The ISC Int'l Journal of Information Security
ISeCure

Manuscript template of ISeCure Journal (pp. 1–8)
http://www.isecure-journal.org

Learning to Locate: GNN-Powered Vulnerability Path Discovery in Open Source Code

Nima Atashin, Behrouz Tork Ladani *, and Mohammadreza Sharbaf
[1] *Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran*

ARTICLE INFO.

Keywords:
Vulnerability Path Discovery, Explainable AI, Graph Neural Networks, Program Slicing, Vulnerability Detection

Abstract

Detecting security vulnerabilities in open-source software is a critical task that is highly regarded in the related research communities. Several approaches have been proposed in the literature for detecting vulnerable codes and identifying the classes of vulnerabilities. However, there is still room to work in explaining the root causes of detected vulnerabilities through locating vulnerable statements and the discovery of paths leading to the activation of the vulnerability. While frameworks like SliceLocator offer explanations by identifying vulnerable paths, they rely on rule-based sink identification that limits their generalization. In this paper, we introduce VulPathFinder, an explainable vulnerability path discovery framework that enhances SliceLocator's methodology by utilizing a novel Graph Neural Network (GNN) model for detecting sink statements, rather than relying on predefined rules. The proposed GNN captures semantic and syntactic dependencies to find potential sink points (PSPs), which are candidate statements where vulnerable paths end. After detecting PSPs, program slicing can be used to extract potentially vulnerable paths, which are then ranked by feeding them back into the target graph-based detector. Ultimately, the most probable path is returned, explaining the root cause of the detected vulnerability. We demonstrated the effectiveness of the proposed approach by performing evaluations on a benchmark of the buffer overflow CWEs from the SARD dataset, providing explanations for the corresponding detected vulnerabilities. The results show that VulPathFinder outperforms both original SliceLocator and GNNExplainer (as a general GNN explainability tool) in discovery of vulnerability paths to identified PSPs.

1 Introduction

Modern software systems are increasingly exposed to security vulnerabilities. Many of these are reported through the Common Vulnerabilities and Exposures (CVE) database [1]. To defend against these threats, researchers have developed different automated vulnerability detection methods. Graph-based methods, in particular, have shown superior success due to their ability to capture the structural and semantic dependencies in code [2]. Despite their effectiveness in detecting vulnerable code, most current graph-based models act as black boxes, offering little to no insight into why a particular code is flagged as vulnerable.

* Corresponding author.
Email addresses: nima.atashin@eng.ui.ac.ir, ladani@eng.ui.ac.ir, m.sharbaf@eng.ui.ac.ir
ISSN: 2008-2045

Without such an explanation, it would be difficult for developers to debug and mitigate detected flaws.

Vulnerability detection techniques can generally be grouped into two main categories: rule-based methods, which include both static and dynamic analysis, and data-driven approaches [5]. Because it is difficult to define vulnerabilities, rule-based methods suffer from high false-positive rates, especially on complex code [5]. In contrast, data-driven methods such as deep learning have emerged as powerful alternatives capable of generalizing from large code corpora. This capability is enabled by the extensive availability of open-source vulnerability data, which provides a rich foundation for training and analysis [6]. Data-driven approaches can learn the latent information from vulnerable patterns and have shown better performance compared to static tools that utilize predefined rules [5].

Among data-driven approaches, both sequence-based and graph-based approaches have been widely explored [5]. Sequence-based methods serialize code into tokens and apply neural networks to identify vulnerability patterns. Graph-based models have proven effective by representing code as abstract syntax trees (ASTs), control-flow graphs (CFGs), or program dependence graphs (PDGs), enabling them to capture structural and semantic code dependencies [7]. However, despite their success, these models often yield coarse-grained predictions and lack transparency, making it difficult for developers to understand why a function or code snippet is flagged as vulnerable. This black-box nature poses significant challenges for analyzing root cause, trust, and fixing.

To address the limitation mentioned above, we propose VulPathFinder, a Graph Neural Network (GNN)-based approach for identifying most probable paths from the potential sources to the detected vulnerability sink statements. VulPathFinder enhances the vulnerability path discovery method used by SliceLocator [4] by utilizing a GNN model to first detect potential sink points (PSPs), i.e., the statements that are more likely to be the last chain of a vulnerable trace in the code. Unlike rule-based methods such as SliceLocator, which consider a set of predefined rules to identify candidate sink points, our method will be context-aware and capable of generalizing to unseen sink statements. Indeed, by training a GNN model to find PSPs, VulPathFinder better captures complex vulnerability patterns, retaining control and data dependencies between statements that might not be covered by rule-based approaches. After finding PSPs, inspired by the SliceLocator, we perform backward slicing starting from each of the sink points in the list. As a result, we will have a list of candidate paths leading to a sink point that make some corresponding subgraphs. Subgraphs are then fed into off-the-shelf graph-based detectors to compute their likelihood of being vulnerable. The subgraph with the most likelihood of being vulnerable is finally chosen. It shows the corresponding best candidate vulnerable path to be considered as the explanation of the detected vulnerability.

To evaluate the performance of the proposed model for sink point detection, we used a set of standard classification metrics. Moreover, to show the end-to-end performance of the explanation method (explainability) against the rival methods, we used the Triggering Line Coverage (TLC) metric [4] to compare the achieved results with the original SliceLocator as well as GNNExplainer [8]. The latter is a model-agnostic explanation method for GNNs that is most influential for a given prediction. The results achieved show that VulPathFinder not only brings in acceptable precision and recall in sink point detection but also brings in higher end-to-end performance in terms of TLC that shows better explainability.

The rest of the paper is organized as follows: Section 2 reviews the related work in conventional static and dynamic approaches, deep learning, and explainable AI approaches. In Section 3, the proposed method is explained. In Section 4, experimental setup, evaluation metrics, and implementation details are explained. The results are shown in Section 5. Limitations are addressed in Section 6, and finally, we conclude the paper in Section 7.

2 Related Work

2.1 Conventional Static and Dynamic approaches

Static analysis tools such as CodeQL [9] and FindBugs [10] use fixed rules to find vulnerabilities without executing the code; however, they suffer from high false positives and may miss complex vulnerabilities because defining vulnerable patterns is a challenging task [5]. Dynamic analysis tools such as Valgrind [11] and AddressSanitizer [12] find vulnerabilities at runtime, but they depend on test cases and may miss unexecuted paths.

2.2 Deep learning-based approaches

The use of deep learning for detecting vulnerable functions and code snippets has increased rapidly in recent years, thanks to the abundant vulnerable open-source datasets [6]. Graph Neural Networks (GNNs), in particular, have shown strong capability in capturing patterns inside graphs and have been widely applied to tasks such as traffic analysis [13] and social network modeling [14]. By representing source code as a graph,

graph-based models can be leveraged to find intrinsic semantic and structural patterns by retaining control and data dependency inside code [7]. There exist different graph representations, such as abstract syntax tree (AST), control flow graph (CFG), control dependence graph (CDG), data dependence graph (DDG), and code property graph (CPG). CPG integrates AST, CFG, CDG, and DDG to create a unified view that encodes the syntactic and semantic dependencies [7]. Some works have used solely the sequence of tokens as their code representation, but by mapping code to a graph $G = (V, E)$, where V are nodes which denote entities like variables or statements, and E are edges inside the graph which show dependencies between two entities, we can better represent dependencies among statements.

Several works have used GNN, such as Devign [15], which is a Gated Graph Recurrent Network-based method that represents source code in a composite graph of ASTs, CFGs, and DFGs. Reveal [2] first extracts rich syntax-semantic features using Gated Graph Neural Network and embeds these features via code property graphs, then maximizes the separation between vulnerable and non-vulnerable code in real-world datasets using representation learning. DeepWukong [16] first generates the CFG and Variable Flow Graph (VFG) to construct a PDG. It then conducts forward and backward traversals to create an Extended Flow Graph (XFG). Then it converts statement tokens into Doc2Vec vectors. Subsequently, information from XFG edges, along with vectorized code tokens, is used as input for k-GNNs. IVDetect [17] is an interpretable method that first produces sub-token sequences of the code and then leverages BGRU with an attention mechanism to integrate ASTs, variable names, type features, sub-token sequences, and data/control dependencies into a comprehensive code representation. The representation is then processed by the Feature-Attention Graph Convolutional Network (GCN) model for training. Then it employs an explanation model called GNNExplainer [8] to identify critical sub-graphs as the explanation.

2.3 Explanation approaches

Despite the effectiveness of GNN-based detectors at flagging vulnerable code, the interpretations and explanations of the cause of vulnerability remain unknown; these models just output a prediction score for each input without explaining the cause of the prediction. Recently, explainable AI (XAI) has emerged to address this gap [18]. Several techniques, such as GNNExplainer [8], CFExplainer [19], demonstrate the cause of predictions yielded by models by highlighting parts of the input that influence model outputs. GNNExplainer learns a minimal subgraph and a subset of node features that alone are sufficient to yield the same prediction as the full graph [8]. CFExplainer is a counterfactual explanation that identifies the smallest modifications to a graph's structure needed to reverse the model's prediction [19]. In the scope of vulnerability detection, it highlights which structural modifications could transform a code snippet from being classified as vulnerable to non-vulnerable, or vice versa. However, these methods often struggle with granularity and usability when applied to complex source code. This is because these models capture the difference between vulnerable code and non-vulnerable code without capturing the intrinsic behavior of vulnerabilities and their execution paths, and a slight change in input results in drastically different explanations. Also, most of the explainers deal with the models themselves, ignoring insights about the taint tracking and slicing. So, static analysis concepts such as taint propagation and slicing can be a promising complement to explainers.

3 The Proposed Approach

In this section, we present our framework, VulPathFinder, which enhances vulnerability path discovery by utilizing a GNN model to detect PSPs. Previous works (including SliceLocator) [3, 4] considered predefined rules—such as those related to library/API call, array usage, pointer usage, and arithmetic operations—to locate candidate sink points. In contrast, we employ a data-driven approach to locate candidate vulnerability sink points. This is the most important difference between our work and previous ones. By training a GNN model to find PSPs, our method better captures complex vulnerability patterns, retaining control and data dependencies between statements that might not be covered by rule-based approaches [7]. VulPathFinder offers several advantages over rule-based approaches. First, by training a specific model to identify PSPs, we can have a context-aware model that can capture vulnerable patterns. Second, VulPathFinder can be generalized to find unseen sink points across various vulnerability types.

The overall framework is depicted in Figure 1, which consists of four main phases:

(1) Training GNN model for Sink Point Detection,
(2) Identification of PSPs: In this phase, taking advantage of the trained GNN model from the previous step, we find a list of PSPs,
(3) Flow Path Generation: By having the list of PSPs, we perform backward slicing starting from each of the sink points in the list. At the end of this phase, we will have a list of candidate paths leading to a sink point.,
(4) Flow Path Selection: In the last phase, the pre-

diction score of each path is separately fed into the graph-based detector, and the path with the prediction score closest to the prediction score of the whole original graph will be chosen as the vulnerable path and considered as the explanation of the vulnerability.

Figure 2 shows an example of a buffer overflow function, and its corresponding CPG is illustrated in Figure 3. We can see that the sink line is line 8, and several paths can be extracted by performing backward slicing, such as 1 –> 5 –> 8, 1 –> 6 –> 8, 7 –> 8, etc. So by ranking these paths based on their prediction score, we can select the path with the highest score as the explanation for the given vulnerable function. Each step is detailed below:

3.1 GNN Training for Sink Point Detection

In order to train the GNN model, we used the Software Assurance Reference Dataset (SARD) [20]. SARD provides ground-truth annotations for vulnerability-triggering statements; these were used to label corresponding CPG nodes as 'sink' (triggering points) or 'non-sink'. These labels are used in the training process as node labels for the node classification task to classify each node as either sink or non-sink. To achieve robustness and overcome class imbalance, we preprocessed the dataset to balance positive and negative samples to overcome the imbalance issue. We used a Graph Convolutional Network architecture, which uses message passing to capture dependencies among neighboring nodes in the CPG. We used 6 GCN layers, each followed by batch normalization and ReLU activation function to stabilize training and also to introduce nonlinearity. We also added Dropout with a probability of 0.5 after each hidden layer to prevent overfitting. As for node features, we trained 128-dimensional Word2Vec embeddings by using a random walk to encode node types (e.g., Identifier, CallExpression) and their content (e.g., variable name or function calls). The final output of the final layer is 1 or 0, representing sink or non-sink class for each node. Figure 4 presents GNN's architecture.

Once the GNN model was trained and deployed to detect sink points, the approach proceeded as follows:

3.2 Identification of PSPs

We utilized the pretrained GNN model from the previous step to detect candidate sink points. At the end of this step, a list of PSPs is returned.

3.3 Backward Slicing

Inspired by SliceLocator [4], we generate a list of potential vulnerable paths by performing backward slicing from each of the predicted sink points in the previous step, all the way up to the source of the path. Then, we will have a list of candidate paths to further examine in the next step.

3.4 Flow Path Scoring and Selection

Following the methodology of SliceLocator, the selection of the most probable path is determined by leveraging a target graph-based vulnerability detector such as Devign, Reveal, or IVDetect to assign an importance score to each path. Finally, the path with the highest importance score will be returned as the explanation of the vulnerable input code. To be precise, we calculate the probability of the given code graph G as follows

$$p_G = \Phi(\text{vec}(G))$$

where Φ represents the target detector model, and vec denotes the Word2Vec embedding function that transforms G into its vector representation. Then, for each path, we calculate the same probability, but this time only for the subgraph corresponding to that path. Then we calculate the importance score for each path as follows:

$$\text{IS}_g = 1 - (p_G - p_g)$$

The closer the probability p_g of each subgraph g is to that of the original graph G, the higher the likelihood that the subgraph contains vulnerable statements.

4 Experimental Evaluation

In this section, the experimental setup used to evaluate our approach is shown. We describe the dataset, the configuration of the training process, the metrics used for evaluation, and the baselines. All of the developed codes of VulPathFinder and datasets used in this work are available in our GitHub repository [21].

4.1 Dataset

For our experiment, we used the SARD dataset [20]. We included six C/C++ weaknesses: CWE-121 to CWE-126, which are different sorts of buffer overflow. This selection resulted in a total of 9660 vulnerable functions, with each function containing multiple statements that are represented as nodes in our program graphs. Source code is parsed into graphs using Joern [22] and SVF [23], with duplicates removed via MD5 hashing.

4.2 Evaluation Metric

To evaluate the performance of our GNN model for sink point detection, we use standard classification metrics, including Precision, Recall, and F1-Score [5]. To evaluate the end-to-end performance of our expla-

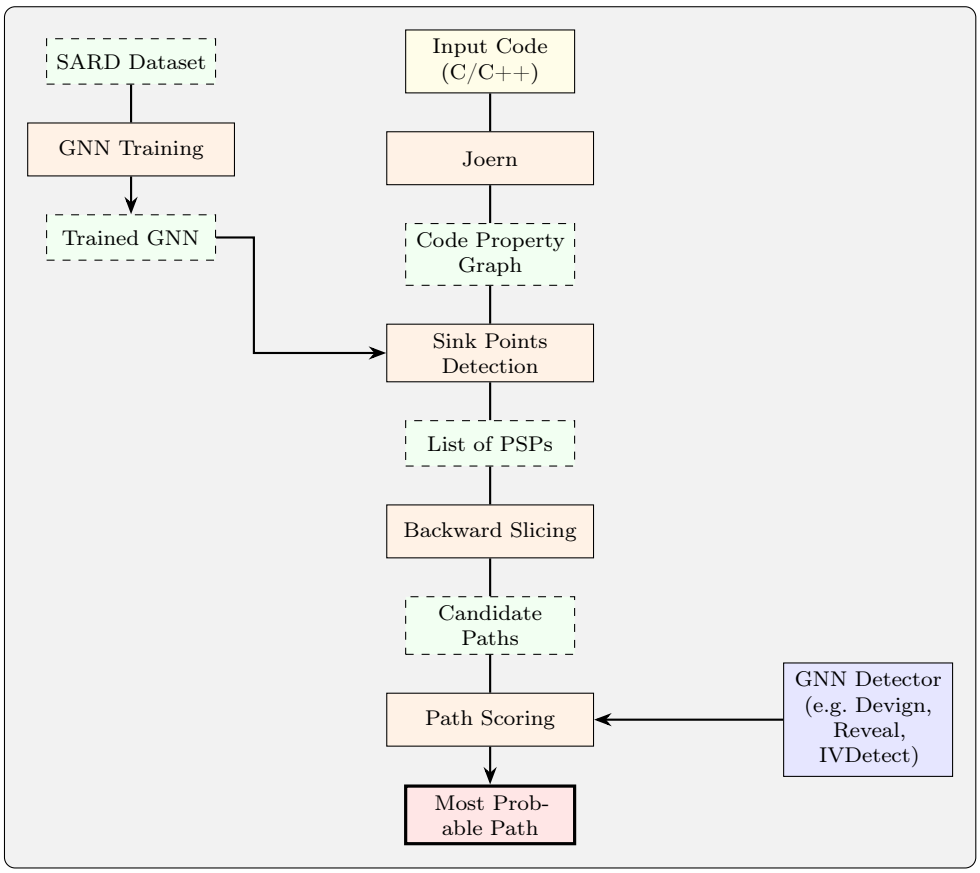


Figure 1. Overview of the VulPathFinder Vulnerability Path Discovery Framework.

```
  void CWE121_Stack_Based_Buffer_Overflow()
  {
1     int * data;
2     int * dataBadBuffer = (int *)ALLOCA(50*sizeof(int));
3     int * dataGoodBuffer = (int *)ALLOCA(100*sizeof(int))
        ;
4     if(globalReturnsTrueOrFalse())
      {
5         data = dataBadBuffer;
      }
      else
      {
6         data = dataGoodBuffer;
      }
      {
7         int source[100] = {0};
8         memmove(data, source, 100*sizeof(int));VULN
9         printIntLine(data[0]);
      }
  }
```

Figure 2. Buffer overflow example in C

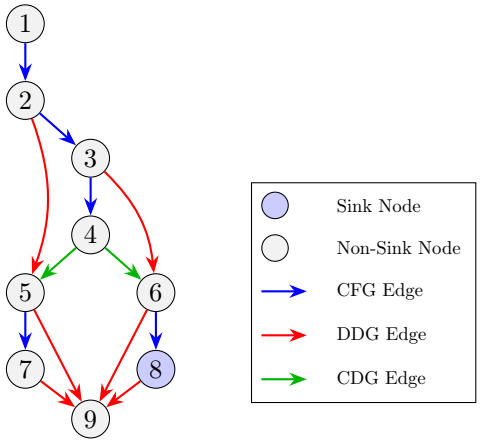


Figure 3. Control flow graph with CFG, DDG, and CDG edges.

nation method, we adopt the Triggering Line Coverage (TLC) metric, which is also used by the baseline method, SliceLocator, allowing for a fair comparison [4]. TLC measures the overlap between the reported path, which serves as an explanation, and the actual ground truth statements that trigger the vulnerability. TLC is calculated with the following equation:

$$\text{TLC} = \frac{|s^{\text{e}} \cap s^{\text{v}}|}{|s^{\text{v}}|}$$

where s^e denotes the set of statements in the predicted vulnerable path and s^v represents the set of labeled triggering statements as ground truths.

4.3 Target Vulnerability Detectors

To thoroughly evaluate VulPathFinder's ability to provide explanations for different black-box models, we adopted three state-of-the-art graph-based vulnerability detectors as our targets: Devign [15], Reveal [2],

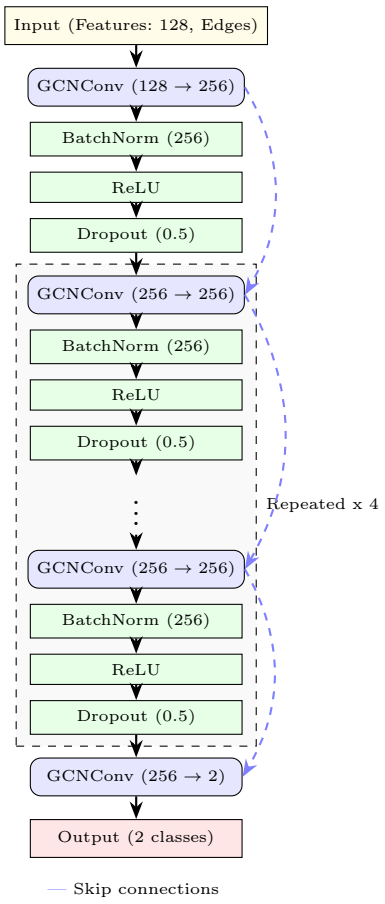


Figure 4. Illustration of the GNN (6-layer GCN) architecture with skip connections and repeated blocks.

and IVDetect [17]. These models were chosen because they represent prominent deep learning approaches for vulnerability detection and were also utilized as target detectors in the SliceLocator study, allowing for direct comparison [4]. For each of these detectors, we used their publicly available implementation. These models then served as the 'black-box' detectors for which VulPathFinder generated explanations for the vulnerability path discovery task.

4.4 Baselines

We compare VulPathFinder against two baselines to benchmark its performance in providing vulnerability explanations:

SliceLocator: A state-of-the-art technique that employs a rule-based approach to identify PSPs and then uses backward slicing to generate explanations for vulnerabilities [4].

GNNExplainer: A model-agnostic explanation method for GNNs that identifies a critical subgraph that is most influential for a given prediction [8].

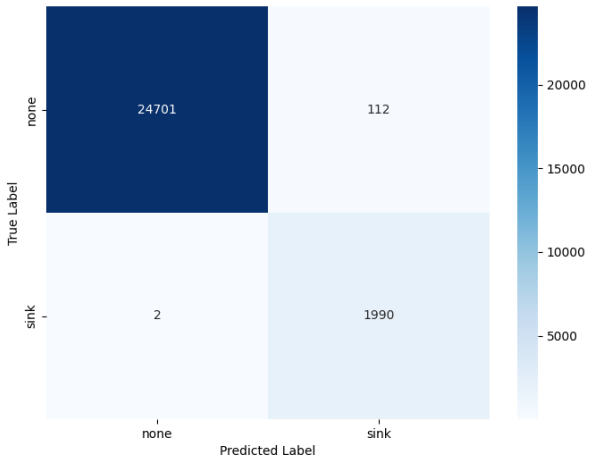


Figure 5. Confusion matrix on the test set.

4.5 Implementation and Training Details

The dataset was partitioned into training (70%), validation (10%), and test (20%) sets. Class imbalance was addressed through a combination of oversampling the minority class in the training data and using a weighted loss function during training. All models were trained on a single NVIDIA RTX 3070ti GPU with a batch size of 64, using the Adam optimizer.

5 Results

In this section, we present the experimental results of our evaluation. First, we report the performance of our GNN model for sink point detection, followed by results for vulnerability path discovery, comparing VulPathFinder against the baselines.

5.1 Sink Point Detection Performance

We first evaluated our trained GNN model on the task of classifying graph nodes as sinks. The performance of the GNN model on the test set is summarized in Table 1. The model achieved a high precision of 0.97 and a macro F1-score of 0.98. The model's ability to detect the majority of true sink nodes is highlighted by its 0.99 recall score. This high recall is required because the correct vulnerable path cannot be included for analysis if its sink is not identified. The confusion matrix is shown in Figure 5, and it confirms this low rate of false negatives for the sink class.

5.2 Vulnerability Explanation Performance

In the second part of our evaluation, we assessed the end-to-end performance of VulPathFinder in explaining vulnerabilities against the baselines. Table 2 shows the average TLC scores across the test set for all methods. VulPathFinder achieved an average TLC score of 98%, outperforming both SliceLocator and GNNEx-

plainer. Although SliceLocator achieves a respectable average TLC of 92%, its rule-based nature of sink identification prevents it from generalizing to unforeseen vulnerability patterns. GNNExplainer shows the lowest performance, with an average TLC of 81%. The reason for the low performance of GNNExplainer can be attributed to the lack of explicit modeling of taint flow and dependencies within code, which are important for understanding many vulnerabilities. This result highlights a key challenge for applying general-purpose XAI techniques in the domain of software security. This observation validates the need to incorporate program analysis concepts, such as slicing, to give insightful explanations of software vulnerabilities.

Table 1. Model Performance Metrics

Metric	Value
Precision	0.97
Recall	0.99
F1-Macro	0.98

6 Limitations and Threat to Validity

First, the SARD dataset we used is an academic dataset that includes synthetic code that might not be used in real-world software programs [20]. Second, we only evaluated 6 types of CWEs that are mostly related to buffer overflow vulnerability. Third, we only evaluated on c/c++ codes, although we can easily extend this work to use more programming languages such as Java, Python, etc.

7 Conclusion

In this paper, we introduced VulPathFinder, a GNN-based framework for explainable vulnerability path discovery that outperforms traditional rule-based methods. By training a dedicated GNN model to identify potential sink points (PSPs) in code, our approach moves beyond the limitations of fixed heuristics and learns to recognize complex, context-aware vulnerability patterns. By integrating this learned sink detection with program slicing and path ranking, VulPathFinder successfully identifies and highlights the most probable vulnerable execution paths, providing developers with actionable insights.

Our experiments on the SARD dataset demonstrate the superiority of this data-driven approach. The sink detection model achieved high precision and recall, and the full VulPathFinder framework significantly outperformed both the rule-based SliceLocator and the general-purpose GNNExplainer in terms of Triggering Line Coverage. This work underscores the potential of combining deep learning with program analysis principles to build not only accurate but also interpretable tools for software vulnerability analysis.

Table 2. Comparison of TLC scores (explanation power) of different approaches with three underlying state-of-the-art graph-based vulnerability detectors

Approach	IVDetect	Devign	Reveal
VulPathFinder	0.98	0.99	0.98
SliceLocator	0.90	0.97	0.91
GNNExplainer	0.71	0.86	0.86

References

[1] National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov/, 2020. Accessed: 2020.

[2] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

[3] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[4] Baijun Cheng, Kailong Wang, Cuiyun Gao, Xiapu Luo, Li Li, Yao Guo, Xiangqun Chen, and Haoyu Wang. Slicelocator: Locating vulnerable statements with graph-based detectors. *arXiv e-prints*, pages arXiv–2401, 2024.

[5] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming, and Nachiappan Nagappan. A survey on automated software vulnerability detection using machine learning and deep learning. *arXiv preprint arXiv:2306.11673*, 2023. Available at https://arxiv.org/abs/2306.11673.

[6] Open Source Security Foundation. Open source vulnerabilities database. https://osv.dev/, 2020. Accessed: 2020.

[7] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[8] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[9] GitHub. Codeql: Security analysis platform. https://codeql.github.com/, 2023.

[10] David Hovemeyer and William Pugh. Finding

bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[11] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6): 89–100, 2007.

[12] LLVM Project. Addresssanitizer: A fast memory error detector. https://clang.llvm.org/docs/AddressSanitizer.html, 2023.

[13] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert systems with applications*, 207:117921, 2022.

[14] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[15] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[16] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–33, 2021.

[17] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[18] David Gunning, Mark Stefik, Jaesik Choi, Timothy Miller, Simone Stumpf, and Guang-Zhong Yang. Xai—explainable artificial intelligence. *Science robotics*, 4(37):eaay7120, 2019.

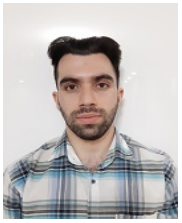
[19] Ana Lucic, Maartje A Ter Hoeve, Gabriele Tolomei, Maarten De Rijke, and Fabrizio Silvestri. Cf-gnnexplainer: Counterfactual explanations for graph neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 4499–4511. PMLR, 2022.

[20] National Institute of Standards and Technology. Software assurance reference dataset (sard). https://samate.nist.gov/SARD/, 2020. Accessed: 2020.

[21] Nima Atashin. Vulpathfinder source codes and datasets. https://github.com/NimaNA11/VulPathFinder/, 2025. Accessed: 2025-07-21.

[22] Joern Team. Joern: A robust code analysis platform. https://joern.io/, 2023. Accessed: 2023.

[23] SVF Team. Svf: Static value-flow analysis framework. https://github.com/SVF-tools/SVF, 2023. Accessed: 2023.



Nima Atashin received his bachelor's degree in Computer Engineering from Isfahan University of Technology in 2022. He is currently pursuing an M.Sc. in Software Engineering in the Faculty of Computer Engineering at the University of Isfahan. His research interests include explainable AI, graph neural networks, and software vulnerability detection.

Behrouz Tork Ladani received his bachelor's degree in computer engineering from the University of Isfahan (UI), Isfahan, Iran, in 1996, M.Sc. degree in software engineering from the Amirkabir University of Technology, Tehran, Iran, in 1998, and Ph.D. degree in software engineering from the University of Tarbiat Modarres, Tehran, Iran, in 2005. He joined UI in 2005, where he is currently a professor of Software Engineering. He is the author of more than 70 articles. His research interests are around modeling, analysis, and verification of security in information systems, including software security (vulnerability detection and malware analysis) and soft security (computational trust, rumor control, and opinion formation in social networks). Behrouz is a member of the Iranian Society of Cryptology (ISC), and he is the Managing Editor of the Journal of Computing and Security (JCS).

Mohammadreza Sharbaf is an Assistant Professor in Computer Engineering at the University of Isfahan (UI). He is interested in Model-Driven Software Engineering, Collaborative Modeling, Low-Code Development Platforms, Software Development Methodologies, Design Patterns, and Semantic Web (Semantic Reasoning). His current research is focused on software testing, inconsistency management, and multi-view modeling. Mohammadreza received his B.Sc. from the Isfahan University of Technology, Isfahan, Iran, in 2013, and his M.Sc. and Ph.D from the UI, Isfahan, Iran, in 2016 and 2022, both in Software Engineering. Now, he is the director of Model-Driven Software Engineering Research Group (MDSERG) at UI.