VulEXplaineR: XAI for Vulnerability Detection on Assembly Code

Samaneh Mahdavifar[1](✉), Mohd Saqib[1], Benjamin C. M. Fung[1], Philippe Charland[2], and Andrew Walenstein[3]

[1] Data Mining & Security Lab, McGill University, Montreal, Canada, H3A 1X1
samaneh.mahdavifar@affiliate.mcgill.ca, mohd.saqib@mail.mcgill.ca, ben.fung@mcgill.ca
[2] Defence Research and Development Canada philippe.charland@drdc-rddc.gc.ca
[3] BlackBerry Limited, Waterloo, Canada walenste@ieee.org

Abstract. Software vulnerabilities have posed significant threats to on-premise as well as cloud servers and applications. So far, numerous studies have focused on identifying and addressing software vulnerabilities at the binary level. Traditional approaches often involve highly complicated static and dynamic analysis techniques. Current intelligent methods are not explainable to reverse engineers, making them incapable of validating the detected vulnerabilities. In this paper, we propose VulEXplaineR, an XAI method for vulnerability detection based on assembly code. It employs BERT for block embedding, augmented with TFIDF of blocks and operand types information, to provide an effective vulnerability detection/explanation framework. VulEXplaineR takes a trained GCNN and its predictions and returns an explanation in the form of a small subgraph of the input graph. It is based on PGExplainer, a perturbation-based global explanation model for GNNs. We augment edge distribution with the edge feature in the form of intra-function jumps between blocks or inter-function calls between functions. The experimental results on the NDSS2018 and Juliet Test datasets demonstrate that VulEXplaineR outperforms the current state-of-the-art baselines in vulnerability detection. Unlike other baseline models, VulEXplaineR provides a high level of explainability as a complementary aid to a reverse engineer, for a more accurate function analysis. We measure fidelity to demonstrate how much two predictions from the extracted subgraph and the original graph match. Furthermore, we conduct a case study to show that VulEXplaineR not only identifies functions and basic blocks that cause the vulnerability, but also highlights interdependencies between those functions and blocks.

Keywords: Vulnerability · explainability · assembly code · BERT · block embedding · graph neural network · subgraph · TFIDF.

1 Introduction

A software vulnerability refers to a weakness in system design, implementation, or operational management that, if exploited, can result in various attacks or system crashes. Software vulnerabilities are released through the CVE database.

When it comes to vulnerability detection methods, traditional approaches, such as reverse engineering, are manually intensive and time-consuming, making them impractical for mitigating zero-day vulnerabilities. Using deep learning and machine learning, we can automate the process of identifying software vulnerabilities and streamlining the process. However, adding intelligence to vulnerability detection has never been an effortless task. There exists a multitude of challenges on the route of designing an efficient effective detector that is capable of identifying zero-day vulnerabilities. The existing methods do not explain why a binary is vulnerable. To date, several Artificial Intelligence (AI)-based binary vulnerability detection methods have been proposed at both the source code and binary level [9], [17]. However, they are not fully explainable to reverse engineers. If they rarely exist, the explanations are low-level and not abstract enough. Thus, reverse engineers cannot benefit from the transparency these methods provide to dissect the observable factors and characteristics of the assembly code. If we can add a human-friendly explanation of the learning representations and the input assembly code, we would enhance the trustworthiness and reliability of the vulnerability detection method we provide. One of the most straightforward explanation approaches is to highlight those parts of the input file that have led to our decision. In this case, the reverse engineer does not need to review all parts of the code instruction by instruction to find vulnerable functions. He can only focus on those parts of the code we have identified by our explainable model. As a result, it increases accuracy and saves time, energy, and resources. From the detection perspective, there have been some attempts to use Graph Neural Networks (GNNs) for training on vulnerability datasets [5], [10]. However, existing graph-based detection methods do not consider edge types in the underlying Control Flow Graph (CFG) and treat all jumps and calls equally. This assumption prevents the GNN from modeling the edge distributions correctly and fails to detect unknown vulnerabilities. We believe that knowing whether an edge represents a jump between two blocks in a function or a call between two functions in a CFG has an impact on identifying some specific vulnerabilities.

The primary objective of this research is to create and implement an eXplainable AI (XAI) model to detect binary vulnerabilities. The key aims are twofold: (1) providing evidence and predictions for reverse engineers and (2) identifying vulnerable behaviors rather than solely focusing on features. To achieve this, we introduce VulEXplaineR, a framework designed to identify vulnerabilities by analyzing assembly code. The initial step involves extracting CFG from the assembly code, followed by using BERT for block embedding, enriched with TFIDF incorporating block information and operand types. This integrated approach aims to establish an effective and scalable framework for vulnerability detection and explanation. Furthermore, we offer explainability through a subgraph of Graph Convolutional Neural Networks (GCNNs), ensuring a high level of comprehension for reverse engineers and an accurate representation of the relationships and interdependencies between blocks. Graph visualization provides a high level of explainability and is very much understandable to humans. Also, it provides an intermediary means for rule explanation. The main contributions of this paper are as follows:

- To the best of our knowledge, this is the first work that provides explainability for vulnerability detection in terms of subgraph of the CFG based on a graph explanation model called PGExplainer. This is also the first work to use block jumps and function calls as the edge distribution for the GCNN, unlike previous methods that treat all jumps and calls equally. The edge distribution provides information about how these connections are distributed throughout the graph.
- We use operand type frequency and TFIDF that provide a lightweight feature vector for detecting and explaining vulnerability. TFIDF and operand types alone demonstrate promising results on datasets with less sophisticated vulnerabilities, such as the Juliet Test Suite.
- We augment operand type frequency and TFIDF with BERT for block embedding. Experimental results show that VulExplaineR provides almost the same classification performance as the best performing baseline model, while providing a high level of explainability.
- We evaluate explainability in terms of fidelity, provide a case study to analyze the extracted graph explanation, and validate it using expert knowledge.

The remainder of the paper is organized as follows: we review related work in Section 2. The VulExplaineR design is described in Section 3. Section 4 provides the experimental results. Finally, Section 5 concludes the paper.

2 Related Work

2.1 Vulnerability Detection

Researchers employ machine learning and deep learning techniques to detect vulnerabilities, utilizing source or binary code analysis. In source code-based vulnerability detection, researchers analyze the software code and extract relevant features. For example, Li et al. [12] propose converting the source code into a numeric vector using representation learning, with the aim of reducing false negatives. Harer et al. [7] apply machine learning to detect vulnerabilities in C and C++ programs. In contrast, Cao et al. [3] utilize deep learning with Bidirectional GNN (BGNN) to capture syntax and semantic information for vulnerability detection. Another study by Garcia et al. [6] employs representation learning, specifically principal component analysis, to generate informative representations for the C source code. Similarly, in the case of Java code, Pang et al. [16], and Hovsepyan et al. [8] represent the code using n-grams and synthetic features, before applying a Support Vector Machine (SVM) for vulnerability detection. Russell et al. [17] explore the use of CNNs and Recurrent Neural Networks (RNNs) on real-world C/C++ code datasets. Although these approaches achieve good evaluation metrics, they heavily rely on access to source code, which is only sometimes available. Therefore, researchers have begun to explore alternative approaches based on binary-level vulnerability detection. Binary-level vulnerability detection has gained popularity in research, due to its advantage of using a standard representation for programs across different programming languages. Scholars have

hypothesized that assembly code shares similarities with natural language processing by focusing on the assembly code obtained through disassembly. For example, Dahl et al. [4] employed representation learning on assembly code and utilized RNNs to detect vulnerabilities. Similarly, Lee et al. [9] used CNNs to process binary code by converting instructions into vectors.

While these approaches have shown promising results, they can only sometimes capture the semantic relationships among code blocks. Researchers have shifted towards graph-based detection methods to address this limitation, aiming for greater accuracy and precision. For example, Diwan et al. [5] employed representation learning using RoBERTa [13] to encode code blocks into vectors and applied message-passing neural networks to process the entire CFG. Previously, Diwan et al. [5] also explored graph-based techniques at the source code level, as demonstrated in [20]. The research conducted by Diwan et al. [5] achieved high accuracy, but one of the challenges of binary-level detection is the difficulty in interpreting the results. Reverse engineers still need help understanding the underlying reasons for the detected vulnerabilities. Furthermore, deep learning algorithms such as GNNs, RNNs, or CNNs are often perceived as black boxes, further complicating the task for reverse engineers. XAI techniques have been considered as potential solutions to explain why black-box models classify software as vulnerable.

2.2 XAI for Vulnerability Detection

Integrating XAI techniques into vulnerability detection has been a relatively understudied area within the existing literature. Most available XAI algorithms have been predominantly developed and tailored for real-world data domains, such as text and images. However, the unique characteristics of vulnerability data require modifications and adaptations of these XAI algorithms to suit this specific domain. To address this gap, researchers have proposed custom algorithms specifically designed to explain source code-level vulnerabilities. For example, Zou et al. [21] introduced a heuristic searching-based tree generation approach to explain vulnerability detection outcomes. Similarly, Li et al. [11] presented a detailed interpretation framework incorporating subgraphs from the Program Dependency Graph (PDG) containing critical statements related to the identified vulnerabilities. Notably, Li et al. [10] made a significant contribution by pioneering the exploration of binary-level vulnerability detection and explanation. However, their approach relied on attention-based graph classification, which may need to possess the sensitivity for local-level explainability in vulnerability detection. In contrast, our proposed algorithm overcomes this limitation by leveraging power gradient analysis and providing individual explanations tailored to vulnerability detection scenarios.

3 Methodology

Explainability is defined as the capacity to convey information clearly to a human audience. It is argued that efficient explanations should be discerning, requiring the choice of 'one or two causalities' from a potentially vast array of causes [15].

3.1 Method of Communication for Vulnerability Detection

In the literature, there are different ways of organizing how an explanation is communicated to an audience, out of which the three following methods could be tailored to the area of vulnerability detection.

- *Input feature explanation* could be based on the whole instruction or tokens (Fig 1). Although it is highly adaptable to different problems, it is limited to providing abstract explanations and controlling the flow of the program.
- *Rule explanations* attempt to explain the model by a simple set of rules. Rules offer the highest level of abstract explanation and are powerful at approximating non-linear decision boundaries. Although they are a local approximation-based explanation method, they can be generalized to the entire dataset and provide global explanations. However, they are complex when applied directly to the assembly code and have high time complexity.
- *Graph visualization* matches the nature of the graphs of assembly code from two perspectives: (1) the structure and semantic information of the assembly code and (2) the relationships and dependencies between blocks. From the visual interpretation viewpoint, graphs are more human understandable and are an intermediary for rule explanation.

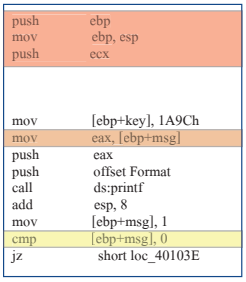

Fig. 1: Input feature explanation

3.2 Graph Explanation

GNNs have been noticeably employed for representation learning in applications [19] that involve graph-structured data, such as social network data and genomic data. The main idea of learning a representation of graph-structured data is to use a message-passing scheme to enable each node in the graph to capture the feature vector of the neighbor nodes. This way, the GNN can capture both the node features and the topology of the graph. GNNs could be used for node classification, link prediction, graph classification, and graph generation. However, GNNS, like other deep neural networks, suffer from not being explainable to humans, because of their black-box nature. In terms of their application to cybersecurity, the lack of explainability hinders security experts to comprehend how complex decisions are made, leading to decreased levels of trust and reliability in the system. A few attempts have been made in the literature to explain graph predictions based on important subgraphs and sets of features. GNNExplainer [18]

is the first general model-agnostic-based approach for explaining GNNs. GNNExplainer explains in terms of a compact subgraph and a small subset of node features that are essential for predicting a specific instance, i.e., a node or a graph in a GNN. However, since GNNExplainer focuses on providing local explainability for a single instance individually, this makes it difficult for the explanations to be generalized to other nodes. Therefore, this approach would not be suitable for comprehending the trained model globally and because of looking at each instance independently, this may generate sub-optimal generalized explanations.

3.3 VulEXplaineR

PGExplainer [14] parametrizes the generation process for explanations that explain multiple instances collectively with a global view of the GNN model. Therefore, it benefits from a higher generalization power. The PGExplainer model enables the inference of explanations for unexplained nodes in an inductive setting, without requiring the retraining of the explanation model. PGExplainer extracts G_s as the explanatory graph, which is the underlying subgraph that makes important contributions to the predictions of a GNN. We have adopted PGExplainer as the base model and exploited the concept of edge distribution to take into account the type of edge in the CFG. For the sake of simplicity, we assume that G_s follows a Gilbert random graph model. In this model, the selection of edges from the original input graph G_0 is treated as conditionally independent of each other. Let V denotes the node set and $E \in V \times V$ be the edge set of Graph G. $e_{ij} \in V \times V$ represents the binary variable that indicates whether the edge is selected, with $e_{ij} = 1$ if the edge (i, j) is selected and 0 otherwise. Further, assume e'_{ij} is the binary variable representing whether the edge is an inter-function call, with $e'_{ij} = 1$ if the edge (i, j) is an inter-function call, and 0 if it is an intra-function call (between blocks). Given G be the random graph variable, based on the above assumptions, the probability of a graph G is factorized as follows:

$$P(G) = \Pi_{(i,j)\in E} P(e_{ij}).P(e'_{ij}) \tag{1}$$

Probabilities $P(e_{ij})$ and $P(e'_{ij})$ could be modeled as Bernoulli distributions, where $e_{ij} \sim Bern(\theta_{ij})$ and $e'_{ij} \sim Bern(\theta'_{ij})$. $P(e_{ij}) = \theta_{ij}$ and $P(e'_{ij}) = \theta'_{ij}$ indicate the probability that edge (i, j) exists in G and is an inter-function call, respectively. The objective function in the PGExplainer algorithm is described as follows [14]:

$$\min_{G_s} H(Y_0|G = G_s) = \min_{G_s} \mathbb{E}_{G_s}\left[H(Y_0|G = G_s)\right] \approx \min_{\Theta} \mathbb{E}_{G_s \sim q(\Theta)}\left[H(Y_0|G = G_s)\right], \tag{2}$$

where $q(\Theta)$ is the distribution of the explanatory graph parameterized by θ and θ'. We apply a relaxation technique to the discrete variable G_s where edge weights, initially binary, are relaxed to continuous variables in the range $(0, 1)$. A reparameterization method is then employed to optimize the objective function efficiently using gradient-based methods. The approach involves using a deterministic function with parameters Ω, temperature τ, and an independent random variable ϵ to approximate the sampling process $G_s \sim q(\theta)$. Therefore, G_s is approximated by $\hat{G}_s = f_\Omega(G_0, \tau, \epsilon)$ and the weight $\hat{e}_{ij} \in (0, 1)$ of edge (i, j) in $\hat{G}_s$ is calculated by:

$$\hat{e}_{ij} = \sigma((log\epsilon - log(1 - \epsilon) + \omega_{ij})/\tau), \;\; \epsilon \sim \text{Uniform}(0, 1), \tag{3}$$

where $\sigma(.)$ is the Sigmoid function, and $\omega_{ij} \in \mathbb{R}$ is the parameter. Thus, with the reparametrization technique, the objective equation in Eq. 2 becomes:

$$\min_{\Omega} \mathbb{E}_{\epsilon \sim \text{Uniform}(0,1)} H(Y_0 | G = G_s). \tag{4}$$

Fig. 2 depicts the explanation process. It takes an input graph G_0 to compute Ω which serves as the latent variable in edge distributions, essentially representing explanations. To extract the explanatory subgraph, the latent variables are employed to select the highest-rank edges. Then a random graph $\hat{G}_s$ is drawn from the edge distributions and fed to the trained GNN to obtain the prediction $\hat{Y}_s$. Finally, the parameters in the explanation network are optimized by minimizing the cross-entropy between the original prediction Y_0 and the updated prediction $\hat{Y}_s$ [14].

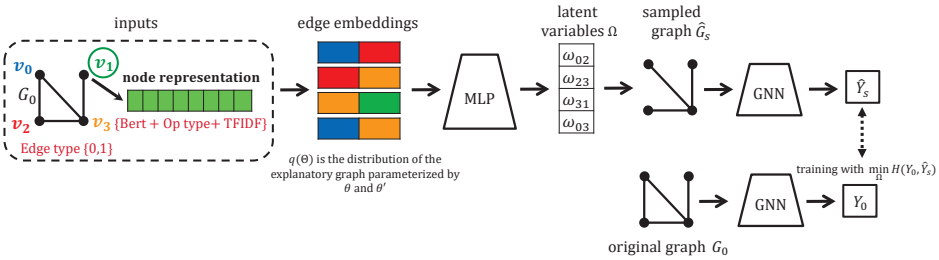

Fig. 2: Extracting explanation graph

Edge distribution is often used to describe how features are propagated between nodes through the edges of the graph. The edge distribution can define how neighboring nodes influence each other during message passing in a GNN. The distribution may determine how much weight or importance is assigned to the information coming from each neighbor. For example, a uniform edge distribution may imply that each neighbor contributes equally to the node's updated representation, while a learned distribution may assign different weights to different neighbors based on the model's learning. In a CFG, the probability distribution of the edges is very important to capture meaningful dependencies between blocks in the graph and therefore, detect zero-day vulnerabilities. To augment the edge distribution in a CFG, we concatenate the edge type as an entry in the edge embedding from which latent variables are computed. Edge types are either intra-function calls between blocks, indicated as zero, or inter-function calls between functions, indicated as one in the edge embedding. Fig. 3 shows the overview of the VulEXplaineR framework to predict and explain vulnerability in terms of a subgraph of a GCNN that represents a CFG. First, the input binary files are disassembled into assembly code blocks using a disassembler. Then, we extract the CFG of the program that represents the graphical representation of its different execution paths. Each node in a CFG stands for a basic block and an edge of the graph connects these basic blocks that shows the flow of the program execution. We generate the CFG of the entire assembly file by connecting the functions through the basic blocks of each function that call one another. Using this representation, we can discover risky program execution paths of a file.

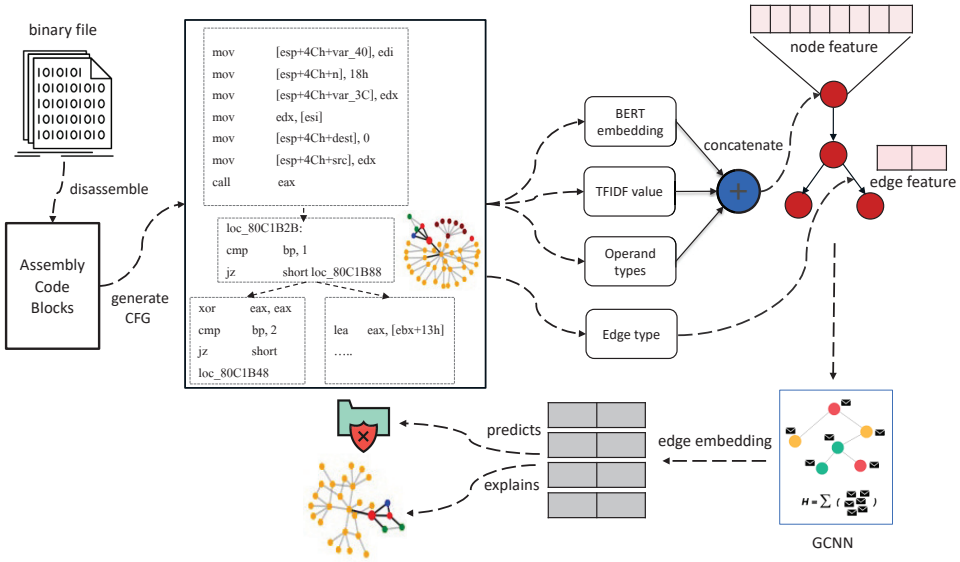

Fig. 3: A model overview of the VulEXplaineR framework

After that, we need to create the embeddings for the nodes (blocks) and the edges (connectivity between blocks) [5] of the CFG and arrange them in a structure to serve as an input to a GCNN. The node embeddings consist of several components, including the TFIDF value, the BERT embedding, and the frequency of operand types in a basic block. All of these feature vectors are concatenated and serve as block embedding in each node. The edge feature would be the call type (intra-function or inter-function) which is incorporated into the edge embedding for the explanation part. The GCNN employs a message-passing mechanism to generate an enhanced binary file representation of the input graph and then predicts whether the input file is vulnerable or not. In a GCNN, each node which represents a block receives information from its neighbors and accumulates the vectors to create a block embedding.

Finally, the node embeddings and the edge embeddings are used to create the edge distributions and create an explanatory subgraph. Using the edge distribution, we can determine the weight of the information coming from each neighbor. To compute the TFIDF value of each block, we assume each block is a word and employ TFIDF on the whole binary file. Using this approach, we can capture the semantic relationship between the blocks, and similar blocks would have a TFIDF value at a proximity. BERT embedding is applied to the instructions in a block and then an average is taken between the embeddings of the instructions. Using pre-trained BERT methods, we can ensure to preserve the semantics between different tokens in assembly instructions so that similar basic blocks have embeddings close to each other. For the operand types, we calculate the frequency of each distinct operand type and create a bag-of-words type of feature vector. There exist seven operand types: (0) No Operand, (1) General Register (al, ax,

es, ds...) (reg), (2) Direct Memory Reference (DATA) (addr), (3) Memory Ref [Base Reg + Index Reg] (phrase), (4) Memory Reg [Base Reg + Index Reg + Displacement] (phrase + addr), (5) Immediate Value (value), (6) Immediate Far Address (CODE) (addr), and (7) Immediate Near Address (CODE) (addr).

4 Experimental Results

4.1 Dataset

The NDSS18 dataset originated from the National Institute of Standards and Technology (NIST) and the Software Assurance Reference Dataset (SARD) project. It includes a total of 32,281 binary files, which can be categorized into Windows and Linux platforms and are associated with two specific Common Weakness Enumerations (CWEs), CWE-119 and CWE-399. Notably, the dataset is well balanced in terms of vulnerability labels both the Windows and Linux platforms. CWE-399 is related to a specific type of software vulnerability, known as "Resource Management Error" and CWE-119 is related to improper restriction of operations within the bounds of a memory buffer. The Juliet Test Suite is the second dataset we use that contains 118 distinct CWEs. This synthetic dataset comprises a total of 83,624 binary files, with an equal distribution of labels between vulnerable and non-vulnerable files. We analyzed our model on CWE-121 and CWE-190 from the Juliet Test Suite and on CWE-119 from the NDSS18 dataset as benchmarks. CWE-121 (Stack-based Buffer Overflow) is related to situations where excessive data is written to a stack buffer, potentially leading to a buffer overflow. Moreover, CWE-190 (Integer Overflow or Wraparound) involves errors in the calculation of integer values, leading to overflow or wraparound. A graph structure proves to be more appropriate for identifying and understanding these vulnerabilities, due to multiple functions. Table 1 shows the distribution of both datasets in terms of vulnerable and non-vulnerable samples. To normalize the assembly instructions,

Table 1: Dataset distributions

Dataset	# **Vulnerable samples**	# **Non-vulnerable samples**
NDSS18	8978	8999
Juliet Test Suite	7060	7060

the following steps have been done on the instructions in each block: (1) removing the heading memory address, (2) replacing constants with 'const', e.g., '0Bh' is replaced with the word 'const', and (3) replacing effective addresses with 'addr', e.g., $[ebp + Var\_C]$ is replaced with the word 'addr'. Skipping the instruction normalization step would result in generating a huge list of distinct vocabularies for the language modeling task and not allowing the BERT algorithm to extract well-represented block embeddings.

4.2 Experimental Setup

The experiments were performed on an AMD Ryzen Threadripper PRO 3975WX 32-Core 3.50 GHz with 512 GB RAM running Windows Server 2022. VulEX-

plaineR was implemented using Python and Pytorch. We used the PGExplainer Github code repository to implement VulEXplaineR and revised the code [4] for pre-processing of the binary files (instruction normalization, creating block embedding, generating CFGs, generating node embedding), adding edge distributions, and computing fidelity metrics. To disassemble the binary files, we used IDA Pro version 8.2.230124. For implementing BERT models, 'bert-mini' from Hugging Face was employed, which is one of the smaller Pytorch pre-trained BERT variants. The graph convolutional layers were implemented using the GCNConv library from 'torch_geometric.nn'.

To evaluate the performance of our model, we split our dataset into training (80%), validation (10%), and testing (10%). The vulnerability detection task is a binary classification that classifies binary files as vulnerable (positive) or non-vulnerable (negative). We add explainability on top of the classification model by generating subgraphs of the initial graph. We use accuracy, recall, precision, and F1 score as evaluation metrics to calculate the classification performance. To assess explainability, we compute the classification performance of the generated subgraphs. Also, we use the fidelity metric to show the ratio of test examples that both GCNN and VulEXplaineR can agree on for the classification result.

Hyperparameter Fine-tuning We used cross-entropy loss for the loss function and Adam as the optimization algorithm for GCNN. We managed to fine-tune the learning rate in different ranges of [0.1, 0.01, 0.001, 0.0001]. More fine-tuning was conducted for the batch size [64, 100, 150, 200] and the number of epochs [200, 400, 600, 1000]. We chose the best setting that resulted in the best accuracy in the validation set. Thus, 0.001, 100, and 600 were selected as the best settings for the learning rate, batch size, and number of epochs, respectively. We needed to fine-tune the architecture of the GCNN including the convolutional layers. The best model consists of three stacked GCN layers followed by a linear layer. Max and Mean pooling is then used between the GCN outputs and the linear layers.

Models to Compare We compare two variants of our model with two different combinations of node features with GCNN as the base GNN for both variants, VulEXplaineR$_{FS1}$ and VulEXplaineR$_{FS2}$. The first group of node features (FS1) includes BERT embedding, operand types, and TFIDF Value, while the second group (FS2) comprises operand types and TFIDF Value. The baseline models are VulGCNN$_{FS1}$ which is a vulnerability detector using a GCNN and employs BERT embedding, operand types, and TFIDF Value as the node features, VulTFIDF$_{RF}$ that uses Random Forest (RF) as a classifier and TFIDF embeddings of all blocks as the feature vector, VDGraph2Vec [5] with GCNN, VulANalyzeR [10], and i2v-TCNN [5] which uses Instruction2Vec [9] for embedding the assembly instructions and TextCNN for classifying the samples into benign and vulnerable.

4.3 Results

In Tables 2 and 3, we compare both variants of VulEXplaineR with the baseline algorithms VulGCNN$_{FS1}$, VulTFIDF$_{RF}$, VDGraph2Vec, i2v-TCNN, and VulAN-

[4] https://github.com/Sam-Mah/VulExplainer

alyzeR on the NDSS18 and Juliet Test datasets, respectively. For the baseline algorithms, namely VDGraph2Vec, i2v-TCNN, and VulANalyzeR, we chose the setting with the best results in the papers. Both variants of VulEXplaineR are the only models that provide full explainability for the vulnerability detection task. VulANalyzeR provides a local-level explanation at the instruction level, while VulEXplaineR provides a global-level explanation, which preserves the local fidelity. As shown in Table 2, VulEXplaineR$_{FS1}$ achieves an accuracy of 96.02% and precision 97.31%, 0.54% higher accuracy and 1.66% higher precision acquired from the best performing baseline algorithm, i.e., VDGraph2Vec. As for other measures such as recall and F1, there is a negligible difference of $\approx$ 0.2% and 0.4%. Considering the fact that VDGraph2Vec does not provide any level of explainability, we can conclude that VulEXplaineR is highly efficient in terms of all measures. Similarly, in Table 3, VulEXplaineR provides nearly the same performance of the best performing baseline model, VDGraph2Vec, which lacks explainability. Overall, the results are representative of the fact that VulEXplaineR is capable of reproducing the accuracy of the GNNs from which it was derived.

Table 2: Performance results on the NDSS18 dataset

Models	**Accuracy**	**Recall**	**Precision**	**F1**
VulEXplaineR$_{FS1}$	**96.02**	**96.02**	**97.31**	**95.51**
VulEXplaineR$_{FS2}$	78.75	78.75	79.67	78.35
VulGCNN$_{FS1}$	93.02	93.02	95.81	92.99
VulTFIDF$_{RF}$	70.59	70.59	70.59	70.46
VDGraph2Vec	95.48	96.21	95.65	95.92
i2v-TCNN	81.41	82.50	83.72	83.11
VulANalyzeR	89.53	94.18	85.36	90.10

Table 3: Performance results on the Juliet Test dataset

Models	**Accuracy**	**Recall**	**Precision**	**F1**
VulEXplaineR$_{FS1}$	**99.80**	**99.80**	**99.81**	**99.80**
VulEXplaineR$_{FS2}$	94.79	96.65	93.40	95.00
VulGCNN$_{FS1}$	98.55	98.55	98.56	98.55
VulTFIDF$_{RF}$	87.47	87.47	88.25	87.42
VDGraph2Vec	100	100	100	100
i2v-TCNN	92.81	93.1	93.59	93.31
VulANalyzeR	99.68	100	99.38	99.69

4.4 Graph Explanation

In this section, we present the results of two sets of experiments, namely qualitative and quantitative, to evaluate the quality of the graph explanations. In quantitative evaluations, we calculate the fidelity of the subgraphs to show how much it mimics the behaviour of the original graph it is extracted from. The qualitative evaluation is the ground truth analysis of the extracted subgraph with

respect to the expert knowledge. In this task, a reverse engineer thoroughly examines the subgraph associated with the Buffer Overflow Vulnerability (CWE 121) to validate the extracted subgraph and assess its level of representation, compared to the original graph from which it is derived.

Fidelity Analysis. Fidelity refers to the degree to which the predictions of the extracted subgraphs align with the actual GNN. The higher the fidelity metric, the more faithful the extracted subgraph is to the original graph. Fidelity ($\mathcal{F}$) values range between zero and one, based on Eq. 5:

$$\mathcal{F} = cnt/len(\mathcal{O}_p) \ , \tag{5}$$

where *cnt* is the number of the times $(\mathcal{O}_p - \mathcal{M}_p)$ vector is zero. $\mathcal{O}_p$ are the prediction vectors based on the original graph and $\mathcal{M}_p$ are the prediction vectors based on the current explanation, on the test dataset. The fidelity values of VulEXplaineR$_{FS1}$ and VulEXplaineR$_{FS2}$ on the NDSS18 dataset and the Juliet Test Suite are depicted in table 4. The results show that the extracted subgraph

Table 4: Fidelity of VulEXplaineR

Dataset	**VulEXplaineR**$_{FS1}$	**VulEXplaineR**$_{FS2}$
NDSS18	0.95	0.80
Juliet Test Suite	0.99	0.99

and the underlying GNN highly agree with the predictions they make, except for the scenario with the lightweight feature set *FS2* (operand type frequency and TFIDF) on the NDSS18 dataset that contains non-synthetic vulnerabilities.

Ground Truth Analysis of Buffer Overflow Vulnerabilities. Buffer overflow vulnerabilities are a common type of security weakness in software programs that can be exploited by attackers to execute their own code or overwrite critical data. In this paper, we focus specifically on buffer overflow vulnerabilities caused by improper bounds checking, weak input validation, and lack of stack protection. In addition to detecting whether a program is vulnerable to buffer overflow attacks, we provide explainability by extracting a subgraph from the CFG and offering justifications for vulnerability. We also link each vulnerability to the CWE list, which is a community-developed list of software and hardware weaknesses.

Summary of Extracted Subgraph. VulEXplaineR extracted a subgraph of the main CFG of a vulnerable binary, as shown in Fig. 4[5]. In these explanations, bold black edges indicate top-k edges ranked by their importance weights, where k is set to 50 by hyper-parameter fine tuning. The disassembled code consists of several blocks of instructions, each labeled with an address in memory. There are function call names in the code blocks that indicate the vulnerability. We have removed these debug symbols before conducting experiments. The first block at memory address 4096 checks if a certain value is null and calls a function if it

[5] Details about the subgraph and the code are provided in the supplementary data.

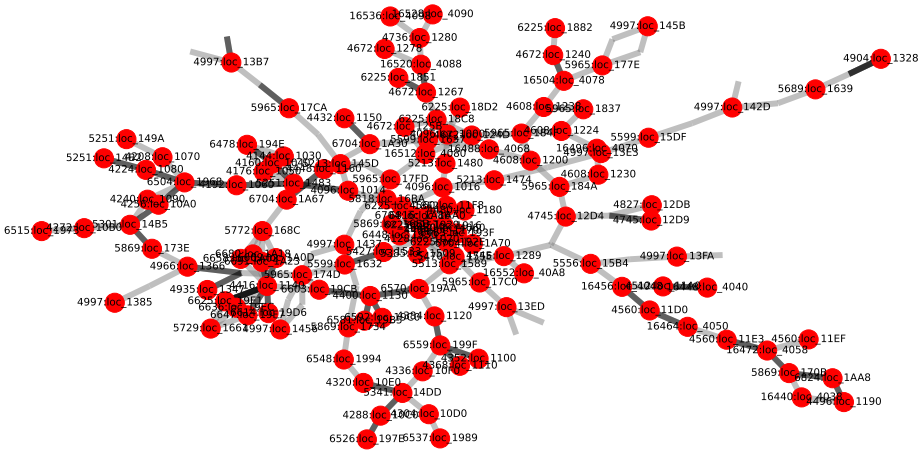

Fig. 4: Extracted explanations as a subgraph indicated by bold black edges calculated as top-k edges. Each node of the graph is a block represented as *function id: block name*

is not. The next several blocks starting at memory address 4128 and ending at memory address 4496 seem to be setting up some constants. The block at memory address 4512 appears to be the main function, which initializes some values and calls other functions. The blocks at memory addresses 4560 and 4608 appear to be functions that manipulate the data in the BSS section of the program. The block at memory address 4672 appears to be a function that calls other functions to clean up the program. The block at memory address 4736 appears to be a function that registers some clones. Providing a justification or ground truth for a subgraph extracted from a model is challenging in the absence of complete code details. To determine the actual occurrences and functions that can serve as a backdoor for an attacker, it may be necessary to execute the code in a sandbox environment with the aid of an executable file. However, this approach is not always feasible. An alternative approach involves a manual reading of the extracted code to identify vulnerabilities. Using this method, we establish a ground truth for the explainability extracted from the model. A list of vulnerable functions along with their vulnerable basic blocks is described as follows:

```
4827:loc_12DB
['endbr64', 'push rbp', 'mov rbp rsp', 'sub rsp const', 'mov addr edi', 'mov
    addr rsi', 'mov edi const', 'call _time', 'mov edi eax', 'call _srand', '
    lea rdi acallingbad', 'call printline', 'mov eax const', 'call
    cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54_bad', '
    lea rdi afinishedbad', 'call printline', 'mov eax const', 'leave', 'retn']
```

This function block has instructions that push values onto the stack `'push rbp', 'sub rsp const', 'mov addr edi', 'mov addr rsi', 'lea rdi acallingbad', 'call printline', 'mov eax const'` and call other functions that could write to the stack without checking the size of the buffer, such as 'printline'. This leaves the program vulnerable to a stack-based buffer overflow attack. The function `cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54_bad` describes

a weakness related to stack-based buffer overflows (CWE121), where a buffer overflow can occur if an attacker can cause more data to be written to a buffer than it can hold [2]. In this scenario, the weakness is caused by a loop that reads input into a structure, with no validation to ensure that the input does not exceed the size of the structure's buffer. This can lead to the overwriting of adjacent memory areas and cause unexpected behavior or crashes [2].

```
4935:loc_1347
['endbr64', 'push rbp', 'mov rbp rsp', 'sub rsp const', 'mov addr rdi', 'mov
    rax addr', 'mov rdi rax', 'call
    cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54d_badsink
    ', 'nop', 'leave', 'retn']
```

```
4904:loc_1328
['endbr64', 'push rbp', 'mov rbp rsp', 'sub rsp const', 'mov addr rdi', 'mov
    rax addr', 'mov rdi rax', 'call
    cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54c_badsink
    ', 'nop', 'leave', 'retn']
```

```
4966:loc_1366
['endbr64', 'push rbp', 'mov rbp rsp', 'sub rsp const', 'mov addr rdi', 'mov
    rax addr', 'mov rdi rax', 'call
    cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54e_badsink
    ', 'nop', 'leave', 'retn']
```

The above function blocks have instructions that push values onto the stack `'push rbp', 'mov rbp rsp', 'sub rsp const', 'mov addr rdi'` and call `cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54c_bad` which could potentially write to the stack without checking the size of the buffer (CWE-805). This leaves the program vulnerable to a stack-based buffer overflow attack. The function `cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54c_badsink` describes a similar scenario where a bad sink is present, causing the same vulnerability (CWE-805) [1]. A sink is any place in a program where data is received from an untrusted source, and a bad sink is one that does not properly validate or sanitize the incoming data, allowing attackers to inject malicious input [1].

```
4096:loc_1000
['endbr64', 'sub rsp const', 'mov rax cs:__gmon_start___ptr', 'test rax rax', '
    jz loc_1016']
```

The above function has an instruction that jumps to `__gmon_start___ptr`, which could potentially be modified to point to an attacker-controlled address. If this happens, an attacker could execute arbitrary code by crafting a specially-crafted object file that causes the program to jump to the attacker-controlled address.

```
4672:loc_125B
['mov rdi cs:__dso_handle', 'call __cxa_finalize']
```

This function has an instruction that calls `__cxa_finalize`, passing in a value from `__dso_handle`. If attackers can overwrite the value of `__dso_handle`, they could execute arbitrary code by crafting a specially-crafted shared library that causes the program to execute the attacker's code when `__cxa_finalize` is called.

```
4512:loc_11A0
['endbr64', 'xor ebp ebp', 'mov r9 rdx', 'pop rsi', 'mov rdx rsp', 'and rsp
    const', 'push rax', 'push rsp', 'lea r8 __libc_csu_fini', 'lea rcx
    __libc_csu_init', 'lea rdi main', 'call cs:__libc_start_main_ptr', 'hlt']
```

The above function has instructions that push values onto the stack `push rsp, lea r8 __libc_csu_fini, lea rcx __libc_csu_init, lea rdi main` and call other functions that could write to the stack without checking the size of the buffer, such as printf, scanf, and wprintf. This leaves the program vulnerable to a stack-based buffer overflow attack.

Mainly, two functions `cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54_bad` and `cwe121_stack_based_buffer_overflow__cwe805_struct_declare_loop_54c_badsink` both belong to CWE lists [2], [1] and describe similar weaknesses related to stack-based buffer overflows caused by inadequate input validation, with one involving a loop and the other involving a bad sink. To mitigate such vulnerabilities, it is essential to validate all user input to ensure that it does not exceed the size of the buffer and that it is correctly sanitized to remove any potentially harmful content.

5 Conclusion

In this paper, we propose VulEXplaineR, an XAI method for vulnerability detection in assembly code. Utilizing BERT and TFIDF, it offers an efficient framework to represent relationships between blocks and functions. Inspired by PGExplainer, VulEXplaineR produces explanations in the form of subgraphs of GCNNs, incorporating edge embeddings for enhanced accuracy. Experimental results on the NDSS2018 and Juliet Test datasets show that VulEXplaineR outperforms state-of-the-art baselines, providing high explainability that matches the graph nature of the assembly code and is valuable for reverse engineers. Qualitative and quantitative evaluations, including fidelity metrics, demonstrate the method's effectiveness. A case study highlights VulEXplaineR's ability to identify vulnerabilities and dependencies within the extracted subgraph. One of the directions of future work is to design motifs as the ground truth to conduct a quantitative evaluation of the extracted subgraph in the form of binary edge classification. Those edges that fall inside the motifs are positive edges, and those that fall outside the motifs are negative edges. Another line of research would be to model the underlying GNN as a directed graph. A directed graph imposes an ordering on a pair of nodes that is useful, as it further describes the relationship between the nodes.

Acknowledgments. This research is supported by BlackBerry Limited, Defence Research & Development Canada, and NSERC Alliance Grants (ALLRP 561035-20).

References

1. Cwe-805: Buffer access with incorrect length value. https://cwe.mitre.org (A-2024)
2. Cwe-121: Stack-based buffer overflow. https://cwe.mitre.org (Accessed-2024)
3. Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. **136**, 106576 (2021)

4. Dahl, W.A., Erdodi, L., Zennaro, F.M.: Stack-based buffer overflow detection using recurrent neural networks. arXiv preprint arXiv:2012.15116 (2020)
5. Diwan, A., Li, M.Q., Fung, B.C.: Vdgraph2vec: Vulnerability detection in assembly code using message passing neural networks. In: 21st Int. Conf. on Machine Learning and Applications (ICMLA). pp. 1039–1046. IEEE (2022)
6. Garcia, F.D., de Koning Gans, G., Verdult, R.: Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning (2011)
7. 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:1803.04497 (2018)
8. Hovsepyan, A., Scandariato, R., Joosen, W., Walden, J.: Software vulnerability prediction using text analysis techniques. In: 4th Int. Workshop on Security Measurements and Metrics. pp. 7–10 (2012)
9. Lee, Y., Kwon, H., Choi, S.H., Lim, S.H., Baek, S.H., Park, K.W.: Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN. Appl. Sci. **9**(19), 4086 (2019)
10. Li, L., Ding, S.H., Tian, Y., Fung, B.C., Charland, P., Ou, W., Song, L., Chen, C.: Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. ACM Trans. Priv. Secur. **26**(3), 1–25 (2023)
11. Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations. In: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303 (2021)
12. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv:1801.01681 (2018)
13. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
14. Luo, D., Cheng, W., Xu, D., Yu, W., Zong, B., Chen, H., Zhang, X.: Parameterized explainer for graph neural network. Advances in Neural Information Processing Systems **33**, 19620–19631 (2020)
15. Madsen, A., Reddy, S., Chandar, S.: Post-hoc interpretability for neural nlp: A survey. ACM Comput. Surv. **55**(8), 1–42 (2022)
16. Pang, Y., Xue, X., Namin, A.S.: Predicting vulnerable software components through n-gram analysis and statistical feature selection. In: 14th Int. Conf. on Machine Learning and Applications (ICMLA). pp. 543–548. IEEE (2015)
17. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 17th Int. Conf. on Machine Learning and Applications (ICMLA). pp. 757–762. IEEE (2018)
18. Ying, Z., Bourgeois, D., You, J., Zitnik, M., Leskovec, J.: Gnnexplainer: Generating explanations for graph neural networks. Advances in Neural Information Processing Systems **32** (2019)
19. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M.: Graph neural networks: A review of methods and applications. AI open **1**, 57–81 (2020)
20. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in Neural Information Processing Systems **32** (2019)
21. Zou, D., Zhu, Y., Xu, S., Li, Z., Jin, H., Ye, H.: Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. ACM Trans. Softw. Eng. Methodol. (TOSEM) **30**(2), 1–31 (2021)