWalk [56] is the one of the first proposed algorithms to learn node embeddings in an unsupervised manner. It is based on the idea that the distribution of nodes in a graph follow a power law [1]. It resembles the word embedding in terms of the training process. DeepWalk uses local information obtained from truncated random walks to learn latent representations. It consist of two phases: 1) identifying the context of a target node and 2) learning embeddings that maximize the likelihood of predicting context nodes. The method that is used to make predictions is skip-gram model [52]. DeepWalk regards the nodes in the network as words in the sentence, and the sequence obtained by the random walk of the network cor-

responds to the sequence of sentences. However, unlike words in a sentence, the order of the nodes in a context window for the random walks is not important.

Grover and Leskovec [29] introduced the idea of flexible and biased random walks that can trade off between the local and global views of the network. The *Node2Vec* model uses parameters that allow the random walk probabilities to smoothly interpolate between walks like the breadth-first search (BFS) or depth-first search (DFS) over the graph. BFS is ideal for learning local neighbors, while DFS is better for learning global variables. Node2Vec model also emphasizes that a good network representation learning algorithm must satisfy two conditions: 1) nodes in the same community should be similar, and 2) nodes with similar structural features indicate similarity. These two characteristics are called homogeneity and structural similarity, respectively.

However, these node encoding models utilize shallow encoders that are inherently transductive and the encoders do not incorporate node features. These methods can only generate embeddings for nodes that were present during the training phase [30]. To overcome these aforementioned problems, GNNs were proposed to encode the information in a graph inductively. Some of the popular graph neural networks include *Graph Convolutional Network* (*GCN*) [36], *GraphSAGE* [31], and *Graph Attention Network* (*GAT*) [71]. Unlike the shallow node embedding methods, the GNN framework requires that we input the node features $x_v, \forall v \in X$ to the model.

Graph Neural Networks

The GNNs generate node embeddings based on *local network neighborhoods*. Nodes aggregate information from their neighbors using neural networks, and every node defines its computation graph based on its neighborhood. This is known as *neighborhood aggregation*. A basic approach for it is to average neighbor messages for each node, and apply a neural network. For node v in a graph G with node features x_v and its neighbors $N(v)$, its hidden state at layer $t+1$, h_v^{t+1} is given by,

$$h_v^{t+1} = \sigma\left(W_t \sum_{u \in N(v)} \frac{h_u^t}{|N(v)|} + B_t h_v^t\right)$$

where σ is a non-linear activation function, and W_t and B_t are trainable weight matrices that we learn. W_t is the weight matrix for neighborhood aggregation and B_t is the weight matrix for transforming the hidden vector of self. The initial layer embeddings are initialized to the node features, hence

$$h_v^0 = x_v$$

GraphSage

Hamilton et al. [31] introduced more flexible aggregation with GraphSage. Instead of mean, it provides a simple aggregation function and concatenates the neighbors embedding with its own embedding. The hidden layer at layer $t+1$ is given by,

$$h_v^{t+1} = \sigma\left[W_t \cdot AGG(h_u^t, u \in N(v)), B_t h_v^t\right]$$

Some of the popular variants for the aggregation function (*AGG*) are:

- **Mean** - It takes a weighted average of neighbors,

$$AGG = \sum_{u \in N(v)} \frac{h_u^t}{|N(v)|}$$

- **Pooling** - It transforms the neighbor vectors and applies a symmetric vector function,

$$AGG = \gamma(MLP(h_u^t), \forall u \in N(v))$$

  where γ is element-wise mean or maximum operation.

- **LSTM** - It applies LSTM [33] to reshuffled neighbors,

$$AGG = ([LSTM(h_u^t), \forall u \in \pi(N(v))])$$

Graph Attention Networks

Velvickovic et al. [71] argued that all neighbors of a node are not equally important. The authors introduced graph attention networks in which attention weights are used to specify importance of the different neighbors of each node in the graph. If

α_{vu} represents the attention weight of neighbor u for node v, then the hidden layer at layer $t+1$ is given by,

$$h_v^{t+1} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} W_t h_v^t\right)$$

The *attention* mechanism was first introduced by *Bahdanau et al.* [4] for neural machine translation. Prior to that, neural machine translation was based on encoder-decoder RNNs [68]. These models are also known as *sequence-to-sequence* (*seq2seq*). Both the encoder and decoder are stacks of RNN units. The encoder is used to read the input sentence and encodes it into a context vector which summarizes the input. This context vector is passed to the decoder which translates it to a sequence. However, it fails to capture the long-range dependencies. *Cho et al.* [13] also demonstrated that the performance of the encoder-decoder network degrades as the length of the input sequence increases. Attention mechanism was proposed as a solution to overcome the problem. For generating a sentence, the model searches for a set of positions in the encoder hidden states where the most relevant information is available. The attention mechanism retains and utilizes all the hidden states of the input sequence during the decoding process. Given the impressive improvement in results for machine translation by attention, its effectiveness was also gauged in the field of computer vision [76]. In the recent times, researches have experimented and explored various other forms of attention mechanisms [9, 11, 50, 70].

For GNNs, the attention weights α_{vu} are computed using the attention coefficients e_{vu} across the pair of nodes u and v based on their messages:

$$e_{vu} = a(W_t h_u^t, W_t h_v^t)$$

where a is the attention mechanism and e_{vu} indicates the importance of neighbor u for node v. The attention coefficients are normalized into the final attention weights using the softmax function such that $\sum_{u \in N(v)} \alpha_{vu} = 1$,

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

The attention mechanism a has trainable parameters, which can be learned jointly with the weight matrices in an end-to-end manner. Additionally, we can use the multi-head attention approach also which will use K heads instead of one and finally take the average (or concatenate) of all the feature vectors learned from the K heads.

Next, we discuss the MPNN framework and its two popular algorithms that we use in our research.

4.2.2 Message Passing Neural Networks

MPNN [27] is a popular framework that generalizes most graph neural models based on the idea of getting enhanced node representations by aggregating infor-

mation from the neighbors. An overview of the concept of message aggregation is illustrated in Figure 4.3.

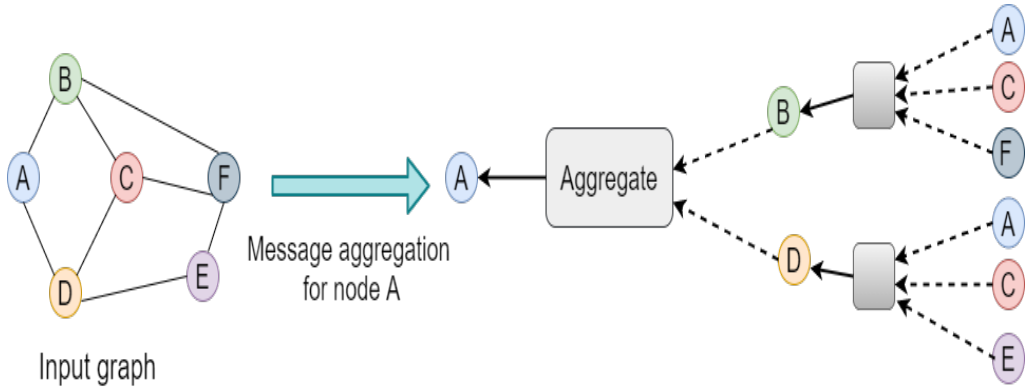


Figure 4.3: A 2-layer message aggregation in an MPNN framework. It provides an overview of how a single node aggregates messages from its local neighborhood.

The architecture of an MPNN consists of two primary phases: a message phase and a readout phase. A graph G has node features x_v and edge features e_{vw}. At every time step t, a node has an associated hidden state, h_v^t. During the message passing phase, the hidden states are updated based on the messages m_v^{t+1} obtained from the neighbors.

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw})$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1})$$

The readout phase computes the feature vector for the entire graph using a readout function R.

$$\hat{y} = R(\{h_v^T | v \in G\})$$

The message update function M_t, the node update function U_t, and the readout function R are all learned differentiable functions. This framework is quite robust because it provides the feasibility to use different messages and update functions. In this work, we perform our experiments specifically with two graph neural networks, *Graph Convolutional Networks* (*GCN*s) [36] and *Gated Graph Neural Networks* (*GGNN*s) [44].

4.2.3 Graph Convolutional Network

GCN generalizes the idea of *convolutional neural networks* (*CNN*s) [41] to the complex graph networks. In a GCN layer, the weights are shared for all nodes, and the feature vector for a node is computed by performing mathematical operations on its neighborhood nodes. If A is the adjacency matrix for the graph, and x_v represents the initial representation for node v, GCN computes the feature vector h_v^{t+1} for $t+1$ layer as,

$$h_v^{t+1} = \sigma\big(\sum_{w \in N(v)} h_w^{(t)} W^{(t)}\big) = \sigma\big(A h_w^{(t)} W^{(t)}\big)$$

where $W^{(t)}$ is the weight matrix used for the layer t, and $h_v^0 = x_v$.

As we can see, the weights are shared for all the nodes in a given layer similar to conventional convolutional filters, and the feature vector of a node is computed upon performing some mathematical operation on its neighbourhood nodes like in a CNN. CNNs cannot be directly applied on graphs because of the complex

topology of the graph, implying that there is no spatial locality. Moreover, two problems are evident with the update function here: 1) while computing the feature vector for a node, we do not consider its own feature vector unless a self-loop is there, and 2) the adjacency matrix used here is not a normalized one, hence it can cause scaling problem or gradient explosion due to the large values of the parameters.

To overcome these problems,

- The self loops are enforced using the identity matrix I, which is of the shape of A,

$$\hat{A} = A + I$$

- The adjacency matrix A is normalized to avoid the scaling problem by,

$$\hat{A} = D^{(-1/2)}\hat{A}D^{(-1/2)}$$

where D is the diagonal matrix with the degrees of all nodes in $\hat{A}$.

In each layer, the information is passed to a node from its neighborhood resulting in neighborhood aggregation.

4.2.4 Gated Graph Neural Networks

Gated graph neural networks are used to build sequential models in which each node v is updated using the previous node state (h_v^t) and the current message state (m_v^{t+1}) with a *gated recurrent unit* (*GRU*) [13].

$$h_v^{t+1} = GRU(h_v^t, m_v^{t+1})$$

Further, we can experiment with other techniques such as *batch normalization* [34] to stabilize neural network training, *dropout* [67] to prevent overfitting, and *attention* to control the importance of a message.

GRU cell

A GRU is a variant of the recurrent neural network [13]. In fact, LSTM and GRU cells are proposed to address two main drawbacks of the recurrent neural network, which are to deal with long-term dependencies and with the vanishing and exploding gradient issue. GRU is similar to LSTM as it also uses gates to control the flow of information. However, unlike LSTM, it does not have a separate cell state C_t. It only has a hidden state h_t. Due to the simpler architecture, GRUs are faster to train. On the other hand, an LSTM cell also manages the cell state C_t, which is updated additively. In an LSTM, the hidden state h_t can be seen as short-term memory, while the cell state C_t can be interpreted as long-term memory. Furthermore, forgot and input gates control the information in both short and long-term memory.

GRU combines the forget and input gates into a single update gate in order to reduce computational cost while maintaining power of representation. Moreover, it merges the cell state and short-term memory into one hidden state. The differences in the architecture of the two cells is depicted in Figure 4.4.

(a) LSTM architecture **(b)** GRU architecture

Figure 4.4: Difference in the architectures of the LSTM and GRU cells. LSTM uses an additional cell state, whereas GRU uses only the hidden states.

An LSTM cell is composed of three gates: input gate, output gate, and forget gate. The detailed architecture of the LSTM cell is shown in Figure 4.5. If we represent hidden state as h_t, input as x_t, cell state as C_t, W_j and b_j as the weight and bias used for gate j, then the formal equations for the LSTM gates are given by,

- Forget gate: It is responsible for removing information from the cell state.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- Input gate: It is responsible for the addition of information to the cell state. It is done in three steps. First, a sigmoid layer called the input gate layer

decides the values to be updated.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

A tanh layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state.

$$\widetilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

Next, we need to update the old cell state C_{t-1} into the new cell state, C_t. This is done by multiplying the old state by f_t, forgetting the things decided by the forget gate. Then, the new candidate values are scaled by how much we update each state.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$$

- Output gate: It is responsible for deciding the output of the cell. A sigmoid layer is used, which decides the parts of the cell state that will go to the output.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Finally, a tanh layer is used for the cell state and it is multiplied by the output of the sigmoid gate.

$$h_t = o_t * tanh(C_t)$$

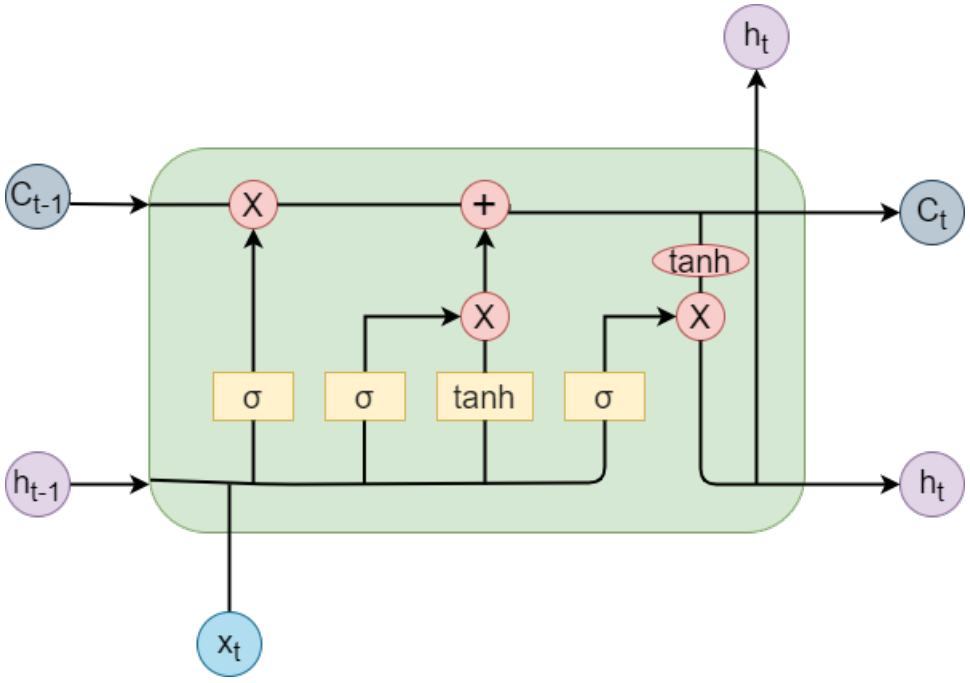


Figure 4.5: The architecture of an LSTM cell.

On the other hand, a GRU cell does not need to maintain the cell state. Instead, GRU uses only two gates: update gate and reset gate. These are the two vectors that decide on the information that should be passed to the output. The detailed architecture of the GRU cell is shown in Figure 4.6.

- Update gate: It helps the model to determine the past information that needs to be passed to the future.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

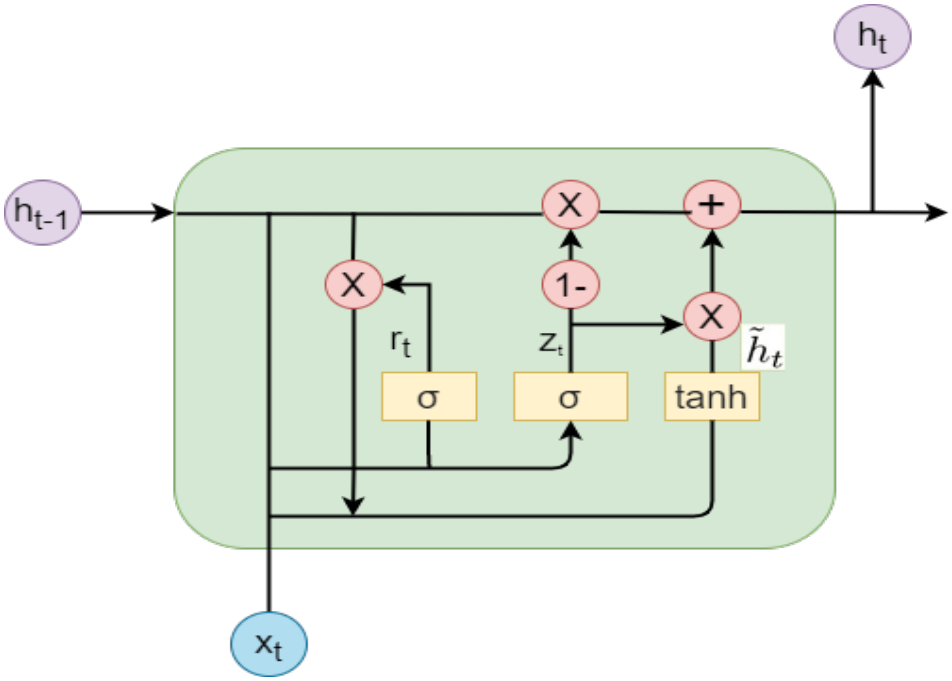


Figure 4.6: The architecture of a GRU cell.

- Reset gate: It is used to decide how much of the past information to forget.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

- Candidate hidden state: It takes in the input and the hidden state from the previous timestamp, which is multiplied by the reset gate output. A tanh layer is applied to it for the candidate hidden states.

$$\widetilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t])$$

- New hidden state: The new hidden state is calculated using the update gate, previous hidden state and the candidate hidden states.

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h}_t$$

  It is evident that only one gate controls how much information can come from the old hidden state and from the new state.

Therefore, in gated graph neural networks, a GRU cell is used which allows for neighborhood aggregation with recurrent state update. GGNNs are useful for complex networks representation including logical formulas and programs.

4.3 Word Embeddings

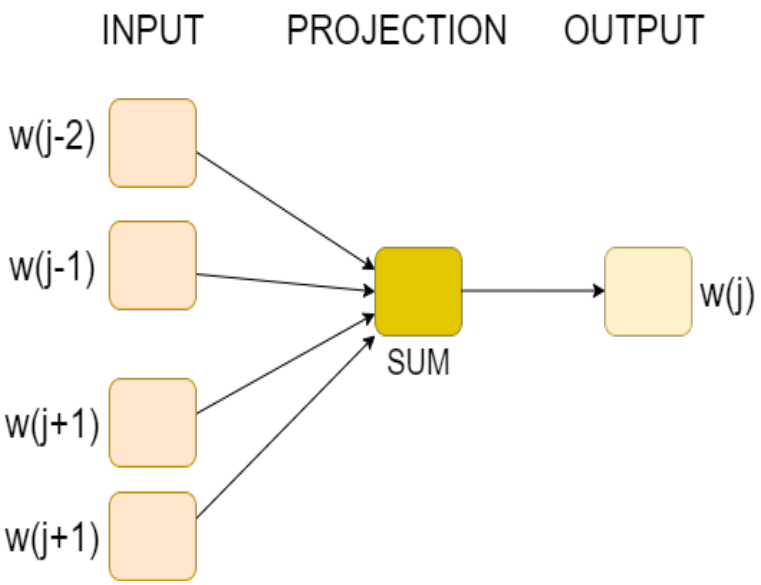
In natural language processing, we often employ methods to efficiently represent words as a meaningful feature vector, such that the complex relationships of a word in a text are captured carefully. These meaningful feature representations are termed as word embeddings. Initially, vector space models and other methods such as Latent Dirichlet Allocation (LDA) [7] and Latent Semantic Analysis (LSA) [16] were used for estimating continuous representations of words. Neural networks have taken over the mainstream NLP since 2014 with the introduction of Word2Vec [52] and GloVe [55] models for generating the word embeddings.

4.3.1 Word2Vec

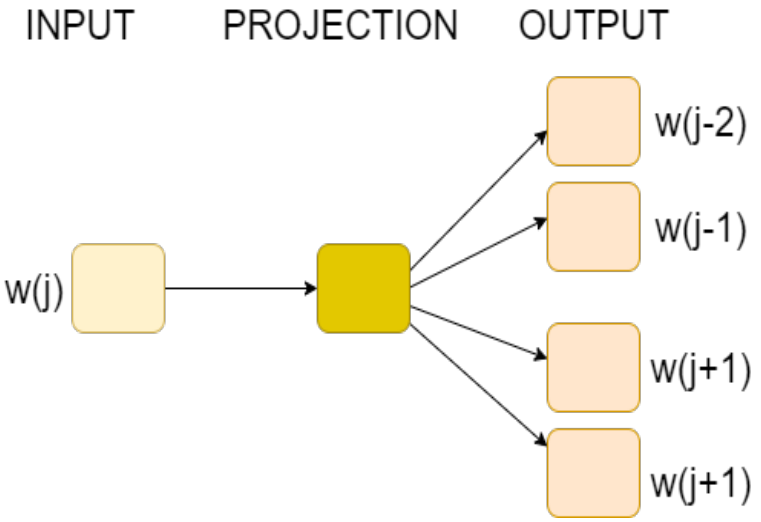
Word2Vec [52] is an extremely popular algorithm in natural language processing to capture dense learned representations of text in such a way that words with the same meaning tend to have similar representations. The word2vec model is based on the idea of distributional similarity. It is a shallow neural network with a single hidden layer, where the hidden layer is a fully-connected layer whose weights are the word embeddings. There are two types of methods described in this paper to learn a low dimensional feature vector for each word: *skip-gram* and *continuous bag-of-words (CBOW)* models. The idea of the skip-gram model is to use the current word to predict words around it. The continuous bag-of-words model works on the reverse principle, it predicts the current word on the basis of the neighboring words. In both skip-gram and CBOW, we learn the embedding of the target word. However, CBOW works well for smaller datasets, while the skip-gram model performs better with larger data. The architectures of both models are shown in Figure 4.7.

If we represent the target word vector as v_j, context word vectors as c_k, then using the skip-gram model we are trying to learn the following objective function to predict the context word w_k, given the target word w_j:

$$p(w_k|w_j) = \frac{exp(c_k|v_j)}{\sum_{i \in |V|} exp(c_i|v_j)}$$



(a) CBOW model architecture



(b) Skip-gram model architecture

Figure 4.7: Architecture of the Word2Vec models. The CBOW architecture predicts the target word based on the context, and the Skip-gram predicts surrounding words given the target word.

However, the denominator is too expensive to compute. To overcome this, we employ the negative sampling technique. For this, we sample a target word j, true

context word c, and some negative context words from the entire vocabulary. We try to optimize the function,

$$log(\sigma(c \cdot v_j)) + \sum_i E_{w_i \approx p(w)}[log(\sigma(-c_i \cdot v_j))]$$

where σ is the logistic function. Gradient descent is used to learn the parameters. The sampling procedure of negatives during the training impacts the performance of the model, and hence, $p^{\frac{3}{4}}(w)$ works better than the standard $p(w)$. The *word2Vec* pre-trained word embeddings are used for various downstream tasks like analogical reasoning. Despite this, the *word2Vec* model fails to capture differences like polysemy. This distributional similarity gives us a measure of relatedness, which often works well, but it suffers from the problem that antonyms and synonyms share similar distributional properties. To overcome this shortcoming, context informed word embeddings were introduced with *Transformer* [70] models.

4.3.2 Transformer

Transformer is a novel architecture that solves the problem of long-range dependencies with ease. It is the first transduction model which relies entirely on self-attention to compute representations of its input and output. The transformer model consists of encoders and decoders stacked up. Both the encoder stack and the decoder stack have the same number of units. The architecture of the encoder and decoder is shown in Figure 4.8. The encoder block has one layer of a multi-

head attention, followed by a layer of feed forward neural network. The decoder, on the other hand, has an extra multi-head attention, that helps the decoder to focus on relevant parts of the input sentence. As there is no recurrence, the position of each token in a sequence is represented with the positional encodings. The word embeddings of the input sequence are passed to the first encoder, where they are transformed and propagated to the next encoder. The output from the last encoder in the encoder-stack is passed to all the decoders in the decoder-stack.

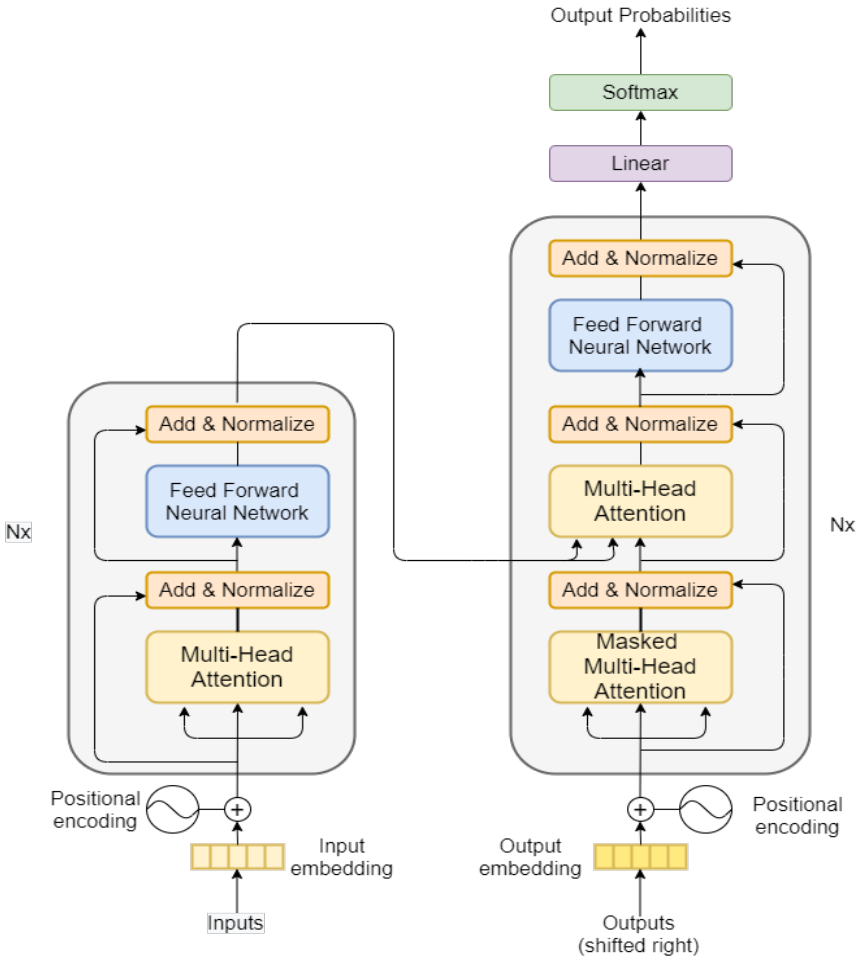


Figure 4.8: Transformer architecture.

Multi-headed attention-mechanism is used in creating the transformer model. To understand multi-head attention, we first explicate on self-attention. Self-attention is an attention mechanism that relates different positions of a single sequence in order to compute a representation of the sequence. For calculating the self-attention, we first create three vectors from each of the encoder's inputs: *Query Vector (Q)*, *Key Vector (K)* and *Value Vector (V)*. The query is the hidden state of the decoder. Key is the hidden state of the encoder, and the corresponding value is a normalized weight, representing how much attention a key gets. The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

However, this calculation is actually done using the matrix form for faster processing. The process to calculate the scaled-dot product attention is depicted in Figure 4.9.

Self-attention is computed not once but multiple times in the Transformer's architecture, in parallel and independently. It is referred to as Multi-head Attention. The attention mechanism is repeated h times with linear projections of Q, K and V, yielding d_o dimensional output values. The outputs are then concatenated, and once again projected. This allows the model to jointly attend to information using

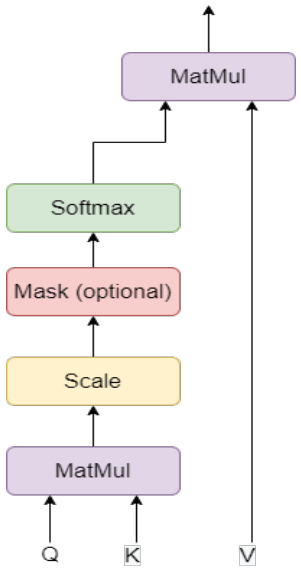


Figure 4.9: Scaled dot-product attention.

different key, value and query matrix for each encoder, thus creating different subspaces. The multi-attention mechanism is shown in Figure 4.10. Therefore, Transformer architectures can learn longer-term dependency by using the multi-head attention mechanism. *Al-Rfou et al.* [2] proposed to use the Transformer model for language modeling. The impressive results of transfer learning for computer vision motivated research for language model pre-training for improving the performance of natural language processing tasks. Next, we discuss the deep learning models that have given state-of-the-art results on a wide variety of natural language processing tasks which utilize the Transformer architecture.

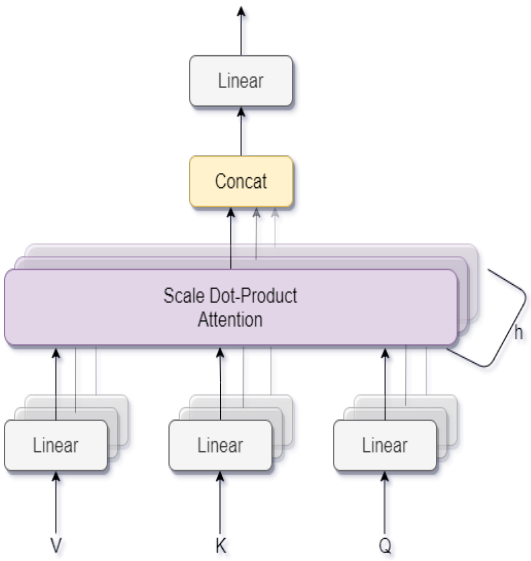


Figure 4.10: Multi-head attention.

4.3.3 RoBERTa

A model that revolutionized pre-trained language models in NLP is BERT [17]. The key innovation of the model is the bidirectional training of Transformer for language modeling. The model takes into consideration both left and right context of the words, resulting in more accurate feature representations. BERT utilizes the Transformer with its attention mechanism to learn contextual relations between words in a text. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. There are two steps in the framework: pre-training and fine-tuning. The model has been pre-trained on Wikipedia (2,500M words) and BooksCorpus (800M words) [84] over two pre-training tasks: *masked language modeling* (*MLM*) and *next sentence prediction* (*NSP*). The authors of this paper investigate a novel technique called MLM, in which some of the tokens from the

input are masked, and the objective is to predict the original vocabulary ID of the masked token. Before feeding word sequences into BERT, 15% of the words in each sequence are chosen at random. 80% of the time tokens are actually replaced with the token $[MASK]$, 10% of the time tokens are replaced with a random token, and 10% of the time tokens are left unchanged. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. Next sentence prediction task is a binary classification task in which, given a pair of sentences, it is predicted if the second sentence is the next sentence of the first sentence. For fine-tuning, the BERT model is first initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labeled data from the downstream tasks. Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters.

In BERT, a WordPiece tokenizer [75] is used with 30,000 token vocabulary. The first token of every sequence is always a special classification token *[CLS]*. The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks. Sentence pairs are packed together into a single sequence. The sentences are separated using a special token $[SEP]$. Additionally, a learned embedding is added to every token indicating whether it belongs to sentence A or sentence B. The pre-training of the BERT model is illustrated in Figure 4.11.

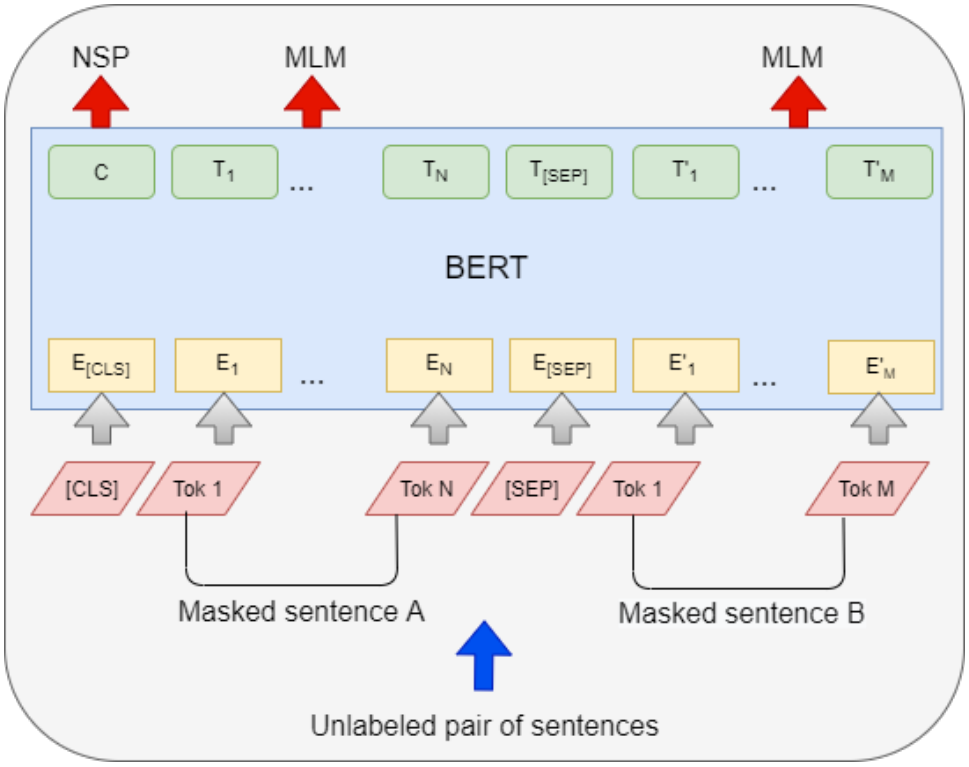


Figure 4.11: Pre-training the BERT model.

Based on BERT's masking strategy, RoBERTa is an optimized pre-training language model that has made breakthroughs in NLP. RoBERTa allows training with much larger mini-batches and learning rates by tuning the BERT model. This allows RoBERTa to improve on the masked language modeling objective, compared with BERT. Instead of the WordPiece tokenizer, RoBERTa uses a Byte-Pair Encoding (BPE) [64]. In both cases, the vocabulary is initialized with all the individual characters in the language, and then the most frequent combinations of the symbols in the vocabulary are iteratively added it. The difference between BPE and WordPiece lies in the way the symbol pairs are chosen for adding to the vocabulary. The BPE algorithm counts the occurrence of every symbol pair and chooses

the one with the highest frequency, while WordPiece chooses the one which maximizes the likelihood of the training data. RoBERTa model is trained with a larger byte-level BPE vocabulary containing 50K subword units.

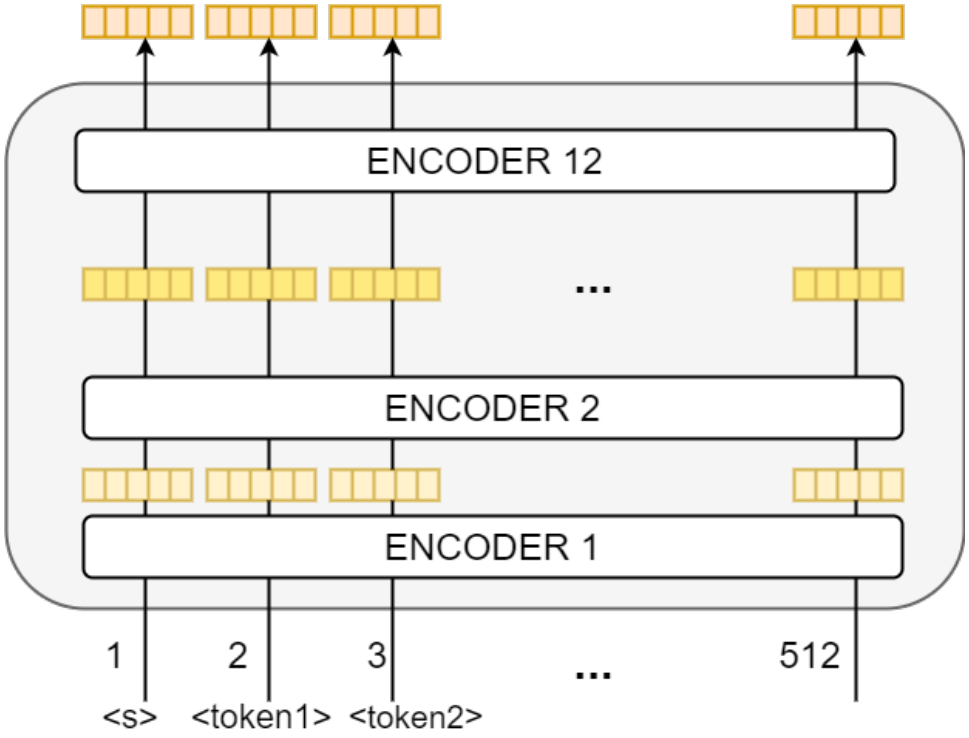


Figure 4.12: Architecture of the RoBERTa model with its stacked encoders.

RoBERTa removes the NSP task from BERT's pre-training and introduces dynamic masking so that the masked token changes during the training epochs. Moreover, RoBERTa uses 160 GB of text for pre-training, including 16GB of BooksCorpus and Wikipedia used in BERT. The additional data included CC-News dataset[4] (63M articles, 76 GB), OpenWebText corpus[5] (38 GB) and Stories from Common Crawl [69] (31 GB). Specifically, RoBERTa is trained with dynamic masking, full

[4]http: //web.archive.org/save/http://commoncrawl.org/2016/10/newsdataset-available
[5]http://web.archive.org/ save/http://Skylion007.github.io/ OpenWebTextCorpus.

sentences without NSP loss, large mini-batches and a larger byte-level BPE. The inputs of the model take pieces of 512 contiguous token that may span over documents. The beginning of a new document is marked with $< s >$ and the end of one by $< /s >$. The *BASE* model contains 12 bidirectional transformer encoders with large feed-forward units (768 hidden states) and 12 attention heads. The architecture of the model is shown in Figure 4.12. As input RoBERTa takes a sequence of words that keep flowing up the stack. Each layer applies self-attention, passes its results through a feed-forward network, and then hands it off to the next encoder.

4.4 VDGraph2Vec

We leverage and integrate the above discussed concepts and knowledge for building our model, VDGraph2Vec, which is used for "Vulnerability Detection" by utilizing graph neural networks on control flow graphs and generating their vector representations. In order to get effective representations of assembly code, VDGraph2Vec learns both the structural and semantic aspects of the assembly code. We are able to accomplish this using a graph structure, such as CFG, and representing each of its basic blocks with a dense vector representation by using a language model. Further, both these components are integrated by using a graph neural network that assimilates the messages from neighbouring nodes to give richer embeddings for the graph. Finally, we use these enriched embeddings for our task of vulnerability detection.

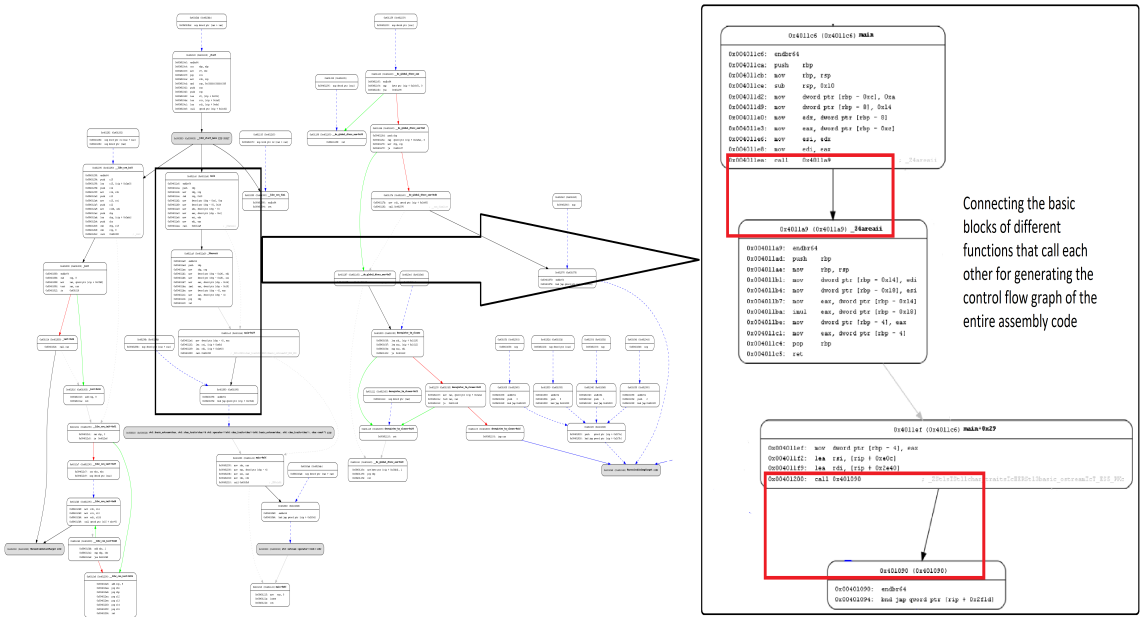


Figure 4.13: An example of a control flow graph created using angr for the entire binary file. We connect the edges between the basic blocks of functions that call one another.

We begin by disassembling the binary file to get its CFG and use angr[6] to create the CFG of the entire assembly file by connecting the basic blocks between different functions that call one another. An example of a CFG generated by angr for an entire binary file is shown in Figure 4.13. To train our language model, we consider an assembly instruction as a word and the entire basic block with its instructions as a sentence. We employ Word2Vec in our setting for learning the block embeddings by taking the average of all the instruction embeddings in the block. Following [5], we also experiment by applying attention to acquire the basic block representations. However, averaging over the block instructions works better in our case, and hence we report our results in the experimental section using an average. A

[6]https://github.com/angr/angr

possible reason for this is that every block contains a different number of instructions. Unlike other approaches, we do not set a maximum limit for the number of instructions in a basic block.

For training the RoBERTa model on assembly code, a corpus containing one million x86 assembly instructions was built. We train the model on the MLM objective, and we retrieve the block (sentence) representations from it by averaging the tokens and concatenating the last four hidden states of the model. We train our language models on the x86 assembly instructions. A possible limitation of our model is that it is mono-architecture based. Other architectures and optimization levels were not taken into consideration, but it is feasible to extend it. Additionally, data dependency edges can be employed along with the control flow execution paths to incorporate more structural information of the assembly codes.

The connectivity between the blocks (edges) and the embeddings for each block (nodes) serve as inputs to our message passing neural network. The MPNN framework accumulates the information from the neighbouring blocks and uses it to generate enriched block representations. For obtaining a graph embedding, we apply a global pooling (readout) layer. We experiment with {add, average, attention} readout layers, and *average* worked better in the case of GCN, and *attention* worked better for gated graph neural network. We then train our embeddings for the downstream task of vulnerability detection using the objective function of minimizing the cross entropy loss between the predicted and actual labels.

$$L(y, \hat{y}) = -\frac{1}{N}\sum^{N}(y \cdot log(\hat{y}) + (1-y) \cdot log(1-\hat{y}))$$

The weights of the neural network are optimized using *Adam* optimizer, which is the de-facto optimization method used for training deep learning models. The neural architecture of the VDGraph2Vec model is shown in Figure 4.14.

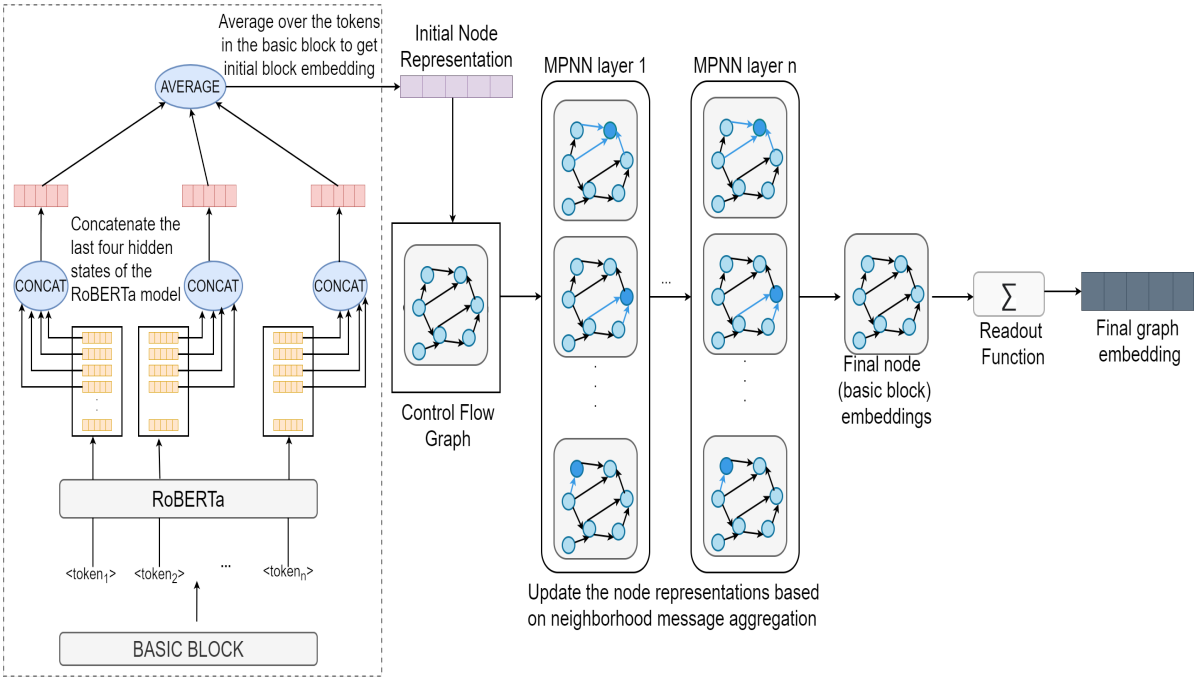


Figure 4.14: Architecture of the VDGraph2Vec model. The first part of the model captures the initial node embeddings using the RoBERTa model. These embeddings are fed to the message passing neural network that further enhances the representations for each node by aggregating the messages from its neighbors. Lastly, the final node embeddings are passed through the readout layer to generate the graph embedding for the entire control graph of the assembly code.

Chapter 5

Experiments

In this research, we conduct extensive experimentation by examining different node embedding methods and graph neural networks. In this chapter we demonstrate why it is important to incorporate the syntax as well as the semantics of the assembly code in our vector representations. Thus, the objectives of the experiments are to evaluate the performance of VDGraph2Vec for vulnerability detection and to compare our methodology with the state-of-the-art vulnerability detection works. We consider vulnerability detection as a binary classification task for each *Common Weakness Enumeration* (*CWE*). CWE[1] is a categorization of software weaknesses and vulnerabilities. Each of the weaknesses has its separate characteristics, and hence it is better if we train models separately to learn these distinguishing features. Thus, we test the effectiveness of VDGraph2Vec on the three most commonly encountered weaknesses. We use *PyTorch* [54] and *PyTorch Geometric* [25] to

[1]http://cwe.mitre.org/about/index.html

implement our models. We train the models on a server with two Xeon E5-2697 CPUs, 384 GB RAM, and four Nvidia Titan XP graphics cards.

5.1 Data Preparation

Data collection is a preliminary task of any research. Thus, we first collect a dataset that contains examples of vulnerable versions of the software. The *Juliet Test Suite*[2] is a collection of vulnerability datasets created by the *National Institute of Standards and Technology* (*NIST*) and organized into 118 different CWEs. The code is categorized into good and bad cases to make it suitable for supervised learning. Since the Juliet Test Suite contains more synthetic examples, we also evaluate our model in a more challenging and realistic scenario. We use the *NDSS18* dataset, which is also maintained by NIST and extracted from the National Vulnerability Database (NVD)[3] and Software Quality Assurance Dataset (SARD)[4]. This dataset was originally available in source code format [47]. *Le et. al* [40] compiled the source code into binaries for Windows OS and Linux OS platforms. The NDSS18 dataset contains a total of 32,281 binary files for CWE-119 and CWE-322 over both platforms.

We conduct our analysis on three of the CWEs obtained from two different datasets. Particularly, we use CWE-121 and CWE-190 from the Juliet Test Suite, and CWE-119 from the NDSS18 dataset to benchmark our model's performance.

[2]https://samate.nist.gov/SARD/testsuite.php
[3]https://nvd.nist.gov/
[4]https://samate.nist.gov/SARD/

CWE-121 is a weakness caused by stack-based buffer overflow. An integer overflow or wraparound results in the vulnerability CWE-190. CWE-119 is related to improper restriction of operations within the bounds of a memory buffer. Buffer overflow and integer overflow vulnerabilities are commonly encountered in software and exploited, leading to various adverse attacks. Additionally, these vulnerabilities usually span more than one function. Consequently, a graph structure is more suitable for discovering these vulnerabilities. Moreover, we also note from Figures 5.1 and 5.2 that these datasets have a varying number of edges and nodes. CWE-121 and CWE-190, from the Juliet Test Suite, have a higher average number of nodes and edges as compared to CWE-119, from the NDSS18 dataset. The statistics of these datasets are listed in Table 5.1.

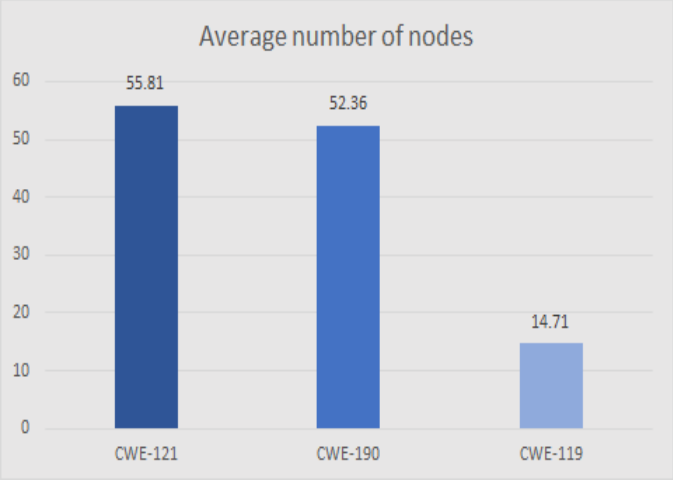


Figure 5.1: Average number of nodes in the datasets.

As assembly code shares similarities with normal text, it is important that we perform pre-processing on assembly codes, similarly to the case of textual data.

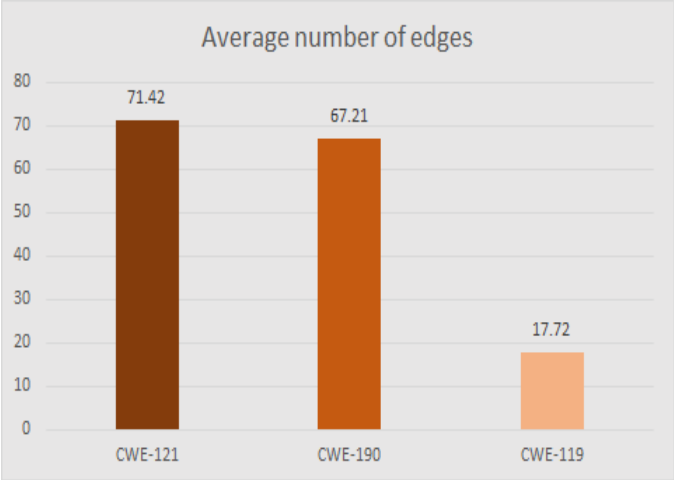


Figure 5.2: Average number of edges in the datasets.

Table 5.1: Statistics of our datasets used for the empirical analysis.

CWE	Vulnerable samples	Non-vulnerable samples
CWE-121	3100	3100
CWE-190	3960	3960
CWE-119	6521	5861

Thus, we first convert all instructions to lower case. In order to avoid learning different representations for all different hexadecimal addresses, we replace the hexadecimal addresses with the token ⟨ADDR⟩, and the numerical constants with ⟨CONST⟩. This improves the semantic quality of our embeddings.

5.2 Evaluation Metrics

We evaluate the performance of our models by splitting the datasets as follows: 80% training, 10% validation, and 10% testing. We initialize different random seeds, and results are averaged for 5 runs. The vulnerability detection task that

we consider here is a binary classification task, where we treat vulnerable samples as positive and non-vulnerable as negative. The following evaluation metrics were used to examine the performance of our models:

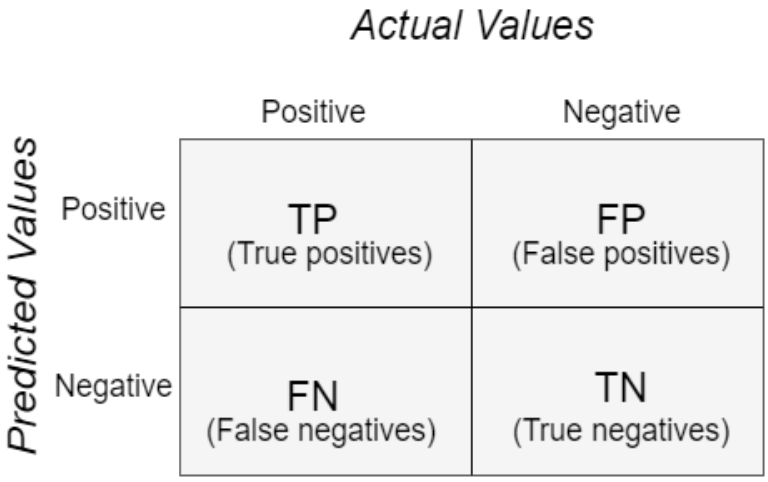


Figure 5.3: Confusion matrix depicting how to find TP, FP, FN, and TN from the actual and predicted values for the task of binary classification. For vulnerability detection, FN is more critical than FP since we want the models to not misclassify any vulnerable cases.

- Accuracy: The proportion of true predictions among the total number of samples considered. It is a fair measure to compare the performance of models when the datasets are not skewed or imbalanced.

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

- Precision: The proportion of predicted positives that are actually positive. It is a good measure to compare model performance when there is a high cost

associated with false positives.

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

- Recall: The proportion of actual positives that are correctly predicted. It is a good measure to compare model performance when there is a high cost associated with false negatives.

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

- F1-score: The harmonic mean of precision and recall, which is used to maintain a balance between the precision and recall. There is always a trade-off between recall and precision, and the F1-score seeks to ensure a balance between the precision and recall.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- AUC-ROC score: ROC (Receiver operating characteristic) is a probability curve between sensitivity (false positive rate) and (1-specificity) (true positive rate), and AUC (Area under curve) represents the degree or measure of separability. Thus, AUC is calculated by plotting the false positive rate against

the true positive rate at various threshold settings. It indicates the capability of the model to distinguish between the classes.

We also study the impact of hyper-parameter tuning by evaluating the models on the validation set. We try different settings of learning rate, batch size, epochs, channels for a GCN convolution layer, number of layers for gated graph neural networks, and dropout. The different hyper-parameter settings are enlisted in Table 5.2. We select the setting that results in the best accuracy on the validation set. The highest achieved accuracies obtained on different parameter settings are reported for each model.

Hyper-parameters	**Set of Values**
Batch Size	[100, 128, 256]
Learning Rate	[0.01, 0.001, 0.0001]
Epochs	[50, 75, 100, 150]
Channels in GCN	[16, 32, 64, 128]
Layers of GGNN	[2, 3]
Dropout	[0, 0.3, 0.4, 0.5]

Table 5.2: Different set of hyper-parameter settings used our model.

5.3 Models for Comparison

We implement the state-of-the-art binary vulnerability detection models to compare with our VDGraph2Vec model. In order to demonstrate the competence of our semantic and structural components, we compare the potential of our model at the node embedding and classification level. As in [77], we investigate an approach

based on employing handcrafted features for generating our basic block embeddings. We use the following features for our basic blocks: 1) number of transfer instructions, 2) number of function calls, 3) total number of instructions in the block, 4) number of arithmetic instructions, 5) number of logical operations in the block, 6) number of constants, and 7) number of strings. However, using this representation we lose all the pivotal information expressed in the assembly instructions. We also compare our model against our baseline model presented in [42], which uses Instruction2Vec[5] for embedding the assembly instructions and TextCNN for classifying the samples into benign and vulnerable. We try all possible settings and report the best results these methods can achieve to compare with our model. Following [5], we use another state-of-the-art model for binary code representation based on word2vec for node representation and Structure2Vec [66] for CFG representation. Although the authors evaluate their model for binary clone detection and compiler provenance on different datasets, we will analyze the effectiveness of these embeddings for vulnerability detection. Thus, we incorporate Structure2Vec in our experiments to compare it with GCN and Gated Graph Neural Network (GGNN), and Word2Vec to contrast it with RoBERTa node embeddings. We seek to try different variations of node embeddings and classification models to gauge the subtle differences in performances caused by each of these components. Specifically, we compare the two variants of our model, ***VDGraph2Vec-GCN*** and ***VDGraph2Vec-GGNN***, with the following variants:

[5]https://github.com/firmcode/instruction2vec

- *Handcrafted features with GCN* ***(HF-GCN)***
- *Handcrafted features with GGNN* ***(HF-GGNN)***
- *Instruction2Vec with TextCNN* ***(i2V-TCNN)***
- *Word2Vec with Structure2Vec* ***(w2v-s2v)***
- *Word2Vec with GCN* ***(w2v-GCN)***
- *Word2Vec with GGNN* ***(w2v-GGNN)***
- *RoBERTa with Structure2Vec* ***(RoS2v)***

The VDGraph2Vec-GCN model utilizes a GCN for the message passing component, while VDGraph2Vec-GGNN employs a gated graph neural network.

5.4 Results and Analysis

The results of vulnerability detection on CWE-121, CWE-190, and CWE-119 for various combinations of node embeddings and classification methods are shown in Tables 5.3, 5.4, and 5.5, respectively. The last two rows of the tables denote the performance of our VDGraph2Vec model. We also investigate if the difference in accuracies between our model and other state-of-the-art methods is statistically significant. Furthermore, when the models are deployed for real world application, they are often trained on a different dataset and evaluated on the actual data.

Therefore, we perform a cross-dataset evaluation to assess the generalization capability of the models. For this experimental setting we collect a test dataset containing 1,000 samples obtained from [15]. We train our models on the entire CWE-121 dataset from the Juliet Test Suite and evaluate it on the test dataset[6]. The results of the experiment are reported in Table 5.6.

Model	Accuracy	Precision	Recall	F1-score	AU-ROC score	p-value
HF-GCN	70.96	79.28	60.85	68.85	71.55	•
HF-GGNN	71.77	66.81	92.35	77.53	70.58	•
i2v-TCNN	94.83	97.12	92.96	95.0	94.94	•
w2v-s2v	95.32	94.08	97.24	95.63	95.21	•
w2v-GCN	95.81	94.13	98.16	96.11	95.66	•
w2v-GGNN	97.58	98.14	97.24	97.69	97.59	•
RoS2v	97.90	**100.0**	96.02	97.97	98.01	•
VDGraph2Vec-GCN*	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	○
VDGraph2Vec-GGNN*	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	N/A

Table 5.3: Vulnerability detection results on CWE-121 (Juliet Test Suite). $*$ denotes the performance of our proposed model, *VDGraph2Vec*. • and ○ respectively denote whether the difference in accuracy between *VDGraph2Vec* and baseline models is statistically significant or not.

VDGraph2Vec achieves statiscally significantly better accuracy than the other models in all experiments, as the p-values in t-test are much smaller than 0.01. Thus, our model outperforms all other models on all three CWEs. Moreover, it is evident from our results that manually extracted features do not offer good representational quality for the assembly code; hence it is important to incorporate the meaningful contextual representations of the assembly instructions. We

[6]https://github.com/williamadahl/RNN-for-Vulnerability-Detection

Model	Accuracy	Precision	Recall	F1-score	AU-ROC score	p-value
HF-GCN	67.67	69.02	72.89	70.90	67.21	•
HF-GGNN	69.19	71.19	72.19	71.69	68.92	•
i2v-TCNN	90.78	90.06	93.22	91.61	90.56	•
w2v-s2v	93.43	93.11	94.85	93.98	93.31	•
w2v-GCN	95.07	94.71	96.26	94.97	95.48	•
w2v-GGNN	95.41	95.58	96.02	95.81	95.40	•
RoS2v	94.57	94.25	95.79	95.01	94.46	•
VDGraph2Vec-GCN*	99.74	99.53	**100.0**	99.76	99.72	○
VDGraph2Vec-GGNN*	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	N/A

Table 5.4: Vulnerability detection results on CWE-190 (Juliet Test Suite). $*$ denotes the performance of our proposed model, *VDGraph2Vec*. • and ○ respectively denote whether the difference in accuracy between *VDGraph2Vec-GGNN* and baseline models is statistically significant or not.

Model	Accuracy	Precision	Recall	F1-score	AU-ROC score	p-value
HF-GCN	64.83	69.84	64.13	66.86	64.92	•
HF-GGNN	66.77	64.66	88.04	74.56	64.23	•
i2v-TCNN	81.41	83.72	82.50	83.11	81.32	•
w2v-s2v	85.0	85.91	87.17	86.54	84.74	•
w2v-GCN	89.03	90.32	89.79	90.05	88.94	•
w2v-GGNN	90.48	92.77	89.79	91.25	90.56	•
RoS2v	86.86	86.0	90.42	88.15	86.55	•
VDGraph2Vec-GCN*	92.9	93.08	94.16	93.61	92.58	•
VDGraph2Vec-GGNN*	**95.48**	**95.65**	**96.21**	**95.92**	**95.27**	N/A

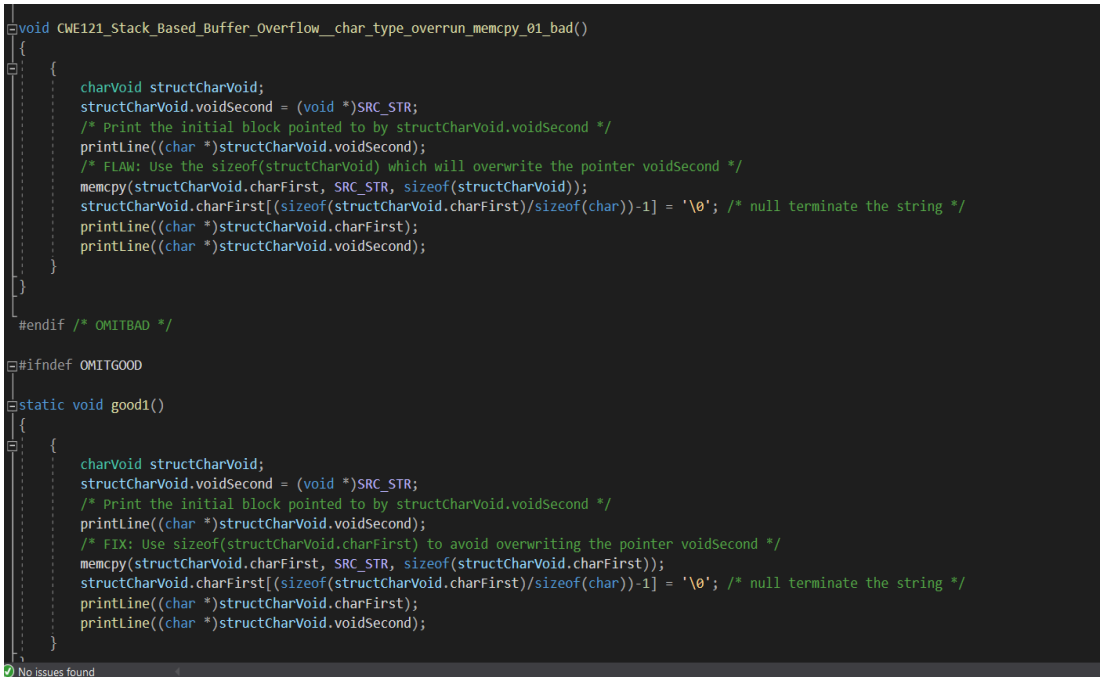
Table 5.5: Vulnerability detection results on CWE-119 (NDSS dataset). $*$ denotes the performance of our proposed model, *VDGraph2Vec*. • and ○ respectively denote whether the difference in accuracy between *VDGraph2Vec-GGNN* and baseline models is statistically significant or not.

also observe that RoBERTa block representations boost the performance more than Word2Vec. Additionally, in comparison to TextCNN and Structure2Vec, message passing neural networks are able to better embed the nuanced relationships be-

Model	Accuracy	Precision	Recall	F1-score	AU-ROC score	p-value
HF-GCN	61.0	62.73	54.2	58.15	60.99	•
HF-GGNN	62.2	71.32	40.8	51.98	62.2	•
i2v-TCNN	75.7	**100.0**	51.4	67.89	75.7	•
w2v-s2v	72.2	**100.0**	44.4	61.4	72.2	•
w2v-GCN	83.8	**100.0**	67.6	80.66	83.8	•
w2v-GGNN	89.3	**100.0**	78.6	88.01	89.3	•
RoS2v	78.7	**100.0**	57.4	72.93	78.69	•
VDGraph2Vec-GCN*	91.1	**100.0**	82.2	90.23	91.1	•
VDGraph2Vec-GGNN*	**94.9**	**100.0**	**89.8**	**94.6**	**94.9**	N/A

Table 5.6: Cross dataset results on CWE-121 with the model trained on samples from the Juliet Test Suite and tested on samples from another dataset. $*$ denotes the performance of our proposed model, *VDGraph2Vec-GGNN*. • and ○ respectively denote whether the difference in accuracy between *VDGraph2Vec-GGNN* and baseline models is statistically significant or not.

tween different parts of an assembly code with its flow of code execution. Even in our different experimental setting of cross-dataset evaluation, VDGraph2Vec outperforms the other methodologies. Most of the models are able to achieve a perfect precision, implying that they are able to detect the vulnerable samples. In that setting we also observe that the gated graph neural network surpasses the generalizability power of the graph convolution network by a wide margin. Further, we notice that our model is able to achieve 100% accuracy on the CWE-121 and CWE-190 datasets from the Juliet Test Suite. Intuitively, we believe the reason for this is that the samples in the dataset are synthetic and man-made, thus the distinguishing characteristics between the vulnerable and benign samples are easily learned by the model. As shown in Figure 5.4, the samples in Juliette Test Suite have small



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

#endif /* OMITBAD */

#ifndef OMITGOOD

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

No issues found

Figure 5.4: Example of a vulnerable and non-vulnerable sample from CWE-121.

fixes and the difference between the vulnerable and non-vulnerable samples is easy to learn. Also, the actual vulnerability is statically detectable. Nonetheless, our model is able to perform better than the baseline models, which is further validated by its effective performance on the more natural dataset of CWE-119 from the NDSS18 dataset.

Chapter 6

Conclusion and Future Work

In this thesis, we present our method, VDGraph2Vec, for vulnerability detection at the assembly code level. We leverage message passing neural networks and combine it with RoBERTa model to generate effective graph embeddings. We perform thorough experimentation to investigate the performance of our model on vulnerability detection. We empirically prove that VDGraph2Vec is able to spot vulnerabilities successfully because both semantics and the innate hierarchical structure of assembly code are being taken into consideration. The control flow graph helps in finding the vulnerable execution paths. MPNN gives better comprehensive representations by aggregating messages from all neighbors. We also demonstrate the effectiveness and generalization ability of VDGraph2Vec by conducting a cross-dataset evaluation. Our model is able to achieve high performance in different experimental settings, surpassing the recent works in this direction.

Despite these impressive results, we believe there are certain open challenges that hinder research for vulnerability detection at the binary level.

- The datasets in this area mostly encompass the source code level. Furthermore, most of these datasets that are available in source code format cannot be compiled to their equivalent binaries. In a real-world scenario, we generally do not have access to the source code. Therefore, there is a need to curate datasets for binary vulnerability detection so that we have more data to train our deep learning models.

- We can incorporate more structural information of the graph with the data flow dependencies. This can lead to further improvement in the performance of the model.

- Our work caters to x86 assembly instructions. In the future, we can extend it for all target machine architectures. The lack of sufficient data hinders research for cross-architecture vulnerability detection.

Bibliography

[1] ADAMIC, L. A., HUBERMAN, B. A., BARABÁSI, A., ALBERT, R., JEONG, H., AND BIANCONI, G. Power-law distribution of the world wide web. *science 287*, 5461 (2000), 2115–2115.

[2] AL-RFOU, R., CHOE, D., CONSTANT, N., GUO, M., AND JONES, L. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 3159–3166.

[3] ALLAMANIS, M., BROCKSCHMIDT, M., AND KHADEMI, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[4] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] BALDONI, R., DI LUNA, G. A., MASSARELLI, L., PETRONI, F., AND QUERZONI, L. Unsupervised features extraction for binary similarity using graph embedding neural networks. *arXiv preprint arXiv:1810.09683* (2018).

[6] BAN, X., LIU, S., CHEN, C., AND CHUA, C. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience 31*, 19 (2019), e5103.

[7] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *the Journal of machine Learning research 3* (2003), 993–1022.

[8] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[9] BRITZ, D., GOLDIE, A., LUONG, M.-T., AND LE, Q. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906* (2017).

[10] BROMLEY, J., GUYON, I., LECUN, Y., SÄCKINGER, E., AND SHAH, R. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems 6* (1993), 737–744.

[11] CHENG, J., DONG, L., AND LAPATA, M. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733* (2016).

[12] CHERNIS, B., AND VERMA, R. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics* (2018), pp. 31–39.

[13] CHO, K., VAN MERRIËNBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase represen-

tations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[14] COOPER, K. D., HARVEY, T. J., AND WATERMAN, T. Building a control-flow graph from scheduled assembly code. Tech. rep., 2002.

[15] DAHL, W. A., ERDODI, L., AND ZENNARO, F. M. Stack-based buffer overflow detection using recurrent neural networks. *arXiv preprint arXiv:2012.15116* (2020).

[16] DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. Indexing by latent semantic analysis. *Journal of the American society for information science 41*, 6 (1990), 391–407.

[17] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[18] DING, S. H. H., FUNG, B. C. M., AND CHARLAND, P. Kam1n0: MapReduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)* (August 2016), pp. 461–470.

[19] DING, S. H. H., FUNG, B. C. M., AND CHARLAND, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfus-

cation and compiler optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 472–489.

[20] EILAM, E. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[21] FARHADI, M. R., FUNG, B. C. M., CHARLAND, P., AND DEBBABI, M. BinClone: Detecting code clones in malware. In *Proceedings of the 8th IEEE International Conference on Software Security and Reliability (SERE)* (San Francisco, CA, June 2014), IEEE Reliability Society, pp. 78–87.

[22] FARHADI, M. R., FUNG, B. C. M., FUNG, Y. B., CHARLAND, P., PREDA, S., AND DEBBABI, M. Scalable code clone search for malware analysis. *Digital Investigation (DIIN): Special Issue on Big Data and Intelligent Data Analysis 15* (December 2015), 46–60.

[23] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 480–491.

[24] FENG, Z., GUO, D., TANG, D., DUAN, N., FENG, X., GONG, M., SHOU, L., QIN, B., LIU, T., JIANG, D., ET AL. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[25] FEY, M., AND LENSSEN, J. E. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428* (2019).

[26] GEURTS, P., ERNST, D., AND WEHENKEL, L. Extremely randomized trees. *Machine learning 63*, 1 (2006), 3–42.

[27] GILMER, J., SCHOENHOLZ, S. S., RILEY, P. F., VINYALS, O., AND DAHL, G. E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212* (2017).

[28] GRAVES, A., MOHAMED, A.-R., AND HINTON, G. Speech recognition with deep recurrent neural networks. In *Proceedings of the IEEE international conference on acoustics, speech and signal processing* (2013), Ieee, pp. 6645–6649.

[29] GROVER, A., AND LESKOVEC, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining* (2016), pp. 855–864.

[30] HAMILTON, W. L. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning 14*, 3 (2020), 1–159.

[31] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216* (2017).

[32] HARER, J. A., KIM, L. Y., RUSSELL, R. L., OZDEMIR, O., KOSTA, L. R., RANGAMANI, A., HAMILTON, L. H., CENTENO, G. I., KEY, J. R., ELLINGWOOD, P. M., ET AL. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[33] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[34] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International conference on machine learning* (2015), PMLR, pp. 448–456.

[35] KIM, Y. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[36] KIPF, T. N., AND WELLING, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[37] KRSUL, I. V. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[38] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (2004), IEEE, pp. 75–86.

[39] LE, Q., AND MIKOLOV, T. Distributed representations of sentences and documents. In *Proceedings of the International conference on machine learning* (2014), pp. 1188–1196.

[40] LE, T., NGUYEN, T., LE, T., PHUNG, D., MONTAGUE, P., DE VEL, O., AND QU, L. Maximal divergence sequential autoencoder for binary software vul-

nerability detection. In *Proceedings of the International Conference on Learning Representations* (2018).

[41] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[42] Lee, Y., Kwon, H., Choi, S.-H., Lim, S.-H., Baek, S. H., and Park, K.-W. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences 9*, 19 (2019), 4086.

[43] Li, Y., Gu, C., Dullien, T., Vinyals, O., and Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning* (2019), PMLR, pp. 3835–3845.

[44] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[45] Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., and Jin, H. A comparative study of deep learning-based vulnerability detection system. *IEEE Access 7* (2019), 103184–103197.

[46] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756* (2018).

[47] LI, Z., ZOU, D., XU, S., OU, X., JIN, H., WANG, S., DENG, Z., AND ZHONG, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[48] LIN, G., WEN, S., HAN, Q.-L., ZHANG, J., AND XIANG, Y. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE 108*, 10 (2020), 1825–1848.

[49] LIU, Y., OTT, M., GOYAL, N., DU, J., JOSHI, M., CHEN, D., LEVY, O., LEWIS, M., ZETTLEMOYER, L., AND STOYANOV, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[50] LUONG, M.-T., PHAM, H., AND MANNING, C. D. Bilingual word representations with monolingual quality in mind. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing* (2015), pp. 151–159.

[51] LYNN, H. M., PAN, S. B., AND KIM, P. A deep bidirectional gru network model for biometric electrocardiogram classification based on recurrent neural networks. *IEEE Access 7* (2019), 145395–145405.

[52] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[53] NASSIF, A. B., SHAHIN, I., ATTILI, I., AZZEH, M., AND SHAALAN, K. Speech recognition using deep neural networks: A systematic review. *IEEE access 7* (2019), 19143–19165.

[54] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch.

[55] PENNINGTON, J., SOCHER, R., AND MANNING, C. D. Glove: Global vectors for word representation. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 1532–1543.

[56] PEROZZI, B., AL-RFOU, R., AND SKIENA, S. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), pp. 701–710.

[57] QIU, X., SUN, T., XU, Y., SHAO, Y., DAI, N., AND HUANG, X. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* (2020), 1–26.

[58] REDMOND, K., LUO, L., AND ZENG, Q. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652* (2018).

[59] ROY, C. K., CORDY, J. R., AND KOSCHKE, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming 74*, 7 (2009), 470–495.

[60] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[61] RUSSELL, R., KIM, L., HAMILTON, L., LAZOVICH, T., HARER, J., OZDEMIR, O., ELLINGWOOD, P., AND MCCONLEY, M. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018), IEEE, pp. 757–762.

[62] SCARSELLI, F., GORI, M., TSOI, A. C., HAGENBUCHNER, M., AND MONFARDINI, G. The graph neural network model. *IEEE transactions on neural networks 20*, 1 (2008), 61–80.

[63] SCHUSTER, M., AND PALIWAL, K. K. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing 45*, 11 (1997), 2673–2681.

[64] SENNRICH, R., HADDOW, B., AND BIRCH, A. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Berlin, Germany, Aug. 2016), Association for Computational Linguistics, pp. 1715–1725.

[65] SINGH, A. *Identifying malicious code through reverse engineering*, vol. 44. Springer Science & Business Media, 2009.

[66] SONG, L. Structure2vec: Deep learning for security analytics over graphs. *Atlanta, GA: USENIX Association* (2018).

[67] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 56 (2014), 1929–1958.

[68] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215* (2014).

[69] TRINH, T. H., AND LE, Q. V. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847* (2018).

[70] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[71] VELIČKOVIĆ, P., CUCURULL, G., CASANOVA, A., ROMERO, A., LIO, P., AND BENGIO, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[72] VOULODIMOS, A., DOULAMIS, N., DOULAMIS, A., AND PROTOPAPADAKIS, E. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience 2018* (2018).

[73] WANG, W., LI, G., MA, B., XIA, X., AND JIN, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020), IEEE, pp. 261–271.

[74] WU, F., WANG, J., LIU, J., AND WANG, W. Vulnerability detection with deep learning. In *Proceedings of the 3rd IEEE International Conference on Computer and Communications (ICCC)* (2017), IEEE, pp. 1298–1302.

[75] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[76] XU, K., BA, J., KIROS, R., CHO, K., COURVILLE, A., SALAKHUDINOV, R., ZEMEL, R., AND BENGIO, Y. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International conference on machine learning* (2015), PMLR, pp. 2048–2057.

[77] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 363–376.

[78] YAN, J., YAN, G., AND JIN, D. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), IEEE, pp. 52–63.

[79] YOUNG, T., HAZARIKA, D., PORIA, S., AND CAMBRIA, E. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine 13*, 3 (2018), 55–75.

[80] ZENG, P., LIN, G., PAN, L., TAI, Y., AND ZHANG, J. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access* (2020).

[81] ZHENG, J., PANG, J., ZHANG, X., ZHOU, X., LI, M., AND WANG, J. Recurrent neural network based binary code vulnerability detection. In *Proceedings of the 2nd International Conference on Algorithms, Computing and Artificial Intelligence* (2019), pp. 160–165.

[82] ZHOU, J., CUI, G., HU, S., ZHANG, Z., YANG, C., LIU, Z., WANG, L., LI, C., AND SUN, M. Graph neural networks: A review of methods and applications. *AI Open 1* (2020), 57–81.

[83] ZHOU, Y., LIU, S., SIOW, J., DU, X., AND LIU, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the Advances in Neural Information Processing Systems* (2019), pp. 10197–10207.

[84] ZHU, Y., KIROS, R., ZEMEL, R., SALAKHUTDINOV, R., URTASUN, R., TORRALBA, A., AND FIDLER, S. Aligning books and movies: Towards story-like

visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 19–27.