VNU Journal of Science: Comp. Science & Com. Eng. Vol. 40, No. 1 (2024) 47–59

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

Silent Vulnerability-fixing Commit Identification Based on Graph Neural Networks

Hieu Dinh Vo*, Trong Thanh Vu, Son Nguyen

VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Received 13 September 2023
Revised 28 November 2023; Accepted 28 March 2024

Abstract: In this paper, we present VFFinder, a novel graph-based approach for automated silent vulnerability fix identification. To precisely capture the meaning of code changes, the changed code is represented in connection with the related unchanged code. In VFFinder, the structure of the changed code and related unchanged code are captured and the structural changes are represented in annotated Abstract Syntax Trees (αAST). VFFinder distinguishes vulnerability-fixing commits from non-fixing ones using attention-based graph neural network models to extract structural features expressed in αASTs. We conducted experiments to evaluate VFFinder on a dataset of 11K+ vulnerability fixing commits in 507 real-world C/C++ projects. Our results show that VFFinder significantly improves the state-of-the-art methods by 272–420% in Precision, 22–70% in Recall, and 3.2X–8.2X in F1. Especially, VFFinder speeds up the silent fix identification process by up to 121% with the same effort reviewing 50K LOC compared to the existing approaches.

Keywords: Silent vulnerability fixes, vulnerability fix identification, code change representation, graph-based model.

1. Introduction

With the escalating dependence of software projects on external libraries, ensuring their security has emerged as an imperative priority. Vulnerabilities hidden within these libraries can have far-reaching consequences, as exemplified by the infamous Log4Shell exploit. One critical challenge in addressing these vulnerabilities is the time gap between their fixes and public disclosures [1]. For instance, the Log4Shell vulnerability's resolution was introduced four days prior to its public revelation. Another illustration involves the Apache Struts Remote Code Execution vulnerability[1], which led to the Equifax breach in 2017, was disclosed in August 2018, but was patched in June 2018[2]. This two-month window provides ample opportunity for the potential exploitation of vulnerable software. If the library's

*Corresponding author.
E-mail address: hieuvd@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.1432

[1]https://nvd.nist.gov/vuln/detail/CVE-2018-11776
[2]https://github.com/apache/struts/commit/6e87474

users are aware of the potential exploitation, they can prevented it by updating to the latest version of the component.

Despite the importance of the vulnerability fix identification task in open-source libraries, only a very small portion of maintainers file for a Common Vulnerability Enumeration (CVE) ID after releasing a fix, while 25% of open-source projects silently fix vulnerabilities without disclosing them to any official repository [2, 3]. This situation raises concerns about the visibility and proactive management of vulnerabilities within the software ecosystem. The open-source libraries' users rely on several tools and public vulnerability datasets like the Open Web Application Security Project or National Vulnerability Database (NVD). However, CVE/NVD and public databases miss many vulnerabilities [3].

To address this problem, several vulnerability fix identification techniques have been proposed. Following the good practice of coordinated vulnerability disclosure [1], the related resources of commits, such as commit messages or issue reports, should not mention any security-related information before the public disclosure of the vulnerability. Thus, silent fix identification methods must not leverage these resources to classify commits [4–6]. The state-of-the-art techniques, such as VulFixMiner [4], CoLeFunDa [6], and Midas [5], represent changes in the lexical form of code and apply CodeBERT [7] to capture code changes semantics and determine if they are vulnerability-fixing commit or not. Meanwhile, the existing studies have shown that the semantics of code changes could be captured better in the tree form of code [8].

This paper, which is the extended version of our previous conference paper [9], proposes VFFinder, a novel graph-based approach for automated vulnerability fix identification. Our idea is to capture the semantic meaning of code changes better, we represent changed code in connection with the related unchanged code and explicitly represent the changes in code structure. Particularly, for a commit c, the code version before (after) c is analyzed to identify the code necessary for representing the code change, which is the deleted (added) lines of code and their related unchanged lines of code. After that, the structure of the necessary code of the changes in c is represented by the Abstract Syntax Trees (ASTs). These ASTs are mapped to build an annotated AST (αAST), a fine-grained graph representing the changes in the code structure caused by c. In αASTs, all AST nodes and edges are annotated *added*, *deleted*, and *unchanged* to explicitly express the changes in the code structure. To learn the meanings of code changes expressed in αASTs, we develop a graph attention network model [10] to extract semantic features. Then, these features are used to distinguish vulnerability-fixing commits from non-fixing ones. Compared with the previous paper [9], this paper provides a more comprehensive example for αASTs and more experiment results which are conducted on a much larger dataset.

We conducted several experiments to evaluate VFFinder's performance on a dataset containing 100K+ commits (with 11K+ vulnerability fixing commits) extracted from 507 real-world C/C++ projects. Our experiment results show that VFFinder significantly improves the state-of-the-art techniques [4, 5, 11, 12] by 272–420% in Precision, 22–70% in Recall, and 3.2X–8.2X in F1. Especially, VFFinder speeds up the silent fix identification process up to 121% with the same review effort reviewing 50K lines of code (LOCs) (0.02% of the total changed LOCs) compared to the existing approaches [4, 5, 11, 12].

In brief, our contributions are:

1. VFFinder: A novel graph-based approach for identifying silent vulnerability fixes.

2. Extensive experiments which show that the performance of VFFinder is much better than those of the state-of-the-art methods for vulnerability-fix identification.

```
1  1    void str_cpy(char *str, size_t n) {
2  2        char buf[BUF_SIZE], *ar;
3  3        size_t len = strlen(str);
4     -     if(len < 2*BUF_SIZE) {
      4  +  if(len < BUF_SIZE) { {
5  5            memcpy(buf, str, len);
6  6        }
7  7    }
```

Figure 1. A commit deleting a line of code and adding another to fix a buffer overflow vulnerability.

The rest of this paper is organized as follows. Section 2 describes our novel code change representation. The graph-based vulnerability fix identification model is introduced in Section 3. After that, Section 4 states our evaluation methodology. Section 5 presents the experimental results following the introduced methodology and some threats to validity. Section 6 provides the related work. Finally, Section 7 concludes this paper.

2. Code Change Representation

Essentially, vulnerability fixing commits tend to correct the vulnerable code which already exists in repositories. Thus, *to precisely capture the semantics of code changes, besides the changed parts, the unchanged code is also necessary* [13, 14]. Indeed, the unchanged code could connect the related changed parts and help to understand the code changes as a whole. Additionally, once changed statements are introduced to a program, they change the program's behaviors by interacting with the unchanged statements via certain code relations such as control/data dependencies. Hence, precisely distinguishing similar changed parts could require their relations with the unchanged parts. In Figure 1, the unchanged parts, including lines 3 and 5 in both the versions before and after the commit, are necessary for understanding the meaning of the changed code.

Thus, for a commit, we additionally consider the code statements that are semantically related to the changed statements in the versions (v_b and v_a) before and after the commit. Particularly, the considering statements in v_b are the deleted statements and the related ones via control and data dependencies in v_b, while the added statements and their related ones via control and data dependencies in v_a are considered. These statements in v_b and v_a are analyzed to construct corresponding Abstract Syntax Trees (ASTs) and capture the structural changes in an *Annotated AST - αAST*.

Definition 1 (Annotated AST - *αAST*)**.** *For a commit changing code from one version (v_b) to another (v_a), the annotated abstract syntax tree (annotated AST) is an annotated graph constructed from the ASTs of v_b and v_a. Formally, for $AST_o = \langle N_b, E_b \rangle$ and $AST_n = \langle N_a, E_a \rangle$ which are the ASTs of v_b and v_a, respectively, the αAST $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$ is defined as followings:*

- *$\mathcal{N}$ consists of the AST nodes in the old version and the new version, $\mathcal{N} = N_b \cup N_a$.*
- *$\mathcal{E}$ is the set of the edges representing the structural relations between AST nodes in AST_o and AST_n, $\mathcal{E} = E_b \cup E_a$.*
- *Annotations for nodes and edges in $\mathcal{T}$ are either unchanged, added, or deleted by the change. Formally, $\alpha(g) \in$ {unchanged, added, deleted}, where g is a node in $\mathcal{N}$ or an edge in $\mathcal{E}$:*
  - *$\alpha(g)$ = added if g is a node and $g \in N_a \setminus N_b$, or g is an edge and $g \in E_a \setminus E_b$*
  - *$\alpha(g)$ = deleted if g is a node and $g \in N_b \setminus N_a$, or g is an edge and $g \in E_b \setminus E_a$*
  - *Otherwise, $\alpha(g)$ = unchanged*

Figure 2 shows ASTs of the versions before and after the commit shown in Figure 1. The annotated AST constructed from these ASTs is shown in Figure 3. The *αAST* expresses the change in the structure of the code. Particularly, the right-hand-side of the `less-than` expression (`2*BUF_SIZE`) is replaced by expression `BUF_SIZE`. The *αAST* also expresses the structure of the related unchanged statements, which clarify the meaning of the changed code.

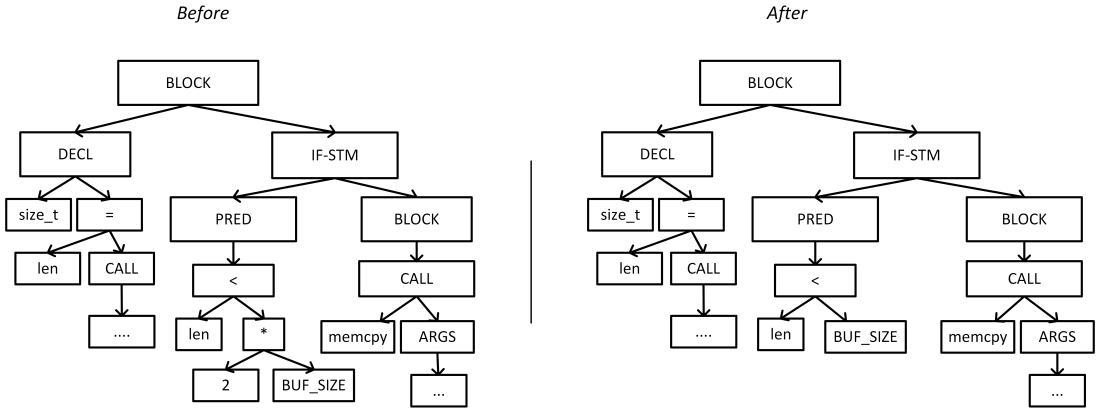

Figure 2. The ASTs of the versions before and after the commit shown in Figure 1.

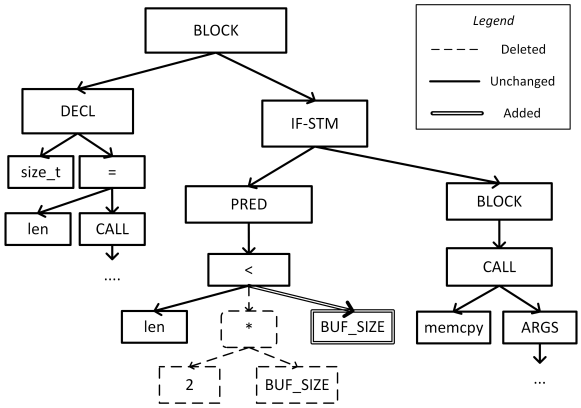

Figure 3. The αAST corresponding to the commit shown in Figure 1.

3. Vulnerability-Fix Identification Model

Figure. 4 illustrates the overview of VFFinder for vulnerability fix identification. First, the given commits and their repositories are used to construct their corresponding αASTs (*Change representation*). To construct αASTs, the code versions before and after commits must be syntactically valid. This could be checked by directly applying code parsers for the code. Each AST node in αASTs is embedded in the corresponding vectors (*Embedding*). After that, a Graph Neural Network (GNN) is applied to extract structural features from constructed αASTs (*Feature extraction*). Finally, the extracted structural features are used for learning and predicting vulnerability-fixing commits (*Prediction*).

Particularly, in the *Embedding* step, for each αAST, $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$, every node in $\mathcal{N}$ is embedded into d-dimensional hidden features n_i produced by embedding the content of the nodes. Constructing the vectors for nodes' content could be done by applying code embedding techniques [7, 15–18]. In this work, we use Word2vec [17], one of the most popular code embedding techniques for code [18]. The reason is that the number of AST nodes in αASTs could be huge. Thus, for a practical embedding step for αASTs, we apply Word2vec, which is known as an efficient embedding technique [18]. Then, to form the node feature vectors, the node embedding vectors are annotated with the change operators (*added*, *deleted*, and *unchanged*) by concatenating corresponding one-hot vector of the operators to the embedded vectors, $h_i^0 = [n_i \| \alpha(n_i)]$, where $\|$ is the concatenation operation and α returns the one-hot vector corresponding the annotation of node i.

In the *Feature Extraction* step, from each αAST, $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$, we develop a Graph Attention Network (GAT) [10] model to extract the structural features H. Particularly, the embedded vectors of the nodes from the *Embedding* step are fed to a GAT model. Each GAT layer computes

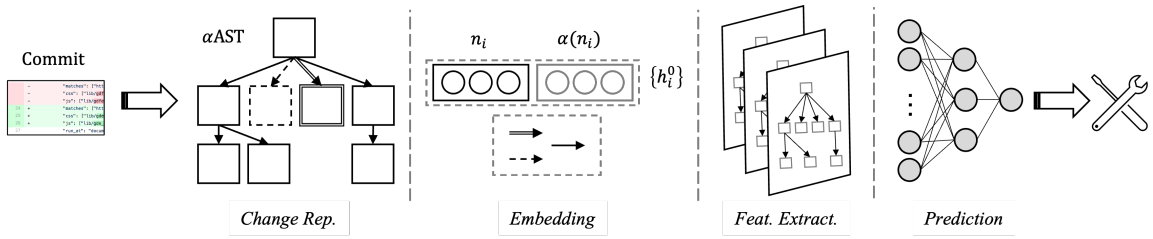

Figure 4. Graph-based Vulnerability-fix Identification Model.

the representations for the graph's nodes through message passing [10, 19], where each node gathers features from its neighbors to represent the local graph structure. Stacking L layers allows the network to build node representations from each node's L-hop neighborhood. From the feature vector h_i of node i at the current layer, the feature vector h'_i at the next layer is:

$$h'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} h_j\right)$$

where $\mathbf{W}$ is a learnable weight matrix for feature transformation, $\mathcal{N}_i$ is the set of neighbor indices of node i including node i itself via *self-connection*, which is a single special relation from node i to itself. σ is a non-linear activation function such as ReLU. Meanwhile, α_{ij} specifies the weighting factor (importance) of node j's features to node i. α_{ij} could be explicitly defined based on the structural properties of the graph or learnable weight [19, 20]. In this work, we implicitly define α_{ij} based on node features [10] by employing the self-attention mechanism, where unnormalized coefficients E_{ij} for pairs of nodes i, j are computed based on their features:

$$E_{ij} = \text{LeakyReLu}(\mathbf{a}^T \cdot [\mathbf{W}h_i \| \mathbf{W}h_j]),$$

where $\|$ is the concatenation operation and $\mathbf{a}$ is a parametrizing weight vector implemented by a single-layer feed-forward neural network. E_{ij} indicates the importance of node j's features to node i. The coefficients are normalized across all choices of j using the softmax function:

$$\alpha_{ij} = \text{softmax}_j(E_{ij}) = \frac{\exp(E_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(E_{ik})}$$

After L GAT layers, a d-dimensional graph-level vector representation H for the whole CTG $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$ is built by averaging over all node features in the final GAT layer, $H = \frac{1}{|\mathcal{N}|} \sum_{i \in [1, |\mathcal{N}|]} h_i^L$. Finally, in the *Prediction* step, the graph features are then passed to a Multilayer perceptron (MLP) to classify if αAST $\mathcal{T}$ is a fixing commit or not.

4. Evaluation Methodology

To evaluate our vulnerability-fixing commit identification approach, we seek to answer the following research questions:

RQ1: ***Accuracy and Comparison.*** How accurate is VFFinder in identifying vulnerability-fixing commits? And how is it compared to the state-of-the-art approaches [4, 5]?

RQ2: ***Intrinsic Analysis.*** How do the consideration of related unchanged code and the GNN model in VFFinder impact VFFinder's performance?

RQ3: ***Sensitivity Analysis.*** How do various factors of the input, including training data size and changed code complexity, affect VFFinder's performance?

RQ4: ***Time Complexity.*** What is VFFinder's running time?

4.1. Dataset

In this work, we collect the vulnerability-fixing commits from various public vulnerability

datasets [21–23]. In total, we collected the commits in real-world 507 C/C++ projects, including about 11K fixing commits for the vulnerabilities reported from 1990 to 2022. Table 1 shows the overview of our dataset[3].

4.2. Procedure

For **RQ1. Accuracy and Comparison**, we compared VFFinder against the state-of-the-art vulnerability-fix identification approaches:

1) **MiDas** [5] establishes distinct neural networks for varying levels of code change granularity, encompassing commit-level, file-level, hunk-level, and line-level alterations, aligning with their inherent categorization. It employs an ensemble model that amalgamates all foundational models to produce the ultimate prediction.

2) **VulFixMiner** [4] and CoLeFunDa [6] use CodeBERT to automatically represent code changes and extract features for identifying vulnerability fixes. However, as the implementation of CoLeFunDa has not been available, we cannot compare VFFinder with CoLeFunDa. This is also the reason that Zhou *et al.* was not able to compare MiDas with CoLeFunDa in their study [5].

Additionally, we applied the same procedure, adapting the state-of-the-art just-in-time defect detection techniques for vulnerability-fix identification as in Midas *et al.* [5]. In this work, the additional baselines include:

3) **JITLine** [12]: A simple but effective method utilizing changed code and expert features to detect buggy commits.

4) **JITFine** [11]: A DL-based approach extracting features of commits from changed code and commit message using CodeBERT as well as expert features.

Note that we did not utilize commit messages when adapting JITLine and JITFine for silent vulnerability fix identification in our experiments. For VFFinder, we set the number of GNN layers $L = 2$ for a practical evaluation.

[3]https://github.com/UETISE/VFFinder

In this comparative study, we consider the impact of time on the approaches' performance. Particularly, we divided the commits into those before and after the time point t. The commits before t were used for training, while the commits after t were used for evaluation. We selected a time point t to achieve a random training/test split ratio of 80/20 based on time. Specifically, the commits from Aug 1998 to Mar 2017 are used for training, and the commits from Apr 2017 to Aug 2022 are for evaluation.

However, the vulnerability-fixing commits account for a very small proportion of the whole dataset, about 0.3%. Consequently, the approaches work on a severely imbalanced classification dataset. This causes poor performance of classification approaches for the vulnerability detection task [24]. To mitigate this issue, Yang *et al.* have recommended under-sampling for training classification models [24]. Thus, we applied under-sampling for all the considered approaches for a fair comparison.

For the testing dataset, the number of commits from Apr 2017 to Aug 2022 is huge, 2,462,900 commits, including only 2,828 vulnerability-fixing commits. We applied the approaches to the set containing all the fixing commits and the set of non-vulnerability-fixing commits manageable for all the considering approaches given our hardware resources. Specifically, we considered the testing dataset containing 2,828 vulnerability-fixing commits and about 80K+ non-fixing commits (Table 1). In our dataset, there are about 2,600 changed LOC in a commit on average, while the largest number of changed LOC is about 101K. Additionally, we applied the same procedure in the existing work by Zhou *et al.* [23] to project the relative comparison trend for the approaches with the real-world non-fix/fix ratio.

For **RQ2. Intrinsic Analysis**, we investigated the impact of the consideration of related unchanged parts and the GNN model on VFFinder's performance. We used different vari-

Table 1. Dataset statistics

Phase	*#Time*	*#Fixes*	*#Non-fixes*	*#Changed LOCs*
Training	Aug 1998 – Mar 2017	8,471	8,471	3,869,454
Testing	Apr 2017 – Aug 2022	2,828	86,395	271,182,635
Total	Aug 1998 – Aug 2022	11,299	94,866	275,052,089

ants of αAST and graph neural networks such as GAT [10], GCN [19], GIN [25], and GraphSAGE [26] to study the impact of those factors on VFFinder's performance.

For **RQ3. Sensitivity Analysis**, we studied the impacts of the training size and change size in the number of changed LOC on the performance of VFFinder. To systematically vary these factors, we gradually added more training data and varied the range of the change size.

4.3. Metrics

Essentially, the task of vulnerability fix identification could be considered as a binary classification task. Thus, to evaluate the vulnerability fix identification approaches, we measure the classification *accuracy*, *AUC* (*Area Under Curve*), *precision*, and *recall*, as well as *F1*, which is a harmonic mean of precision and recall. Particularly, the classification accuracy (*accuracy* for short) is the fraction of the (fixing and non-fixing) commits that are correctly classified among all the tested commits. *AUC* represents the area under the curve that plots the True Positive Rate (Sensitivity) against the False Positive Rate (1 - Specificity) for various threshold settings. For detecting fixing commits, *precision* is the fraction of correctly detected fixing commits among the detected fixing commits, while *recall* is the fraction of correctly detected fixing commits among the fixing commits. Formally $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$, where TP is the number of true positives, FP and FN are the numbers of false positives and false negatives, respectively. *F1* is calculated as $F1 = \frac{2 \times precision \times recall}{precision + recall}$. Additionally, we also applied a cost-aware performance metric, CostEffort@L (*CE@L*), which is used in [4, 5]. *CE@L* counts the number of detected vulnerability-fixing commits, starting from commit with high to low predicted probabilities until the number of lines of code changes reaches L lines of code (LOCs). In this work, we considered, $C@50K$, $L = 50,000$, about 0.02% of the total changed LOCs in our dataset, for simplicity.

5. Experimental Results

5.1. Performance Comparison (RQ1)

Table 2 shows the performance of VFFinder and the state-of-the-art vulnerability-fix identification approaches. As can be seen, VFFinder significantly outperforms the state-of-the-art vulnerability-fix identification approaches. Particularly, the VFFinder achieves a recall of 0.99. In other words, 99/100 vulnerability fixing commits are correctly identified by VFFinder, which is more than **22–70%** better than the recall rates of the existing approaches. Additionally, VFFinder is still much more precise than the existing approaches with about **272–420%** improvement in the precision rate. These show that VFFinder can not only find more vulnerability-fixing commits but also provide much more precise predictions.

Furthermore, the *CE@50K* of VFFinder is **71%**, which is **83-121%** better than the corresponding figures of MiDas and VulFixMiner. This means that given the effort reviewing 50K LOC, the number of the fixing commits found by using VFFinder is much larger compared to those found by using MiDas and VulFixMiner.

Table 2. Comparison Results

Approach	*Pre.*	*Rec.*	*F1*	*Acc.*	*AUC*	*CE@50K*
JITLine	0.06	0.66	0.11	0.72	0.66	0.35
JITFine	0.07	0.81	0.13	0.65	0.80	0.39
VulFixMiner	0.05	0.05	0.05	0.62	0.65	0.32
MiDas	0.06	0.63	0.11	0.67	0.70	0.34
VFFinder	**0.26**	**0.99**	**0.41**	**0.91**	**0.98**	**0.71**

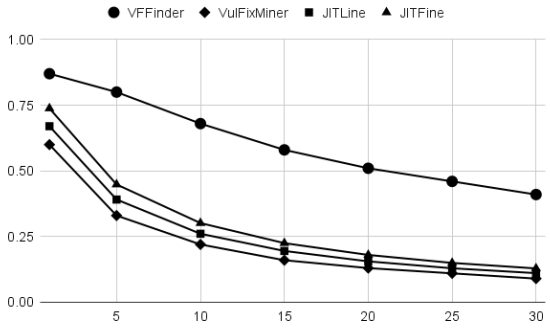

Figure 5. The performance of VFFinder and the existing approaches in different imbalance degrees (non-fix/fix rates)

As seen in Table 2, the AUCs of the approaches are quite high, while the corresponding F1 scores are low. The reason is that the high AUCs result from our imbalanced dataset, where negative samples (non-fixing commits) significantly outnumber positive ones (fixing commits). In such cases, the AUCs are inflated by a surplus of true negatives, leading to a high false positive rate. Consequently, the F1 score remains low due to the challenge of capturing true positives in the context of imbalanced data. It is crucial to recognize that the effectiveness of the classifier might vary across different thresholds, and optimizing solely for AUC may not guarantee a well-performing model (with high F1) [27]. This phenomenon also happens in cases of fraud detection prediction, where the positive samples - the ones we are interested in - do not occur as many times as the negatives (the negatives are too high).

To project the approaches' performance on the real-world non-fix/fix rate, we investigated their performance on various datasets with different rates of fixing and non-fixing commits (Figure 5). As seen, all the approaches perform worse when the proportion of non-fixing commits increases. VFFinder's F1 declines about 56% when testing on the dataset with the non-fix/fix rates from 1:1 to 30:1. This phenomenon occurs for all the vulnerability-fix identification approaches. However, the performance of the existing approaches declines much faster than VFFinder's performance. Indeed, the other approaches' performance declines from 83–85%. Furthermore, VFFinder performs much better than the existing approaches for all the considered rates.

Overall, VFFinder is more effective than the state-of-the-art approaches in identifying vulnerability fixes. This confirms our strategy explicitly representing the code structure changes and using graph-based models to extract features for vulnerability fix identification.

5.2. Intrinsic Analysis (RQ2)

To investigate the contribution of the related unchanged code in αAST, we used two variants of αAST: one considering both changed lines and unchanged lines (αAST), the other considering only changed lines ($\widehat{\alpha AST}$). Table 3 shows the performance of VFFinder using the two αAST variants: VFFinder$_{\alpha AST}$ and VFFinder$_{\widehat{\alpha AST}}$. For simplicity, in this experiment, we used the same GAT model for both representation variants and the dataset with the non-fix/fix rate of 1:1.

As seen, additionally considering the related unchanged lines along with changed lines in αAST significantly improves the performance of VFFinder using αAST with only changed lines. Particularly, VFFinder$_{\alpha AST}$ achieves an equivalent recall rate but a much better *precision* rate, which is 21% higher than that of VFFinder$_{\widehat{\alpha AST}}$. The related unchanged code provides valuable information and helps the model not only understand code changes more precisely but also discover more vulnerability-fix patterns. This confirms our observation 2 on the important role of related unchanged code.

In order to investigate the impact of different GNN models on the vulnerability-fix identification performance, we compare three variants of graph neural networks: VFFinder with GCN [19], GAT [10], GIN [25], and GraphSAGE [26]. In this experiment, we use the dataset with the non-fix/fix rate of 1:1 for simplicity. The results of those four variants are shown in Table 4. As expected, VFFinder obtains quite stable performance with the F1 of 0.88–0.91 and the accuracy of 0.87–0.90. Moreover, while VFFinder obtains quite similar *recall* with GCN, GAT, GIN, and GraphSAGE, it archives the highest *precision*, yet lowest *recall* with GAT. Thus, lightweight relational graph neural networks such as GCN [19] should be applied to achieve cost-effective vulnerability-fix identification performance.

5.3. Sensitivity Analysis (RQ3)

To measure the impact of training data size on VFFinder's performance. In this experiment, the training set is randomly separated into five folds. We gradually increased the training data size by adding one fold at a time until all five folds were added for training. In this experiment, we used GAT and the dataset with the non-fix/fix rate of 1:1. Figure 6 shows VFFinder's performance is improved when expanding the training dataset. The precision, recall, and F1 increase by more than 30% when the training data expands from one fold to five folds. The reason is that with larger training datasets, VFFinder has observed more and performs better. However, the training time of VFFinder with five folds is about 4.53 more than that with a fold.

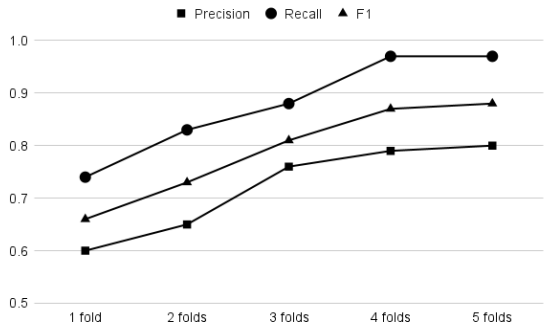

Figure 6. Impact of training data size on VFFinder's performance.

Additionally, we investigate the sensitivity of VFFinder's performance on the input size in the number of changed (i.e., added and deleted) lines of code (LOCs) (Fig. 7). As seen, there are much fewer commits with a larger number of changed LOCs. The *precision* of VFFinder is quite stable when handling commits in different change sizes. Particularly, VFFinder's F1 slightly varies between 0.88 and 0.89 when increasing the change size from 1 to 500 changed LOC. For the fixing commits changing a large part of code, VFFinder can still effectively identify with the recall of 0.73 and the precision of 0.84. Additionally, VFFinder's F1 for the vulnerability fixing commits having more than 500 changed LOC is only about 10% lower than VFFinder's performance for other vulnerability fixing commits.

5.4. Time Complexity (RQ3)

In this work, all our experiments were run on a server running Ubuntu 18.04 with an NVIDIA Tesla P100 GPU. In VFFinder, training the model took about 4–6 hours for 50 epochs. Additionally, VFFinder spent 1–2 seconds to classify whether a commit is a fixing commit or not.

Table 3. Impact of related unchanged code

	Pre.	*Rec.*	*F1*	*Acc.*	*AUC*
VFFinder$_{\alpha AST}$	0.8	0.97	0.88	0.87	0.85
VFFinder$_{\widehat{\alpha AST}}$	0.66	0.97	0.78	0.73	0.74

Table 4. Impact of GNN Models

	Pre.	*Rec.*	*F1*	*Acc.*	*AUC*
GCN [19]	0.84	0.98	0.90	0.90	0.90
GAT [10]	0.80	0.97	0.88	0.87	0.85
GIN [25]	0.81	0.99	0.89	0.88	0.91
GraphSAGE [26]	0.85	0.97	0.91	0.90	0.91

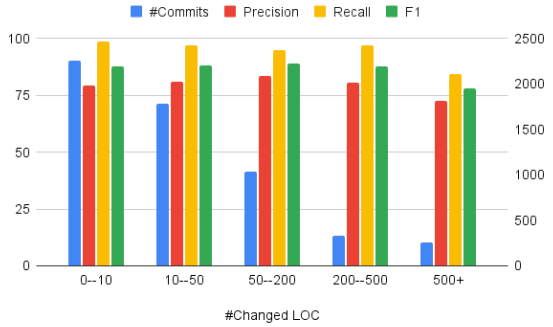

Figure 7. Impact of change size (left axis: *Precision* and *Recall*; right axis: No. of commits)

5.5. Threats to Validity

The main threats to the validity of our work consist of internal, construct, and external threats.

Threats to internal validity encompass the potential impact of the adopted method for building Abstract Syntax Trees (ASTs). To mitigate this challenge, we employ the extensively recognized code analyzer Joern [28]. Another threat lies in the correctness of the implementation of our approach. To reduce such a threat, we carefully reviewed our code and made it public [4] so that other researchers can double-check and reproduce our experiments.

[4]https://github.com/UETISE/VFFinder

Threats to construct validity relate to the suitability of our evaluation procedure. We used *precision*, *recall*, *F1*, *AUC*, *accuracy*, and CostEffort@L. They are the widely-used evaluation measures for vulnerability fix identification and just-in-time defect detection [4, 5, 11, 12]. In addition, a threat may come from the adaptation of the baselines. To mitigate this threat, we directly obtain the original source code from their GitHub repositories or replicate exactly their description in the paper [4, 5]. Also, we use the same hyper-parameters specified in the original papers [5, 11, 12, 29].

Threats to external validity mainly lie in the selection of graph neural network models employed in our experiments. To mitigate this threat, we have chosen widely recognized models with established track records in natural language processing and software engineering domains [10, 19, 25, 26]. Moreover, our experiments are conducted on only the code changes of C/C++ projects. Thus, the outcomes may not be universally applicable to different programming languages. To overcome this limitation, our future research agenda involves performing additional experiments to validate the findings across various programming languages that belong to different paradigms.

6. Related Work

VFFinder directly relates to the vulnerability fix identification works. Representative works in this directions are VulFixMiner [4] and CoLeFunDa [6]. These tools utilize CodeBERT to automatically represent code changes and extract features for identifying vulnerability-fixing commits. On the other hand, Midas [5] constructs different neural networks for each level of code change granularity, corresponding to commit-level, file-level, hunk-level, and line-level, following their natural organization. It then utilizes an ensemble model that combines all base models for the final prediction.

VFFinder also relates to the work on just-in-time vulnerability detection. DeepJIT [29] automatically extracts features from commit messages and changed code and uses them to identify defects. Pornprasit *et al.* propose JITLine, a simple but effective just-in-time defect prediction approach. JITLine utilizes the expert features and token features using bag-of-words from commit messages and changed code to build a defect prediction model with a random forest classifier. LAPredict [30] is a defect prediction model by leveraging the information of "lines of code added" expert feature with the traditional logistic regression classifier. Recently, Ni *et al.* introduced JITFine [11], combining the expert features and the semantic features which are extracted by CodeBERT [7] from changed code and commit messages.

Different from all prior studies in vulnerability fix identification and just-in-time bug detection, our work presents VFFinder which explicitly represents code changes in code structure and applies a graph-based model to extract the features distinguishing fixing commits from non-fixing ones.

Several studies have been proposed for specific software engineering tasks, including code suggestion/completion [31–33], code summarization [34–36], program synthesis [37], pull request description generation [38, 39], code clones [40], fuzz testing[41], code-text translation [42], and bug/vulnerability detection [14, 43, 44].

7. Conclusion

In conclusion, this paper has addressed the critical challenge of identifying silent vulnerability fixes in software projects heavily reliant on third-party libraries. The existing gap between fixes and public disclosures, coupled with the prevalence of undisclosed vulnerability fixes in open-source projects, has hindered effective vulnerability management. We have introduced VFFinder, a novel graph-based approach designed for the automated identification of vulnerability-fixing commits. To precisely capture the meaning of code changes, the changed code is represented in connection with the related unchanged code. In VFFinder, the structure of the changed code and related unchanged code are captured and the structural changes are represented in annotated Abstract Syntax Trees. By leveraging annotated ASTs to capture structural changes, VFFinder enables the extraction of essential structural features. These features are then utilized by graph-based neural network models to differentiate vulnerability-fixing commits from non-fixing ones. Our experimental results show that VFFinder improves the state-of-the-art methods by 272–420% in Precision, 22–70% in Recall, and 3.2X–8.2X in F1. Especially, VFFinder speeds up the silent fix identification process by up to 121% with the same effort reviewing 50K LOC compared to the existing approaches. These findings highlight the superiority of VFFinder in accurately identifying vulnerability fixes and its ability to expedite the review process. The performance of VFFinder contributes to enhancing software security by empowering developers and security auditors with a reliable and efficient tool for identifying and addressing vulnerabilities in a timely manner.

References

[1] A. D. Householder, G. Wassermann, A. Manion, C. King, The Cert Guide to Coordinated Vulnerability Disclosure, Software Engineering Institute, Pittsburgh, PA (2017).

[2] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, Y. Le Traon, SSPCatcher: Learning to Catch Security Patches, Empirical Software Engineering, Vol. 27, No. 6, 2022, pp. 151.

[3] L. Tal, The State of Open Source Security Report, Tech. rep., Snyk (2019).

[4] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, A. E. Hassan, Finding a Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes, in: 36th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2021, pp. 705–716.

[5] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan, et al., Multi-Granularity Detector for Vulnerability Fixes, IEEE Transactions on Software Engineering, Vol. 49, No. 8, (2023).

[6] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, A. E. Hassan, CoLeFunDa: Explainable Silent Vulnerability Fix Identification (2023).

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547.

[8] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, D. Hao, FIRA: Fine-grained Graph-based Code Change Representation for Automated Commit Message Generation, in: The 44th International Conference on Software Engineering, 2022, pp. 970–981.

[9] S. Nguyen, T.-T. Y. Vu, D.-H. Vo, VFFINDER: A Graph-based Approach for Automated Silent Vulnerability-Fix Identification, in: Proceedings of the 15th IEEE International Conference on Knowledge and Systems Engineering, 2023.

[10] P. V. G. C. A. Casanova, A. R. P. Lio, Y. Bengio, Graph Attention Networks, ICLR. Petar Velickovic Guillem Cucurull Arantxa Casanova Adriana Romero Pietro Liò and Yoshua Bengio (2018).

[11] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, D. Lo, The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 672–683.

[12] C. Pornprasit, C. K. Tantithamthavorn, Jitline: A Simpler, Better, Faster, Finer-grained Just-in-time Defect Prediction, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 369–379.

[13] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, M. Hilton, Graph-based Mining of In-the-wild, Fine-grained, Semantic Code Change Patterns, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, IEEE, 2019, pp. 819–830.

[14] S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, H. D. Vo, Code-centric Learning-based Just-In-Time Vulnerability Detection, arXiv preprint arXiv:2304.08396 (2023).

[15] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning Distributed Representations of Code, Proceedings of the ACM on Programming Languages, Vol. 3, No. POPL, 2019, pp. 1–29.

[16] Y. Wang, W. Wang, S. Joty, S. C. Hoi, CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation, in: The 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[17] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, in: Y. Bengio, Y. LeCun (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, 2013.

[18] Z. Ding, H. Li, W. Shang, T.-H. P. Chen, Can Pre-trained Code Embeddings Improve Model Performance? Revisiting the Use of Code Embeddings in Software Engineering Tasks, Empirical Software Engineering, Vol. 27, No. 3, 2022, pp. 1–38.

[19] T. N. Kipf, M. Welling, Semi-Supervised Classification with Graph Convolutional Networks, in: International Conference on Learning Representations, 2016.

[20] J. Chen, T. Ma, C. Xiao, Fastgcn: Fast Learning with Graph Convolutional Networks via Importance Sampling, arXiv preprint arXiv:1801.10247 (2018).

[21] G. Bhandari, A. Naseer, L. Moonen, CVEfixes: Automated Collection of Vulnerabilities and Their Fixes From Open-Source Software, in: The 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30–39.

[22] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[23] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, Advances in neural information processing systems, Vol. 32, (2019).

[24] X. Yang, S. Wang, Y. Li, S. Wang, Does Data Sampling Improve Deep Learning-based Vulnerability De-

tection? Yeas! and Nays!, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2287–2298.

[25] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How Powerful are Graph Neural Networks?, in: International Conference on Learning Representations, 2018.

[26] W. Hamilton, Z. Ying, J. Leskovec, Inductive Representation Learning on Large Graphs, Advances in neural information processing systems, Vol. 30, (2017).

[27] L. A. Jeni, J. F. Cohn, F. De La Torre, Facing Imbalanced Data–Recommendations for the Use of Performance Metrics, in: 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction, Vol. 40, 2013, pp. 47–60.

[28] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

[29] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, N. Ubayashi, DeepJIT: an End-to-end Deep Learning Framework for Just-in-time Defect Prediction, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 34–45.

[30] Z. Zeng, Y. Zhang, H. Zhang, L. Zhang, Deep Just-in-time Defect Prediction: How Far Are We?, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 427–438.

[31] S. Nguyen, H. Phan, T. Le, T. N. Nguyen, Suggesting Natural Method Names to Check Name Consistencies, in: 2020 42nd International Conference on Software Engineering, IEEE, 2020, pp. 1372–1384.

[32] S. Nguyen, T. Nguyen, Y. Li, S. Wang, Combining Program Analysis and Statistical Language Model for Code Statement Completion, in: 34th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2019, pp. 710–721.

[33] S. Nguyen, C. T. Manh, K. T. Tran, T. M. Nguyen, T.-T. Nguyen, K.-T. Ngo, H. D. Vo, ARist: An Effective API Argument Recommendation Approach, Journal of Systems and Software 2023, pp. 111786.

[34] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer, Summarizing Source Code Using a Neural Attention Model, in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.

[35] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, G. Bavota, Studying the Usage of Text-to-text Transfer Transformer to Support Code-Related Tasks, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 336–347.

[36] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, P. S. Yu, Improving Automatic Source Code Summarization via Deep Reinforcement Learning, in: The 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 397–407.

[37] T. Gvero, V. Kuncak, Synthesizing Java Expressions from Free-form Queries, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2015, pp. 416–432.

[38] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep Code Comment Generation, in: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), IEEE, 2018, pp. 200–20010.

[39] Z. Liu, X. Xia, C. Treude, D. Lo, S. Li, Automatic Generation of Pull Request Descriptions, in: 34th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2019, pp. 176–188.

[40] L. Li, H. Feng, W. Zhuang, N. Meng, B. Ryder, Cclearner: A Deep Learning-based Clone Detection Approach, in: International Conference on Software Maintenance and Evolution, IEEE, 2017, pp. 249–260.

[41] P. Godefroid, H. Peleg, R. Singh, Learn&fuzz: Machine Learning for Input Fuzzing, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 50–59.

[42] H. A. Nguyen, H. D. Phan, S. S. Khairunnesa, S. Nguyen, A. Yadavally, S. Wang, H. Rajan, T. Nguyen, A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications, in: 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.

[43] Y. Li, S. Wang, T. N. Nguyen, S. Van Nguyen, Improving Bug Detection via Context-based Code Representation Learning and Attention-based Neural Networks, Proceedings of the ACM on Programming Languages, Vol. 3, No. OOPSLA, 2019, pp. 1–30.

[44] H. D. Vo, S. Nguyen, Can an Old Fashioned Feature Extraction and a Light-weight Model Improve Vulnerability Type Identification Performance?, Information and Software Technology, Vol. 164, 2023, pp. 107304.