The ISC Int'l Journal of Information Security

ISeCure

October 2025, Volume 17, Number 3 (pp. 1–9)

http://www.isecure-journal.org

Presented at the ISCISC'2025 in TEHRAN, Iran.

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

Nima Atashin [1], Behrouz Tork Ladani [1,*], and Mohammad Reza Sharbaf [1]

[1] *Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran.*

ARTICLE INFO.

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

Type:

doi:

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 code and identifying classes of vulnerabilities. However, there is still room to improve the explanation of the root causes of detected vulnerabilities by locating vulnerable statements and discovering the paths that lead 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 generalisation. In this paper, we introduce VulPathFinder, an explainable vulnerability path discovery framework that enhances SliceLocator's methodology by utilising 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 demonstrate 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 the original SliceLocator and GNNExplainer (as a general GNN explainability tool) in discovering 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

* Corresponding author.

**The ISCISC'2025 program committee effort is highly acknowledged for reviewing this paper.

Email addresses: nima.atashin@eng.ui.ac.ir, ladani@eng.ui.ac.ir, m.sharbaf@eng.ui.ac.ir

ISSN: 2008-2045

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. 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 [3]. Because it is difficult to define vulnerabilities, rule-based methods suffer from high false-positive rates, especially on complex code [3]. In contrast, data-driven methods such as deep learning have emerged as powerful alternatives capable of generalising 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 [4]. Data-driven approaches can learn the latent information from vulnerable patterns and have shown better performance compared to static tools that utilise predefined rules [3].

Among data-driven approaches, both sequence-based and graph-based approaches have been widely explored [3]. Sequence-based methods serialise 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 [5]. 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 analysing root cause, trust, and fixing.

To address the limitation mentioned above, we propose VulPathFinder, a Graph Neural Network (GNN)-based approach for identifying the most probable paths from potential sources to detected vulnerability sink statements. VulPathFinder enhances the vulnerability path discovery method used by SliceLocator [6] by utilising a GNN model to detect potential sink points (PSPs) first, 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 is context-aware and capable of generalising 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. Crucially, this context-aware learning allows the model to generalise to new vulnerability instances by identifying risky patterns, rather than memorising specific code snippets. After finding PSPs, inspired by SliceLocator, we perform backwards slicing starting from each sink point in the list. As a result, we obtain a list of candidate paths leading to sink points that form corresponding subgraphs. Subgraphs are then fed into off-the-shelf graph-based detectors to compute their likelihood of being vulnerable. The subgraph with the highest likelihood of being vulnerable is finally chosen. This 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 rival methods, we used the Triggering Line Coverage (TLC) metric [6] to compare the achieved results with the original SliceLocator as well as GNNExplainer [7]. The latter is a model-agnostic explanation method for GNNs that identifies the most influential subgraph for a given prediction. The results show that VulPathFinder not only achieves acceptable precision and recall in sink point detection but also demonstrates higher end-to-end performance in terms of TLC, indicating better explainability.

The rest of the paper is organised 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 [8] and FindBugs [9] 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 [3]. Dynamic analysis tools such as Valgrind [10] and AddressSanitizer [11] find vulnerabilities at run-

time, 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 [4]. 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 [12] and social network modelling [13]. 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 [5]. 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 [5]. Some works have used solely the sequence of tokens as their code representation. However, 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 recent studies have leveraged GNNs for vulnerability detection and localisation, sharing similarities with our approach in graph representations but differing in focus and methodology. For instance, Devign [14] employs Gated Graph Recurrent Networks for function-level vulnerability identification using Code Property Graphs, emphasising detection accuracy on datasets like SARD. Similarly, LineVD [15] refines granularity to statement-level detection as a node classification task with Graph Attention Networks (GAT) and transformer-based embeddings (e.g., CodeBERT), jointly optimising function- and statement-level predictions to resolve conflicts, achieving significant F1 improvements on real-world C/C++ CVEs. CFExplainer [16] adds explainability through counterfactual reasoning, generating "what-if" code edits to flip predictions, while VulChecker [17] localises vulnerabilities to instructions via GNNs with data augmentation, tested on CVE-linked code. Coca [18] enhances GNN robustness by addressing spurious correlations, providing causal subgraph explanations. In contrast to these detection- or localisation-focused works, VulPathFinder uniquely applies a GNN for context-aware sink point detection, replacing rule-based methods in SliceLocator [6] and integrating backwards slicing with off-the-shelf detectors for ranked vulnerability paths, enabling root-cause explanations beyond classification or attributions.

While the mentioned approaches show strong performance, their outputs are coarse-grained and mostly detect vulnerable code at the function level, lacking actionable explanations. VulPathFinder differs in two ways: (1) it introduces a data-driven mechanism for sink detection, which removes reliance on fixed rules, and (2) it produces fine-grained vulnerable paths by integrating program slicing with graph-based vulnerability detectors.

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 [19]. Several techniques, such as GNNExplainer [7] and CFExplainer [16], 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 [7]. CFExplainer is a counterfactual explanation that identifies the smallest modifications to a graph's structure needed to reverse the model's prediction [16]. 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 behaviour of vulnerabilities and their execution paths, and a slight change in input results in drastically different explanations. Also, most explainers deal with the models themselves, ignoring insights about taint tracking and slicing. Therefore, static analysis concepts such as taint propagation and slicing can be a promising complement to explainers. VulPathFinder is designed to bridge this exact gap, integrating learned sink detection with program slicing to create explanations that are both data-driven and semantically grounded in program analysis.

3 The Proposed Approach

In this section, we present our framework, VulPathFinder, which enhances vulnerability path discovery by utilising a GNN model to detect PSPs. Previous works (including SliceLocator) [6, 20] con-

sidered 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 [5]. 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 generalised 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 backwards slicing starting from each sink point in the list. At the end of this phase, we obtain a list of candidate paths leading to sink points
(4) Flow Path Selection: In the last phase, the prediction 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 is 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 backwards slicing, such as $1 \to 5 \to 8$, $1 \to 6 \to 8$, $7 \to 8$, etc. 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

To train the GNN model, we used the Software Assurance Reference Dataset (SARD) [21]. 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. We used a Graph Convolutional Network architecture, which uses message passing to capture dependencies among neighbouring nodes in the CPG. We used six GCN layers, each followed by batch normalisation and ReLU activation function to stabilise training and introduce nonlinearity. We also added dropout with a probability of 0.5 after each hidden layer to prevent overfitting. While deeper GCNs are prone to oversmoothing, we mitigated this risk by incorporating skip connections, batch normalisation, and dropout, which helped preserve feature diversity and stabilise training in our experiments. For node features, we trained 128-dimensional Word2Vec embeddings using random walks 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 the GNN architecture.

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

3.2 Identification of PSPs

We utilised 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 [6], we generate a list of potential vulnerable paths by performing backward slicing from each predicted sink point in the previous step, all the way up to the source of the path. Then, we obtain 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 is returned as the explanation of the vulnerable input code. Precisely, 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

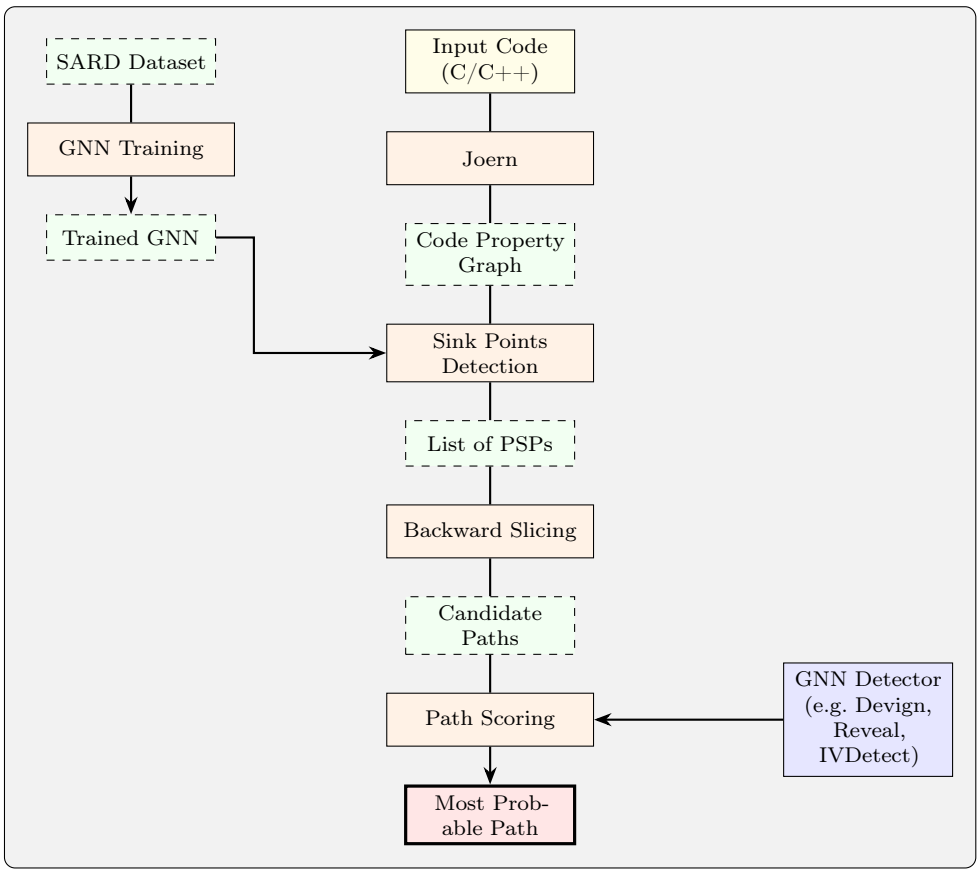


Figure 1. Overview of the VulPathFinder Vulnerability Path Discovery Framework.

```
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

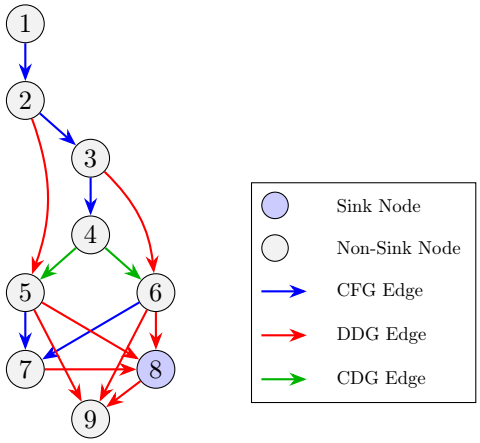


Figure 3. Control flow graph with CFG, DDG, and CDG edges.

path. We calculate the importance score for each path as follows:

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

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, we present the experimental setup used to evaluate our approach. We describe the dataset, the configuration of the training process, the metrics used for evaluation, and the baselines. All developed codes of VulPathFinder and datasets used in this work are available in our GitHub repository [22].

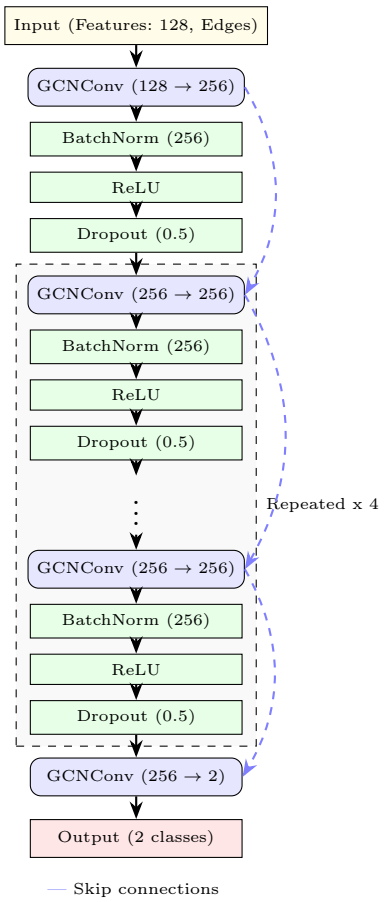


Figure 4. Illustration of the GNN (6-layer GCN) architecture with skip connections and repeated blocks.

4.1 Dataset

For our experiment, we used the SARD dataset [21]. 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 [23] and SVF [24], 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 [3]. To evaluate the end-to-end performance of our explanation method, we adopt the Triggering Line Coverage (TLC) metric, which is also used by the baseline method, SliceLocator, allowing for a fair comparison [6]. 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 labelled 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 [14], Reveal [2], and IVDetect [25]. These models were chosen because they represent prominent deep learning approaches for vulnerability detection and were also utilised as target detectors in the SliceLocator study, allowing for direct comparison [6]. 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 backwards slicing to generate explanations for vulnerabilities [6].

GNNExplainer: A model-agnostic explanation method for GNNs that identifies a critical subgraph that is most influential for a given prediction [7].

4.5 Implementation and Training Details

The dataset was partitioned into training (70%), validation (10%), and test (20%) sets. Class imbalance was addressed by 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 optimiser. Word2Vec training took approximately 3.5 minutes, and GNN training required approximately 17.8 minutes over 87 epochs, totalling approximately 21.3 minutes. GNN inference on a single test sample averaged approximately 752 milliseconds. For the explanation phase (769 samples), the average per-sample inference time across detectors was approximately 2.65 seconds (total times ranging from approximately 22 to 45.8 minutes). These results indicate moderate training costs and variable inference times, with batch processing as a potential optimisation for deployment efficiency.

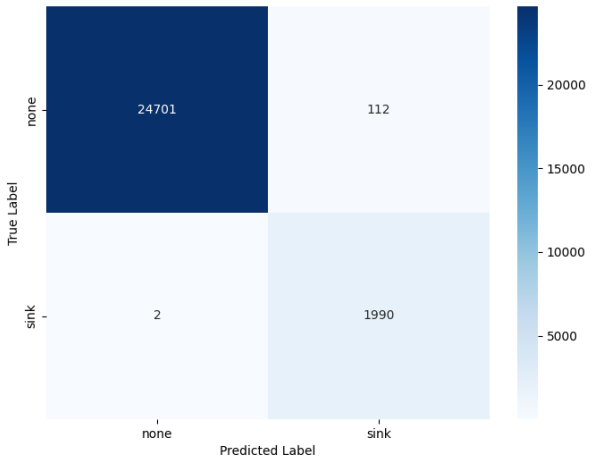


Figure 5. Confusion matrix on the test set.

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 summarised 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 GNNExplainer. Although SliceLocator achieves a respectable average TLC of 92%, its rule-based nature of sink identification prevents it from generalising 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 modelling 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 [21]. Second, we only evaluated six types of CWEs that are mostly related to buffer overflow vulnerability. However, since our framework learns to detect potential sink points and vulnerable paths in a data-driven manner, it can be retrained to support other categories of vulnerabilities, such as injection or cross-site scripting (XSS). Third, we only evaluated C/C++ codes, although we can easily extend this work to use more programming languages such as Java, Python, etc. Finally, while we adopted a 6-layer GCN to capture deep structural dependencies, such architectures are known to suffer from oversmoothing. To mitigate this, we employed batch normalisation, dropout, and skip connections (see Fig. 4), which stabilised training and preserved feature diversity.

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 recognise 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. In future work, we plan to extend VulPathFinder's evaluation to larger real-world datasets such as Big-Vul and to a broader set of CWE categories. These steps will further validate the framework's robustness and applicability in practical vulnerability discovery scenarios.

References

[1] National Institute of Standards and Technology.

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

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] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

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

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

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

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

[14] 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.

[15] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

[16] 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.

[17] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. {VulChecker}: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574, 2023.

[18] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

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

[20] 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.

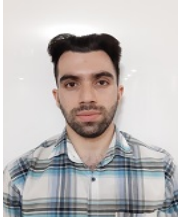
[21] National Institute of Standards and Technology. Software assurance reference dataset (sard). https://samate.nist.gov/SARD/, 2020. Accessed: 2020.

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

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

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

[25] 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.



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 modelling, analysis, and verification of security in information systems, including software security (vulnerability detection and malware analysis) and soft security (computational trust, rumour control, and opinion formation in social networks).

Mohammad Reza Sharbaf is an Assistant Professor in Computer Engineering at the University of Isfahan (UI). He is interested in Model-Driven Software Engineering, Collaborative Modelling, 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 modelling. 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 the Model-Driven Software Engineering Research Group (MDSERG) at UI.