MGVul: a Multi-Granularity Detection Framework for Software Vulnerability

Xiangyu Zhao*, Yanjun Li*, Zhengpeng Zha*†, Zhenhua Ling‡
* Institute of Advanced Technology, University of Science and Technology of China
Email: {xyzhao23, ballinli}@mail.ustc.edu.cn,
† Corresponding author. Email: zhazp@ustc.edu.cn,
‡ School of Information Science and Technology, University of Science and Technology of China
E-mail: zhling@ustc.edu.cn

***Abstract*—Source code vulnerability detection is a critical issue in software security. Existing detection methods primarily focus on the function-granularity, neglecting inter-function call information and lacking the ability to detect inter-function vulnerability. The isolation of functions is one of the reasons that limits the performance of deep learning models. In this work, we propose MGVul: a Multi-granularity detection framework for software vulnerability. This framework treats a project as a whole graph, files as subgraphs, functions as nodes, and function call relationships as edges. By using a multi-expert mixture graph neural network to learn inter-function call information, it not only improves the model's detection performance at the function-granularity but also is capable of handling detection tasks at the file-granularity and project-granularity. Preliminary evaluation shows that introducing inter-function call relationships bring an average F1-score improvement of 7.99% for function-level detection. Meanwhile, our framework also shows promising results at both the file-granularity and the project-granularity. The source code will be released upon paper acceptance.**

I. Introduction

The development of the Internet industry has led to the widespread influence of open-source software. According to information from CVE (Common Vulnerabilities and Exposures) [1], the number of vulnerabilities has rapidly increased in recent years [2]. Research on source code vulnerability detection is also growing steadily. With the development of artificial intelligence technology, detection methods have shifted from expert-dominated static and dynamic analysis to machine learning and deep learning models [3].

In recent years, most research based on deep learning, using either pre-trained language models or graph neural networks, has focused on individual functions[4]–[7]. Specifically, the models take a single function as input and output whether the function contains a vulnerability. Pre-trained language models treat functions as text and perform classification through steps such as tokenization and embedding [6], [8]. Graph neural networks detect vulnerabilities by learning from the Abstract Syntax Tree (AST) and other representations parsed from a single function [9], [10]. Despite significant progress in function-granularity detection in recent years, there remains an insurmountable barrier [7]. We argue that the problem lies in the absence of some critical information: the function call relationships are not provided to the model for learning and the model cannot observe the internal conditions of related functions. Since vulnerabilities often occur during function calls [11]–[13], it is reasonable to view a project as a whole rather than as multiple individual entities.

In this paper, we propose MGVul, which enhances the model's detection capability at the function-granularity by using function call graphs. Additionally, by learning the graph structure from different levels (whole graph, subgraphs, and nodes), we can naturally accomplish vulnerability detection tasks at multiple granularities (project, file, and function) by aggregating information at the corresponding level. Firstly, we use code parsing tools to analyze software projects and capture the complex call relationships between multiple functions in multiple files. Then, we encode functions as fixed-length vectors using pre-trained language models, treating them as function features. Moreover, we utilize a graph neural network (GNN) with a multi-expert mixture model to learn the function call graph from different perspectives. Furthermore, we introduce a pairwise training method to deepen the model's understanding of vulnerabilities in the training parse. Finally, we treat the project as a whole graph, files as subgraphs, and functions as nodes, conducting classification tasks at all these granularities for vulnerability detection.

Experimental results show that MGVul can effectively improve function-granularity detection performance compared to baseline models. Additionally, our framework is also competent for file-granularity and project-granularity detection.

In summary, the contributions of this work are as follows:

- We introduce function call information into function-level vulnerability detection tasks by using GNN to learn the function call graph.
- By aggregating information at the corresponding granularity, our framework is capable of handling detection tasks at the file-granularity and project-granularity.
- Through preliminary evaluation, incorporating function call information improves the model's function-granularity detection performance, with an average F1-score improvement of 7.99%. For the other two granularity levels of detection, MGVul also demonstrates encouraging results.

II. PRELIMINARY

This section introduces intra-function vulnerability and inter-function vulnerability with their examples. Additionally, we provide the definition of the multi-granularity source code vulnerability detection.

A. Intra-function Vulnerability and Inter-function Vulnerability

Intra-function vulnerability refers to a vulnerability that occurs within a single function. Figure 1(a) illustrates an example (CVE-2014-1266) [14]. Due to the extra line *"goto fail;"*, the signature verification always fails. Another scenario is when vulnerabilities arise during a function call, even though both the caller and callee function are flawless. Figure 1(b) provides another example (CVE-2014-3513) [15]. Both the *ssl3_read_n()* and *ssl3_get_record()* functions are benign on their own. However, when *ssl3_get_record()* calls *ssl3_read_n()*, if the length field of the read data packet is incorrect, *ssl3_read_n()* will enter an infinite loop, resulting in a Denial of Service (DoS) attack.

```
static OSStatus SSLVerifySignedServerKeyExchange(
    SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
    uint8_t *signature, UInt16 signatureLen)
{
    OSStatus err;
    ...
    if ((err =
    SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err =
    SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        /* MISSING BRACE - leads to always failing */
        goto fail;
    ...
}
```

(a) An example of the intra-function vulnerability.

```
int ssl3_read_n(SSL *s, int n, int max, int extend) {
    ...
    if (s->packet_length < (unsigned int)n) {
        ret = ssl3_read_n(
        s, n-s->packet_length, max-s->packet_length, 1);
    }
    ...
    return ret;
}

int ssl3_get_record(SSL *s) {
    int n;
    ...
    n = ssl3_read_n(
    s, SSL3_RT_HEADER_LENGTH, SSL3_RT_HEADER_LENGTH, 0);
    ...
}
```

(b) An example of the inter-function vulnerability.

Fig. 1: Examples of different categories of function vulnerabilities, the blue-highlighted lines cause the vulnerability.

B. Multi-granularity Source Code Vulnerability Detection

Given a project $P = \{F_1, F_2, \ldots, F_T\}$, where F_t is a file belonging to P and T is the number of files. Each file F_t contains a set of functions $\{f_t^1, f_t^2, \ldots, f_t^{N_t}\}$, where f_t^n is a function belonging to F_t and N_t is the number of functions of F_t. We aim to learn a mapping to predict whether the project P, each file F_t, and each function f_t^n contain vulnerabilities. Each of these predictions is a binary classification problem.

III. METHODOLOGY

A. Overview

We use a code parsing tool to transform the project P into a function call graph $G = \{SG_1, SG_2, \ldots, SG_T\}$, where each subgraph SG_t corresponds to a file F_t in P. Each subgraph $SG_t = \{ND_t^1, ND_t^2, \ldots, ND_t^{N_t}\}$ consists of nodes ND_t^n that correspond to a function f_t^n. The edges in G represent the function call relationships. We employ a GNN to learn from G and perform classification tasks at three granularities: the whole graph level, the subgraph level and the node level. The framework of our approach is illustrated in Figure 2. The framework consists of three main components: function call graph generation, input representation for code, MGVul training and prediction.

B. Function Call Graph Generation

A clear function call relationships is crucial for vulnerability detection. We chose Cflow [16], a widely used and free parsing software. It can resolve not only intra-file call relationships, but also inter-file function calls. By parsing the files to be detected in a project, we obtained the function call graph, where each function represents a node, and the nodes belonging to the same file form a subgraph.

C. Input Representation

The goal of this phase is to convert functions into compact, uniform-length feature vectors that preserve semantic and syntactic information. This uniform length is crucial because the feature vectors are used as node features in the graph.

To achieve this, we utilize pre-trained language models for code representation. Trained on large code corpora, these models can effectively capture the intricate semantic and syntactic details of the source code. The code is input into the model, which outputs a sequence of contextual embeddings. Such embeddings are then converted to code features.

D. MGVul Training and Prediction

The goal of this phase is to leverage the aggregation capabilities of GNN to learn from functions with call relationships. Since function calls can have multiple layers and long-range dependencies, we use Gated Graph Neural Network (GGNN) [17] to effectively learn and integrate the embeddings due to their excellent depth perception.

Our multi-granularity detection requires learning features from multiple levels, a single network may not effectively meet these requirements. We need to aggregate information from different perspectives (granularities) to suit different focuses. Therefore, we introduce the multi-expert mixture model[18]. Specifically, each f_t^n is transformed into $\mathbf{e}_t^n$ after encoding.

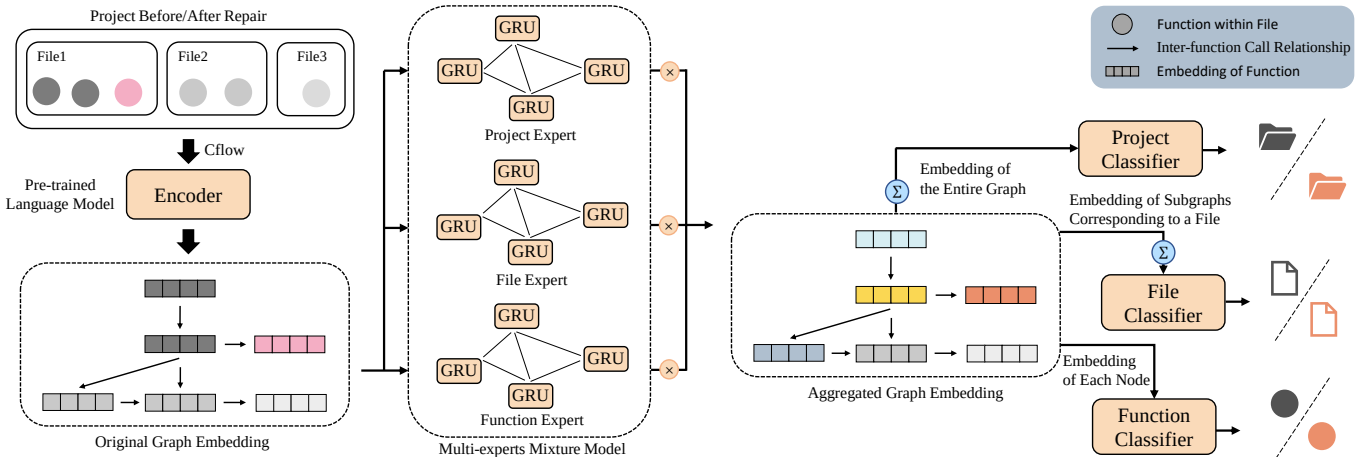

Fig. 2: Overview of the MGVul detection framework.

These embeddings $\mathbf{e}_t^n$ are then processed through three independent expert networks (GGNN) for node communication and feature integration, generating $(\mathbf{e}_t^n)_1, (\mathbf{e}_t^n)_2, (\mathbf{e}3_t^n)_3$, the final representation of the node $(\mathbf{e}_t^n)'$ can be formulated as:

$$(\mathbf{e}_t^n)' = \sum_{i=1}^{3} g(x)_i (\mathbf{e}_t^n)_i. \tag{1}$$

Here, $(\mathbf{e}_t^n)_i, i = 1, 2, 3$ are 3 expert GGNNs and g represents a gating network that ensembles the results from all experts. The term $\sum_{i=1}^{3} g(x)_i = 1$, and $g(x)_i$, the i-th logit of the output of $g(x)$, indicates the weight for representation $(\mathbf{e}_t^n)_i$.

After obtaining the final representation of the nodes, we generate the features of the subgraph $(\mathbf{e}_t)'$ based on the nodes' affiliation. Specifically, we compute $(\mathbf{e}_t)'$ as the average of $(\mathbf{e}_t^n)'$:

$$(\mathbf{e}_t)' = \frac{1}{n} \sum_{n=1}^{N} (\mathbf{e}_t^n)'. \tag{2}$$

The feature of the entire graph $(\mathbf{e}_P)'$ is generated by averaging all the node features. Specifically, we compute $(e_P)'$ as follows:

$$(\mathbf{e}_P)' = \frac{1}{T} \sum_{t=1}^{T} \left(\frac{1}{N_t} \sum_{n=1}^{N_t} (\mathbf{e}_t^n)' \right). \tag{3}$$

After obtaining the representations at each granularity, we complete our task by adding classification layers separately for each granularity. During training, we compute the weighted sum of the loss functions for the three granularities.

Since many of the vulnerability detection datasets are built by crawling historical patches [12], [19], the advantage of these approaches is that they provide code changes before and after vulnerabilities are fixed, and we hope that the model can observe such subtle changes in order to enhance the model's understanding of the vulnerabilities. To this end, we propose a pairwise training strategy. Specifically, we train the model by packing pairs of data in a batch so that the model learns not only to detect anomalies in the vulnerability code but also to understand the logic of fixing the code. $Loss_{all}$ is the sum of $Loss_{before}$ and $Loss_{after}$. The specific computation can be formulated as:

$$Loss_{all} = Loss_{before} + Loss_{after} \tag{4}$$

$$Loss_{stage} = \alpha * Loss_{graph} + \beta * Loss_{subgraph} + \gamma * Loss_{node}, \tag{5}$$

where α, β, and γ are hyperparameters indicating the contribution of different granularities. The *stage* can be either *before* or *after*. During the inference phase, the input data consists of individual graphs rather than pairs, so the pairwise training strateg is only used in the training phase.

IV. EVALUATION

- **RQ1:** Can learning inter-function call relationships help improve the model's function-granularity vulnerability detection capability?
- **RQ2:** How effective are the multi-expert mixture model and the pairwise training strategy in our proposed method on multi-granularity detection?

A. Dataset Details

Most current datasets are still at the function-granularity [8], [20], where individual functions and their labels are used for training. To the best of our knowledge, ReposVul [12] is the first repository-level vulnerability dataset. It contains a large number of real security incidents, meeting our requirements for call parsing. We believe that this form of data organisation is more reasonable for the vulnerability detection task. We selected the C/C++ portion of ReposVul for our experiments.

We treated each record in ReposVul, consisting of files related to C/C++ before and after fixing, as a project pair. After parsing, each file in these projects is regarded as a subgraph, and each function within these files is regarded as a node. By contacting the authors, we determined the organisation of the dataset with the meaning of the relevant targets. We excluded records with files marked as -1, indicating discrepancies between large language models and static code analysis tools. Similarly, we excluded records where Cflow

parsing was unsuccessful. The target of a project depends on the files it includes: if it contains any file with a target of 1, the project's target is 1 (indicating the presence of one or more vulnerabilities), otherwise, it is 0 (indicating no vulnerabilities). After processing, we obtained 2,220 graphs (project), containing 3,933 subgraphs (file), and a total of 132,088 nodes (function). We split the dataset into train, validation, and test sets in an 8:1:1 ratio.

B. Experimental Settings and Metrics

For the pre-trained language model, we used the embedding of the special token ([CLS]) as the input representation of the function, which served as the node feature for the GGNN, with a dimension of 768. The hyperparameters α, β, and γ were set to 50, 10 and 1. We used a two-layer MLP for the classification layers. The loss function was the Cross-entropy loss, and the optimizer was Adam with a learning rate of 2×10^{-5}. The batch size for training was set to 32. All experiments were conducted on an NVIDIA RTX 3090 GPU.

For evaluation metrics, we used Precision, Recall, and F1-score. Precision is the proportion of correctly predicted positive samples among all predicted positive samples. Recall is the proportion of correctly identified positive samples among all actual positive samples. The F1-score is the harmonic mean of Precision and Recall.

C. Effectiveness of Introducing Inter-function Call Information at Function Granularity (RQ1)

To the best of our knowledge, vulnerability detection at the function granularity is still the mainstream nowadays. To address RQ1, we validate the effectiveness by comparing to baseline models. For fairness, our encoder is set to be the same as the baseline. Specifically, for the baseline models, we use the encoder to convert functions into vectors and then directly apply a classifier. For the MGVul framework, we use the same encoder to convert functions into vectors, then use GGNN for feature communication and aggregation, and finally apply the classification layer on the node features. We selected four pre-trained code models: CodeBERT [21], GraphCodeBERT [22], PDBERT [5] and UniXcoder [23]. These models are widely used as encoders in vulnerability detection tasks.

Table 1 presents our experiment results. We observed that all four baseline models showed significant improvements in function-granularity detection performance after incorporating inter-function call information. The UniXcoder model exhibited the most substantial improvement, with an F1-score increase of 12.80%. Compared to recall, the precision rate was generally lower, indicating a higher likelihood of false positives in the current dataset. Surprisingly, CodeBERT achieved the highest recall. One possible reason is that the structural information introduced during the pre-training phase may have misled the model.

TABLE I: Evaluation results for the effectiveness of MGVul.

Method	Precision	Recall	F1-score
CodeBERT	3.06	57.50	5.81
GraphCodeBERT	5.17	30.23	8.83
PDBERT	4.39	32.50	7.74
UniXcoder	10.62	36.25	16.43
MGVul			
+CodeBERT	4.79	**61.25**	9.74
+GraphCodeBERT	6.35	52.33	11.32
+PDBERT	12.63	54.00	20.47
+UniXcoder	**20.10**	53.55	**29.23**

TABLE II: Evaluation results of ablation studies on the components of MGVul.

Method	Node_F1	Subgraph_F1	Graph_F1
MGVul	**29.23**	**31.25**	**33.33**
w/o PTS	28.79	29.73	30.30
w/o MeMM	28.77	31.15	29.09
w/o PTS & MeMM	26.64	29.73	29.19

Observation 1. Learning inter-function call relationships using GGNN improves the model's function-granularity vulnerability detection capability.

D. Effectiveness of MGVul and the ablation studies of its components (RQ2)

MGVul can perform detection tasks not only at the function-granularity but also at the file and project granularity. To answer RQ2, we use UniXcoder as the encoder to validate the detection capability of the MGVul framework and verify the effectiveness of the multi-expert mixture model (MeMM) and the pairwise training strategy (PTS). For the case without MEMM, we use only a single GGNN. For the case without PTS, we use disrupted data.

Table 2 presents our experiment results. We observed that both MeMM and PTS can improve the model's detection capability at three function granularities. Specifically, MeMM focuses on enhancing detection at the project-granularity, while PTS focuses on improving file-granularity detection. When both methods are used together, the model achieves the best performance. The F1-score increased by 2.59%, 1.52%, and 4.14%, respectively.

Observation 2. The MGVul framework shows promising results across all three detection granularities. And MeMM and PTS are useful for our framework.

V. RELATED WORK

In recent work, learning-based detection methods have gradually become mainstream [24]–[26]. Among these methods, most operate at the function level. VulCNN [27] learns vulnerability features by converting functions into images. MVulD

[28] proposes a multimodal function-level vulnerability detection method. Meanwhile, some studies operate at other granularities. Vuldeepecker [29] uses code gadgets to represent programs for vulnerability detection. Ghaffarian et al. [30]provide a detection method at the file level using parsed graphs and graph neural networks. The work at different granularity levels lacks a unified framework, which our work attempts to address. Our work designs the vulnerability detection task as a unified framework by learning function call relationships through graph neural networks and code semantics through pre-trained models.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose the MGVul framework, which enhances detection performance at the function-granularity and shows promising results at the file and project granularities. Our framework combines pre-trained language models and GGNN, using a multi-expert mixture model and pairwise training strategy. Evaluations show our method outperforms baseline models. However, precision is notably lower than recall, possibly due to the imbalance in the number of vulnerable and non-vulnerable classes. In the future, we plan to adopt data augmentation methods to reduce class imbalance. At the same time, the introduction of graph neural networks may lead to efficiency problems, which is the focus of the future research.

REFERENCES

[1] MITRE Corporation, *Common Vulnerabilities and Exposures*, https://cve.mitre.org/, Accessed: 2024-06-09, 2024.

[2] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103 009, 2021.

[3] B. Wu and F. Zou, "Code vulnerability detection based on deep sequence and graph models: A survey," *Security and Communication Networks*, vol. 2022, no. 1, p. 1 176 898, 2022.

[4] 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.

[5] Z. Liu, Z. Tang, J. Zhang, X. Xia, and X. Yang, "Pre-training by predicting program dependencies for vulnerability analysis tasks," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[6] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*, IEEE, 2022, pp. 1–8.

[7] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[8] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.

[9] X. Wang, Y. Wang, Y. Wan, *et al.*, *Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. corr abs/2205.02029 (2022)*, 2022.

[10] X. Duan, J. Wu, S. Ji, *et al.*, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities.," in *IJCAI*, 2019, pp. 4665–4671.

[11] Z. Li, N. Wang, D. Zou, *et al.*, "On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[12] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, "Reposvul: A repository-level high-quality vulnerability dataset," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 472–483.

[13] J. Sun, J. Chen, Z. Xing, Q. Lu, X. Xu, and L. Zhu, "Where is it? tracing the vulnerability-relevant files from vulnerability reports," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[14] Apple, *Improper Certificate Validation*, https://nvd.nist.gov/vuln/detail/CVE-2014-1266, 2024.

[15] OpenSSL, *Improper Input Validation*, https://nvd.nist.gov/vuln/detail/CVE-2014-3513, 2023.

[16] S. Poznyakoff, *Gnu cflow*, https://www.gnu.org/software/cflow/, 2005.

[17] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[18] J. Ma, Z. Zhao, X. Yi, J. Chen, L. Hong, and E. H. Chi, "Modeling task relationships in multi-task learning with multi-gate mixture-of-experts," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1930–1939.

[19] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: 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.

[20] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/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.

[21] Z. Feng, D. Guo, D. Tang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[22] D. Guo, S. Ren, S. Lu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[23] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[24] X.-C. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, and Z. Gu, "When less is enough: Positive and unlabeled learning model for vulnerability detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 345–357.

[25] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 297–308.

[26] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[27] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2365–2376.

[28] C. Ni, X. Guo, Y. Zhu, X. Xu, and X. Yang, "Function-level vulnerability detection through fusing multi-modal knowledge," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 1911–1918.

[29] Z. Li, D. Zou, S. Xu, *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[30] S. M. Ghaffarian and H. R. Shahriari, "Neural software vulnerability analysis using rich intermediate graph representations of programs," *Information Sciences*, vol. 553, pp. 189–207, 2021.