Hindawi
Security and Communication Networks
Volume 2022, Article ID 1919907, 13 pages
https://doi.org/10.1155/2022/1919907

Research Article

HGVul: A Code Vulnerability Detection Method Based on Heterogeneous Source-Level Intermediate Representation

Zihua Song,[1] **Junfeng Wang**,[2] **Shengli Liu**,[3] **Zhiyang Fang**,[1] **and Kaiyuan Yang**[2]

[1]*School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China*
[2]*College of Computer Science, Sichuan University, Chengdu 610065, China*
[3]*State Key Laboratory of Mathematical Engineering and Advance Computing, Zhengzhou 450000, China*

Correspondence should be addressed to Junfeng Wang; wangjf@scu.edu.cn

Received 3 November 2021; Accepted 22 March 2022; Published 11 April 2022

Academic Editor: Gu Zhaoquan

Vulnerability detection on source code can prevent the risk of cyber-attacks as early as possible. However, lacking fine-grained analysis of the code has rendered the existing solutions still suffering from low performance; besides, the explosive growth of open-source projects has dramatically increased the complexity and diversity of the source code. This paper presents HGVul, a code vulnerability detection method based on heterogeneous intermediate representation of source code. The key of the proposed method is the fine-grained handling on heterogeneous source-level intermediate representation (SIR) without expert knowledge. It first extracts graph SIR of code with multiple syntactic-semantic information. Then, HGVul splits the SIR into different subgraphs according to various semantic relations, which are used to obtain semantic information conveyed by different types of edges. Next, a graph neural network with attention operations is deployed on each subgraph to learn representation, which captures the subtle effects from node neighbors on their representation. Finally, the learned code feature representations are utilized to perform vulnerability detection. Experiments are conducted on multiple datasets. The F1 of HGVul reaches 96.1% on the sample-balanced Big-Vul-VP dataset and 88.3% on the unbalanced Big-Vul dataset. Further experiments on actual open-source project datasets prove the better performance of HGVul.

1. Introduction

The explosive growth of open-source projects has made their code security faces severe challenges. In 2020, more than 60 million new repositories were created on GitHub, and the number of new contributions exceeded 9.1 billion [1], which led to the number of attacks against open-source projects continuing to increase. Besides, open-source is not only a key fuel for digital innovation, but also an ideal target for software supply chain attacks. Supply chain attacks on open-source projects surged by 650% in the last years, inflicting many severe damages, e.g., SolarWinds, Codecov, and Kaseya events [2]. Furthermore, software code vulnerabilities are the main foundation for launching supply chain attacks. Code vulnerabilities often act as "door openers" allowing attacks to gain lateral movement and deploy malware for the disruptive attacks. Detecting the vulnerability of source code efficiently is significant for locating software security problems as early as possible, ensuring stable operation of software systems, and securing confidential information from theft. Many classical approaches have been used to detect vulnerabilities, such as static analysis [3–5], symbolic execution [6–8], and fuzzy testing [9–11]. However, these are still inefficient in practical detection, and the false-positive and false-negative are still high due to the lack of processing the subtle syntax-semantic information of the source code. The efficiency of the source code vulnerability detection still needs to be further improved.

Finding vulnerabilities in the software code has always been a challenging task. Vulnerability pattern-based detection approaches are widely used in the industry, but they strongly rely on the artificially constructed vulnerability pattern library [4, 12, 13], which makes them unable to cope with a large

amount of the emerging open-source code. Symbolic execution [6–8] and fuzzy testing [9–11] are also commonly used vulnerability detection approaches, but the huge overhead of computing makes their detection performance low in real-world use. The data-driven approach based on machine learning (ML) provides an alternative way to identify vulnerability, which can be further divided into traditional ML-based approach and Deep Learning- (DL-) based approach. Traditional ML-based approaches are first proved to be feasible in vulnerability detection [14–16], which takes features extracted from the code as input and detects vulnerabilities based on ML algorithms. The quality of code features is critical to the approaches, but it entirely depends on the expert experience, and in addition, extracting features is a generally time-consuming and error-prone task. In contrast to the traditional ML-based approach, the DL-based approach has the stronger ability to learn vulnerability feature representations, which can automatically extract feature representations from data without manual intervention. Besides, the large amount of open-source code can provide sufficient corpus for DL, which accelerates this approach being applied to vulnerability detection task [3, 5, 17–21].

The data-driven approach provides a profitable way to detect the vulnerability, in which the key is capturing the syntax and semantic information of code. Some approaches treat the code as the flat sequence, such as function call sequence [22, 23] and the different traversal sequence based on the SIR of code [20, 24, 25], and then extracting vulnerability information based on recurrent neural networks (RNN) [20, 25, 26] or convolutional neural networks (CNN) [17, 24]. The code itself has complex structural properties, yet treating the code as just a sequence does not represent its syntax and semantic information well. It can lose code structure properties, which are often crucial for vulnerability detection. Therefore, to better capture vulnerability characteristics from the structural properties of the code, algorithms that can learn directly on the complex structure are required.

Graph neural networks (GNN) can meet such needs, and some works are already using GNN for vulnerability detection [19, 27]. Devign [19] expands the Abstract syntax tree (AST) of the function code into a graph structure and uses a variant GNN to identify vulnerability based on the expended graph structure. BGNN4VD [27] further improves the SIR of function code, which treats it as a bidirectional graph, and then uses a variant of GNN to detect vulnerability. Despite the advance, the graph-based approach still struggles to improve efficiency and performance, which is essential for real-world detection. At present, the syntactics and semantics processed by the graph-based approaches are relatively coarse-grained, and the vulnerability information hidden in the code cannot be fully utilized, thus leading to a high rate of false positives and false negatives. Capturing fine-grained syntactic-semantic information can yield more valuable vulnerability information since the vulnerable code only accounts for a very small portion of the entire function.

This paper presents HGVul, a source code-oriented vulnerability detection method based on heterogeneous source-level intermediate representation graph. It has the capability to improve detection effectiveness because HGVul can capture more subtle syntactic-semantic information. First, HGVul focuses on the function-level code with appropriate granularity [18, 19, 24, 28], because most of the vulnerabilities-related codes only involve part of the code of a single function [29]. And HGVul characterizes function source code with SIR graphs; that is, it combines code property graph and natural code sequence (CPG+), which contain abundant syntactic-semantic information. Second, HGVul treats the CPG+ as a heterogeneous graph with multiple types of edges and extracts subgraphs by different edge types. For each type of subgraph, the node feature representation is generated based on GNN with the attention mechanism as a way to catch the slight effect by different neighbors on semantics. Third, HGVul merges the corresponding node representation of each type subgraph and read out the whole graph representation as the function feature, which further captures the subtle semantic information since each type of relation conveys different semantic information. Therefore, through meticulously processing the SIR of function, HGVul can acquire more valuable information hidden in the code, which can improve the performance of vulnerability detection. The main contributions of this paper are as follows:

(i) A source code vulnerability detection framework based on heterogeneous SIR is designed to extract valuable information of function code. It provides better code information representation capability than existing methods.

(ii) A method for deriving fine-grained syntactic-semantic information of codes is proposed. It not only distinguishes different semantic information of multiple edges, but also captures the different effects of internode in SIR.

(iii) We implement the prototype and evaluate the effectiveness of HGVul on multiple datasets. The experimental results show that HGVul has better balanced performance, the F1 of HGVul is the best on balanced and unbalanced datasets as 96.1% and 88.3%, respectively, and it has the ability for detecting practical open-source projects.

The rest of this paper is organized as follows: Section 2 reviews the previous related work. The preliminaries for vulnerability detection are presented in Section 3. Section 4 introduces the details of the methodology. The experimental evaluation is given in Section 5. Finally, Section 6 concludes this paper.

2. Related Work

Vulnerability detection has been a key concern in the field of cyberspace security. Targeting software source code draws a large number of researchers' attention because it can avoid potential vulnerability security threats as early as possible. Existing source code-oriented approaches can be categorized as pattern-based matching approach, code similarity-based analyzing approach, and learning-based detection approach.

2.1. Pattern-Based Approach. This approach identifies vulnerability relying on a large vulnerable code pattern rule database. The predefined pattern database allows the approaches to quickly detect known vulnerabilities; hence, it has a widespread use by code scanners, such as RATS [30], Flawfinder [31], and Checkmarx [32]. The vulnerability-related pattern is crucial for this approach, and researchers are also exploring different methods of pattern extraction [4, 12, 13]. However, vulnerability can exhibit multiple variants, and the pattern of complex vulnerability is challenging to construct; building a sufficient comprehensive pattern dataset is a laborious and unachievable task. So, it can only detect what exists in the pattern database and cannot cope with unknown vulnerabilities. Compared with such approaches, HGVul does not need to build the pattern rule database based on expert knowledge, which can dramatically reduce labor costs; besides, it has the potential to find unknown vulnerabilities.

2.2. Similarity-Based Approach. It discovers the vulnerability based on the similarity of code. It discovers the vulnerability based on the similarity of code. Instead of using the original code directly for similarity comparison, this approach usually extracts the abstract representation of code or the corresponding semantic syntax properties for similarity analysis. Redebug [33] detects vulnerabilities by extracting basic tokens from the source code and comparing the similarity of the token sets. VUDDY [34] calculates the hash value of the string sequence and compares the hash value to achieve fast vulnerability identification. Some researchers [35, 36] extract complex metrics for calculating similarity with vulnerability code. A suitable code abstract representation or code metric is the key to this approach. Therefore, it is easily susceptible to obfuscation techniques and weak in detecting unknown vulnerabilities. HGVul has better robustness because its feature representation is learned from abundant data.

2.3. Learning-Based Approach. Learning-based approach combines ML algorithms to learn vulnerability information hidden in the code data. The early learning-based approach uses code feature as input for vulnerability prediction, e.g., different lengths of sequence code [37, 38], features from the function call sequence [16, 39]. Feature extraction is a time-consuming and error-prone work, while the DL is proven to have the ability to generate features automatically [40–43], so the DL-based approaches are gradually being applied in vulnerability detection. Russell et al. [24] form source code token extracted by a lexical parser as an image and then identify vulnerability using CNN algorithm. More researchers [17, 20, 25, 26] consider that the sequence of code contains more information, such as function call sequence and different traversal sequence of code representation, and use RNN algorithms to detect vulnerability. How to better capture the syntactic-semantic information hidden in code is the key to learning-based approach.

Because the graph-structured representation of code can well represent the syntactic-semantic properties of the code, some researchers [19, 27, 44] began to explore using GNN to detect vulnerabilities based on the SIR of source code. Zhou et al. [19] extended the code graph representation structure of the code based on AST and used a variant GGNN network to implement vulnerability detection. Wu et al. [44] extract simplified Code property graph (CPG) from code and then use GNN to identify vulnerabilities. Cao et al. [27] combine the AST, Control flow graph (CFG), and Data flow graph (DFG) of the code into Code Composite Graph (CCG). The authors believe that the valuable backpropagation information on CCG is also worth tackling, and they employ GNN to learn the representation of vulnerabilities. Compared with the existing GNN-based approach, HGVul not only distinguishes the heterogeneous features of SIR, but also applies attention mechanisms in each semantic information subgraph to obtain fine-grained code semantic information, which in turn improves the efficiency of vulnerability detection.

3. Preliminaries for Vulnerability Detection on SIR

3.1. Problem Formulation. The goal of the proposed method is to determine whether the function-level code is vulnerable or not. The sample of data is represented as $\{(f, y) | f \in F, y \in Y\}$, where $F = \{f_1, f_2, f_3, \ldots f_n\}$ is a series of functions, $Y = \{0, 1\}^n$ is the set of corresponding labels in which 0 denotes the not vulnerable and 1 otherwise, and n is the number of samples, so the target of HGVul is to find the optimal mapping φ: $F \Rightarrow Y$. We extract the graph-based SIR of function, which can be formulated as follows:

Definition 1. **(Function)** A function can be symbolized by its SIR as $f = g(V, E, A)$, where V is the set of nodes, E is the set of edges, and A is the set of all node attributes.

In particular, the SIR used in this paper is extracted based on multiple semantic information, which we regard as a directed heterogeneous graph with multiple edge types. The heterogeneous graph can be formally described as follows:

Definition 2. **(Heterogeneous Graph)** The heterogeneous graph can be represented as $G = (V, E), V \in O, E \in R$, V is the node set, E is the edge set, and O and R denote the set of all node types and the set of all edge types, where $|O| + |R| > 2$. Specifically, the SIR in this paper has multiple types of edges, i.e., $|R| \geq 2$.

Therefore, it searches the optimal mapping by minimizing the loss function and can be defined as follows:

$$\min \sum_{i=1}^{n} \ell\left(\varphi\left(g_i(V, E, A), y_i | g_i\right)\right) + \lambda\omega(\varphi). \tag{1}$$

where $\ell(\cdot)$ is the cross entropy loss function, λ is the adaptive weight, and $\omega(\cdot)$ is a regularization.

3.2. Source-Level Intermediate Representation of Code

3.2.1. Abstract Syntax Tree (AST). AST is an ordered tree representation of the abstract syntactic of code. Each node in

the AST represents the smallest lexical unit, and each edge of AST denotes the parent-child relationship between nodes.

3.2.2. Control Flow Graph (CFG). CFG is a graph representation of code, which accounts for all possible paths during its execution [19]. The nodes of a CFG represent basic blocks that can be statements or conditions. The edges of CFG indicate the transfer of control through directed connections.

3.2.3. Program Dependence Graph (PDG). PDG is a program representation that makes data dependencies and controls dependencies explicitly [45]. It comprises two types of relationships: data dependency (DD) and control dependency (CD). The edges of data dependency are used to represent the relevant data flow relationship. The control-dependent edges are utilized to denote the essential control flow relationship.

3.2.4. Code Property Graph (CPG). CPG merges the AST, CFG, and PDG into a single joint data structure [12]. The node of CPG is the same as AST, and the edge of CPG is combined with other SIRs. So, the CPG is a heterogeneous graph that contains multitype edges.

3.2.5. Natural Code Sequence (NCS). NCS connects all token nodes of AST by the natural sequential order of the source code. It reflects the programming logic of the function from the order in which the code appears in the function code. The nodes of NCS are the leaf node of AST, and the edges of NCS connect them according to the natural sequential order.

Besides, there are various extended forms of basic SIR, e.g., the SIR combining AST and NCS (we call it AST+ for convenience), the SIR integrating CPG and NCS (called CPG+ for convenience). This paper chooses CPG+ as the SIR of the function code because it contains more syntactic-semantic information. A visual example of CPG+ is shown in Figure 1.

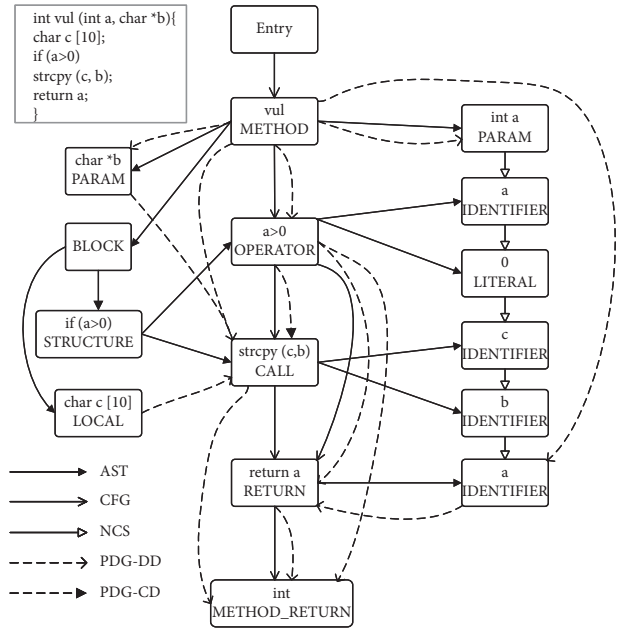

Figure 1: The CPG+ of an example function code.

4. Methodology

4.1. Overview of HGVul. The overall framework of the method is shown in Figure 2, including three major processes: Preparing SIR, Learning Representation, and Detecting Vulnerability. Preparing SIR collects function code from open-source project, then extracts the SIR of code at the function-level granularity, and initializes the primary features of each node in SIR. Learning Representation takes the graph structure corresponding to the SIR of function as input and outputs the feature representation of the function. It starts by using GNN to update node representations based on the different edge-typed subgraphs, which can distinguish semantic information from different types of edges, then merges it, and reads it out as the function representation. While updating the node representation, the attention operation is employed for each node to separate the influence of different neighbors. Detecting vulnerability takes the function representation as input. It trains the detection model in the training state and uses the trained model in the detection state to determine whether the function is vulnerable.

4.2. Preparing SIR of Function. This process mainly transforms the original code into a graph structure with node attributes. It includes two steps:

4.2.1. Extracting SIR of Function. For function-level code, this paper treats the entire function as a basic processing unit and extracts its corresponding SIR as the handle object. Specifically, HGVul takes CPG as the prototype and combines it with NCS to constitute a more comprehensive graph representation of code (called CPG+). CPG+ contains a variety of edge types with abundant syntactic-semantic information.

4.2.2. Embedding the Code Statement as Node Initial Representation. This step is to transform the code of the nodes into quantifiable vectors and use it as the initial features of the nodes. Firstly, HGVul uses a lexical analyzer to obtain the basic tokens in the node code. Then, the function and variable names in tokens are mapped to symbolic names (e.g., "FUN," "VAR") to prevent them from interfering with the initial feature of the node, because the user-defined function and variable names contain program-specific naming characteristics. Next, HGVul uses a pre-trained word2vec model to obtain the primary embedding of each node. For the presence of multiple tokens in the node code, the average of each dimension of the multiple token vectors is calculated to form a new vector as the node primary embedding. The corpus of the pre-rained word embedding model consists of the mapped tokens of all the

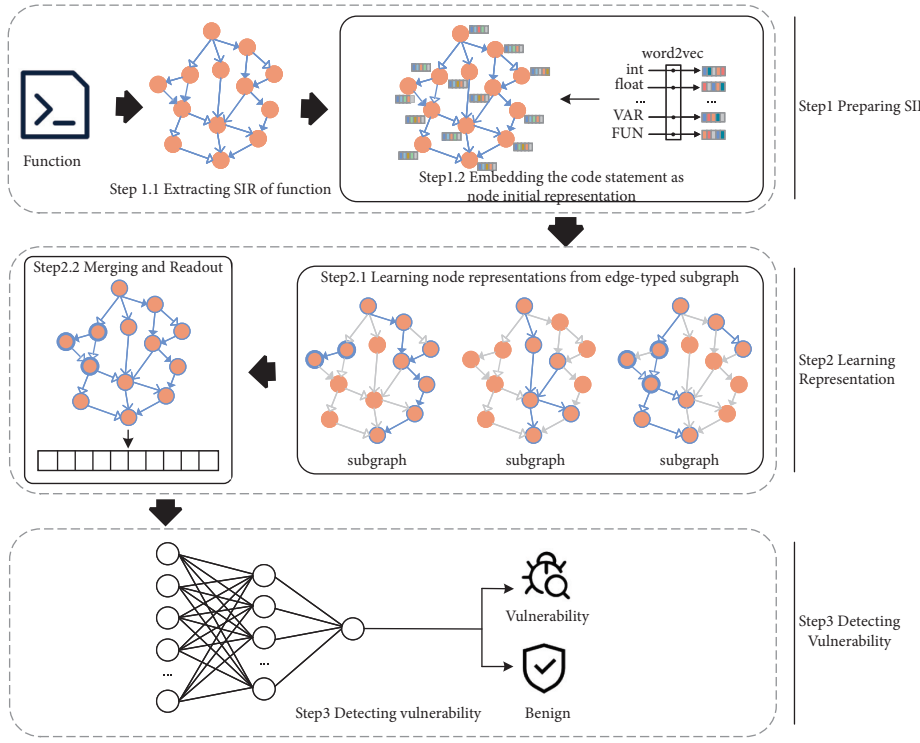

Figure 2: Overview of HGVul.

training samples, and the dimension of the token is set as 100. Finally, to capture the feature type hiding information of the nodes, we encode each type as an integer and concatenate the encoding of node types and the obtained node embedding as the feature representation of the node.

4.3. Learning Representation. The process is to acquire feature representation of the function by taking function-level SIR with node features as input. There are the step of node representation updating and the function representation generating step.

4.3.1. Learning Node Representation from Edge-Typed Subgraph. In this step, the node aggregates the neighbor information along the edges in the SIR and updates its own feature representation with it. HGVul extracts subgraphs according to different edge types and then performs the node learning process on each subgraph separately. Hence, the SIR of a function is represented as $g = \cup_{r\in R} g^r, r$ denotes the edge type. The initial representation of node v_i in subgraph g^r is set as $h_{i,r}^0$. So, the representation of node v_i at t state is $h_{i,r}^t$, and $h_{i,r}^{t+1}$ represents it at $t+1$ state, which aggregates along the edge in the subgraph g^r.

$$h_{i,r}^{t+1} = \text{aggregator}\left(h_{j,r}^t, \forall v_j \in N_{i,r}\right), \tag{2}$$

where $N_{i,r}$ denotes the neighbors of node v_i in the subgraph g^r.

When updating the feature representation of nodes in each subgraph, this paper employs the attention operation to distinguish the impact of neighbors. Firstly, the correlation coefficient e between the nodes and their direct neighbor in g^r is calculated. For a specific node v_i, the correlation coefficient e_{ij}^r with neighbor v_j is calculated as

$$e_{ij}^r = m\left(\left[Wh_{i,r}^t \| Wh_{j,r}^t\right]\right), j \in N_{i,r}, \tag{3}$$

where W is a shared parameter; it increases the embedding dimensionality for generating enhanced node representation. The operation $[\cdot\|\cdot]$ is to concatenate the transformed features of v_i and v_j. The $m(\cdot)$ maps high-dimensional embedding to an actual number. Then, calculate the attention coefficient $a_{i.}$ of each neighbor node relative to v_i.

$$a_{ij}^r = \text{softmax}\left(e_{ij}^r\right) = \frac{\exp\left(\sigma\left(e_{ij}^r\right)\right)}{\sum_{k\in N_{i,r}} \exp\left(\sigma\left(e_{ij}^r\right)\right)}, \tag{4}$$

where σ denotes the activation function. After obtaining the attention coefficients, the linear transformation is performed on the node initial representation and then updates the node representation by combing the attention coefficients. In fact, we adopt the multihead scheme to ensure the stability of the attention operation.

$$h_{i,r}^{t+1} = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in N_{i,r}} a_{ij}^{rk} W^k h_{j,r}^t\right), \tag{5}$$

where a_{ij}^{rk} denotes the k-th head of a_{ij}^r. W^k corresponds to the k-th head of W.

In addition, for extending the receptive field of nodes learning neighboring features, HGVul updates the node feature representation by repeating steps (3)–(5) to aggregate the information from multistep neighbors of the nodes.

And there are the training state and the detection state in this step. In the detection state, the GNN model trained in the training state is directly used to obtain the node feature representation.

4.3.2. Merging and Readout Graph Representation as Function Feature Representation. The feature representation of the function is generated in this step through reading out the nodes feature on SIR. Because the node representation is learning on different edge-typed subgraphs, it is necessary to merge the representation on an entire graph. Common merge operations include average, maximum or minimum, summation, and concatenate; this paper chooses to average.

$$h_i' = \frac{1}{|R|} \sum_{r \in R} h_{i,r}, \tag{6}$$

where h_i' represents the updated feature representation of node v_i by aggregating features from neighbors with different edge types.

Then, read out the feature representation of function from the whole SIR since each node of the SIR represents a basic block with syntactic and semantic information. Here, HGVul reads out the feature of function by averaging each node feature in the SIR.

$$H = \frac{1}{|V|} \sum_{i \in V} h_i'. \tag{7}$$

H represents the feature representation of the sample function.

4.4. Detecting Vulnerability. This process performs graph-level classification to determine whether a function is vulnerable. It takes the feature representation of the function as input and trains a classifier for output whether the function is vulnerable or not. The classifier employs a linear transformation on the function feature representation to extract function-level abstract features further. The proposed method uses multilayer perception (MLP) to further extract the function-level features and choose the sigmoid function for classification.

$$\overline{y} = \text{Sigmod}(\text{MLP}(H)), \tag{8}$$

where $\overline{y}$ is the final detection result, and H is the feature representation of the function.

Similarly, it includes both training and detection states. The classifier is training at the training state, and it is used directly at the detection state.

5. Evaluation

5.1. Experimental Setup

5.1.1. Datasets. Obtaining enough high-quality function samples with vulnerability labels is essential for both training and evaluating the model, but it is never trivial. This paper collected 3 different datasets for validating the model performance, evaluating its efficiency on the actual project, and testing the ability to detect the functions corresponding to CVEs.

Dataset I: this paper trains and evaluates the models based on the Big-Vul dataset [46], which has a lot of sample functions with vulnerability labels and can be used publicly in entirety. Since the distribution of positive and negative samples is very uneven in the dataset, this paper extracts two datasets with balanced and unbalanced samples on Big-Vul for better validating HGVul. The balanced dataset is composed of the vulnerability function and its corresponding patch function called Big-Vul-VP. The unbalanced dataset is the original Big-Vul dataset. The experiment uses Joern [47] to extract the basic SIR of function, and HGVul uses only the samples that can be handled right by Joern. For convenience, we refer to the two datasets as Dataset I, whose details are shown in Table 1.

Table 1 lists the number of positive (Vulnerable) and negative (non-Vulnerable) samples in the two datasets. In the experiments, we performed the 5-fold cross-validation to conduct the experiments in Big-Vul-VP. For the larger Big-Vul dataset, it divided the dataset into the training set, validation set, and test set in the ratio of 2 : 1 : 1.

Dataset II: to test the actual detection effect of the model, this paper extracted the actual test functions from 6 open-source projects based on the D2A dataset [48], which include ffmpeg, openssl, libav, httpd, nginx, and libtiff. Each function extracted from the D2A dataset has a "touched_by_commit" flag, so it is regarded as vulnerable when its flag is set to "true" and regards the correspondingly repaired function as not vulnerable. In particular, the D2A not only contains multiple versions of code for each project, but also produces inter-procedural analysis, so that one vulnerability may contain multiple vulnerable functions. We strictly removed the duplicate functions and rigorously confirmed the number of vulnerable functions. Again, only samples that Joern could handle were used in the experiment. Specific information about the data in each project is shown in Table 2.

Dataset III: in addition, to further test the ability of the HGVul to detect the functions corresponding to CVEs and explore whether it has the potential to detect unknown vulnerable functions, we manually scraped the latest 10 open CVEs of the six projects from CVE Details [49], so it obtains 60 CVEs containing 73 vulnerable functions. It should be noted that some projects do not disclose the details of the latest CVE such as httpd, and we try to collect the latest public vulnerability functions as much as possible, but there are still some outdated vulnerabilities. The used CVEs of dataset III are shown in Table 3.

5.1.2. Baselines. We compared HGVul against 6 different approaches that cover vulnerability analysis tools, similarity-based approach, sequence learning-based approach, and graph learning-based approach: (1) RAT, a well-known static analyzer [30]; (2) Flawfinder, a widely utilized vulnerability analyzer [31]; (3) VUDDY, a similarity-based approach [34]; (4) VulDeePecker, a sequence learning-based approach [20]; (5) BGNN4VD, a variant graph learning-based approach [27]; (6) Devign, a graph learning-based approach [19].

Table 1: The details of Dataset I.

Datasets	# Positive funcs	# Negative funcs	# Total
Big-Vul-VP	10207	9288	19495
Big-Vul	10207	166618	176825

Table 2: The details of dataset II.

Project	# Positive funcs	# Negative funcs	Total
Ffmpeg	1583	1476	3059
Openssl	1075	897	1972
Libav	801	719	1520
Httpd	105	85	190
Nginx	78	72	150
libtiff	54	47	101

Table 3: The details of dataset III.

Project	CVE
ffmpeg	CVE-2021-33815, CVE-2021-30123, CVE-2020-35965, CVE-2020-24020, CVE-2020-22054, CVE-2020-22051, CVE-2020-22049, CVE-2020-22029, CVE-2020-22026, CVE-2020-22020
openssl	CVE-2021-23841, CVE-2021-23840, CVE-2021-3450, CVE-2021-3449, CVE-2020-1971, CVE-2020-1967, CVE-2019-1563, CVE-2019-1559, CVE-2019-1547, CVE-2018-0737
libav	CVE-2017-16803, CVE-2017-9051, CVE-2016-8676, CVE-2016-8675, CVE-2016-7499, CVE-2016-7424, CVE-2016-7393, CVE-2016-6832, CVE-2016-3062, CVE-2015-5479
httpd	CVE-2017-9798, CVE-2016-8740, CVE-2016-4979, CVE-2015-3185, CVE-2015-3183, CVE-2015-0253, CVE-2015-0228, CVE-2014-8109, CVE-2012-0031, CVE-2012-0021
nginx	CVE-2021-23017, CVE-2019-20372, CVE-2019-11839, CVE-2019-11837, CVE-2017-20005, CVE-2017-7529, CVE-2016-4450, CVE-2014-3556, CVE-2014-0088, CVE-2013-2070
libtiff	CVE-2020-35524, CVE-2020-35523, CVE-2019-17546, CVE-2019-7663, CVE-2019-6128, CVE-2018-18557, CVE-2018-17101, CVE-2018-17100, CVE-2018-8905, CVE-2018-7456

5.1.3. Evaluation Metrics and Implementation. This paper uses 6 widely used metrics to evaluate the performance of the HGVul, including Accuracy (ACC), Precision (P), Recall, F1-measure (F1), False positive rate (FPR), and False negative rate (FNR).

This paper chose the open-source tool Joern [47] to construct the basic SIR of the function. DGL [50] v0.6 package is using to store and deal with the graph-based data. The GNN-based vulnerability detection model is implemented by using Pytorch [51] v1.8.1. All experiments are performed on a multicore server with a 20-core 2.2 GHz Intel Xeon CPU and an Nvidia Tesla V100 GPU.

5.2. Experimental Results

5.2.1. Comparing with the Different Approaches. In this experiment, we applied the 7 different methods on dataset I for comparing the efficiency of HGVul and the others. Moreover, to observe the stability of approaches, the experiment performed the 5-fold cross-validation on balanced Big-Vul-VP and averaged the final result. For the unbalanced Big-Vul dataset, we conducted 10 independent experiments, which shuffled and divided the training/validation/test set into 2 : 1 : 1 before each experiment, and exhibited the average value with max-min bars. Figure 3 reports the experimental results.

Figure 3 exhibits the evaluation results of the seven methods on the two datasets, respectively. It can make the following observations. First, on both the Big-Vul-VP and Big-Vul, the performance of RATs and Flawfinder is worse whose Recall and F1 are both lower than 25%. It still has a high FNR and FPR due to the limitations of the vulnerability pattern dataset, while HGVul does not suffer from the limitations and is therefore significantly better than such approaches. Secondly, VUDDY has the highest precision and the lowest FPR. But it has the highest FNR, which is caused by the nature of the clone-based method based on code similarity. VUDDY is unable to cope with changing forms of exploit code even with slight changes, while HGVul has the ability to work with variant code. Thirdly, the sequence-based approaches have better performance than vulnerability pattern-based approaches because they combine DL techniques to extract more complex information. The FNR of these approaches is obviously lower; especially the FNR of VulDeePecker is only 14.3% on Big-Vul-VP. But, it can not suppress the FPR well because it fails to effectively exploit the semantic information in the code. Lastly, the graph-based approaches have better performance than the others, and the better F1 value indicates that BGNN4VD, Devign, and HGVul all have a more balanced detection effect. In the three GNN-based approaches, HGVul still has the best performance. The FNR and FPR of HGVul are both below 5% on the Big-Vul-VP dataset, and the FNR is still the

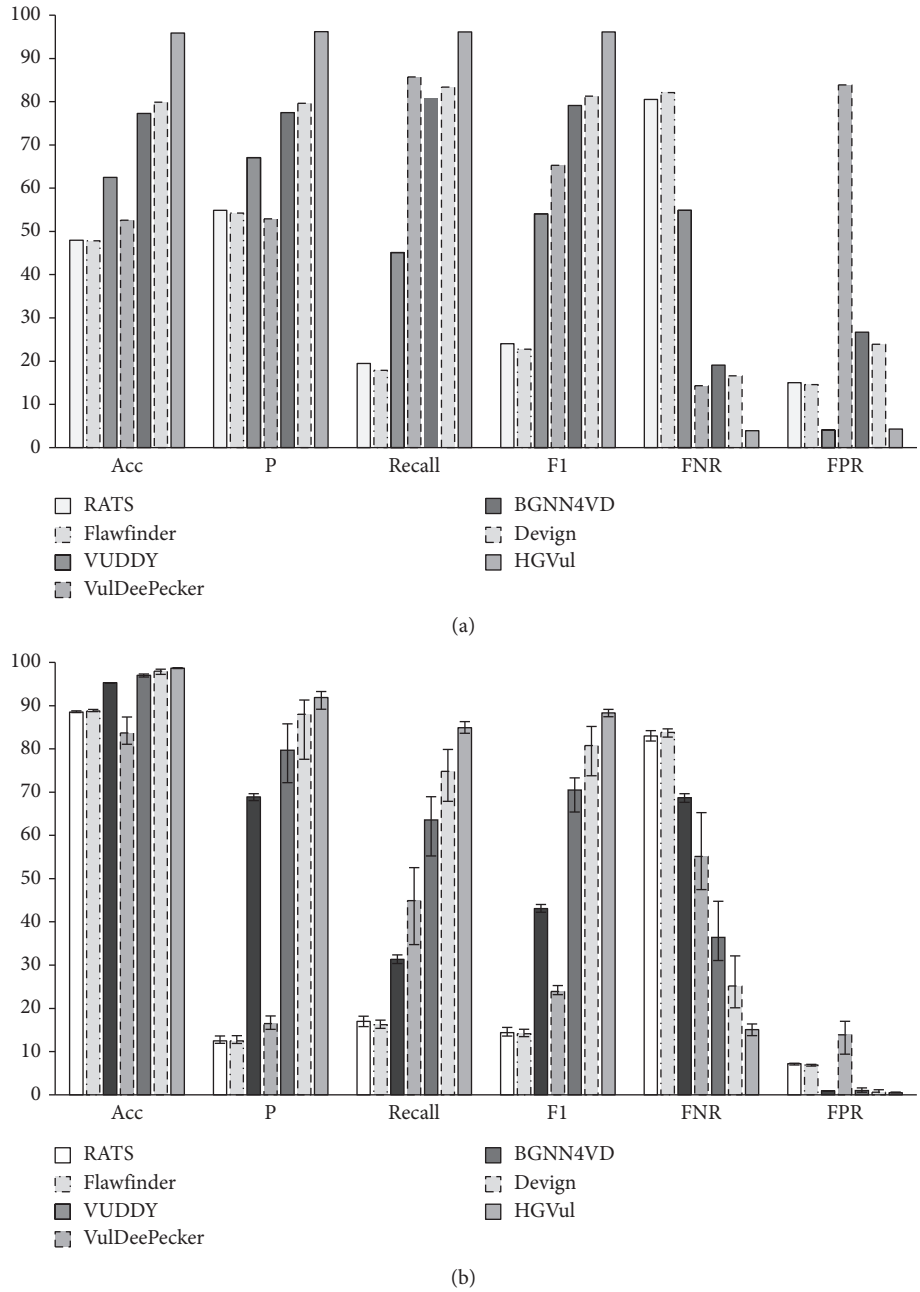

Figure 3: Performance comparison of the approaches. (a) Performance comparison of the approaches on Big-Vul-VP (%). (b) Performance comparison of the approaches on Big-Vul (%).

best 15.1% on the Big-Vul with extremely unbalanced samples. Compared with BGNN4VD and Devign, HGVul uses heterogeneous GNN to collect different semantic information and applies attention mechanism to obtain subtle code information in each semantic subgraph, while the other two approaches generate code representation on the raw graph without being able to better distinguish the subtle heterogeneous features of the code. Therefore, it can get the following finding that HGVul is more effective than the state-of-the-art vulnerability detection methods.

5.2.2. Performance of the Different SIR. This experiment tested separately on different SIRs to compare the vary influence on the effectiveness of vulnerability detection, including AST, CFG, PDG, CPG, AST+, and CPG+. We chose the gated graph neural network (GGNN) [52] to generate functional feature representations, which do not have attention operations that can affect the results. The experiment was also performed on dataset I and performed the ablation setting to reduce the influence from other factors. In other words, the settings of the network were the

same except that the SIR of the inputs was different. The experiment results are shown in Tables 4 and 5.

Tables 4 and 5 list the detection results by using different SIR as input. Each SIR of the code induced different detection performance on both Big-Vul-VP and Big-Vul. The detection results for the experiments based on CPG+ are better than those using the other SIR as input. On the Big-Vul-VP, the Accuracy, Precision, Recall, and F1 were higher than 92% when using CPG+ for detection. And the FNR and FPR were relatively low, with FPR being the best compared to the others at 8.4%. Recall and FNR were the best when using AST+ as input with 92.9% and 7.1%, respectively. On the Big-Vul, although the detection result is decreased due to the large bias of the samples, the performance is still maintained at a good level when using CPG+ as SIR. Its Accuracy is 98.2%, Precision is best 89.0%, F1 is 83.4%, and its FPR is only 0.6%, which is significantly better than those of the method using the other SIR as input. When using CPG as input, its Recall and FNR reach the best 81.3% and 18.7%, respectively. Therefore, it can make a conclusion that the detection performance is greatly influenced by the different SIR of code. And the vulnerability detection performance is better when using CPG+ as input.

5.2.3. Different Influence of Internode in SIR. This experiment is to verify that the node representation is differentially affected by neighbors on SIR and prove the positive impact of the attention operation on the vulnerability detection. We controlled different GNNs for the ablation experiment, so all other experimental settings were the same. The three networks GCN [53], GGNN, and GAT [54] were chosen for the experiments, where GAT contains attention mechanism. Specifically, the experiment focused only on the differences in CPG+. The detection results are listed in Tables 6 and 7.

Tables 6 and 7 show the detection performance of different methods in which learning node representation is based on different GNN. It exhibits better results on both Big-Vul-VP and Big-Vul when considering the different influence of internode for node representation. On the Big-Vul-VP, the detection result based on GAT is better than that of the method of learning node feature representation using GCN or GGNN, its F1 is the best 94.2%, the Recall is 93.9%, and the FNR and FPR are also obviously lower than those of the other two methods, which are both 6.1%. On the Big-Vul, it is also obviously better than other methods when considering the nodes to be influenced differently by different neighboring nodes of SIR. Regarding the Acc and P of the method in which learning node representation based on GAT is higher than 90%, its Recall and F1 are also best at 80.8% and 85.2%, respectively. FNR and FPR also remain low, with its FPR being only 0.5%. Therefore, the experimental results can prove that capturing the different impacts of node representation from the different neighbors in the SIR can enhance the vulnerability characterization of node features, which can improve the performance of vulnerability detection.

5.2.4. Improvement of Heterogeneous SIR. This experiment is performed to examine whether it has improved on detection that treats SIR of function as a heterogeneous graph with multiple types of edges. We compared the effect of treating SIR as a heterogeneous and homogeneous graph. To reduce the influence of other factors on the results, it still only controlled the graph neural network part and chose GAT as a comparison. Besides, the experiment was performed on only two types of graph AST+ and CPG+, because these two SIRs contain different types of edges with more detailed semantic information. The comparison results of the experiment are displayed in Tables 8 and 9.

Tables 8 and 9 list the experiment results of whether paing attention to the heterogeneous edge delivers different information in SIR. Compared to the methods that combine only attention mechanisms, the performance is better than that of the method that can capture the heterogeneous nature of the edges in SIR on both Big-Vul-VP and Big-Vul. On the Big-Vul-VP dataset, it can be directly found that the two SIRs (AST+, CPG+) show higher detection performance in Acc, P, Recall, and F1 when considering different types of edges conveying different information; correspondingly, FNR and FPR are obviously lower. The best is the method that treats CPG+ as a heterogeneous graph with multiple types of edge. Its FNR and FPR are below 5%. On the Big-Vul, the method is based on the heterogeneous graph, with multiple edge types still having better detection performance, and the method that processes CPG+ as a heterogeneous graph can obtain the best detection effect. Compared with the method using CPG+ as input and updating node representation based only on GAT, its detection effect is obviously better, with Recall and F1 being 84.9% and 88.3%, respectively. Therefore, we can conclude that considering the different information delivered by different types of edges can obtain more subtle code information. Thereby, it can enhance the representation of function features and improve the performance of vulnerability detection.

5.2.5. Performance on the Open-Source Projects. We applied the trained models on 6 open-source projects to examine its ability of actual function-level vulnerability detection. This experiment used dataset II, whose functions are grouped by project. The functions of each project are input into the trained model for detection. Table 10 shows the details of the detection results.

Table 10 lists the detection results of seven methods on six practical open-source projects. The experimental results show that the proposed method can detect most of the function-level vulnerabilities, which is better than the other methods. HGVul can detect 3004 vulnerable functions in a total of 3696 samples with vulnerabilities, and the average F1 can reach 69.7%. Moreover, we find that most of the undetected vulnerable function samples are integer overflow type vulnerabilities, and the proposed method shows poor detection performance for this type of

Table 4: Performance of different IR. Performance of different IR on Big-Vul-VP (%).

IR	Acc	*P*	Recall	F1	FNR	FPR
AST	90.8	90	92	91.4	8	10.6
AST+	90.2	89.5	**92.9**	91.1	**7.1**	12.9
CFG	87.4	86	91.3	88.5	8.7	16.9
PDG	81	80.2	86.6	82.9	13.4	25.1
CPG	87.1	85.7	90.4	88	9.6	16.6
CPG+	**92.3**	**92.4**	92.8	**92.6**	7.2	**8.4**

Table 5: Performance of different IR. Performance of different IR on Big-Vul (%).

IR	Acc	*P*	Recall	F1	FNR	FPR
AST	97.2	80.5	67.2	73.3	32.8	1
AST+	97.6	80.3	77	78.6	23	1.1
CFG	97.4	82.7	69.5	75.5	30.5	0.9
PDG	96.5	72	62.9	67.1	37.1	1.5
CPG	98.1	84	**81.3**	82.6	**18.7**	0.9
CPG+	**98.2**	**89**	78.4	**83.4**	21.6	**0.6**

Table 6: Performance of different GNN. Performance of different GNN on Big-Vul-VP (%).

GNN	Acc	*P*	Recall	F1	FNR	FPR
GCN	91.6	91.7	92.3	92.0	7.7	9.1
GGNN	92.3	92.4	92.8	92.6	7.2	8.4
GAT	**93.9**	**94.4**	**93.9**	**94.2**	**6.1**	**6.1**

Table 7: Performance of different GNN. Performance of different GNN on Big-Vul (%).

GNN	Acc	*P*	Recall	F1	FNR	FPR
GCN	95.3	65.4	38.4	48.4	61.6	1.2
GGNN	98.2	89	78.4	83.4	21.6	0.6
GAT	**98.4**	**90.2**	**80.8**	**85.2**	**19.2**	**0.5**

Table 8: Performance of the heterogeneous IR. Performance of the heterogeneous IR on Big-Vul-VP (%).

Method	Acc	*P*	Recall	F1	FNR	FPR
GAT-AST+	91.8	92.3	92.1	92.2	7.9	8.4
GAT-CPG+	93.9	94.4	93.9	94.2	6.1	6.1
HG-AST+	94.2	94	95	94.5	5.0	6.7
HG-CPG+ (ours)	**95.9**	**96.2**	**96.1**	**96.1**	**3.9**	**4.3**

Table 9: Performance of the heterogeneous IR. Performance of the heterogeneous IR on Big-Vul (%).

Method	Acc	*P*	Recall	F1	FNR	FPR
GAT-AST+	98.1	87.5	76.8	81.8	23.2	0.7
GAT-CPG+	98.4	90.2	80.8	85.2	19.2	**0.5**
HG-AST+	98.2	87.6	80.4	83.9	19.6	0.7
HG-CPG+ (ours)	**98.7**	**92.9**	**84.9**	**88.3**	**15.1**	**0.5**

vulnerabilities. The possible reason for this situation is that integer overflow vulnerabilities are closely related to the type of variables and rely on runtime input characteristics, which makes it hard to be detected by graph-based model. So, it can get the finding that the proposed method has the feasibility to detect the practical vulnerable functions.

5.2.6. Performance for the Functions of Actual CVEs. In addition, we further explored the detection capability of the proposed method for the vulnerable functions of actual CVEs. In this experiment, the trained models were applied on dataset III containing some functions of the new CVEs\enleadertwodots. Detection results are shown in Table 11.

Table 10: Performance on six open-source projects.

Project	# Vulnerable funcs	RATS		Flawfinder		VUDDY		VulDeePecker		BGNN4VD		Devign		HGVul	
		# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)
ffmpeg	1583	364	31.8	350	30.9	47	5.7	827	52.3	**1270**	**62.1**	928	53.3	1148	60.6
openssl	1075	526	51.7	383	43.4	64	10.7	609	56.2	838	63.9	826	65.3	**1048**	**69.7**
libav	801	155	28.2	158	28.6	22	5.3	424	53.2	**672**	64.9	602	**71.2**	597	58.2
httpd	105	38	43.2	38	43.4	2	3.7	50	52.4	88	67.2	90	67.2	**104**	**82.2**
nginx	78	0	0	11	22.2	0	0	41	53.2	66	64.4	**67**	78.4	63	**86.9**
libtiff	54	4	12.9	4	12.9	4	12.7	21	44.7	43	66.2	38	**66.7**	**44**	60.7
Total/ Avg	3696	1087	28.0	944	30.2	98	6.4	1972	50.0	2977	64.8	2551	67.0	**3004**	**69.7**

Table 11: Performance for the vulnerable functions of actual CVEs.

Project	# Vulnerable funcs	RATS		Flawfinder		VUDDY		VulDeePecker		BGNN4VD		Devign		HGVul	
		# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)	# Detected	F1 (%)
ffmpeg	11	1	15.4	1	15.4	0	0	6	52.2	8	61.5	7	51.9	**9**	**66.7**
openssl	16	6	42.9	4	33.3	8	**66.7**	7	43.8	**14**	**66.7**	**14**	63.6	12	57.1
libav	10	2	28.6	2	28.6	6	**75.0**	5	50.0	5	43.5	8	61.5	**7**	56.0
httpd	13	4	38.1	4	38.1	10	**87.0**	6	46.2	10	58.8	9	58.1	**12**	66.7
nginx	10	2	28.6	2	28.6	1	18.2	8	**69.6**	**10**	69.0	8	64.0	9	64.3
libtiff	13	1	13.3	1	13.3	5	55.6	4	42.1	10	58.8	**11**	64.7	**11**	**68.8**
Total/ avg	73	16	27.8	14	26.2	30	50.4	36	50.7	56	58.8	54	59.0	**60**	**63.3**

Table 11 lists the detection results of the methods for the latest 10 CVEs vulnerability functions. Limited by the scale of the vulnerability pattern library, the detection of RATs and Flawfinder is less effective. VUDDY has a better detection effect on openSSL, libav, and httpd because the latest version of VUDDY is updated with the newest vulnerabilities in these projects. However, emerging vulnerabilities on ffmpeg are not updated to VUDDY that results in a dramatic decrease in its detection ability, which indicates that VUDDY relies heavily on the clone template database and its severe detection delay. BGNN4VD and Devign can detect more vulnerable functions than VulDeePecker since they obtain the function code features from the source-level graph structure representation. Among the 73 vulnerable functions, HGVul can identify 60 functions with threats, and the average F1 of 6 projects can achieve 63.3%; it is better than the other methods because the fine-grained ones are handled on function code. Thus, it indicates that HGVul still performs better for detecting vulnerable functions of actual CVEs.

6. Conclusion

This paper presents the HGVul, a novel function-level source code-oriented vulnerability detection method based on heterogeneous SIR. To cope with the increasing complexity and diversity of code caused by the surge of open-source projects, HGVul fine-grained processes the SIR of function code. It captures the syntax and semantic information implied by the code from different types of subgraphs. A set of experiments shows that HGVul outperforms 6 existing methods with significantly improving both FNR and FPR. In the future, we will improve our study in many ways, including further enhancing vulnerability detection, extending the scope of vulnerability detection, and providing interpretable vulnerability detection models.

Data Availability

The data sets used in this paper are public, free, and available at https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset, https://github.com/IBM/D2A, and https://www.cvedetails.com/.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work was supported in part by the National Key Research and Development Program under Grant 2019QY1400; in part by the National Natural Science Foundation of China under Grant U2133208; in part by the Sichuan Youth Science and Technology Innovation Team under Grant 2022JDTD0014; and in part by the Basic Research Program of China under Grant 2020-JCJQ-ZD-021.

References

[1] GitHub, "The 2020 state of the octoverse," 2021, https://octoverse.github.com/.

[2] Sonatype, "State of the Software Supply Chain," 2021, https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021.

[3] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 499–510, Association for Computing Machinery, Berlin Germany, November 2013.

[4] X. Du, B. Chen, Y. Li et al., "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 60–71, IEEE Press, Montreal, Canada, May 2019.

[5] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and S. Fu, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 2017 IEEE/ACM Thirty Nineth International Conference on Software Engineering (ICSE)*, pp. 462–472, IEEE Press, Buenos Aires, Argentina, May 2017.

[6] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 725–741, IEEE, San Jose, CA, USA, May 2015.

[7] N. Stephens, J. Grosen, C. Salls et al., "Driller: augmenting fuzzing through selective symbolic execution," *NDSS*, vol. 16, pp. 1–16, 2016.

[8] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: correctness checking for real code," in *Proceedings of the Twenty Fourth USENIX Security Symposium (USENIX Security 15)*, pp. 49–64, USENIX Association, Washington, D. C. USA, August 2015.

[9] H. Chen, Y. Xue, Y. Li et al., "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2095–2108, Association for Computing Machinery, Toronto Canada, October 2018.

[10] Y. Li, Y. Xue, H. Chen et al., "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 Twenty Seventh ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 533–544, Association for Computing Machinery, Tallinn Estonia, August 2019.

[11] H. Wang, X. Xie, Yi Li et al., "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 999–1010, IEEE, Seoul, Republic of Korea, July 2020.

[12] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, IEEE, Berkeley, CA, USA, May 2014.

[13] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pp. 797–812, IEEE, San Jose, CA, USA, May 2015.

[14] M. G. Seyed and H. Reza Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey," *ACM Computing Surveys*, vol. 50, no. 4, pp. 1–36, 2017.

[15] Y. Pang, X. Xue, and A. S. Namin, "Predicting vulnerable software components through n-gram analysis and statistical feature selection," in *P2015 IEEE Fourteenth International Conference on Machine Learning and Applications (ICMLA)*, pp. 543–548, IEEE, Miami, FL, USA, December 2015.

[16] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of the Fifth USENIX conference on Offensive technologies*, p. 13, USENIX Association, San Francisco, CA, USA, August 2011.

[17] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 67–85, 2021.

[18] G. Lin, J. Zhang, W. Luo et al., "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," 2019, https://arxiv.org/abs/1909.03496.

[20] Z. Li, D. Zou, S. Xu et al., "Vuldeepecker: A Deep Learning-Based System for Vulnerability Detection," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, February 2018.

[21] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[22] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 1298–1302, IEEE, Chengdu, China, December 2017.

[23] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 85–96, New Orleans, LA, USA, March 2016.

[24] R. Russell, L. Kim, L. Hamilton et al., "Automated vulnerability detection in source code using deep representation learning," in *Proceedings of the 2018 Seventeenth IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762, IEEE, Orlando, FL, USA, December 2018.

[25] G. Lin, J. Zhang, W. Luo et al., "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2019.

[26] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "vuldeepecker: a deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.

[27] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, Article ID 106576, 2021.

[28] H. Wang, G. Ye, Z. Tang et al., "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[29] L. Cui, H. Zhiyu, J. Yang, H. Fei, and X. Yun, "Vuldetector: detecting vulnerabilities using weighted feature graph comparison," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2004–2017, 2020.

[30] S. Software, "Rough Audit Tool for Security," 2021, https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[31] D. A. Wheeler, "Flawfinder," 2021, https://www.dwheeler.com/flawfinder/.
[32] Checkmarx, "Checkmarx," 2021, https://www.checkmarx.com/.
[33] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, San Francisco, CA, USA, May 2012.
[34] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: a scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, IEEE, San Jose, CA, USA, May 2017.
[35] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 72–83, Paderborn, Germany, September 2017.
[36] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: a neglected metric in effort-aware just-in-time defect prediction," in *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 11–19, IEEE, Toronto, Canada, November 2017.
[37] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.
[38] S. Wen, M. Sayad Haghighi, C. Chen, X. Yang, W. Zhou, and W. Jia, "A sword with two edges: propagation studies on both positive and negative information in online social networks," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 640–653, 2014.
[39] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 529–540, Alexandria VA USA, November 2007.
[40] Z. Guo, Y. Shen, A. K. Bashir et al., "Robust spammer detection using collaborative neural network in internet-of-things applications," *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 9549–9558, 2021.
[41] Z. Cui, X. Jing, P. Zhao, W. Zhang, and J. Chen, "A new subspace clustering strategy for ai-based data analysis in iot system," *IEEE Internet of Things Journal*, vol. 8, no. 16, pp. 12540–12549, 2021.
[42] K. Yu, L. Lin, M. Alazab, L. Tan, and B. Gu, "Deep learning-based traffic safety solution for a mixture of autonomous and manual vehicles in a 5g-enabled intelligent transportation system," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4337–4347, 2021.
[43] Z. Guo, L. Tang, T. Guo, K. Yu, M. Alazab, and A. Shalaginov, "Deep graph neural network-based spammer detection under the perspective of heterogeneous cyberspace," *Future Generation Computer Systems*, vol. 117, pp. 205–218, 2021.
[44] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in c/c++ source code with graph representation learning," in *Proceedings of the 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 1519–1524, IEEE, Nevada, NV, USA, January 2021.
[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.
[46] 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 Seventeenth International Conference on Mining Software Repositories*, pp. 508–512, Seoul Republic of Korea, June 2020.
[47] Joern, "Joern," 2021, https://joern.io/.
[48] Y. Zheng, S. Pujar, B. Lewis et al., "D2a: a dataset built for ai-based vulnerability detection methods using differential analysis," in *Proceedings of the 2021 IEEE/ACM Forty Third International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 111–120, IEEE, Spain, May 2021.
[49] MITRE, "Cve Details," 2021, https://www.cvedetails.com/.
[50] D G L, "Deep graph library," 2021, https://www.dgl.ai/.
[51] Pytorch, "Pytorch," 2021, https://pytorch.org/.
[52] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks," in *Proceedings of the International Conference on Learning Representations*, San Juan, Puerto Rico, May 2016.
[53] T. N. Kipf and M. Welling, "Semi-supervised Classification with Graph Convolutional Networks," in *Proceedings of the International Conference on Learning Representations*, Toulon, France, April 2017.
[54] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph Attention Networks," in *Proceedings of the International Conference on Learning Representations*, Vancouver Canada, May 2018.