VDGraph2Vec: Vulnerability Detection in Assembly Code using Message Passing Neural Networks

Ashita Diwan
School of Computer Science
McGill University
Montreal, Canada
ashita.diwan@mail.mcgill.ca

Miles Q. Li
School of Computer Science
McGill University
Montreal, Canada
miles.qi.li@mail.mcgill.ca

Benjamin C. M. Fung
School of Information Studies
McGill University
Montreal, Canada
ben.fung@mcgill.ca

***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. 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 effectively 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.**

I. 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 the 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 [18]. 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 [15]. Thus, the detection of software vulnerabilities garners significant interest from the software security community. Traditionally, software vulnerabilities were detected by reverse engineering, a complex and time-consuming process that requires expert knowledge and extensive experience [11]. Reverse engineering is the process of analyzing the design of software from its binary executables [34]. Security experts often apply this technique to understanding 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 [5]. 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 [27], [26], [4], [37]. Furthermore, researchers tend to detect vulnerabilities mostly at the source code level [15], [5], [32]. 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 [22], [8]. Though these studies offer promising results, they are tailored to apprehend to the semantics of the binaries only. 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 work, 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 [33]. We focus explicitly on *Message Passing Neural Networks* (*MPNN*) [14], which have achieved state-of-the-art performance in various tasks. Our research highlights that by employing them, we are able to improve the representation quality of our code as for each node, they aggregate

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

978-1-6654-6283-9/22/$31.00 ©2022 IEEE
DOI 10.1109/ICMLA55696.2022.00173

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* [29]. 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 code, 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 achieves a new state-of-the-art. The primary focus of the study is the generation of 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 [31].

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 level 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 for both function-level and code-level analysis. Previous approaches mostly cater to representation at the function level [10]. 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.

II. Related Work

With the advances in machine learning, it has become pivotal to assess its capability in the field of cybersecurity. *Harer et al.* [15] 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 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* [30] representations. They further used a *TextCNN* [16] for learning more enhanced features along with an extremely randomized trees classifier [13]. *Russell et al.* [32] 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 [30] and then employed convolution and recurrent feature extractors. Using the neural features as inputs, they finally applied the random forest classifier to classify vulnerabilities. *Chernis and Verma* [5] 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 (character n-grams, word n-grams, and suffix trees). Several researchers also presented comprehensive surveys outlining automated ways to detect software vulnerability [28], [39], [25] .

Li et al. [27] 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* (*LSTM*). 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* [26] framework, which focuses on obtaining program representations that can accommodate syntax and semantic information pertinent to vulnerabilities by leveraging the abstract syntax trees and program dependency graphs. They conducted empirical studies to show the potency of a *Bidirectional Gated Recurrent Unit* (*BGRU*) for vulnerability detection. A major contribution from the authors is a vulnerability detection dataset [2], collected from two data sources, the *National Vulnerability Database* (*NVD*)[3] and the *Software Assurance Reference Dataset* (*SARD*)[4].

Since an assembly code shares some commonalities with natural text, researchers often employ natural language processing models on programs [1], [23]. *Lee et al.* [22] introduced *Instruction2Vec*, a framework for modelling assembly code. It is an improved version of the Word2Vec [30] 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 [16] for detecting software vulnerabilities. *Ding et al.* [10] proposed an assembly code representation method based on the PV-DM model [19] 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

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

flow of code execution.

Recent research demonstrates the success of message passing neural networks for source code representation. *Zhou et al.* [41] explored the efficacy of using *Graph Neural networks* (*GNN*)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 (*GGNN*)[24] model for getting the representations of each node. This is further utilized by the Conv module for graph-level classification. *Allamanis et al.* [2] introduced strategies to learn program structures using graph-based deep learning. They demonstrated the scalability of GGNNs on two tasks, *VARNAMING*, and *VARMISUSE*. 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 GNN models by experimenting with different pretrained models such as RoBERTa [29] for the initial node representations of the basic blocks.

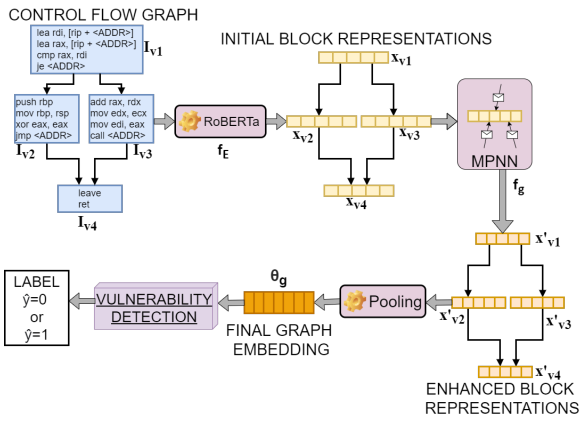
III. Problem Definition

In this section, 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 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. We feed the CFG $G = (X, E)$ to the MPNN, 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, hence $f_E(I_v) = x_v$. 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 pooling layer to generate the embedding for the entire graph, θ_g, which is used for our downstream task of binary vulnerability detection to yield a label, $\hat{y} \in \{0, 1\}$. Thus, $f_g(G) = \hat{y}$. The workflow of our VDGraph2Vec model is illustrated in Figure 1. 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 ($\hat{y} = 0$). ■

IV. 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



```
lea rdi, [rip + <ADDR>]
lea rax, [rip + <ADDR>]
cmp rax, rdi
je <ADDR>
```

```
push rbp
mov rbp, rsp
xor eax, eax
jmp <ADDR>
```

```
add rax, rdx
mov edx, ecx
mov edi, eax
call <ADDR>
```

```
leave
ret
```

Fig. 1: Workflow of our VDGraph2Vec model.

be leveraged for efficient assembly code representation and detection of software vulnerabilities. In this section we discuss the preliminaries requisite for understanding the model. Then, we describe our VDGraph2Vec model in detail.

A. 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 assembly code. There are a variety of disassemblers including *IDA Pro*[5] and *Ghidra*[6] that can be used for obtaining the CFG. We use *angr*[7] in our research because it is open source and provides an easy to use Python interface, and hence it is easier to replicate results. Therefore, we start by extracting the Control Flow Graph (CFG) of the program. A CFG [7] 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. At the high level, this representation is especially beneficial for vulnerability detection because it has the ability to uncover risky and unsafe program execution topologies. Further, we use an MPNN [14] to obtain efficient representations of the assembly code. Conceptually, it works better because for each node, it accumulates the messages 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 have achieved impressive results for various tasks in both natural language processing and in source code representation [12]. To capture the meaning of the assembly instructions, we utilize pre-trained language models (f_E) to extract the initial block embeddings, x_v.

1) Message Passing Neural Networks: The complexity of data structures led to several advancements in machine learning with the introduction of graph neural networks. Owing

[5]https://www.hex-rays.com/products/ida/
[6]https://ghidra-sre.org/
[7]https://angr.io/

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 [40]. MPNN [14] is a popular framework that generalizes most graph neural models based on the idea of getting enhanced node representations by aggregating information from the neighbors. 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 learnable and 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) [17] and *Gated Graph Neural Networks* (*GGNN*s) [24]. GCN generalizes the idea of *convolutional neural networks* (*CNN*s) [21] to non-structured 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,

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

where $W^{(t)}$ is the weight matrix used for the layer t, and $h_v^0 = x_v$. The adjacency matrix A is normalized to avoid the scaling problem by,

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

where D is the diagonal matrix with the degrees of all nodes in A. 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*) [6].

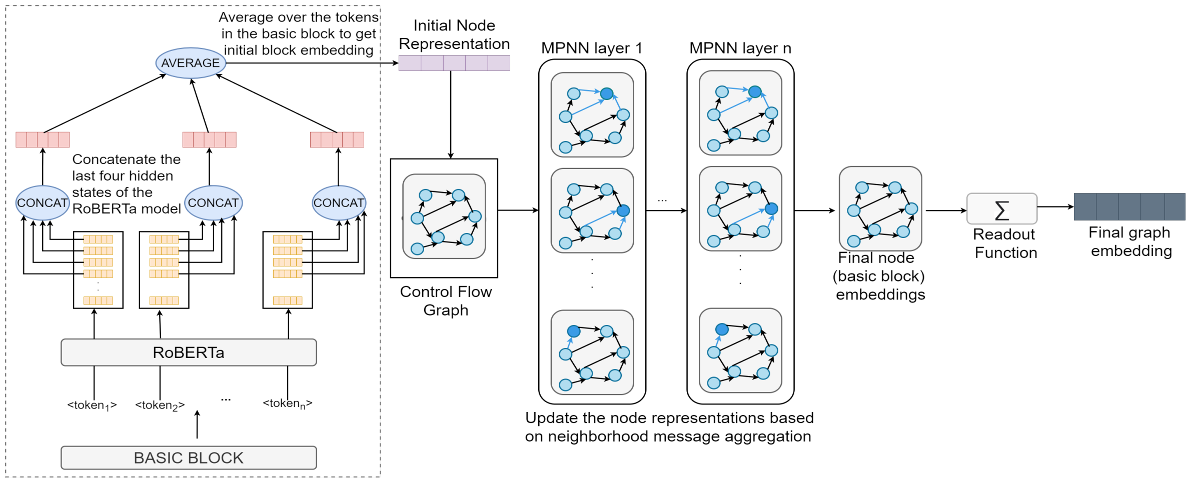
$$h_v^{t+1} = GRU(h_v^t, m_v^{t+1})$$

2) Word Embeddings: Word2Vec [30] 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. It is a shallow, two-layer neural network, and 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* 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. However, the word2Vec model fails to capture differences like polysemy. To overcome this shortcoming, context informed word embeddings were introduced with *Transformer* [36] models. One such model that revolutionized pre-trained language models in NLP is BERT [9]. The model takes into consideration both left and right context of the words, resulting in more accurate feature representations. The authors of this paper investigate a novel technique called *masked language modeling* (*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. 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. The *BASE* model contains 12 bidirectional Transformer encoders with large feed-forward units (768 hidden states) and 12 attention heads. 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.

B. Model

In order to get embeddings of assembly code, VD-Graph2Vec learns both the structural and semantic aspects of the assembly code. We begin by disassembling the binary file and use angr[8] to create the CFG of the entire assembly file by connecting the basic blocks between different functions that call one another. We represent each of its basic blocks with a dense vector representation by using a language model and both these components are integrated by using a graph neural network. 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 *Baldoni et al.* [3], 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 possible reason for this is that every block contains a different number of instructions. Unlike other

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



Average over the tokens in the basic block to get initial block embedding
AVERAGE
Initial Node Representation
MPNN layer 1
MPNN layer n
Concatenate the last four hidden states of the RoBERTa model
CONCAT
CONCAT
CONCAT
RoBERTa
<token$_1$>
<token$_2$>
...
<token$_n$>
BASIC BLOCK
Control Flow Graph
Update the node representations based on neighborhood message aggregation
Final node (basic block) embeddings
Readout Function
Final graph embedding

Fig. 2: Architecture of the VDGraph2Vec model.

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 using examples from various datasets. We train the model on the MLM objective, and we retrieve the block (sentence) representations from it by averaging over 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 code.

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. The neural architecture of the VDGraph2Vec model is shown in Figure 2.

V. Experiments

In this research, we conduct extensive experiments by examining different node embedding methods and graph neural networks. In this section we demonstrate why it is important to incorporate the structure 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[9] 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* and *PyTorch Geometric* to implement our models. We train the models on a server with two Xeon E5-2697 CPUs, 384GB RAM, and four Nvidia Titan XP graphics cards.

A. 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* 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)[10] and Software Quality Assurance Dataset (SARD)[11]. This dataset was originally available in source code format [27]. *Le et al.* [20] 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,

[9]http://cwe.mitre.org/about/index.html
[10]https://nvd.nist.gov/
[11]https://samate.nist.gov/SARD/

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. 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 adversarial 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 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 I.

As assembly code shares similarities with normal text, it is important that we perform pre-processing on assembly code, similarly to the case of textual data. 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 $\langle$ADDR$\rangle$, and the numerical constants with $\langle$CONST$\rangle$. This improves the semantic quality of our embeddings.

CWE	Vulnerable samples	Non vulnerable samples	Average # of nodes	Average # of edges
121	3100	3100	55.81	71.42
190	3960	3960	52.36	67.21
119	6521	5861	14.71	17.72

TABLE I: Statistics of our datasets.

B. 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: Accuracy, Precision, Recall, F1-score, and AUC-ROC score.

We also study the impact of hyperparameter tuning by evaluating the models on the validation set. We try different settings of learning rate [0.01, 0.001, 0.0001], batch size [100, 128, 256], epochs [50, 75, 100, 150], channels for a GCN convolution layer [16, 32, 64, 128], number of layers for gated graph neural networks [2, 3], and dropout [0, 0.3, 0.4, 0.5]. 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.

C. 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. Similar to *Xu et al.* [38], 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 [22], which uses Instruction2Vec[12] for embedding the assembly instructions and TextCNN for classifying the samples into benign and vulnerable. We try with all possible settings and report the best results these methods can achieve to compare with our model. Following *Baldoni et al.* [3], we use another state-of-the-art model for binary code representation based on word2vec for node representation and Structure2Vec [35] 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 **(VGVec-GCN)*** and *VDGraph2Vec-GGNN* **(VGVec-GGNN)**, with the following variants: *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)***, and *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.

D. Results and Analysis

The results of vulnerability detection on Juliet Test Suite (CWE-121 and CWE-190), and NDSS18 (CWE-119) datasets for various combinations of node embeddings and classification methods are shown in Tables II and III, respectively. The last two rows of the tables denote the performance of our VDGraph2Vec model. In these tables, A represents the accuracy, P the precision, R the recall, F the F1-score and AUC the AUC-ROC score. 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 [8]. We train our models on the entire CWE-121 dataset from

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

Model	CWE-121					CWE-190				
	A	**P**	**R**	**F**	**AUC**	**A**	**P**	**R**	**F**	**AUC**
HF-GCN	70.96	79.28	60.85	68.85	71.55	67.67	69.02	72.89	70.90	67.21
HF-GGNN	71.77	66.81	92.35	77.53	70.58	69.19	71.19	72.19	71.69	68.92
i2v-TCNN	94.83	97.12	92.96	95.0	94.94	90.78	90.06	93.22	91.61	90.56
w2v-s2v	95.32	94.08	97.24	95.63	95.21	93.43	93.11	94.85	93.98	93.31
w2v-GCN	95.81	94.13	98.16	96.11	95.66	95.07	94.71	96.26	94.97	95.48
w2v-GGNN	97.58	98.14	97.24	97.69	97.59	95.41	95.58	96.02	95.81	95.40
RoS2v	97.90	**100.0**	96.02	97.97	98.01	94.57	94.25	95.79	95.01	94.46
VGVec-GCN*	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	99.74	99.53	**100.0**	99.76	99.72
VGVec-GGNN*	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

TABLE II: Vulnerability detection results on Juliet Test Suite (CWE-121 and CWE-190). $*$ denotes the performance of our proposed model, *VDGraph2Vec*.

the Juliet Test Suite and evaluate it on different samples from the test dataset[13]. The results of the experiment are reported in Table IV.

Model	A	P	R	F	AUC
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
VGVec-GCN*	92.9	93.08	94.16	93.61	92.58
VGVec-GGNN*	**95.48**	**95.65**	**96.21**	**95.92**	**95.27**

TABLE III: Vulnerability detection results on CWE-119 from NDSS18 dataset. $*$ denotes the performance of our proposed model, *VDGraph2Vec*.

Model	A	P	R	F	AUC
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
VGVec-GCN*	91.1	**100.0**	82.2	90.23	91.1
VGVec-GGNN*	**94.9**	**100.0**	**89.8**	**94.6**	**94.9**

TABLE IV: 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*.

We also investigate if the difference in accuracies between our model and other state-of-the-art methods is **statistically significant**. VDGraph2Vec achieves statistically 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 also observe that RoBERTa block representations boost the performance more than Word2Vec. Additionally, in comparison to TextCNN and Structure2Vec, an MPNN is able to better embed the nuanced relationships between 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 GGNN surpasses the generalizability power of the GCN 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. 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 obtained from the NDSS18 dataset.

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

VI. Conclusion

We perform thorough experiments to investigate the performance of our model on vulnerability detection. We empirically show that VDGraph2Vec is able to successfully spot vulnerabilities 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. The datasets in this area mostly encompass the source code level. Additionally, 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.

Acknowledgment

This research was funded by NSERC Discovery Grants (RGPIN-2018-03872), Canada Research Chairs Program (950-230623), and the Canadian National Defence Innovation for Defence Excellence and Security (IDEaS W7714-217794/001/SV1). The IDEaS program assists in solving some of Canada's toughest defence and security challenges.

References

[1] A. Abusitta, M. Q. Li, and B. C. Fung. Malware classification and composition analysis: A survey of recent developments. *Journal of Information Security and Applications*, 59:102828, 2021.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[3] R. Baldoni, G. A. Di Luna, L. Massarelli, F. Petroni, and L. Querzoni. Unsupervised features extraction for binary similarity using graph embedding neural networks. *arXiv preprint arXiv:1810.09683*, 2018.

[4] X. Ban, S. Liu, C. Chen, and C. Chua. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency and Computation: Practice and Experience*, 31(19):e5103, 2019.

[5] B. Chernis and R. Verma. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 31–39, 2018.

[6] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[7] K. D. Cooper, T. J. Harvey, and T. Waterman. Building a control-flow graph from scheduled assembly code. Technical report, 2002.

[8] W. A. Dahl, L. Erdodi, and F. M. Zennaro. Stack-based buffer overflow detection using recurrent neural networks. *arXiv preprint arXiv:2012.15116*, 2020.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

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

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[13] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[14] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[15] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[16] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

[19] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning*, pages 1188–1196, 2014.

[20] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *Proceedings of the International Conference on Learning Representations*, 2018.

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

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

[23] M. Q. Li, B. C. Fung, P. Charland, and S. H. Ding. I-mad: Interpretable malware detector using galaxy transformer. *Computers & Security*, 108:102371, 2021.

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

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

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

[27] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

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

[29] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[31] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[32] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[33] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[34] A. Singh. *Identifying malicious code through reverse engineering*, volume 44. Springer Science & Business Media, 2009.

[35] L. Song. Structure2vec: Deep learning for security analytics over graphs. *Atlanta, GA: USENIX Association*, 2018.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[37] F. Wu, J. Wang, J. Liu, and W. Wang. Vulnerability detection with deep learning. In *Proceedings of the 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 1298–1302. IEEE, 2017.

[38] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[39] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, 2020.

[40] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[41] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 10197–10207, 2019.