

VFFINDER: A Graph-based Approach for Automated Silent Vulnerability-Fix Identification

Son Nguyen, Thanh Trong Vu, and Hieu Dinh Vo*

Faculty of Information Technology

University of Engineering and Technology, Vietnam National University, Hanoi, Vietnam

{sonnguyen,19020626,hieuvd}@vnu.edu.vn

Abstract—The increasing reliance of software projects on third-party libraries has raised concerns about the security of these libraries due to hidden vulnerabilities. Managing these vulnerabilities is challenging due to the time gap between fixes and public disclosures. Moreover, a significant portion of open-source projects silently fix vulnerabilities without disclosure, impacting vulnerability management. Existing tools like OWASP heavily rely on public disclosures, hindering their effectiveness in detecting unknown vulnerabilities. To tackle this problem, automated identification of vulnerability-fixing commits has emerged. However, identifying silent vulnerability fixes remains challenging. This paper presents VFFINDER, a novel graph-based approach for automated silent vulnerability fix identification. VFFINDER captures structural changes using Abstract Syntax Trees (ASTs) and represents them in annotated ASTs. VFFINDER distinguishes vulnerability-fixing commits from non-fixing ones using attention-based graph neural network models to extract structural features. We conducted experiments to evaluate VFFINDER on a dataset of 36K+ fixing and non-fixing commits in 507 real-world C/C++ projects. Our results show that VFFINDER significantly improves the state-of-the-art methods by 39–83% in Precision, 19–148% in Recall, and 30%–109% in F1. Especially, VFFINDER speeds up the silent fix identification process by up to 47% with the same review effort of 5% compared to the existing approaches.

Index Terms—silent vulnerability fixes, vulnerability fix identification, code change representation, graph-based model

I. INTRODUCTION

With the increasing reliance of software projects on third-party libraries, ensuring their security has become a paramount concern. Vulnerabilities hidden within these libraries can have far-reaching consequences, as exemplified by the infamous Log4Shell¹ exploit. One critical challenge in addressing these vulnerabilities is the time gap between their fixes and public disclosures [1], [2]. For instance, Log4Shell’s patch was pushed four days prior to its public revelation. Another example is that the Apache Struts Remote Code Execution vulnerability², which led to the Equifax breach in 2017, was disclosed to the public in August 2018, but was patched in June, 2018³. Two months is plenty of time for the potential exploitation of vulnerable software. If the library were monitored to identify vulnerability patches, the library’s users would have been aware of the potential exploitation and 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 [3], [4]. 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 Open Web Application Security Project (OWASP) or National Vulnerability Database (NVD). However, CVE/NVD and public vulnerability databases miss many vulnerabilities [4].

Recognizing the importance of identifying vulnerability-fixing commits, several security companies such as Huawei, Veracode, Mend, and Snyk have been monitoring open-source libraries’ commits and other software artifacts to provide their users with early warnings of unpublished vulnerabilities. However, the process of identifying silent vulnerability fixes is very challenging in practice. For example, constructing a dataset of 1,282 vulnerability-fixing commits required approximately four years [5]. Thus, automated vulnerability-fixing commit identification could help researchers maintain and update vulnerability databases, including NVD.

To address this problem, several vulnerability fix identification techniques have been proposed. Following the good practice of coordinated vulnerability disclosure [1], [2], 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. The state-of-the-art techniques, such as VulFixMiner [6], CoLeFunDa [7], and Midas [8], represent changes in the lexical form of code and apply CodeBERT [9] 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 [10].

In this paper, we propose VFFINDER, a novel graph-based approach for automated vulnerability fix identification. Our idea is to capture the semantic meaning of code changes better, we explicitly represent the changes in code structure. Particularly, for a commit c , the structure of the versions before and after c are represented by the Abstract Syntax Trees (ASTs). These ASTs are mapped to build an annotated

*Corresponding author.

¹<https://nvd.nist.gov/vuln/detail/CVE-2021-44228>

²<https://nvd.nist.gov/vuln/detail/CVE-2018-11776>

³<https://github.com/apache/struts/commit/6e87474>

AST (α AST), a fine-grained graph representing the changes in the code structure caused by c . In α ASTs, 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 α ASTs, we develop a graph attention network (GAT) model [11] to extract semantic features. Then, these features are used to distinguish from vulnerability-fixing commits to non-fixing ones.

We conducted several experiments to evaluate VFFINDER’s performance on a dataset of 36K+ fixing and non-fixing commits in 507 real-world C/C++ projects. Our results show that VFFINDER significantly improves the state-of-the-art techniques [6], [8], [12], [13] by 39–83% in Precision, up to 148% in Recall, and 109% in F1. Especially, VFFINDER speeds up the silent fix identification process up to 47% with the same review effort of 5% compared to the existing approaches.

In brief, this paper makes the following major contributions:

- 1) VFFINDER: A novel graph-based approach for identifying silent vulnerability-fixes.
- 2) An extensive experimental evaluation showing the performance of VFFINDER over the state-of-the-art methods for vulnerability-fix identification.

II. CODE CHANGE REPRESENTATION

In this work, we represent the syntactic aspect via the *structure* relation using Abstract Syntax Tree (ASTs).

Definition 1 (Annotated AST - α AST). *For a commit changing code from one version to another, the annotated abstract syntax tree (annotated AST) is an annotated graph constructed from the ASTs of these two versions. Formally, for $AST_o = \langle N_o, E_o \rangle$ and $AST_n = \langle N_n, E_n \rangle$ which are the ASTs of the old version and the new version, respectively, the α AST $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$ is defined as followings:*

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

Fig. 1 shows a piece of changed code, the ASTs before and after the change, and the annotated AST constructed from these ASTs. The α AST expresses the change in the structure of the code at line 4. Particularly, the right-hand-side of the less-than expression (`BUF_SIZE`) is replaced by the multiply-expression (i.e., `2*BUF_SIZE`).

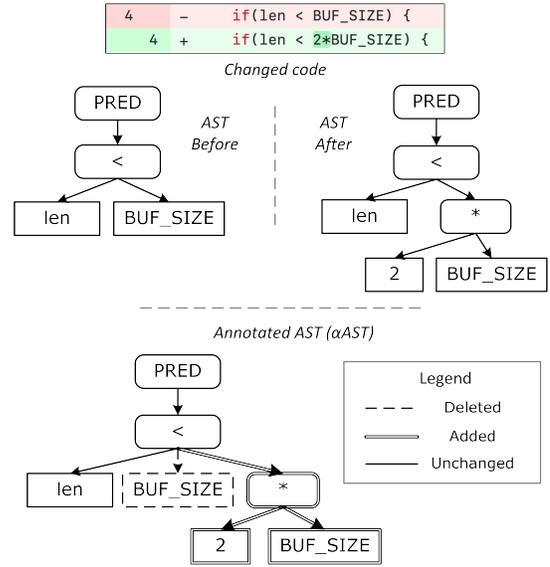


Fig. 1: Annotated AST: An example

III. APPROACH

Fig. 2 illustrates the overview of our approach, VFFINDER, for vulnerability-fixing commit identification. First, the given commits and their repositories are used to construct their corresponding α ASTs (*Change representation*). Each AST node in α ASTs is embedded in the corresponding vectors (*Embedding*). After that, a Graph Neural Network (GNN) is applied to extract structural features from constructed α ASTs (*Feature extraction*). Finally, the extracted structural features are used for learning and predicting vulnerability-fixing commits (*Prediction*).

Particularly, in the *Embedding* step, for each α AST, $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$, every node in \mathcal{N} is embedded into d -dimensional hidden features n_i produced by embedding the content of the nodes. To build the vectors for nodes’ content, we use Word2vec [14], one of the most popular code embedding techniques for code [15]. The reason is that the number of AST nodes in α ASTs could be huge. Thus, for a practical embedding step for α ASTs, we apply Word2vec, which is known as an efficient embedding technique [15]. 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 α AST, $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$, we develop a Graph Attention Network (GAT) [11] 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 the representations for the graph’s nodes through message passing [11], [16], 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

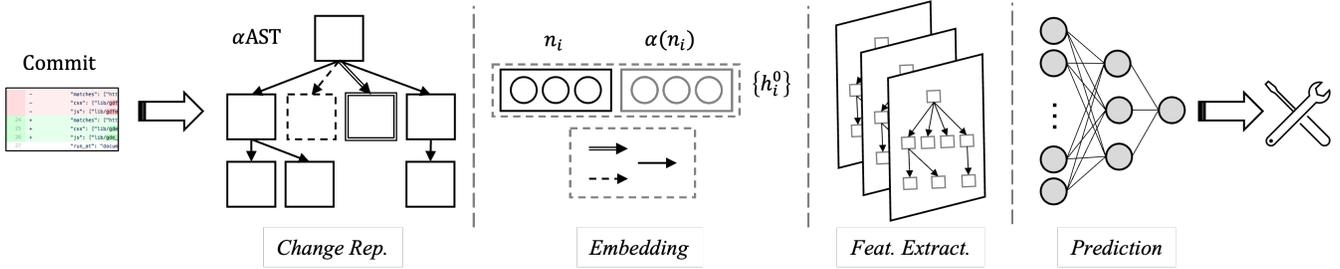


Fig. 2: VFFINDER: A Graph-based Approach for Vulnerability-fixing Commit Identification

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 [16], [17]. In this work, we implicitly define α_{ij} based on node features [11] by employing the self-attention mechanism, where unnormalized coefficients E_{ij} for pairs of nodes i, j are computed based on their features:

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

where \parallel 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 α AST $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$ 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 α AST \mathcal{T} is a fixing commit or not.

IV. 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 [6], [8]?

RQ2: Sensitivity Analysis. How do various factors of the input, including training data size and changed code complexity, affect VFFINDER's performance?

RQ3: Time Complexity. What is VFFINDER's running time?

TABLE I: Dataset statistics

	#Projects	#Fixes	#Non-fixes	#Changed LOC
Training	402	9,176	21,193	3,807,967
Testing	105	2,123	3,878	992,938
Total	507	11,299	25,071	4,800,905

A. Dataset

In this work, we collect the vulnerability-fixing commits from various public vulnerability datasets [18]–[20]. In total, we collected the fixing commits for the vulnerabilities reported from 1990 to 2022 in real-world 507 C/C++ projects. For a pragmatic evaluation, we collected the (remaining) non-fixing commits in the popular projects among these projects, such as FFmpeg, Qemu, Linux, and Tensorflow. The total numbers of collected fixing commits and non-fixing commits are about 11K and 25K, respectively. Table I shows the overview of our dataset. The details of our dataset can be found at: <https://github.com/thanhtlx/VFFinder>.

B. Procedure

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

1) **MiDas** [8] 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 to generate the final prediction.

2) **VulFixMiner** [6] and **CoLeFunDa** [7] utilize CodeBERT to automatically represent code changes and extract features for identifying vulnerability-fixing commits. 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 [8].

Additionally, we applied the same procedure adapting the state-of-the-art just-in-time defect detection techniques for vulnerability fix identification as in the work of Zhou *et al.* [8]. The additional baselines include:

3) **JITLine** [13]: A simple but effective method utilizing changed code and expert features to detect buggy commits.

4) **JITFine** [12]: 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 GAT layers $L = 2$ for a practical evaluation.

In this comparative study, we follow the same cross-project evaluation procedure to construct the training data and testing data from the dataset as in the prior work [6], [8]. Particularly, the whole set of projects is randomly split into 80% (402 projects) for training and 20% (105 projects) for testing. The details of the training set and testing set are shown in Table I.

For **RQ2. Sensitivity Analysis**, we studied the impacts of the following factors on VFFINDER’s performance: training size and change size in the number of changed lines of code (LOCs). To systematically vary these factors, we gradually added more training data and varied the change size.

C. Metrics

To evaluate the vulnerability fix identification approaches, we measure the classification *accuracy*, *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 which are correctly classified among all the tested commits. 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, $CE@L$ ($CE@L$), which is used in [6], [8]. $CE@L$ 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 $L\%$ of total lines of code (LOC).

V. EXPERIMENTAL RESULTS

A. Performance Comparison (RQ1)

Table II shows the performance of VFFINDER and the state-of-the-art vulnerability fix identification approaches. As seen, VFFINDER significantly outperforms the state-of-the-art vulnerability fix identification approaches. Particularly, the VFFINDER achieves a recall of 0.57, which is more than **19–148%** the recall rates of VulFixMiner and MiDas, respectively. Additionally, VFFINDER is still much more precise than the existing approaches with about **39–83%** 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 $CE@5\%$ of VFFINDER is **0.50**, which is **19–47%** better than the corresponding figures of MiDas and VulFixMiner. This means that given 5% effort (in LOC), the number of the fixing commits found by using

TABLE II: Comparison Results

	<i>Pre.</i>	<i>Rec.</i>	<i>F1</i>	<i>Acc.</i>	<i>AUC</i>	<i>CE@5%</i>
JITLine	0.62	0.23	0.33	0.68	0.65	0.23
JITFine	0.58	0.48	0.53	0.69	0.69	0.45
VulFixMiner	0.47	0.42	0.44	0.63	0.64	0.34
MiDas	0.53	0.25	0.34	0.66	0.58	0.42
VFFINDER	0.86	0.57	0.69	0.82	0.79	0.50

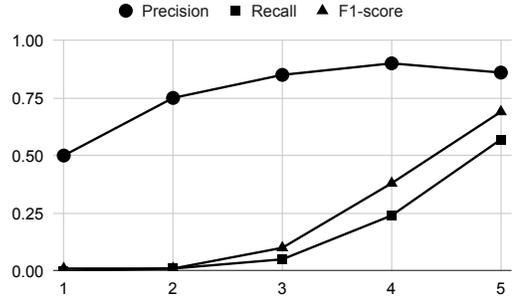


Fig. 3: Impact of training data size on VFFINDER’s performance

VFFINDER is much larger compared to those found by using MiDas and VulFixMiner.

Answer to RQ1: 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.

B. Sensitivity Analysis (RQ2)

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. As shown in Fig. 3, VFFINDER’s performance is improved when expanding the training dataset. The precision increases by 72% when the training data expands. Especially, the recall and F1-score grow much more significantly when increasing the training size from 1 fold to 5 folds. The reason is that with larger training datasets, VFFINDER has observed more and then performs better. However, training with a larger dataset costs more time. The training time of VFFINDER with five folds is about 3X more than that with a fold.

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 (LOC) (Fig. 4). As seen, there are fewer commits with a larger number of changed LOC. The *precision* of VFFINDER is quite stable when handling commits in different change sizes. Meanwhile, the *recall* significantly grows from 24% to 93% when increasing the change size. The reason could be that fixing commits tend to have less changed code [12], [21]. Thus, in the set of commit having a smaller number of changes, the vulnerability fix identification techniques could achieve a lower recall due to a larger number of fixing commits being identified.

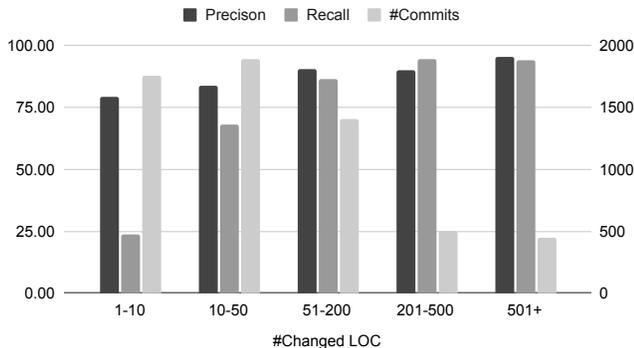


Fig. 4: Impact of change size (left axis: *Precision* and *Recall*; right axis: No. of commits)

Answer for RQ2: VFFINDER performs better when being trained on a larger dataset. Additionally, VFFINDER’s precision is quite stable with different change sizes, while the recall is better with larger code changes.

C. 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.

D. Threats to Validity

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

Threats to internal validity include the influence of the method used to construct AST. To reduce this threat, we use the widely-used code analyzer Joern [22]. Another threat mainly lies in the correctness of the implementation of our approach. To reduce such a threat, we carefully reviewed our code and made it public [23] so that other researchers can double-check and reproduce our experiments.

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 [6], [8], [12], [13]. Besides, 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 [6]. Also, we use the same hyperparameters as in the original papers [8], [12], [13], [24].

Threats to external validity mainly lie in the selection of graph neural network models used in our experiments. To mitigate this threat, we select the representative models which are well-known for NLP and SE tasks [11], [16]. Moreover, our experiments are conducted on only the code changes of C/C++ projects. Thus, the results could not be generalized for other languages. In our future work, we plan to conduct more experiments to validate our results in other languages.

VI. RELATED WORK

VFFINDER relates to the vulnerability fix identification work. VulFixMiner [6] and CoLeFunDa [7] utilize CodeBERT to automatically represent code changes and extract features for identifying vulnerability-fixing commits. Midas [8] 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 to generate the final prediction.

VFFINDER also relates to the work on just-in-time vulnerability detection. DeepJIT [24] 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 [21] 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 [12], combining the expert features and the semantic features which are extracted by CodeBERT [9] 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 SE tasks, including code suggestion/completion [25]–[27], program synthesis [28], pull request description generation [29], [30], code summarization [31]–[33], code clones [34], fuzz testing [35], code-text translation [36], bug/vulnerability detection [37]–[39], and program repair [40], [41].

VII. CONCLUSION

In this paper, we have introduced VFFINDER, an novel graph-based approach designed for the automated identification of vulnerability-fixing commits. By leveraging ASTs to capture structural changes and representing them as annotated ASTs, 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 30–60% in Precision, 2.0X–4.0X in Recall, and 62%–160% in F1. Especially, VFFINDER speeds up the silent fix identification process up to 2.6X with the same review effort of 5%. 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] 14:00-17:00, "ISO/IEC 29147:2018." [Online]. Available: <https://www.iso.org/standard/72311.html>
- [2] A. D. Householder, G. Wassermann, A. Manion, and C. King, "The cert guide to coordinated vulnerability disclosure," *Software Engineering Institute, Pittsburgh, PA*, 2017.
- [3] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. Le Traon, "Sspatcher: Learning to catch security patches," *Empirical Software Engineering*, vol. 27, no. 6, p. 151, 2022.
- [4] L. Tal, "The state of open source security report," Snyk, Tech. Rep., 2019.
- [5] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.
- [6] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.
- [7] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, "Colefunda: Explainable silent vulnerability fix identification," 2023.
- [8] 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*, 2023.
- [9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.
- [10] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: fine-grained graph-based code change representation for automated commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.
- [11] P. V. G. C. A. Casanova, A. R. P. Lio, and Y. Bengio, "Graph attention networks," *ICLR. Petar Velickovic Guillem Cucurull Arantxa Casanova Adriana Romero Pietro Liò and Yoshua Bengio*, 2018.
- [12] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and 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.
- [13] C. Pornprasit and 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.
- [14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013*, Y. Bengio and Y. LeCun, Eds., 2013.
- [15] Z. Ding, H. Li, W. Shang, and 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, pp. 1–38, 2022.
- [16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2016.
- [17] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.
- [18] G. Bhandari, A. Naseer, and L. Moonen, "Cvfixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.
- [19] J. Fan, Y. Li, S. Wang, and 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.
- [20] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.
- [21] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *Proceedings of the 30th International Symposium on Software Testing and Analysis*, 2021, pp. 427–438.
- [22] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
- [23] [Online]. Available: <https://github.com/UETISE/VFFinder>
- [24] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and 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.
- [25] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *2020 42nd International Conference on Software Engineering*. IEEE, 2020, pp. 1372–1384.
- [26] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 710–721.
- [27] S. Nguyen, C. T. Manh, K. T. Tran, T. M. Nguyen, T.-T. Nguyen, K.-T. Ngo, and H. D. Vo, "Arist: An effective api argument recommendation approach," *Journal of Systems and Software*, p. 111786, 2023.
- [28] T. Gvero and 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.
- [29] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–2010.
- [30] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2019, pp. 176–188.
- [31] S. Iyer, I. Konstas, A. Cheung, and 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.
- [32] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyanyk, R. Oliveto, and 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.
- [33] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.
- [34] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *International Conference on Software Maintenance and Evolution*. IEEE, 2017, pp. 249–260.
- [35] P. Godefroid, H. Peleg, and 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.
- [36] H. A. Nguyen, H. D. Phan, S. S. Khairunnesa, S. Nguyen, A. Yadavally, S. Wang, H. Rajan, and 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.
- [37] Y. Li, S. Wang, T. N. Nguyen, and 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, pp. 1–30, 2019.
- [38] S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, and H. D. Vo, "Code-centric learning-based just-in-time vulnerability detection," *arXiv preprint arXiv:2304.08396*, 2023.
- [39] H. D. Vo and S. Nguyen, "Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance?" *Information and Software Technology*, vol. 164, p. 107304, 2023.
- [40] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.
- [41] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020, pp. 275–286.