Hindawi
Security and Communication Networks
Volume 2022, Article ID 1176898, 11 pages
https://doi.org/10.1155/2022/1176898

Review Article

Code Vulnerability Detection Based on Deep Sequence and Graph Models: A Survey

Bolun Wu and Futai Zou

School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

Correspondence should be addressed to Futai Zou; zoufutai@sjtu.edu.cn

Received 16 May 2022; Revised 20 August 2022; Accepted 2 September 2022; Published 14 October 2022

Academic Editor: Xuehu Yan

With the flourishing of the open-source software community, the problem of software vulnerabilities is becoming more and more serious. Hence, it is urgent to come up with an effective and efficient code vulnerability detection method. Source code vulnerability detection techniques used in practice today like symbolic execution and fuzz testing suffer from high false positives and low code coverage, respectively. Traditional machine-learning-based solutions fail to cope with the diversity of vulnerabilities. To overcome these drawbacks, a large number of deep-learning-based code vulnerability detection works have emerged, aiming at building powerful neural network models to fully learn code semantics and vulnerability patterns. In this survey, we mainly focus on code vulnerability detection approaches based on deep sequence modeling and graph modeling technologies. Our goal is to investigate how these two methods are applied to facilitate code vulnerability detection. We also go over current prevailing datasets that are used to evaluate detection models. At last, we identify the current challenges in this field and share our views on future work.

1. Introduction

Nowadays, with the increasing number of disclosed software vulnerabilities, software vulnerability detection technology has become a major concern in the software industry and the field of cyber security. Especially, the booming of the open-source software community produces a large number of supply chain attacks. Recent work [1] has shown that attackers can abuse the open-source package managers to distribute malware, posing significant security risks to both the developers and the users and causing substantial damages financially and socially.

One of the effective mitigation methods is to scan the source code using code vulnerability detection tools before the software deployment. In practice, developers or security engineers mainly rely on code analysis or testing tools to detect and fix bugs [2, 3]. These code analysis techniques can be classified into static, dynamic, and hybrid methods. Static methods like rule-based analysis [4–9] and symbolic execution [10, 11] analyze the source code statically without execution. These methods often suffer from a high false-positive rate [12], leaving engineers with tedious work to verify false alarms. The current prevailing dynamic technique is fuzz testing [13–16]. Though it is an effective approach to discovering 0-day vulnerabilities, it faces the challenges of low analysis efficiency and low code coverage. Hybrid techniques combine static and dynamic analysis to overcome their respective drawbacks, but they are still inefficient to work in practice [17].

To overcome the aforementioned weaknesses of code analysis techniques, several advances have been made in applying machine learning (ML) to identify code vulnerabilities. ML-based methods often regard code vulnerability detection as a binary classification task. They train a supervised ML model and use that model to predict whether an unknown code sample is vulnerable or not. Early research works [18–20] mainly rely on expert experience to construct hand-crafted features which are then fed into machine learning algorithms like random forests [21] to detect vulnerabilities. However, the code patterns vary between

different types of code weaknesses [22]. Thus, it is impractical to design manual features that can express the characteristics of all vulnerability types.

Aiming to improve the detection accuracy and to free human experts from the intense labor of constructing handcrafted features, lots of research works [12, 23–36] employed deep learning (DL) models and investigated the potential of deep neural networks on automated vulnerability detection. Based on source code representation methods and the type of neural networks, DL-based code vulnerability detection approaches can be classified into two categories: sequence-based methods and graph-based methods. Sequence-based approaches [12, 23, 24, 31, 33, 34] firstly preprocess the source code into token sequences. Though works such as [12, 23] exploit data dependency graphs to extract code gadgets, they all end up representing the code gadget as a sequence of tokens. The "token" here can be either a whole line of code or a coding unit split by space. Then, a deep sequence model such as GRU [37], LSTM [38], and Transformer [39] is used to learn contextual information and defect-related semantics of source code. Graph-based methods [17, 25, 26, 28, 35, 36] convert source code into a specific graph structure. Four commonly used graph representations of code are abstract syntax tree (AST), control flow graph (CFG), program dependency graph (PDG), and code property graph (CPG) [17]. Though Zhou et al. [25] added the natural code sequence edges into CPG and Wang et al. [26] proposed an augmented AST, their graph designs are still based on the four graph backbones. After extracting the graph structure, a graph neural network (GNN) model is employed to extract the structural information and implement vulnerability detection.

The current study indicates that sequence models and graph models have comparable performance on code vulnerability detection. In this survey, we mainly focus on these two approaches and investigate how these two kinds of models can be applied to facilitate code defect discovery. We also show current mainstream datasets that can be used to evaluate sequence-based and graph-based detection methods. Then, we highlight the current challenges in DL-based code vulnerability detection and share our potential solutions.

To summarize the work of this paper, the key contribution is three-fold:

(i) firstly, we classify the current deep-learning-based code vulnerability detection approaches into sequence-based and graph-based methods. We review the latest research progress in the two fields, summarize their unified detection frameworks, and compare the different strategies used in detail.

(ii) Secondly, we review current prevailing and available datasets that can be used to evaluate a sequence-based or graph-based code vulnerability detection approach. We also present the characteristics of different datasets to help future researchers choose proper datasets or construct their own.

(iii) Finally, we discuss the challenges that this research field is facing. We also propose future work to address these issues.

2. Preliminaries

In this section, we explain the definition and the target of code vulnerability detection. Then, we present the basic knowledge of prevailing sequence and graph neural network models.

2.1. Code Vulnerability Detection. Code vulnerability detection is formalized as a binary classification problem, i.e., predicting whether a given piece of raw source code is vulnerable or not. As for deep-learning-based code vulnerability detection, let a vulnerable code dataset be defined as $((c_i, y_i)|c_i \in \mathscr{C}, y_i \in \mathscr{Y}),\ i \in \{1, 2, \ldots, n\}$, where $\mathscr{C}$ denotes the set of pieces of codes, $\mathscr{Y} = \{0, 1\}^n$ represents the label set with 1 for vulnerable code and 0 for benign code, and n is the number of instances. The goal is to learn a mapping from $\mathscr{C}$ to $\mathscr{Y}$, $f: \mathscr{C} \mapsto \mathscr{Y}$ to predict whether a piece of code is vulnerable or not. The prediction function f is expressed as a deep neural network, which can be learned by minimizing the loss function shown in

$$\min \sum_{i=1}^{n} \mathscr{L}\left(f\left(c_i, y_i|c_i\right)\right) + \lambda\omega(f), \tag{1}$$

where $\mathscr{L}(\cdot)$ is the cross-entropy loss function, $\omega(\cdot)$ is a regularization to prevent overfitting, and λ is an adjustable weight to determine the regularization degree.

The target piece of code c_i in (1) can be a file, a function, a code gadget [23], or even a line of code. Based on the granularity of the target code, we classify code vulnerability detection tasks into the coarse-grained task and the fine-grained task. In the former task, the model learns to predict whether a source file or a function is vulnerable or not [17, 18, 23, 40]. Here, we also consider the code gadget as coarse-grained data because it contains quite a few lines of inter-procedural code. Besides, in the fine-grained task, the model can identify vulnerable statements in the code [29, 30]. Vulnerable statements are defined as those code lines that are relevant to the vulnerability. It should be noted that in this paper, we mainly review studies related to coarse-grained vulnerability detection. We also discuss current and future potential solutions regarding the fine-grained task in Section 6.5.

2.2. Sequence Models

2.2.1. Recurrent Neural Network. The recurrent neural network (RNN) is a classical model to process serialized data. At each time step, the RNN correlates the input and its "memory" to update "memory" and generate an output. Due to the vanishing gradient, long short-term memory (LSTM) and gated recurrent unit (GRU) are proposed to improve the vanilla RNN. RNN-based code vulnerability detection approaches [12, 23] often split code into tokens which are then fed into models such as bidirectional LSTM (BLSTM) for prediction.

2.2.2. Transformer. With its powerful feature extraction capabilities, the transformer has dominated both NLP [41] and CV [42] fields. We mainly focus on the transformer encoder proposed by [39]. Taking a token sequence X as the input, it first converts the sequence into vectors H^0, which is computed by summing the token embeddings and positional embeddings. Then, the model applies N transformer layers to generate contextual representations $H^n = \text{Transformer}_n(H_{n-1})$, $n \in [1, N]$. Each transformer layer contains a multi-head self-attention operation followed by a feed-forward layer over the input H^{n-1}, which is formalized as

$$G^n = LN\left(\text{MultiAttn}\left(H^{n-1}\right) + H^{n-1}\right), \tag{2}$$

$$H^n = LN\left(MLP\left(G^n\right) + G^n\right), \tag{3}$$

where MultiAttn represents a multi-head self-attention mechanism, MLP is a two-layer fully-connected neural network, and LN is the layer normalization. For the nth transformer layer, the $MultiAttn$ part is computed via

$$\begin{aligned} Q_i &= H^{n-1}W_i^Q, K_i = H^{n-1}W_i^K, V_i = H^{n-1}W_i^V, \\ \text{head}_i &= \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}} + M\right)V_i, \\ \widehat{G}^n &= [\text{head}_1; \ldots; \text{head}_u]W_n^O. \end{aligned} \tag{4}$$

Here, H^{n-1} is linearly projected into query (Q_i), key (K_i), and value (V_i) using trainable parameters W_i^Q, W_i^K, W_i^V, respectively. d_k is the dimension of a head, u is the number of heads, and W_n^O is the model parameters. M is a mask matrix to avoid computation between normal tokens and padding tokens, where M_{ij} is 0 if ith token is allowed to correlate with jth token otherwise $-\infty$.

A typical successful application of transformer is BERT [41], which is a pretraining model firstly trained on a large corpus based on the combined task of masked language modeling (MLM) and next sentence prediction (NSP) and can then be fine-tuned on downstream NLP tasks.

Many recent works apply natural language models to achieve programming language understanding [31, 33, 43, 44]. Inspired by BERT, they generally follow the pretraining scheme in which the model is firstly pretrained on a large corpus containing both programming language and natural language. Then, the model is fine-tuned on a variety of downstream tasks, such as code summary generation, code clone detection, code completion, and code vulnerability detection. It is worth mentioning that these works aim at designing an effective model and pretraining tasks but not code vulnerability detection. Although it is only a downstream task to validate their approach. These researches demonstrate that transformer-based models achieve promising performance in code vulnerability detection. We will show methods based on sequence models in Section 3.

2.3. Graph Neural Network. Graph neural networks (GNNs) are proposed to apply deep learning on graphs, which is a non-Euclidean data structure. GNNs are capable of graph-related tasks like node classification, link prediction, and graph classification. Recently, a lot of research on GNNs has emerged because of its wide application in the fields of social network [45], molecular analysis [46], and so on. GNNs follow the scheme of message passing neural network (MPNN) [47], where each node aggregates embeddings of its neighbors to update its embedding. After k iterations of aggregation, one node embedding involves information from nodes within its k-hop neighborhood. These node embeddings can be utilized to perform link prediction and node classification. As for the graph classification task, a pooling method merges node embeddings into a graph embedding which is then fed into a fully-connected classifier. Studies on improving GNN structure focus on modifying either aggregation or pooling method. Previous works like GCN [48] and GAT [49] aim at enhancing the aggregation algorithm, while sort-pooling [50] and self-attention pooling [51] are proposed to improve the pooling method.

Different from regarding code as token sequences in sequence models, code vulnerability detection based on GNNs often converts source code into a graph-structured format like AST, CFG, PDG, and CPG [17, 25, 26]. Then, the graphs are fed into a GNN to compute the graph representation and the binary classification results. We will show some of these works explicitly in Section 4.

3. Method Based on Sequence Models

In this section, we review studies that adopt deep sequence models for code vulnerability detection. We first conclude a general pipeline from existing methods and then discuss them in terms of strengths and weaknesses.

3.1. Pipeline. Current studies based on deep sequence models generally follow the process of preprocessing, vectorization, and neural network modeling as shown in Figure 1. In data preprocessing, the raw source code is parsed into a sequence of tokens. Slicing and normalization techniques can be employed to produce a finer-grained node snippet and a smaller token vocabulary. Then, a tokenization algorithm is used to split the piece of code into tokens. Note that some work omits the slicing and normalization process, they directly generate tokens from the raw source code. These two different preprocessing strategies are distinguished in Figure 1 by red and blue arrows. After preprocessing, the tokens are transformed into vectors that are amenable to the neural network. As for modeling, an RNN or a transformer-based model is used to learn the contextual information inside token sequences and make the final code defect prediction. We show methods based on sequence models in Table 1.

3.1.1. Serialized Preprocessing. In preprocessing stage, to adapt the sequence input form of a sequence model, the raw source code should be transformed into a sequence of tokens.

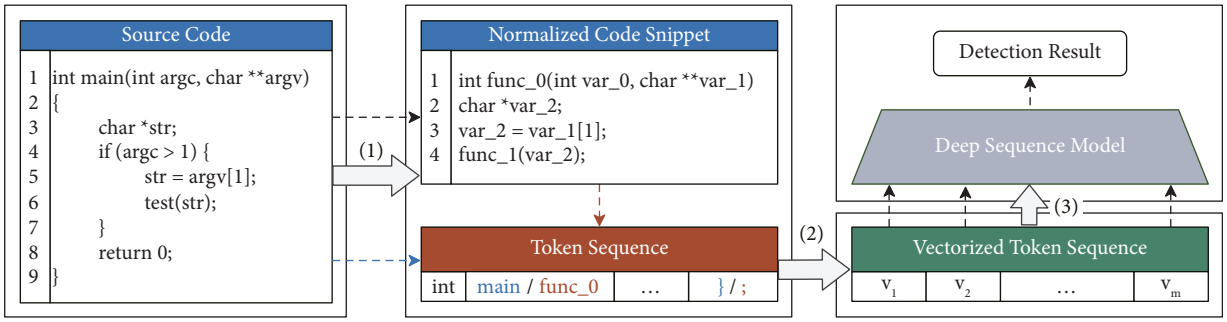

Figure 1: An illustration of code vulnerability detection based on sequence models. (1), (2), and (3) represent serialized preprocessing, token vectorization, and sequence modeling, respectively.

Table 1: Code vulnerability detection approaches based on sequence models.

Method	Slicing	Normalization	Tokenization	Vectorization	Model
VulDeePecker [23]	Code gadget	√	Lexical analysis	word2vec	BLSTM
μ VulDeePecker [12]	Code gadget, Code attention	√	Lexical analysis	word2vec	BLSTM
SySeVR [24]	SeVC	√	Lexical analysis	word2vec	BGRU
CodeT5 [34]	×	×	Byte-level BPE	Transformer's Embedding	Transformer Encoder
CoTEXT [33]	×	×	SentencePiece	Transformer's Embedding	Transformer Encoder
SynCoBERT [31]	×	×	BPE, lexical analysis	Transformer's Embedding	Transformer Encoder

RNN-based works [12, 23, 24] firstly convert the source code into a finer-grained code snippet, namely code gadget used in VulDeePecker [23] and μ VulDeePecker [12], and SeVC proposed in SySeVR [24]. The code snippet consists of statements that are data-dependent or control-dependent on potentially vulnerable statements described by a set of rules. To reduce the size of code vocabulary, they map user-defined variable and function names to symbolic names (e.g., "var_0", "var_1", "func_0", "func_1" in Figure 1), resulting in a normalized code snippet. This normalization operation mitigates the effect of customized names on model robustness because changing these names will not affect the model's prediction results. Then, a tokenization technique is used to split the code snippet into a sequence of tokens. RNN-based works achieve this via lexical analysis, which can identify identifiers, keywords, operators, and symbols automatically.

Transformer-based approaches often omit the code slicing and normalization strategies, they directly perform tokenization on the source code. Thanks to various tokenization strategies applied in the field of NLP, CodeT5 [34] uses byte-level byte-pair-encoding (BPE) [52] to segment the code into tokens. BPE repeats replacing pairs of adjacent symbols that occur most often with a new symbol until the size of vocabulary reaches the expectation. Byte-level BPE applies BPE on raw bytes instead of characters. Besides, CoTEXT [33] uses SentencePiece [53] model to extract tokens. Different from these two methods, SynCoBERT is a multi-modal model which uses both the code token sequence and the AST token sequence. The code token sequence is generated via BPE tokenization, and the AST token sequence is extracted by performing a depth-first traversal on the AST leaf nodes. Since AST generation relies on lexical analysis, we consider SynCoBERT adopts both BPE and lexical analysis in the tokenization stage as shown in Table 1.

3.1.2. Token Vectorization. After serialized preprocessing, tokens should be converted into vectors which can then be fed into a deep learning model. The above three RNN-based approaches use word2vec [54], which is a fully-connected neural network trained on the CBOW or the skip-gram task. Then, for each word, the vector representation computed by the hidden layer is regarded as the word representation. In practice, the vector representations for tokens are fixed and are directly used to perform the downstream code vulnerability detection task. This means that the token vectors cannot indicate vulnerability-related information.

Transformer-based approaches learn the word embedding by pretraining the transformer on more complicated tasks compared to word2vec. Besides, the embedding layer in transformer can be updated when training on downstream tasks. Therefore, when applying Transformer-based models to code vulnerability detection, the token embeddings contain more vulnerability-aware features.

3.1.3. Sequence Modeling. The vectorization technique transforms the sequence of tokens into an input matrix, where each row represents a token vector. Then, these vectors are fed into a sequence neural network to achieve the vulnerability binary classification.

VulDeePecker and μ VulDeePecker exploit the Bidirectional LSTM (BLSTM) instead of the LSTM because a vulnerable program function call may be affected by both the earlier statements and the later statements. BLSTM, with the bidirectional connection between LSTM cells, can perceive such contextual information. Above the BLSTM layers, some dense layers are used to reduce the number of dimensions of the vectors. At last, a SoftMax layer is responsible for providing the classification probabilities. SySeVR employs bidirectional GRU (BGRU) as the sequence modeling which is similar to BLSTM.

As for transformer-based methods, though they design different complex tasks to pretrain the model, they all use the original multi-layer transformer encoder [39] when dealing with the downstream code defect prediction task. The encoder consists of twelve transformer layers as described in Section 2.2 with 768 hidden sizes and 12 attention heads, which encode the input vectors into contextual representations. Then, the output vector regarding the [CLS] token becomes the code representation and is later fed into a dense classifier to achieve the binary classification. Since these works aim at building a novel code understanding model, they pretrain the model with both programming language and natural language (the corresponding code comment) on various pretraining objectives. For example, SynCoBERT adopts identifier prediction and AST edge prediction tasks and builds a contrastive training framework so that the model can learn semantic information from the code. The results from these works demonstrate that semantic features implicitly help the model better distinguish between vulnerable code and normal code.

3.2. Method Discussion. We discuss the strengths and weaknesses from the aspect of serialized preprocessing, token vectorization, and sequence modeling. **First**, approaches that use RNN as sequence modeling (i.e., VulDeePecker, μ VulDeePecker, and SySeVR) obtain more concise code representations by code slicing techniques while transformer-based methods (i.e., CodeT5, CoTEXT, SynCoBERT) directly utilize the source code that may contain many statements that are irrelevant to vulnerabilities. These redundant lines can weaken the impact of the vulnerable statements on the detection results. Besides, RNN-based approaches often apply code normalization to mitigate the inaccuracy brought by customized identifiers. One of the possible concerns of code slicing is that the extracted code representations may not cover all of the vulnerable code snippets. However, the work of SySeVR [24] has proved that SeVCs generated based on library function call, array usage, pointer usage, and arithmetic Expression can cover more than 90% of the vulnerabilities in 126 kinds of CWEs. The vulnerability coverage problem can be further addressed by improving the code slicing strategy. **Second**, token vectorization techniques used by transformer-based methods extract more vulnerability-aware features while word2vec used by RNN-based approaches only grab the semantics. The former solution trains the transformer's embedding layer together with the whole vulnerability detection model. Therefore, the embedding layer can implicitly capture vulnerability-related signals from the source code. By contrast, methods like word2vec are trained by predicting the adjacent tokens, extracting contextual information that is not necessarily effective for vulnerability detection. **Third**, transformers have been shown to have better capacity [39] than RNNs due to a large number of parameters and the attention mechanism. Also, transformers allow for more parallelization than RNNs, resulting in more efficient training and detection.

In general, with the same goal of achieving a precise code vulnerability performance, RNN-based methods are more concerned with code preprocessing which slices the source code into a finer format and replaces the user-defined contents with normalized symbols. On the contrary, those transformer-based approaches can extract more vulnerability-aware signals from code tokens and adopt more expressive models to predict vulnerabilities. Both solutions are proved to be effective and their advantages can be combined to obtain a more effective vulnerability detection approach.

4. Method Based on Graph Models

In this section, we will introduce current GNN-based code vulnerability detection approaches. We explain the pipeline in detail and then follow with the discussion of the methods' strengths and weaknesses.

4.1. Pipeline. As with the sequence models-based methods, GNN-based methods also consist of three steps: preprocessing, vectorization, and neural network modeling. As shown in Figure 2, in the preprocessing stage, the source code is transformed into a graph representation. Common code graph representations are AST, CFG, PDG, and CPG. Besides, similar to the approach explained in Section 3.1.1, code normalization strategy may or may not be used which is distinguished in Figure 2 by red and blue arrows. After preprocessing, the nodes and edges are converted into vectors so that the graph can be fed into a GNN model which can learn structural information and make the final prediction. We list solutions based on graph models in Table 2.

4.1.1. Graphical Preprocessing. Most of the current GNN-based works [25, 26, 28, 35] follow the classical code property graph (CPG) [17], which is a combination of the abstract syntax tree, control flow graph, and program dependency graph. To better understand the necessity of employing CPG, we will explain the structures of these four graphs in detail.

(1) Abstract Syntax Tree (AST). The abstract syntax tree is often the first intermediate representation generated by compilers to examine the code syntactic errors. As shown in Figure 3(a), beginning from the root node representing the function definition, the code is firstly split into statements, declaration, predication, etc., and then, it is divided into the leaf nodes that indicate the most basic code unit such as identifiers, assignment operators, keywords.

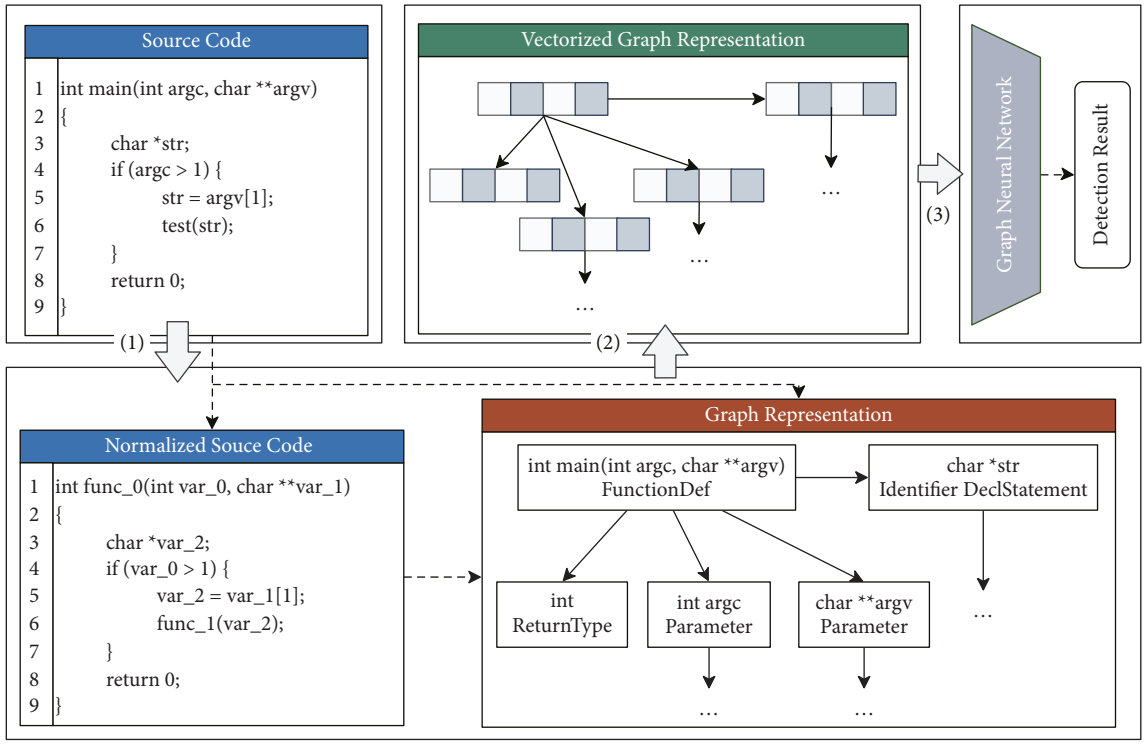

Figure 2: An illustration of code vulnerability detection based on graph models. (1), (2), and (3) represent graphical preprocessing, graph vectorization, and GNN-based graph modeling, respectively.

Table 2: Code vulnerability detection approaches based on graph neural networks.

Method	Normalization	Graph representation	Vectorization	Model
Devign [25]	×	CPG with NCS edges	word2vec, one-hot	GGNN + Conv
FUNDED [26]	√	Augmented AST	word2vec, one-hot	GGNN + Sum
VulSPG [28]	×	SPG	word2vec, one-hot	R-GCN + Attention
AI4VA [35]	×	CPG	word2vec, one-hot	GGNN + Mean
Feng et al. [36]	×	AST, CFG, CPG	word2vec, one-hot	DGCNN

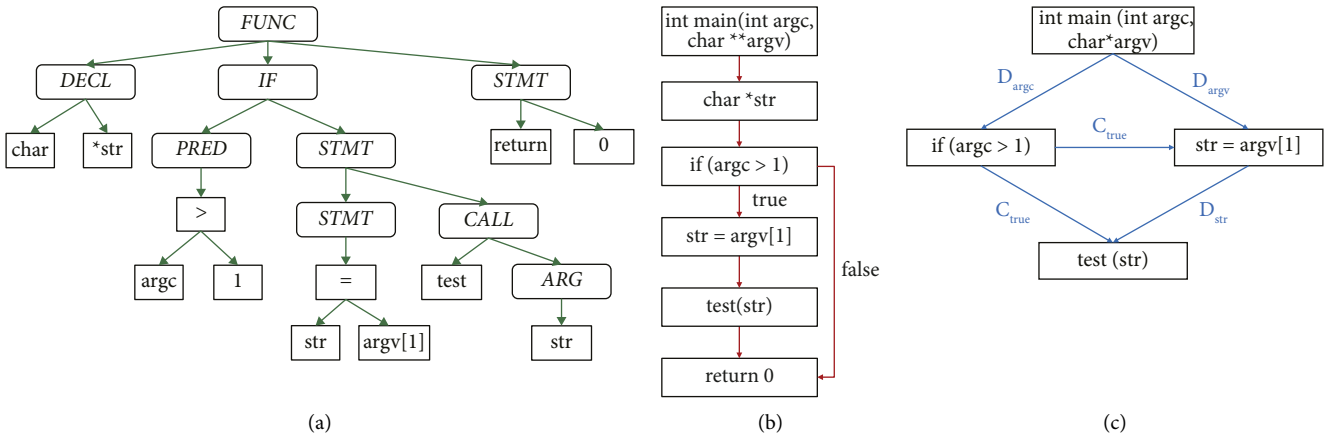

Figure 3: Three common graph representations of the source code is given in Figure 2. In the program dependency graph (c), data and control dependencies are represented by C and D. (a) Abstract syntax tree (AST). (b) Control flow graph (CFG). (c) Program dependency graph (PDG).

(2) Control Flow Graph (CFG). The control flow graph presented in Figure 3(b) explicitly describes the execution order of all code statements as well as the conditions that should be met for all the execution branching. Nodes denote statements and predicates, and the directed edges indicate the transfer of control.

(3) Program Dependency Graph (PDG). The program dependency graph in Figure 3(c) shows the control and data dependencies among the statements and predicates. The two kinds of dependencies are represented by two types of edges. The data dependency edge means some variable is declared in the code of the source node and is later used by the code in the destination node. The control dependency edge shows the influence of predicates on the values of variables.

(4) Code Property Graph (CPG). As displayed in Figure 4, the code property graph [17] uses the nodes of AST and edges of all AST, CFG, and PDG. It combines the properties of the three basic graphs into a joint data structure, which captures both syntactic and structural information of the source code.

The code graph representations that current works employ are presented in Table 2. Methods like AI4VA [35] and Feng et al. [36] directly employ the original version of the four basic graphs as their code representations. Devign [25], which is the first work that adopts GNN to tackle the task of code vulnerability detection, adds the natural code sequence (NCS) edges into CPG to encode the natural sequential information of the source code. Based on AST, FUNDED [26] adds eight additional types of edges including Data and control flows, Jump, ComputedFrom, GuardedBy, NextToken, LastUse, and LastLexicalUse to form an augmented AST. Since vulnerable statements have a very small proportion in real-world functions, CPG contains a large number of structures that are irrelevant to vulnerable patterns. To mitigate this problem, VulSPG [28] defines a novel slice property graph (SPG) and extracts SPG via program slicing technique based on predefined SyVCs criterion.

4.1.2. Graph Vectorization. After graphical preprocessing, the resulting code graph representations need to be converted into vectors to be fed into the subsequent GNN. As shown in Figure 2, in a code graph, every node possesses two attributes: code and type. The code contains the source code represented by the node and the type of node is defined by AST. All of the work listed in Table 2 encode the code of each node through a pretrained word2vec model with the source code corpus built on the whole dataset. And the type of the node is encoded as a one-hot representation. Besides, if CPG is chosen to represent the code, there are multiple types of edges in the graph. Commonly, one-hot encoding is also used to encode the edge types.

4.1.3. GNN-Based Graph Modeling. In this stage, the above extracted vectorized code graphs are fed into a graph neural network to perform vulnerability detection. Devign, FUNDED, and AI4VA make use of GGNN [55], which contains the gated recurrent unit (GRU) to update the node embeddings and the node's hidden state at each aggregation layer. However, they apply different pooling methods to merge node embeddings into a graph embedding. Devign designs a 1-D convolution module to select features that are relevant to the final detection task. FUNDED concatenates the node embeddings computed from all layers and uses their sum as the graph representation. Besides, AI4VA simply regards the average of node embeddings as the final feature. Other than GGNN, VulSPG [28] exploits R-GCN [56] and designs a novel attention mechanism to fusion the graph representation. Fent et al. [36] uses the DGCNN [50] in their graph modeling stage.

4.2. Method Discussion. Methods listed in Table 2 differ mainly in graph representations and GNN models. All of the work either use the original CPG or its enhanced version. Devign improves CPGs by adding natural code sequence edges to infuse contextual semantics. FUNDED enhances CPGs by lots of structural information and produces graph representations with nine kinds of edges. Different from the above strategies of adding structural information, VulSPG proposes to eliminate code that is irrelevant to vulnerabilities. It conducts graph slicing on CPGs to produce SPGs. As for GNN models, GGNN is commonly used due to its effectiveness. VulSPG uses R-GCN to lean different transformations for different types of edges. In general, GNN-based code vulnerability detection methods have the advantage of making the best of the code syntactic and structural information. In the meantime, the semantics are relatively underweighted because GNNs are hard to capture contextual relations between nodes far apart. Also, they often use the word2vec model to produce code embeddings that may not be vulnerability-aware. We argue that the advantages of sequence model-based methods and graph model-based methods can be fused to build a more powerful detection model in Section 6.

5. Dataset Review

To evaluate code vulnerability detection methods based on deep sequence and graph models, a large dataset of vulnerable and non-vulnerable source code is necessary. Table 3 summarizes the characteristics of a few popular publicly available code vulnerability datasets. Except for the type and the source of the dataset, we also investigate whether the dataset contains the vulnerability type of each sample which can be used to build a vulnerability classification model. Besides, we also consider if the fine-grained labels are available which are helpful for further research on fine-grained vulnerable statement localization tasks.

$\hat{x}$ means that the fine-grained labels are not included in the original dataset, but they can be obtained manually.

Juliet [57] and S-Babi [58] are both synthetic datasets that are produced according to predefined patterns. Although a large number of code samples can be synthesized and the vulnerability type and the fine-grained label can be obtained conveniently, the main drawback is the lack of code diversity compared to datasets generated from real-world programs.

The Devign dataset [25] contains real-world function examples from GitHub, which are labeled manually according to commit messages and code diffs. The vulnerability type and the fine-grained label are not available in this dataset. Besides, this dataset is included in a programming language understanding evaluation benchmark called CodeXGLUE [60], so it is widely used by those Transformer-based methods.

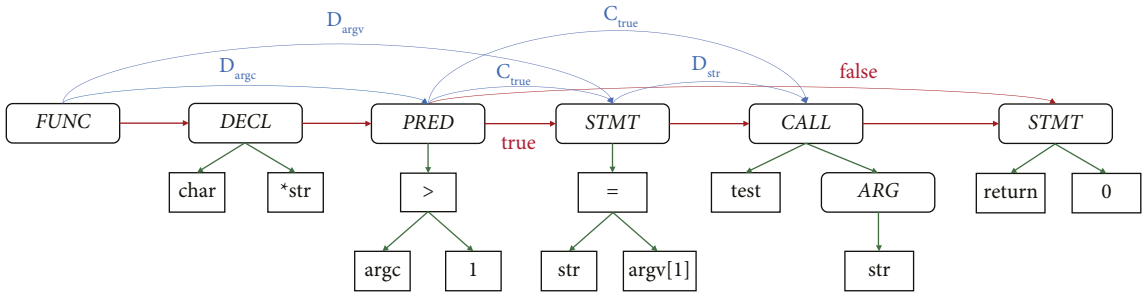

Figure 4: Code property graph of the source code is given in Figure 2.

Table 3: Publicly available dataset for code vulnerability detection.

Dataset	Type	Source	Vul type	Fine-grained label
Juliet [57]	Synthetic	SARD	√	√
S-Babi [58]	Synthetic	—	√	√
Devign dataset [25]	Real-world	GitHub	×	○[1]
CGD [23]	Real-world	NVD, SARD	√	○[1]
μ VulDeePecker dataset [12]	Real-world	NVD, SARD	√	○[1]
SySeVR dataset [24]	Real-world	NVD, SARD	√	○[1]
FUNDED dataset [26]	Real-world	NVD, SARD, GitHub	√	×
D2A [59]	Real-world	GitHub	√	√

○[1] means that the fine-grained labels are not included in the original dataset, but they can be obtained manually.

CGD [23], μ VulDeePecker Dataset [12], SySeVR Dataset [24] are all derived from NVD [61] and SARD [62]. Since the code samples are publicly confirmed bugs, the label quality is reliable. Though the fine-grained labels cannot be directly obtained from these datasets, they can be downloaded directly from SARD. As for samples from NVD, the fine-grained labels can be generated by comparing the code diffs files. Therefore, we consider that these three datasets indirectly contain fine-grained labels.

The FUNDED Dataset [26] consists of samples from NVD, SARD, and GitHub. The authors train a machine learning model to identify whether the commit is related to a vulnerability fix or not and use this model to label the samples from GitHub. The dataset includes ten CWE types that are applicable for vulnerability classification models. However, the source of the code sample is missing, so the fine-grained labels cannot be obtained.

The D2A [59] dataset is constructed based on a differential analysis labeling approach that can label issues reported by static analysis tools. It is built by analyzing version pairs from multiple open-source projects. This dataset also provides abundant vulnerability-related data such as vulnerability types, fine-grained labels, and bug traces.

While the size of the synthetic dataset is not a concern, the model trained on synthetic vulnerable code is difficult to apply to real-world vulnerability detection due to the differences in code patterns. Confirmed vulnerable samples collected from NVD are used to address this issue. However, such samples are quite limited and may not be adequate for model training. As a result, there is an urgent need for vulnerable samples from real-world programs, which can be obtained from GitHub by locating vulnerability-fixing commits. The Devign dataset contains real-world functions, but their labels are unreliable because all functions patched by the defect-fixing commit are labeled as vulnerable. The FUNDED dataset, on the other hand, lacks fine-grained labels for building DL-based vulnerability locators. The D2A dataset is collected from real-world projects based on differential analysis to label issues reported by static analysis tools. It contains the most number of samples among all the datasets and is currently the most effective DL-based code vulnerability detection dataset.

6. Challenges and Future Directions

In this section, we show the challenges that DL-based code vulnerability detection methods currently face. Then, we offer some potentially feasible solutions.

6.1. Code Representation. Current efforts either convert source code into sequence or graph representations. Sequence representations preserve semantic relations but do not include structural information. On the contrary, the semantics are underweighted in graph representations which contain rich structural features. A potential solution is to use both types of representations to utilize both semantic and structural information in code.

6.2. Code Embedding Method. Most of the current methods use pretrained code embedding techniques like word2vec. However, since these models are trained on NLP tasks like predicting neighbor tokens, they are hard to extract vulnerability-aware features from source code. We argue that producing better code embeddings can improve the upper bound of model effectiveness. Therefore, it is essential to design code embedding networks that can be trained along with the subsequent detection model. With the weight update due to the vulnerability detection loss function, the

code embedding networks can produce vulnerability-aware signals, enhancing the model performance.

6.3. Model Selection. The sequence model has the disadvantage of only capturing contextual correlations between code tokens while ignoring the rich control and data dependencies of code. Despite taking structured features into account, the GNN model is poor at learning semantic information. Though Devign tries to solve the problem by adding natural code sequence (NCS) edges, due to the characteristics of the model, GNN is not suitable for learning the sequential features because it is hard to build long-distance relations on tokens. A potentially feasible solution is to combine the advantages of sequence models and graph models to simultaneously learn both semantic and structural information from the source code. We can use a sequence model to extract the semantic representation and employ a GNN model to extract the structural representation. Then, a fusion network can be used to merge both features and make the final prediction.

6.4. Dataset. Due to the lack of a unified benchmark dataset, each study collects its dataset as shown in Table 3 and conducts the evaluation. The fairness and reliability of such experiments are questionable. Though D2A contains large numbers of vulnerable samples collected from real-world projects, it only covers six popular software programs. Therefore, a large-scale real-world dataset that covers multiple programming languages and contains precise coarse-grained and fine-grained labels is needed in this field.

6.5. Fine-Grained Vulnerability Detection. Most of the current works focus on coarse-grained detection. Engineers still have to analyze the entire function and manually locate the vulnerable statements, which is time-consuming. To raise the level, fine-grained vulnerability detection models can predict the exact statements that cause the vulnerability. This technique is more practical because instead of tediously finding the bugs manually, security engineers can efficiently identify the location of the vulnerability and revise them. Current few works on fine-grained detection follow the idea of using an explainable model. IVDETECT [29] adopts an interpretable model called GNNExplainer [63] that can explain why the model has generated its decision. Another work is VulDeeLocator [30], it adds an attention mechanism in the BRNN model and uses the values of attention weight to decide the vulnerable code locations. In general, these methods do not make use of the vulnerability locations provided by the dataset.

Since most datasets contain fine-grained labels, we argue that these labels can be used to train a supervised fine-grained detection model. For sequence-based models, the vulnerability localization task can be converted into a token classification task so that the model can predict which part of the code is vulnerable. Similarly, as for graph-based models, the fine-grained task can be treated as a node classification task. The detected vulnerable statements are the code contained in the vulnerable nodes predicted by the model.

7. Conclusion

In this survey, we provide a detailed description of the background and some preliminary code vulnerability detection techniques. We categorize current works into sequence-based and graph-based methods, and we explicitly review and compare their different detection frameworks and strategies. We argue that sequence-based approaches capture rich semantic features while graph-based models make use of the code's complex structural information. The current dominant datasets and their characteristics are then presented. Finally, we discuss the challenges in terms of code representation, code embedding methods, model selection, dataset, and fine-grained vulnerability detection. And we propose future work to address these issues.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (No. 2020YFB1807500).

References

[1] R. Duan, A. Omar, R. P. Kasturi, and R. Elder, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium*, NDSS, San Diego, California, USA, February 2021.

[2] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2015.

[3] T. D. LaToza, V. Gina, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 492–501, Shanghai, China, May 2006.

[4] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *ACM SIGOPS - Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[5] D. A. Wheeler, "Flawfinder home page," 2016, https://dwheeler.com/flawfinder/.

[6] Checkmarx, "Application Security Testing Company — Software Security Testing Solutions — Checkmarx," 2022, https://checkmarx.com/.

[7] Cppcheck Solutions AB, "Cppcheck - a Tool for Static C/c++ Code Analysis," 2022, https://cppcheck.sourceforge.io/.

[8] S. Vermeulen, "cvechecker," 2022, https://github.com/sjvermeu/cvechecker.

[9] 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, Berlin, Germany, November 2013.

[10] D. A. Ramos and D. Engler, *Under-constrained Symbolic Execution: Correctness Checking for Real Code*, USENIX Association, Washington, D.C, 2015.
[11] "Clang static analyzer," 2020, https://clang-analyzer.llvm.org/.
[12] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: a deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, p. 1, 2019.
[13] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Pearson Education, London, England, 2007.
[14] J. Chen, W. Diao, Q. Zhao, C. Zuo, and Z. Lin, *Iotfuzzer: Discovering Memory Corruptions in Iot through App-Based Fuzzing*NDSS, San Diego, California USA, 2018.
[15] Yu Zhang, W. Huo, K. Jian, and J. Shi, "Srfuzzer: an automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 544–556, San Juan, Puerto Rico, USA, December 2019.
[16] K. George, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, Toronto, ON, Canada, October 2018.
[17] 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.
[18] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pp. 1–8, Bolzano, Italy, September 2010.
[19] 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, Virginia, USA, October 2007.
[20] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
[21] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[22] The MITRE Corporation, "Cwe - common weakness enumeration," 2021, https://cwe.mitre.org/.
[23] Z. Li, D. Zou, S. Xu, and O. Xinyu, "Vuldeepecker: a deep learning-based system for vulnerability detection," 2018, https://arxiv.org/abs/1801.01681.
[24] 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, 2022.
[25] 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.
[26] 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, 2021.
[27] R. Russell, L. Kim, L. Hamilton, and T. Lazovich, "Automated vulnerability detection in source code using deep representation learning," in *Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762, IEEE, Orlando, FL, USA, December 2018.
[28] W. Zheng, Y. Jiang, and X. Su, "Vulspg: Vulnerability Detection Based on Slice Property Graph Representation Learning," in *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, Wuhan, China, October 2021.
[29] Yi Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 292–303, Athens, Greece, August 2021.
[30] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2821–2837, 2022.
[31] X. Wang, Y. Wang, F. Mi, and P. Zhou, "Syncobert: syntax-guided multi-modal contrastive pre-training for code representation," 2021, https://arxiv.org/abs/2108.04556.
[32] Y. Zhuang, S. Suneja, V. Thost, and D. Giacomo, "Software vulnerability detection via deep learning over disaggregated code graph representation," 2021, https://arxiv.org/abs/2109.03341.
[33] L. Phan, H. Tran, D. Le, and H. Nguyen, "CoTexT: multi-task learning with code-text transformer," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pp. 40–47, Association for Computational Linguistics, Bangkok, Thailand, August 2021.
[34] Y. Wang, W. Wang, S. Joty, and S. Hoi, "Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, EMNLP, Punta Cana, November 2021.
[35] S. Suneja, Y. Zheng, Y. Zhuang, Jim Laredo, and A. Morari, "Learning to Map Source Code to Software Vulnerability Using Code-As-A-Graph," 2020, https://arxiv.org/abs/2006.08614.
[36] F. Qi, C. Feng, and W. Hong, "Graph neural network-based vulnerability predication," in *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 800-801, IEEE, Adelaide, SA, Australia, September 2020.
[37] K. Cho, B. Van Merriënboer, C. Gulcehre, and D. Bahdanau, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014, https://arxiv.org/abs/1406.1078.
[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[39] A. Vaswani, N. Shazeer, N. Parmar, and J. Uszkoreit, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998–6008, 2017.
[40] H. Khanh Dam, T. Tran, T. Pham, and S. Wee Ng, "Automatic Feature Learning for Vulnerability Prediction," 2017, https://arxiv.org/abs/1708.02368.
[41] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: pre-training of deep bidirectional transformers for language understanding," 2018, https://arxiv.org/abs/1810.04805.
[42] A. Dosovitskiy, L. Beyer, K. Alexander, and D. Weissenborn, "An Image Is worth 16x16 Words: Transformers for Image Recognition at Scale," 2021, https://arxiv.org/abs/2010.11929.
[43] Z. Feng, D. Guo, D. Tang, and N. Duan, "Codebert: a pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in*

Natural Language Processing: Findings, pp. 1536–1547, Punta Cana, March 2020.

[44] D. Guo, S. Ren, S. Lu, and Z. Feng, "Graphcodebert: pre-training code representations with data flow," in *Proceedings of the International Conference on Learning Representations*, Addis Ababa, Ethiopia, April 2020.

[45] T. Bian, Xi. Xiao, T. Xu et al., "Rumor detection on social media with bi-directional graph convolutional networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 549–556, 2020.

[46] J. You, B. Liu, R. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 6412–6422, Montreal, Canada, December 2018.

[47] J. Gilmer, S. S. Schoenholz, P. F. Riley, Oriol Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the International Conference on Machine Learning*, pp. 1263–1272, PMLR, Sydney, Australia, August 2017.

[48] N. K. Thomas and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, Toulon, France, April 2017.

[49] P. Veličković, G. Cucurull, A. Casanova, and A. Romero, "Graph Attention Networks," in *Proceedings of the International Conference on Learning Representations*, Vancouver, BC, Canada, April 2018.

[50] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," *Thirty-second AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[51] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Proceedings of the International Conference on Machine Learning*, pp. 3734–3743, PMLR, Long Beach, CA, USA, June 2019.

[52] A. Radford, J. Wu, R. Child, and D. Luan, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, 2019.

[53] T. Kudo and J. Richardson, "Sentencepiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing," 2018, https://arxiv.org/abs/1808.06226.

[54] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, https://arxiv.org/abs/1301.3781.

[55] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," 2016, https://arxiv.org/abs/1511.05493.

[56] M. Schlichtkrull, T. N. Kipf, P. Bloem, and R. Berg, "Modeling relational data with graph convolutional networks," in *Proceedings of the European Semantic Web Conference*, pp. 593–607, Springer, Monterey, CA, USA, October2018.

[57] National Institute of Standards and Technology, "Juliet test suite for c/c++ version 1.3," 2018, https://samate.nist.gov/SRD/testsuite.php.

[58] D Carson, W. S. S. Sestili, and N. M. VanHoudnos, "Towards Security Defect Prediction with Ai," 2018, https://arxiv.org/abs/1808.09897.

[59] Y. Zheng, S. Pujar, B. Lewis, and L. Buratti, "D2a: a dataset built for ai-based vulnerability detection methods using differential analysis," in *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '21*, May 2021.

[60] S. Lu, D. Guo, S. Ren, and J. Huang, "Codexglue: a machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. 2102, Article ID 04664, 2021.

[61] National Institute of Standards and Technology, "National Vulnerability Database," 2022, https://nvd.nist.gov/.

[62] National Institute of Standards and Technology, "Software Assurance Reference Dataset," 2022, https://samate.nist.gov/SRD/index.php.

[63] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: generating explanations for graph neural networks," *Advances in Neural Information Processing Systems*, vol. 32, pp. 9240–9251, 2019.